# Evaluation

## Vector Data Structure

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Open file for reading | 1 | 1 | 1 |
| Read each line in file | 1 | n | n |
| Split line into tokens | 1 | n | n |
| Create course object | 1 | n | n |
| Append course to vector | 1 | n | n |
| Total Cost | | | 4n + 1 |
| Runtime | | | O(n) |

Analysis:  
The vector implementation performs a scans linearly through the input file and appends each course to the end of the vector. Each operation executes only once per line. Because of this, the total cost is proportional to the number of courses (n). Since the append operation is constant time, the overall complexity is O(n).  
  
Advantages: Implementation is very simple. Sequential access is straightforward. Offers predictable performance.

Disadvantages: Searching for or validating the prerequisites requires scans to be completed linearly, which can be inefficient at scale.

## Hash Table Data Structure

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Open file for reading | 1 | 1 | 1 |
| Read each line in file | 1 | n | n |
| Split line into tokens | 1 | n | n |
| Create course object | 1 | n | n |
| Insert course into hash table | 1 | n | n |
| Total Cost | | | 4n + 1 |
| Runtime | | | O(n) (average case) |

Analysis:  
Inserting each course into the hash table requires constant time on average, assuming minimal collisions. The total running time is O(n) because each of the n courses is only processed one time. In the worst case of having many collisions, performance can degrade to O(n²), but this is rare as long as your hash function is done well.  
  
Advantages: Lookups and inserts are very fast(constant time on average). This is efficient for searching by course number.

Disadvantages: Hash collisions are possible. Storage can be unordered (data not automatically sorted). Hashing overhead can lead to slightly higher memory usage.

## Binary Search Tree (BST) Data Structure

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Open file for reading | 1 | 1 | 1 |
| Read each line in file | 1 | n | n |
| Split line into tokens | 1 | n | n |
| Create course object | 1 | n | n |
| Insert course into BST | log n | n | n log n |
| Total Cost | | | 3n + n log n + 1 |
| Runtime | | | O(n log n) (average), O(n²) (worst) |

Analysis:  
Traversing the tree is necessary for each course insertion, which takes O(log n) time in a balanced tree. When repeated n times, the total runtime becomes O(n log n). Although, if the tree becomes unbalanced (i.e. if data arrives sorted), insertion time can degrade to O(n²).  
  
Advantages: A BST in this context maintains courses in a sorted (alphanumeric) order and supports efficient range queries.

Disadvantages: Creating a BST can be more complex to implement and slower than a hash table for lookups. Performance can degrade if the tree is not balanced.