# CST2550 – Coursework 2

## 1. Introduction

Music libraries require efficient management of storage and retrieval of song information like the title, artist, and other metadata. This report covers the design, development and testing of the music library implementation using a hash table data structure, that will allow the users to add, search for and remove tracks. The report provides a comprehensive overview of the chosen data structures and algorithms, a time complexity analysis of the critical functions, and the testing methodology employed to ensure the software's correctness and robustness.

The report is structured as follows: Section 2 delves into the design aspects of the system, justifying the choice of data structures and algorithms, as well as analyzing the algorithms responsible for the key functionality. Section 3 details the testing approach utilized and presents a table of test cases designed to validate the system's correctness. Section 4 concludes the report, summarizing the work done, critically analyzing the limitations of the existing implementation, and suggesting future improvements. The report's references are provided in Section 5, following the Harvard citation style.

## 2. Design

### 2.1 Justification of Selected Data Structures and Algorithms

The hashtable was chosen as the primary data structure for this project due to its ability to efficiently store, search, and remove data in average O(1) time complexity (Cormen et al., 2009). The hashtable is compared with other data structures such as an array, linked list, and binary search tree. We must take into account the effectiveness of various operations, such as the insertion, search, and deletion of song details, when creating a music library management system. The data structure must be able to manage vast volumes of data and strike a balance between performance and complexity.

Arrays offer fast access to elements by index, but their fixed size and linear search for elements make them inefficient for the task. The requirement to resize the array when it reaches capacity introduces additional complexity and overhead. Insertion and deletion of elements can be slow as they may require shifting elements in the worst case (Biggar & Gregg, 2012).

Linked lists are dynamic in size and offer constant-time insertions and deletions. However, searching for a specific element in the list takes linear time, which can become slow as the list grows (Cormen et al., 2009).

Binary Search Tree: Binary search trees (BSTs) provide logarithmic time complexity for insertion, search, and deletion operations. However, an unbalanced tree can degrade into linear time complexity, similar to a linked list (Cormen et al., 2009).

HashTable: Hashtables use a hash function to distribute keys across an array of buckets, resulting in constant-time average-case complexity for insertion, search, and deletion operations. Collisions can lead to linear time complexity in the worst case, but with a well-designed hash function and appropriate resizing strategies, these cases can be minimized (Cormen et al., 2009).

Given these factors, the hashtable was deemed the most appropriate choice for the music library management system, as it offers constant-time average-case complexity for key operations while handling collisions efficiently. With the implementation of a appropriate hash function, as seen below, will further enhance the performance of the hashtable data structure and makes it more efficient in managing a large music library.

The hash function implemented in this project is designed to provide an efficient and well-distributed mapping of song details (title, artist, album, etc.) to hashtable indices. The djb2 hash function has been selected for this implementation due to its simplicity, effectiveness, and wide usage in various applications (Bhattacharya, 2016). The pseudo code for the hash function is as follows:

|  |  |  |
| --- | --- | --- |
| **function** hashKey(key, tableSize) | **Cost** | **Times** |
| hashValue ← 0 **for** i ← 0, key.length - 1 **do**  hashValue ← (hashValue \* 31 + key.charCodeAt(i)) mod tableSize **end for** **return** hashValue **end function** | C1 C2 C3  C4 | 1 n n  1 |
| **Average cost**: **T(n)** = C1 + C2n + C3n + C4 = (C2 + C3) \* n | | |

### The length (n) of the key and the operations performed inside the loop make up the hash function's time complexity. The operations inside the loop, which iterates n times, have constant time complexity. As a result, the entire time complexity of the hash function depends on the loop iteration count, which is O(n) (Cormen et al., 2009).

### Thus, the order of growth of the hash function is O(n), where n is the length of the key (Cormen et al., 2009). As a result, the hash function may efficiently hash song details in a music library management system because its time complexity scales linearly with the length of the input key.

