Vector:

Vectors are straightforward to implement and offer fast access times due to their contiguous memory storage. The time used for reading and going through data is O(n), and inserting elements is typically O(1) on average. However, vectors can become inefficient due to the potential overhead of resizing, leading to a worst-case complexity of O(n) for insertions. Vectors are not well-suited for frequent insertions or deletions, which could lead to performance issues as the dataset grows.

Vectors	Line Cost	# Times Executes	Total Cost
Opening and reading a file	1	O(n)	O(n)
Each line and Creating courses Objects	1	O(n)	O(n)

Hash Table:

Hash tables provide excellent average-case performance. Reading and parsing data remains O(n), while insertion into the hash table is O(1) on average. The worst-case complexity for hash tables can degrade to O(n^2) if many collisions occur, although this is highly unlikely with a good hash function. Hash tables are memory efficient and offer rapid lookup, insertion, and deletion times, making them a strong candidate for this project. The primary drawback lies in managing collisions, which can complicate the implementation.

Hash Table	Line Cost	# Times Executes	Total Cost
Opening and reading a file	1	O(n)	O(n)
Each line and Creating courses Objects	O(1)	O(n)	O(n)

Binary Search Tree:

Binary search trees offer efficient search, insert, and delete operations, especially when balanced, with an average-case time complexity of O(log n). The worst-case problem can degrade to O(n^2) if the tree becomes unbalanced, leading to inefficiencies. Trees also require additional memory to store pointers and come with the added complexity of maintaining balance.

Binary Search Tree	Line Cost	# Times Executes	Total Cost
Opening and reading a file	1	O(n)	O(n)
Each line and Creating courses Objects	O(log n)	O(n)	O(n log n)

Based on the analysis, I recommend using a hash table for this project. Hash tables offer the best average-case performance for operations such as insertions, lookups, and deletions, all of which run in O(1) time. They are also efficient in terms of memory usage, particularly when appropriately sized. While hash tables have a worst-case performance of O(n) due to potential collisions, this risk can be minimized with a well-designed hash function and careful load factor management.

In contrast, although a tree structure could provide more consistent performance if balanced, the complexity and overhead involved in maintaining that balance may not be justified for this specific application. A vector is straightforward to implement but may run into inefficiencies due to resizing and slower lookups as the dataset grows.