|  |  |  |  |
| --- | --- | --- | --- |
| **Vector Data Structure - Opening, reading, parsing, and checking format** | **Line Cost** | **# Times Executes** | **Total Cost** |
| ifstream file(csvFile) | 1 | 1 | 1 |
| if file is not open, throw an error | 1 | 1 | 1 |
| string line | 1 | 1 | 1 |
| while getline(file, line) | 1 | n | n |
| if the line is empty, continue | 1 | n | n |
| if the line doesn't end with a comma | 1 | n | n |
| add a comma to the end | 1 | n | n |
| vector<string> tokens | 1 | n | n |
| stringstream ss(line) | 1 | n | n |
| string token | 1 | n | n |
| while getline (ss, token, ',') | 1 | n | n |
| add token to end of tokens vector | 1 | n | n |
| if the tokens vector size is < 2, throw an error | 1 | n | n |
| create CourseInfo course instance | 1 | n | n |
| set course.code = token at index 0 of tokens vector | 1 | n | n |
| set course.name = token at index 1 of tokens vector | 1 | n | n |
| for i=2, i++ while i is < tokens vector size | 1 | n | n |
| if tokens vector at i is not empty | 1 | n | n |
| add token at index i to end of course.prerequisites | 1 | n | n |
| add course to end of courses vector | 1 | n | n |
| file.close(); | 1 | 1 | 1 |
| **Total Cost** | | | 17n+4 |
| **Runtime** | | | O(n) |

**Vector**

|  |  |  |  |
| --- | --- | --- | --- |
| **Hash Table Data Structure - Opening, reading, parsing, and checking format** | **Line Cost** | **# Times Executes** | **Total Cost** |
| ifstream file(csvFile) | 1 | 1 | 1 |
| if file is not open, throw an error | 1 | 1 | 1 |
| string line | 1 | 1 | 1 |
| while getline(file, line) | 1 | n | n |
| if the line is empty, continue | 1 | n | n |
| if the line doesn't end with a comma | 1 | n | n |
| add a comma to the end | 1 | n | n |
| vector<string> tokens | 1 | n | n |
| stringstream ss(line) | 1 | n | n |
| string token | 1 | n | n |
| while getline (ss, token, ',') | 1 | n | n |
| add token to end of tokens vector | 1 | n | n |
| if the tokens vector size is < 2, throw an error | 1 | n | n |
| create CourseInfo course instance | 1 | n | n |
| set course.code = token at index 0 of tokens vector | 1 | n | n |
| set course.name = token at index 1 of tokens vector | 1 | n | n |
| for i=2, i++ while i is < tokens vector size | 1 | n | n |
| if tokens vector at i is not empty | 1 | n | n |
| add token at index i to end of course.prerequisites | 1 | n | n |
| call the Insert(course) method pointed to by hashTable | n | n | n2 |
| file.close(); | 1 | 1 | 1 |
| **Total Cost** | | | n2+16n+4 |
| **Runtime** | | | O(n2) |

**Hash Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **Hash Table Data Structure - Inserting a course object using the Insert() method** | **Line Cost** | **# Times Executes** | **Total Cost** |
| set unsigned int currentKey = hash(the course info to hash) | 1 | 1 | 1 |
| if the course at currentKey has key = max possible value | 1 | 1 | 1 |
| set course at currentKey = Node(course, currentKey) | 1 | 1 | 1 |
| else | 1 | 1 | 1 |
| create Node pointer nextCourse = content of course at currentKey | 1 | 1 | 1 |
| while the course after nextCourse != nullptr | 1 | n | n |
| set nextNode = the course after itself | 1 | 1 | 1 |
| set the course after nextNode = a new pointer for Node(course, currentKey) | 1 | 1 | 1 |
| **Total Cost** | | | n+7 |
| **Runtime** | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Binary Search Tree Data Structure - Opening, reading, parsing, and checking format** | **Line Cost** | **# Times Executes** | **Total Cost** |
| ifstream file(csvFile) | 1 | 1 | 1 |
| if file is not open, throw an error | 1 | 1 | 1 |
| string line | 1 | 1 | 1 |
| while getline(file, line) | 1 | n | n |
| if the line is empty, continue | 1 | n | n |
| if the line doesn't end with a comma | 1 | n | n |
| add a comma to the end | 1 | n | n |
| vector<string> tokens | 1 | n | n |
| stringstream ss(line) | 1 | n | n |
| string token | 1 | n | n |
| while getline (ss, token, ',') | 1 | n | n |
| add token to end of tokens vector | 1 | n | n |
| if the tokens vector size is < 2, throw an error | 1 | n | n |
| create CourseInfo course instance | 1 | n | n |
| set course.code = token at index 0 of tokens vector | 1 | n | n |
| set course.name = token at index 1 of tokens vector | 1 | n | n |
| for i=2, i++ while i is < tokens vector size | 1 | n | n |
| if tokens vector at i is not empty | 1 | n | n |
| add token at index i to end of course.prerequisites | 1 | n | n |
| call the Insert(course) method pointed to by bst | n | n | n2 |
| file.close(); | 1 | 1 | 1 |
| **Total Cost** | | | n2+16n+4 |
| **Runtime** | | | O(n2) |

