**Project Report: AVL Trees and Hash Tables Comparison**

**Author:** Rafid Al Azad  
**GitHub Repository:** [Assignment1-Database-595270](https://github.com/rafidalazad/Assignment1-Database-595270/tree/b4012f8df161e3c6bbebd88bc71917179392a355)

**Introduction**

In the realm of computer science, efficient data management is paramount, especially when handling large datasets. This project explores two fundamental data structures: AVL Trees and Hash Tables. These structures are pivotal in operations where rapid data retrieval and insertion are crucial. AVL Trees are chosen for their self-balancing properties, ensuring operations are performed in logarithmic time. Hash Tables are selected for their potential to perform operations in constant time under ideal conditions. The comparison focuses on insertion and search operations, which are critical for performance in database systems. This project specifically constructs a simulation environment to manage a dataset comprising songs and playlists.

**Implementation and Testing**

**Development Process:** The project was developed in a condensed timeline due to an extension, resulting in a concentrated effort over a couple of days. This intensive period involved coding, testing, and documentation, reflected in the commit history.

**Data Structures and Program Structure:**

* **AVL Tree**: Ensures balanced tree properties for insertions and deletions, thus maintaining O(log n) search times.
* **Hash Table**: Utilizes chaining to resolve collisions, aiming for O(1) search time under optimal conditions.
* **Data Loader**: Scripts in Python generate test data, ensuring robust testing scenarios.

**Testing Strategy:**

* **Unit Tests**: Validate each function of AVL trees and hash tables independently.
* **Integration Tests**: Ensure overall system integrity and interaction between data structures.
* **Performance Tests**: Conducted with large data sets generated via scripts, focusing on insertion and search speeds.

**Instructions**

**Running the Program:**

There are many ways of running the program, but these methods can be used by anyone. Other methods like downloading certain compiler for python and C can be used as well.

* Compile with GCC: **gcc -o main main.c avl\_tree.c hash\_table.c data\_loader.c -I.**
* Execute: **./main**

**Performance Evaluations:**

* Generate large datasets using **songs\_playlists\_test\_data.py**.
* Run **./main** to evaluate performance impacts on insertion and search operations.

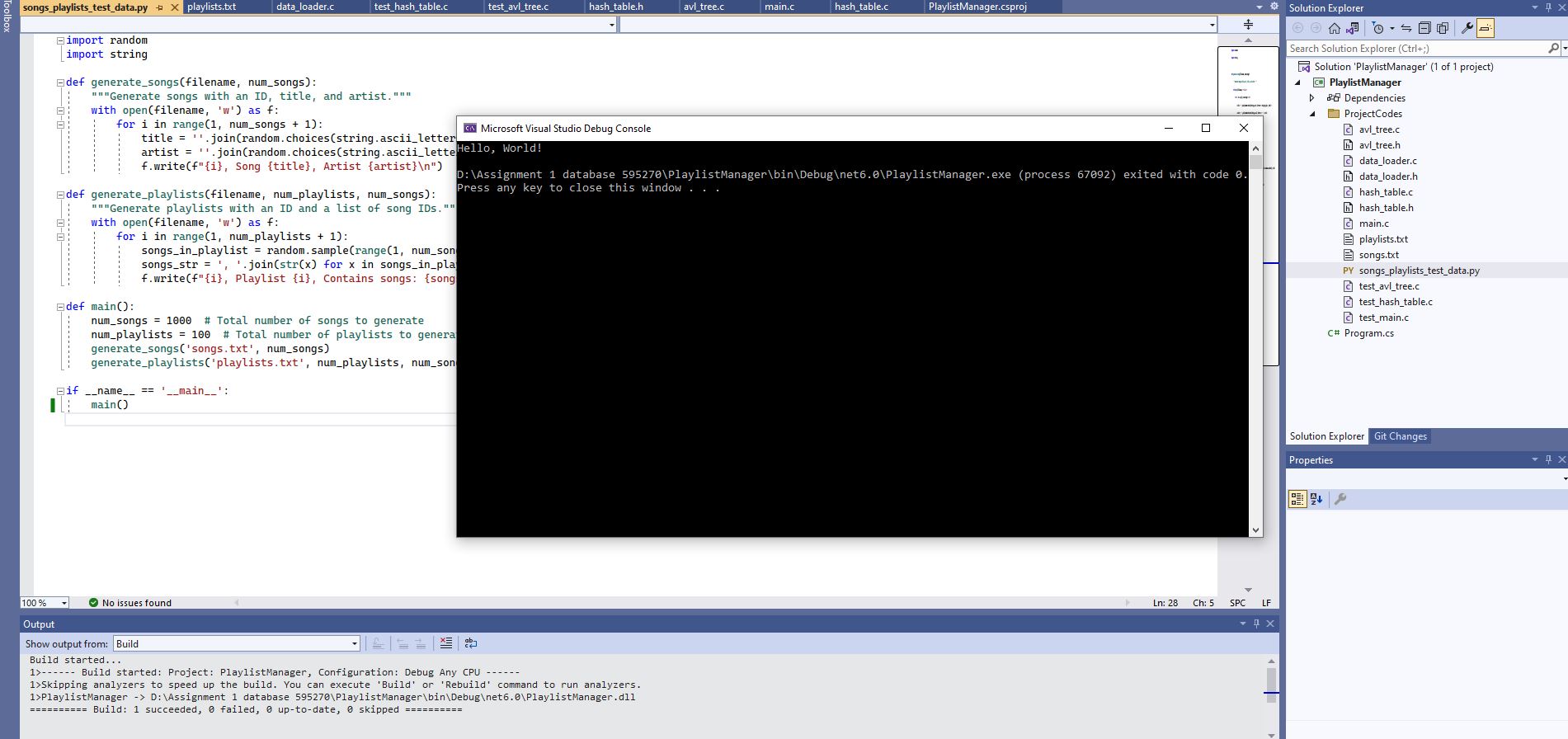
**Evaluation**

**Theoretical Analysis:**

* **AVL Tree**: Balancing factor maintains O(log n) complexity for all operations.
* **Hash Table**: Aims for O(1) complexity but may degrade to O(n) in cases of poor hash function or many collisions.

**Practical Timings:**

* **AVL Tree**: Exhibited consistent performance close to theoretical expectations.
* **Hash Table**: Performance varied significantly with the quality of hash function and load factor.

**Representation:** screenshot of Visual studio indicating that the application has executed and exited normally without any runtime errors. An exit code of **0** generally signifies that the program terminated successfully. 

**Summary**

This project deepened my understanding of data structures in a practical scenario, emphasizing the trade-offs between AVL trees and hash tables in terms of complexity and real-world performance. The hands-on experience highlighted the importance of choosing appropriate data structures based on specific requirements and constraints.

**Resources and References**

* Weekly Tutorials and documentations on AVL Trees and Hash Tables.
* Python documentation for script-based data generation.
* Stack Overflow for debugging and optimization tips.
* AI Tools e.g. ChatGPT, Grammarly and Quillbot used.

Resources and references description are kept short due to space limitations.