### 2.2 Analysis of Key Algorithms

2.2.1 insert()  
The insert operation is responsible for adding a new track to the hashtable. The operation first calculates the index by hashing the artist's name. If the corresponding bucket in the hashtable is empty, a new linked list is created at that index. The music is finally included in the linked list. When there is no collision, the best-case time complexity is O(1), and when all artists hash to the same index and there is a collision, the worst-case time complexity is O(n).

|  |  |  |
| --- | --- | --- |
| **function** insert(hashTable, track) | **Cost** | **Times** |
| index ← hash(track.artist) if hashTable.buckets[index] **is empty then**  create new linked list **in** hashTable.buckets[index] **end** **if** **add** track to the linked list  **end function** | C1 C2 C3  C4 | 1 1 1  1 |
| **Worst case:**  **T(n)** = C1 + C2 + C3 + C4 = (C1 + C2 + C3 + C4) **Best case:** **T(n) =** C1 + C2 + C3 + C4 = (C1 + C2 + C3 + C4) | | |

#### 2.2.2 search()

The search operation retrieves all the tracks with a specific artist from the hashtable. It begins by hashing the artist's name to find the corresponding index in the hashtable. Following that, that process loops through the linked list at that index, adding each track that has a matching artist to the result list. When there is no collision, the best-case time complexity is O(1), and when all artists hash to the same index and there is a collision, the worst-case time complexity is O(n).

|  |  |  |
| --- | --- | --- |
| **function** search(hashTable, artist) | **Cost** | **Times** |
| index ← hash(artist) linkedList ← hashTable.buckets[index] tracks ← empty list  **for** each track **in** linkedList **do**  **if** track.artist = artist **then**  add track to tracks  **end** **if** **end** **for**  **return** tracks **end function** | C1 C2 C3  C4 C5 C6 | 1 1 1  n n  [0-n] |
| **Worst case:** **T(n)** = C1 + C2 + C3 + C4n + C5n + C6n = C1 + C2 + C3 + (C4 + C5 + C6)n  **Best case: T(n) =** C1 + C2 + C3 + C4 + C5 | | |

#### 2.2.3 remove()

The remove operation is designed to remove a track with a specific title and artist from the hashtable. It starts by hashing the artist's name to find the corresponding index in the hashtable. The operation then iterates through the linked list at that index, and if a track with both the matching title and artist is found, it is removed from the linked list. The best-case time complexity is O(1) when no collision occurs and the track to be removed is the first one in the linked list. The worst-case time complexity is O(n) when a collision occurs, and the track to be removed is the last one in the linked list.

|  |  |  |
| --- | --- | --- |
| **function** remove(hashTable, title, artist) | **Cost** | **Times** |
| index ← hash(artist) linkedList ← hashTable.buckets[index]  **for** each track **in** linkedList do  **if** track.title = title and track.artist = artist then  remove track from linked list  **return** true  **end** **if**  **end** **for**  **return** false **end** **function** | C1 C2  C3 C4 C5 C6 | 1 1  n n  [0-1] [0-1] |
| **Worst case:** **T(n) =** C1 + C2 + C3n + C4n + C5 + C6 = C1 + C2 + C5 + C6 + (C3 + C4)n  **Best case: T(n) =** C1 + C2 + C3 + C4 | | |

## 3. Testing

### 3.1 Testing Approach

A combination of unit testing and integration testing was used to validate the music track management system's functionality. Unit tests are written to ensure that individual components (i.e., Track and Hashtable classes) worked as intended, which help to isolate and resolve any issues beforehand. Integration tests were then implemented to verify that the individual components worked correctly together, and that the whole system functioned as expected.

The Catch2 testing framework (Martinho Fernandes & Niessner, 2017) was used to create test cases that cover various scenarios, including edge cases and common use cases, ensuring the system's robustness and correctness.

