**Vector:**

| **Code: Opening and Reading File** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **inputFile.open(“courses.csv”)** | 1 | 1 | 1 |
| if(!inpuFile.isopen) | 1 | 1 | 1 |
| display “File not found” | 1 | 1 | 1 |
| return(1) | 1 | 1 | 1 |
| while(!EOF) | 1 | n | n |
| if (< 2 values on the line) | 1 | n | n |
| display “Incorrect format” | 1 | 1 | 1 |
| return(1) | 1 | 1 | 1 |
| else if(>2 values on the line) | 1 | n | n |
| if (3 or more parameters and already in vector) | 1 | n | n |
| else if (3 or more parameters and not in vector) | 1 | n | n |
| display “Error no previous prerequisite record found”) | 1 | 1 | 1 |
| return 1 | 1 | 1 | 1 |
| **Total Cost** | | | 5n + 7 |
| **Runtime** | | | O(n) |

| **Code: Creating a Course Object** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| while(!EOF) | 1 | n | n |
| Declare courseID | 1 | n | n |
| Declare name | 1 | n | n |
| Declare prerequisites | 1 | n | n |
| get courseId, name, and prerequisite from file line | 1 | n | n |
| Course course(courseId, name, prerequisite) | 3 | 3n | 3n |
| courses.push\_back(course) | 1 | n | n |
| inputFile.close() | 1 | 1 | 1 |
| **Total Cost** | | | 9n + 1 |
| **Runtime** | | | O(n) |

**Hash Table:**

| **Code: Opening and Reading File** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **inputFile.open(“courses.csv”)** | 1 | 1 | 1 |
| if(!inpuFile.isopen) | 1 | 1 | 1 |
| display “File not found” | 1 | 1 | 1 |
| return(1) | 1 | 1 | 1 |
| while(!EOF) | 1 | n | n |
| if (< 2 values on the line) | 1 | n | n |
| display “Incorrect format” | 1 | 1 | 1 |
| return(1) | 1 | 1 | 1 |
| else if(>2 values on the line) | 1 | n | n |
| if (3 or more parameters and already in vector) | 1 | n | n |
| else if (3 or more parameters and not in vector) | 1 | n | n |
| display “Error no previous prerequisite record found”) | 1 | 1 | 1 |
| return 1 | 1 | 1 | 1 |
| **Total Cost** | | | 5n + 7 |
| **Runtime** | | | O(n) |

| **Code: Creating a Course Object** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| while(!EOF) | 1 | n | n |
| Declare courseID | 1 | n | n |
| Declare name | 1 | n | n |
| Declare prerequisites | 1 | n | n |
| get courseId, name, and prerequisite from file line | 1 | n | n |
| Course course(courseId, name, prerequisite) | 3 | 3n | 3n |
| courses.insert(course) | 1 | n | n |
| inputFile.close() | 1 | 1 | 1 |
| **Total Cost** | | | 9n + 1 |
| **Runtime** | | | O(n) |

**BST:**

| **Code: Opening and Reading File** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **inputFile.open(“courses.csv”)** | 1 | 1 | 1 |
| if(!inpuFile.isopen) | 1 | 1 | 1 |
| display “File not found” | 1 | 1 | 1 |
| return(1) | 1 | 1 | 1 |
| while(!EOF) | 1 | n | n |
| if (< 2 values on the line) | 1 | n | n |
| display “Incorrect format” | 1 | 1 | 1 |
| return(1) | 1 | 1 | 1 |
| else if(>2 values on the line) | 1 | n | n |
| if (3 or more parameters and already in vector) | 1 | n | n |
| else if (3 or more parameters and not in vector) | 1 | n | n |
| display “Error no previous prerequisite record found”) | 1 | 1 | 1 |
| return 1 | 1 | 1 | 1 |
| **Total Cost** | | | 5n + 7 |
| **Runtime** | | | O(n) |

| **Code: Creating a Course Object (Recursive)** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| while(!EOF) | 1 | n | n |
| Declare courseID | 1 | n | n |
| Declare name | 1 | n | n |
| Declare prerequisites | 1 | n | n |
| get courseId, name, and prerequisite from file line | 1 | n | n |
| Course course(courseId, name, prerequisite) | 3 | 3n | 3n |
| courses.insert(course) | 1 | n | n |
| inputFile.close() | 1 | 1 | 1 |
| **Total Cost** | | | 9n + 1 |
| **Runtime** | | | O(n) |

As per the advisor’s requirements, three data structures—vectors, hash tables, and likely a binary search tree (BST)—are under consideration. Functionally outlined, each performs similarly in the worst-case scenario for their respective algorithms, but their efficiency varies with different functionalities.

Vectors, adjecent to arrays in syntax. It’s advantages that it can dynamically allocate size on the heap, allowing seamless handling of any number of elements (n) to be reviewed. They're simple structures requiring minimal prior knowledge for data manipulation. However, their disadvantage lies in searching for an element, even when sorted, resulting in a worst-case scenario of O(n), risking a stack overflow for larger datasets due to limited efficiency-improving functions.

Hash tables use a vector as their underlying structure but employ keys and values in a table format. When collisions are absent or managed through modified hash tables, each value has a unique key. The key is obtained by inputting the value into a hash function, transforming it into an index within the vector. In ideal conditions, with no collisions or complexity, ADT functions like insert, remove, and search operate incredibly swiftly with a worst-case time complexity of O(1) due to the hash function. However, complexities arise from techniques like linear probing, quadratic probing, or chaining can escalate the time complexity to O(n) or worse, nullifying the structure's initial advantages.

Trees, particularly BSTs, are more intricate but highly efficient due to their inherent properties. Typically, trees consist of a root node and may have left and right children, extending until leaf nodes devoid of children. In a BST, a node can have a maximum of two children. BSTs excel in executing common ADT functions like insert, remove, and search swiftly, even with substantial data, without encountering stack overflow issues. Insertion and search operations demand O(log(n)) time complexity due to BST properties. However, removal is more intricate, involving various cases such as removing internal nodes, leaves, or the root, each necessitating distinct functionalities, leading to drawbacks. However with BSTs, being the most complex among the provided structures, provide challenges in applying an ADT within a programming task. Additionally, using recursion should be limited to smaller datasets, as the worst-case scenario for a recursive function is O(), extremely slow for larger datasets. For instance, a tree with 12000 nodes could potentially exhaust memory due to recursion.

The data structure that has the most advantages when not considering the time complexity analysis are BSTs. BSTs are incredibly efficient when dealing with large data sets which save time and memory when executing certain functions. Even though it could have the same worst case scenario as the other two data structures it is uncommon for it to occur because it is just worst case. Having a responsive, fast, and reliable data structure is paramount for storing course objects and any functions necessary to manipulate the data.