## Introduction:

In this document, we'll examine three data structures—Linked List (via a Vector), Binary Search Tree (BST), and Hash Table—for organizing and accessing course data. Each structure will be analyzed based on runtime complexity, practicality, and efficiency relative to the project's requirements. We aim to clearly illustrate why each structure performs differently and why one stands out as the best fit.

## Runtime Analysis for LinkedList

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **readFile:** *read and validate* | 2 | n | 2n |
| **courseFactory:**  *Parse* | 1 | n | n |
| **courseFactory**: *prerequisites* | 1 | k | k |
| **readFile:** *append ventor* | 1 | n | n |
| **Total Cost** | | | n(3+k) |
| **Runtime** | | | O(n) |

Reading and validating lines from a file generally involve operations like fetching the line and ensuring it's formatted correctly. Although in reality, fetching data from a file system might be more expensive due to I/O overhead, we simplify here for clarity and assume each operation as a constant cost. Parsing is straightforward—each course entry is split into tokens. The vector append operation is constant time amortized, making it efficient for linear operations. The overall runtime is linear, O(n), making this structure efficient for basic insertions but less ideal when searching or sorting frequently.

## Runtime Analysis BST

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **readFile:** *read and validate* | 2 | n | 2n |
| **courseFactory:**  *create, parse prereq* | 1+k | n | n+nk |
| **insertNode:** | logn | n | nlogn |
| **Total Cost** | | | n(3+k)+nlogn |
| **Runtime** | | | nlogn |

Unlike the linear structure, inserting into a BST has additional complexity due to the nature of its ordered placement. Each insertion operation requires traversing down the tree, causing a logarithmic cost per insertion. While initially more costly than simply appending elements to the vector, this upfront complexity provides substantial long-term benefits. Specifically, this complexity ensures that the data is inherently organized, enabling efficient lookups and sorted traversals. This built-in order reduces subsequent overhead when sorting or searching.

## Runtime Analysis HashTable

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **readFile:** *initialize* | O(tableSize) | 1 | O(tableSize) |
| **readFile:** *read and validate* | 2 | n | 2n |
| **courseFactory:**  *create, parse prereq* | 1+k | n | n+nk |
| **insert:** | O(m) | n | ~n (m is small) |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | n(3+k) |
| **Runtime** | | | O(n) |

Hash tables offer extremely fast insertions and lookups on average due to the hash function efficiently mapping keys to indices. Each insertion is quick, involving a simple calculation and insertion at the bucket head. However, a significant limitation arises because hash tables do not inherently maintain sorted order. To display sorted data, an external sorting operation (like merge sort) must occur, increasing complexity and memory use.

## Conclusion:

Considering the project's primary goals—efficient searching and the ability to display courses in sorted order—the Binary Search Tree (BST) clearly emerges as the most suitable data structure. It adeptly balances insertion complexity with substantial long-term performance gains in searching and sorting.

While the Linked List via Vector is simpler, its linear searching time quickly becomes impractical as the dataset grows. The Hash Table, despite fast searches, is cumbersome when sorting becomes frequent, due to the need for external sorting.

In conclusion, the BST is a clean, scalable, and well-rounded solution, perfectly aligning with the project's needs and providing the necessary efficiency for realistic usage scenarios.