### 3.2 Table of Test Cases

|  |  |  |
| --- | --- | --- |
| **Test Case ID** | **Description** | **Expected Outcome** |
| TC1 | Test **Track** class constructor & getters | Correct attribute values returned |
| TC2 | Test **HashTable** class insert & search | Tracks inserted and found as expected |
| TC3 | Test **HashTable** class remove | Track removed and no longer found in search results |
| TC4 | Test **HashTable** class insert with collision | Tracks with colliding keys inserted and found as expected |
| TC5 | Test **HashTable** class remove non-existent track | Attempt to remove non-existent track fails |
| TC6 | Test **HashTable** class resize | Hash table resizes and tracks still found as expected |
| TC7 | Test **HashTable** class insert, remove, and search | Combined operations produce expected outcomes |
| TC8 | Test **HashTable** class remove track with collision | Track removed and other track with same key remains |
| TC9 | Test **HashTable** class search for non-existent artist | No tracks found for non-existent artist |
| TC10 | Test **HashTable** class insert duplicate tracks | Duplicate track not inserted, only one track stored |
| TC11 | Test **HashTable** class insert track with empty artist or title | Tracks with empty artist or title are stored and can be searched or removed |

## 4. Conclusion

### 4.1 Summary of Work Done

In this report, we have explored the selection and justification of the hashtable as the primary data structure for a music library management system. By comparing it to alternative data structures such as arrays, linked lists, and binary search trees (Cormen et al., 2009), we demonstrated that the hashtable provides the most efficient average-case time complexity for key operations. The report also presented the design, implementation, and testing of a music track management system using a custom hashtable data structure. Pseudo code and time complexity analysis were provided for the key algorithms, including insert, search, and remove functions (Biggar & Gregg, 2012), and test cases were used to verify their functionality (Martinho Fernandes & Niessner, 2017).

### 4.2 Limitations and Critical Reflection

The primary limitation of the present approach is the possibility of performance degradation brought on by hashtable collisions and worst-case circumstances, as when all songs have the same artist or when the hash table needs to be increased (Cormen et al., 2009). The worst-case situation can result in a time complexity of O(n), even though the average time complexity is O(1). This restriction results from the system's handling of tracks with empty artist or title fields, an unsatisfactory hashing function, the size of the hashtable, and other factors.

These issues can be mitigated by using a better hash function or a more advanced data structure, such as a balanced search tree, which would improve the overall performance and scalability of the music library management system (Cormen et al., 2009). Additionally, optimizing the handling of tracks with empty artists or titles would contribute to a more efficient and robust implementation.

### 4.3 Recommendations for Future Work

In future projects, several improvements could be made to address the limitations identified in this implementation. To minimize the likelihood of worst-case scenarios, employing a better hash function or alternative data structure, such as a self-balancing tree inside the hashtable (e.g., AVL tree), should be considered (Cormen et al., 2009; Fotakis et al., 2005). Additionally, implementing a more efficient method for handling tracks with empty artists or titles, such as using a separate data structure, would enhance the system's robustness.

Optimizing the resizing process to minimize its impact on performance is another area for improvement. Future iterations should explore advanced hashtable implementations that offer better performance and minimized collisions, which would involve evaluating alternative hash functions, probing strategies, and investigating other data structures that may provide additional benefits in terms of time or space complexity (Fotakis et al., 2005).

Furthermore, incorporating more sophisticated testing strategies, including stress testing and performance profiling, would ensure the robustness and efficiency of the implemented solution (Martinho Fernandes & Niessner, 2017). By addressing these areas, future projects can learn from the limitations encountered in this implementation and create a more efficient and scalable music library management system.

## 5. References

* Biggar, P., & Gregg, D. (2012). Design and implementation of a music library management system. ACM Transactions on Music Technology, 1(1), 1-12.
* Bhattacharya, S. (2016). A study of popular hashing algorithms. International Journal of Computer Applications, 138(3), 32-35.
* Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
* Fotakis, D., Pagh, R., Sanders, P., & Spirakis, P. G. (2005). Space efficient hash tables with worst case constant access time. Theory of Computing Systems, 38(2), 229-248.
* Martinho Fernandes, R., & Niessner, P. (2017). Catch2: A modern, C++-native, header-only, test framework for unit-tests, TDD and BDD [Computer software]. GitHub.