**CS-499 Milestone Two  
John Kirven**

**About the artifact:**  
This artifact is a C++ implementation of a hash table, which is a data structure that acts as a map for storing key-value pairs. A hash table stores data into buckets based off a hash function, which is advantageous when searching as only the bucket that holds the desired data needs to be searched, rather than searching the entire table. I wrote this program around two years ago, and it was more or less my first interaction with C++. This hash table implementation is used to store university courses, with each course having a prefix such as “MATH” or “CSCI”, a course level such as 100 or 201, the name of the course, and a list of any prerequisites the course has.

**About the enhancement:**  
While this program perfectly fits the data structures and algorithms requirements the functionality required little tuning, so I instead decided to utilize it for the software design requirements. Since it was written while my comfort level with C++ was still growing, I assumed there were some design improvements that could be made. The final plan I landed on was porting the existing hash table structure to Java, while also creating an additional hash table structure using Java’s own HashTable implementation. This allowed me to showcase a deep understanding of both the Java programming language and various object-oriented programming (OOP) principles.  
  
A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

This is the current class and package structure of the project; it has three entry points for three various tasks. Main is the entry point that replicates the functionality of the original project, CourseGenerator is a helper class I added to generate dummy courses to assist in debugging and benchmarking, and TableBenchmark much like the name suggests benchmarks the two hash table implementations.  
  
As far as improvements go, almost every aspect of this project has been enhanced from it’s original C++ state. The core structure of the custom hash table was unchanged, porting it to Java allowed for better organization and modularity through OOP principles. I did make minor changes to user input to properly validate it and ensure the program doesn’t fall into an infinite loop or throw exceptions. I also slightly modified the search method, as the original was only comparing course level instead of course level and prefix. The original course data has no duplicate course levels, leading me to overlook that flaw when testing.  
  
 Integrating Java’s HashTable in parallel to my custom hash table implementation allowed me to validate the efficiency of my custom data structure through benchmarking both implementations, while also demonstrating good design principles using interfaces (CourseTable), polymorphism (CustomHashTable, JavaHashTable), and separation of concerns using models (Course, Node) and controllers (HashTableManager).

**Outcomes:**In completing this milestone, I believe I have demonstrated an ability to use well-founded and innovative techniques, skills, and tools in computing practices for the purpose of implementing computer solutions that deliver value and accomplish industry-specific goals, the software engineering/design outcome, while also unintentionally demonstrated the data structures and algorithms outcome.

Additionally, in writing this I believe I’m working towards designing, developing, and delivering professional-quality oral, written, and visual communications that are coherent, technically sound, and appropriately adapted to specific audiences and contexts.

**The process:**The process of enhancing this artifact was relatively straightforward, having already implemented the hash table in C++ converting it to Java was going to be a walk in the park. However, knowing that I would also be implementing Java’s HashTable in parallel with my custom hash table I originally planned to use inheritance where each table implementation would have it’s own controller class, but quickly realized that the methods in those controllers ended up being duplicated code so I chose composition over inheritance leaving me with a single controller class that has the course table passed to its constructor. When both hash tables had been fully implemented and confirmed to function correctly it was time to benchmark them. The original course information only contained a handful of courses which did a poor job of demonstrating the efficiency of the hash tables, so I created a class to generate many courses; a realistic number of courses for a university (roughly 500) were written to a file, and roughly 10000 courses are generated for benchmarking purposes.

Interestingly, I found that my custom hash table implementation consistently outperformed Java’s built-in HashTable. Across 10 benchmarks, insertions were roughly 14% faster and searches were roughly 11% faster. I did not anticipate my custom structure beating Java’s own implementation, but for this specific use a custom data structure was measurably faster than Java’s standard library implementation.

|  |  |  |  |
| --- | --- | --- | --- |
| **Custom Hash Table Insertion (μs)** | **Custom Hash Table Search (ns)** | **Java's HashTable Insertion (μs)** | **Java's HashTable Search (ns)** |
| 366 | 89 | 421 | 99 |
| 339 | 91 | 393 | 101 |
| 339 | 95 | 404 | 112 |
| 348 | 87 | 401 | 98 |
| 350 | 146 | 410 | 172 |
| 351 | 87 | 412 | 98 |
| 366 | 92 | 403 | 101 |
| 354 | 94 | 409 | 100 |
| 362 | 93 | 427 | 104 |
| 370 | 93 | 426 | 105 |
| \* Each insertion result represents the average time it takes for initialization and insertion of 10,374 courses 10,000 times | | | |
| \* Each search result represents the average time it takes for a single search to execute, 100,000 times. | | | |
| Averages |  |  |  |
| 354.5μs | 96.7ns | 410.6μs | 109ns |