Runtime Analysis

Vector

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Cost | Times Run | Total Time |
| Read line | 1 | n | O(n) |
| Split line | 1 | n | O(n) |
| Validate tokens | 1 | n | O(n) |
| Create Course object | 1 | n | O(n) |
| Append to vector | 1 | n | O(n) [O(n^2) worst] |

Vector is probably the simplest and most efficient for adding items. However, there is a trade-off – it doesn’t keep the courses sorted unless we explicitly add that in there. And, as the vector increases in size, we may hit a space limit, so the vector will need to double in size and copy all of the old elements into the new array.

Hash Table

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Cost | Times Run | Total Time |
| Read line | 1 | n | O(n) |
| Split line | 1 | n | O(n) |
| Validate tokens | 1 | n | O(n) |
| Create Course object | 1 | n | O(n) |
| Insert in hash table | 1 | n | O(n) [O(n^2) worst] |

Hash tables are great for fast insertions in constant time as long as there aren’t enough collisions. If this happens, we resort to storing these elements in a linked list (as we’ve done here). However, if many, *many* course numbers collide, each attempt at inserting a new course number will itself take O(n). Squaring that results in a worst-case scenario of O(n^2). Even further, however, in no scenario are these elements maintaining order, so we still need to sort before printing.

BST

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Cost | Times Run | Total Time |
| Read line | 1 | n | O(n) |
| Split line | 1 | n | O(n) |
| Validate tokens | 1 | n | O(n) |
| Create Course object | 1 | n | O(n) |
| Insert into BST | Log n | n | O(n log n) |

In a standard BST such as the one in the previous pseudocode, we opted to assume a balanced BST for an insertion time of O(log n). We can print the course list in order without extra sorting, but the insertion itself takes longer than the other methods.

Analysis

Our Big O analyses of each data structure revealed that both Vectors and Hash Tables are easy methods of inserting new elements and printing them out (if unordered), the performance of these structures can easily become bogged down as the amount of data inserted into the structure increases, which increases the runtime while sorting through the structures. The BST, on the other hand, guarantees O(n log n) time for both building and searching, all the while keeping the course list sorted, so no extra sorting is needed. Hash Tables can offer faster lookup times for average cases, but considering these courses are somewhat connected to each other through prerequisites, we invite the possibility of worst-case behavior that can degrade this performance by a significant amount, at the same time not even preserving the order of the elements. The BST is consistent, ordered, and straightforward.