**Binary Search Tree**

|  |  |  |  |
| --- | --- | --- | --- |
| **Binary Search Tree Data Structure - Inserting a course object using the Insert() method** | **Line Cost** | **# Times Executes** | **Total Cost** |
| if root = nullptr | 1 | 1 | 1 |
| set root = pass course through a new Node | 1 | 1 | 1 |
| else | 1 | 1 | 1 |
| pass root and course through addNode() | n | 1 | n |
| **Total Cost** | | | n+3 |
| **Runtime** | | | O(n) |
|  | | | |
| **Binary Search Tree Data Structure - addNode() method called by the Insert() method** | **Line Cost** | **# Times Executes** | **Total Cost** |
| create pointer current type Node = node | 1 | 1 | 1 |
| while currrent != nullptr | 1 | n | n |
| if current code > course's code | 1 | n | n |
| if current->left = nullptr | 1 | n | n |
| set current->left = course passed through a new Node() | 1 | 1 | 1 |
| else | 1 | n | n |
| current = current->left | 1 | n | n |
| else | 1 | n | n |
| if current->right = nullptr | 1 | n | n |
| set current->right = course passed through a new Node() | 1 | 1 | 1 |
| else | 1 | n | n |
| set current = current->right | 1 | n | n |
| **Total Cost**  **Runtime** | | | 8n+3 |
| O(n) |

Vectors have the advantage when it comes to simplicity. This simplicity allows for this data structure to be fast and efficient. They are stored in contiguous memory meaning that the elements are stored right next to each other in memory where they can be accessed using an index. This offers a time complexity of O(1) *if* the index is known, otherwise, it would be O(n) since, at worst, the entire vector would need to be traversed to search for a certain element. Vectors, are efficient when compared to another structure like a linked list or structures that use linked lists. This is because of the built in functions vectors have resulting in less overhead to achieve the data structures functionality. Although, this simplicity is a double-edged sword. While having the advantage of faster element access given an index, inserting or deleting an element has a time complexity of O(n) at its worst since every element after the index to insert/delete from would need to shift +/- 1 index value. Because of these advantages and disadvantages, the vector data structure should be used either when inserting/deleting from the end is frequent, or if the information is accessed using index values.

Hash tables use arrays/vectors as the foundation to its structure where the index data is stored at is generated using a hashing function and is within the predetermined size of the array/vector. The index (referred to as a "bucket") can also contain another data structure whether it's another vector or linked list. The big advantage a hash table has as a data structure is that if given a good hashing function, accessing and modifying data is O(1) on average. This is because a well-defined hashing function avoids collisions. Collisions are one of the disadvantages to hash tables. Depending on multiple factors like the size of the hash table, the hash function, and the type of data, the keys used can hash to the same index assigning more than 1 sets of information to the bucket which degrades performance when it comes to searching. Hash tables could also have a lot of unused memory that's allocated because the index hasn't been the result of the hashing function leading to memory overhead. Depending on the use case, hash tables are a bad choice if data needs to be sorted in a certain order. Hash tables are should be used in cases that require fast searching and inserting/deleting of data as long as the hashing function can utilize most of the index range it's provided.

Binary search trees consists of nodes. Each node contains data and a pointer to each of its children nodes. This creates a hierarchical data structure. This structure has the advantage of efficient searching as each node's position relative to their parent node is based on a greater-than/lower-than system to determine if the it's the left or right child node respectively offering a O(log n) time complexity. They also have an ordered structure where the order can be changed just by using a different algorithm to traverse the tree. The disadvantage to using a binary search tree is that they rely on a well-balanced distribution of nodes. Worst case, all nodes are to one side of the root and parent nodes leading to a time complexity of O(n) . Creating a balanced tree is crucial to see any benefits from this data structure. But a balanced tree also requires more memory to store the pointers and to create the structure. If the data being stored needs to be ordered, accessed frequently, and benefits from the parent-child relationship, then a binary search tree is a good choice for that use case.

Given these advantages and disadvantages for the 3 data structures to choose from, I would use the binary search tree data structure. The main reasons are that the courses have a hierarchical structure where the parent is the course and each course has prerequisites as the children , it can maintain ordered data, and performs well when it comes to searching for a node (for inserting, deleting, and retrieving). It's not the best performing when it comes to those functionalities but it does offer them all and does them all well for this use case.