Corey Hyde

003044072

MSCS-532-B01

Project Phase 1

**Application Context**

Predicting the outcomes of sporting events, particularly horse racing, requires historical data analysis, efficient data handling, and proper algorithmic modeling. This project explores the design and implementation of optimized data structures in Python to support a horse racing prediction system. The focus will be on memory usage, scalability, and balancing time complexity to develop an efficient and sustainable application.

This application will focus on predicting horse racing outcomes based on past performance data, incorporating factors such as speed and winning percentage. Since horse racing offers dynamic, real-world data, it presents a valuable opportunity to apply data structures in predicting future race results. The core objective of this application is to leverage historical data to forecast upcoming race outcomes. As Garnica Caparros (2023) observes, “the management, modeling, and complex analysis of data have emerged as key drivers of innovation across various fields. This has led to the rise of data science as a multidisciplinary field, combining elements from computer science, statistics, and domain-specific knowledge” This reinforces the importance of applying structured, multidisciplinary techniques, blending computer science with domain-specific knowledge, to solve complex, real-world problems like forecasting race performance.

**Data Structures**

Several key data structures can support this application, including hash tables, binary search trees, and heaps:

**Hash tables** enable quick lookups of previous race times and other relevant data, such as information on horses, jockeys, and tracks. Retrieve past horse racing statistics and index the stats to the horses by horse\_id enabling average-case O(1) lookups. This structure is well-suited for accessing detailed performance metrics such as average race time and win percentage. Additionally, hash tables support nested dictionaries, enabling the storage of hierarchical data, for instance, segmenting a horse's performance by track type or weather conditions. As noted by Iacono and Pǎtraşcu (2011), “hash tables have been used to solve the dictionary problem faster than binary search for more than half a century,” underscoring their enduring importance in systems that demand rapid data retrieval.

**Binary search trees** allow for efficient range queries and data ranking based on performance metrics. Self-balancing binary search trees, such as AVL or red-black trees, maintain consistent O(log n) performance even as the dataset grows. Their structure facilitates efficient grouping of high and low performers, percentile calculations, and extraction of sorted lists based on race outcomes. These trees also excel at tracking longitudinal metrics, making it easy to compare a horse’s recent results to its historical averages.

**Heaps** offer an efficient solution for monitoring top and bottom performers by preserving maximum or minimum values in O(log n) time. Max-heaps enable rapid identification of the fastest horses or most successful jockeys, while min-heaps highlight underperformers or riskier bets. This functionality supports dynamic leaderboards that update in real time with incoming race data. Furthermore, heaps can drive priority queue implementations for race scheduling or simulation, assigning elevated priority to high-ranking competitors. The heap structure enables dynamic ranking and prioritization through efficient O(log n) insertion and extraction operations, making it an ideal choice for maintaining real-time leaderboards (Williams, 1964).

**Design The Data Structures**

**Hash Tables** – This project utilizes Python’s built-in dict structure to organize horse performance data under a single identifier: horse\_id. Each horse ID maps to a nested dictionary containing race-specific statistics such as race time, track condition, weather, and other relevant metrics. This hierarchical organization enables quick retrieval of context-sensitive insights, including track preferences and winning percentages. With an average time complexity of O(1), Python dictionaries offer rapid access to extensive datasets, support real-time querying, and streamline implementation through an intuitive interface. Their hash-backed design ensures scalability, handling large volumes of data with minimal performance impact.

**Binary Search Tree**- This data structure will manage numeric values like a horse’s average speed and timed finishes over specific distances. Each node will store a tuple, maintaining balanced tree height to ensure efficient data access. To support consistent performance as the dataset scales, a self-balancing binary search tree—such as an AVL or red-black tree—may be employed, offering O(\log n) complexity for insertions and lookups. This setup efficiently handles sorting operations and facilitates targeted queries. For example, identifying horses with average finish times between 65 and 70 seconds can be quickly performed using an in-order traversal.

**Heaps**- This structure will utilize a min heap to track the slowest race times and simulate a max heap (via value inversion) to monitor the fastest average times per horse, both implemented using Python’s heapq module. After each race, the heaps are dynamically updated to ensure rankings reflect the most recent performance data. Heaps excel at adapting to fluctuating metrics, making them ideal for filtering and ranking horses in real time. They also support efficient top-k queries—such as identifying the five fastest horses—with a time complexity of O(k log n), making them a powerful tool for generating dynamic leaderboards.

**Limitations**

There are potential limitations with my setup, particularly regarding scalability and complex queries. While hash tables and heaps are efficient, issues may arise as the dataset grows, requiring more memory-efficient structures or batch updates. Additionally, as the application expands, the need for more complex queries will increase, which could be challenging given the current use of hash tables and heaps.

**Python Implementation**

The Python implementation for this project has several layers, but to summarize, it loads horse racing data from a spreadsheet and uses a hash table for faster lookups. It also employs an AVL tree to rank horses by win ratio, and two heaps (or leaderboards) to display the top horses’ average speeds and win ratios. This code demonstrates the use of the different data structures discussed earlier—hash tables, binary search trees, and heaps.

Hash Table

A computer screen shot of a computer program

AI-generated content may be incorrect.

AVL Tree

A screen shot of a computer program

AI-generated content may be incorrect.

Heap

A screenshot of a computer program

AI-generated content may be incorrect.

Main Code with Results

A screen shot of a computer

AI-generated content may be incorrect.

**References**

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms* (4th ed.). The MIT Press.

Garnica Caparros, M. (2023). *Information systems in analytical applications in sports: A data modeling perspective* [Doctoral dissertation, German Sport University Cologne]. Retrieved from: <https://fis.dshs-koeln.de/en/publications/information-systems-in-analytical-applications-in-sports-a-data-m>

Iacono, J., & Pǎtraşcu, M. (2011). *Using hashing to solve the dictionary problem (in external memory)*. *arXiv preprint*. Retrieved from <https://arxiv.org/abs/1104.2799>

Williams, J. W. J. (1963). Algorithm 232: Heapsort. *Communications of the ACM, 7*(6), 347–348. https://dl.acm.org/doi/10.1145/512274.3734138