**CS 300 Project One – Runtime Analyses and Recommendation**

**Vector Course Loading Runtime Analysis**

| **Code for function** loadCourses | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| define courses vector structure using Course vector | 1 | 1 | 1 |
| open the CSV file at csvPath for parsing | 1 | 1 | 1 |
| start try catch error handling | 1 | 1 | 1 |
| for each row in the CSV file | 1 | n | n |
| if the current row has at least two fields | 1 | n | n |
| define a single course using Course structure | 1 | n | n |
| course.courseNumber = field 1 | 1 | n | n |
| course.name = field 2 | 1 | n | n |
| if the row has more than two fields | 1 | n | n |
| for each prerequisite | 1 | n | n |
| if prerequisite exists in prerequisites | 1 | n | n |
| add the field to course.prerequisites | 1 | n | n |
| add course to the courses vector using function append | O(1) | n | n |
| return courses | 1 | 1 | 1 |
| **Total Cost** | | | 10n + 4 |
| **Runtime** | | | O(n) |

| **Code for function** append | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Create new node | 1 | 1 | 1 |
| If the head pointer is null | 1 | 1 | 1 |
| Set head to new node | 1 | 1 | 1 |
| Set tail to new node | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| Set tail of current node to point to the new node | 1 | 1 | 1 |
| Set tail to new node | 1 | 1 | 1 |
| Increment size | 1 | 1 | 1 |
| **Total Cost** | | | 8 |
| **Runtime** | | | O(1) |

**Hashtable Course Loading Runtime Analysis**

| **Code for function** loadCourses | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| open the CSV file at csvPath for parsing | 1 | 1 | 1 |
| start try catch error handling | 1 | 1 | 1 |
| for each row in the CSV file | 1 | n | n |
| if the current row has at least two fields | 1 | n | n |
| define a single course using Course structure | 1 | n | n |
| course.courseNumber = field 1 | 1 | n | n |
| course.name = field 2 | 1 | n | n |
| if the row has more than two fields | 1 | n | n |
| for each prerequisite | 1 | n | n |
| if prerequisite exists in prerequisites | 1 | n | n |
| add the field to course.prerequisites | 1 | n | n |
| insert course in the courses hashTable | O(n) | n | n |
| **Total Cost** | | | 10n + 2 |
| **Runtime** | | | O(n) |

| **Code for function** Insert | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Create the key for the given bid | 1 | 1 | 1 |
| Create node using key | 1 | 1 | 1 |
| Iterate through the hashtable to check if the bid already exists | 1 | n | n |
| If the bid already exists | 1 | n | n |
| return (does not count in worst-case scenario) | 0 | 0 | 0 |
| Move to the next node | 1 | n | n |
| Insert the bid if it doesn't already exist | 1 | 1 | 1 |
| If node key is equal to UINT\_MAX | 1 | 1 | 1 |
| Set node bid | 1 | 1 | 1 |
| Set node key | 1 | 1 | 1 |
| Set node next | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| Loop to find the next open node | 1 | n | n |
| Set node to next node | 1 | n | n |
| Insert the course | 1 | 1 | 1 |
| **Total Cost** | | | 5n + 9 |
| **Runtime** | | | O(n) |

**Binary Search Tree Course Loading Runtime Analysis**

| **Code for function** loadCourses | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| open the CSV file at csvPath for parsing | 1 | 1 | 1 |
| start try catch error handling | 1 | 1 | 1 |
| for each row in the CSV file | 1 | n | n |
| if the current row has at least two fields | 1 | n | n |
| define a single course using Course structure | 1 | n | n |
| course.courseNumber = field 1 | 1 | n | n |
| course.name = field 2 | 1 | n | n |
| if the row has more than two fields | 1 | n | n |
| for each prerequisite | 1 | n | n |
| if prerequisite exists in prerequisites | 1 | n | n |
| add the field to course.prerequisites | 1 | n | n |
| insert course in the courses BinarySearchTree | 1 | n | n |
| **Total Cost** | | | 10n + 2 |
| **Runtime** | | | O(n) |

| **Code for function** insert | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| If root equals nullptr | 1 | 1 | 1 |
| Set root to new node bid | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| Add node root and bid using function addNode | O(n) | 1 | n |
| **Total Cost** | | | n + 3 |
| **Runtime** | | | O(n) |

| **Code for function** addNode | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| If node is larger then add to left | 1 | 1 | 1 |
| If no left node | 1 | 1 | 1 |
| This node becomes left | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| Recurse down the left node by calling addNode again | O(n) | n | n |
| Else | 1 | 1 | 1 |
| If no right node | 1 | 1 | 1 |
| This node becomes right | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| Recurse down the right node by calling addNode again | O(n) | n | n |
| **Total Cost** | | | 2n + 8 |
| **Runtime** | | | O(n) |

**Comparing Linked Lists, Hash Tables, and Binary Search Trees**

Linked lists are easy to implement in code, can grow or shrink dynamically, and have quick insertion and deletion times. However, they have poor search and retrieval times because the list must be traversed sequentially.

Hash tables are more efficient for large amounts of data because the amount of time for search, insertion, and deletion operations is linear. However, they can require more memory than other data structures, and they do not store the data in a sorted manner.

Binary Search Trees are efficient because search, insertion, and deletion operations have O(log n) time complexity. They also store data in a sorted manner. However, they can require more memory than other data structures; they can be more complex to implement; and they can have poor performance if they are not balanced.

**Recommendation**

From the runtime analyses, we can see that the complexity of all loading data into three data structures is O(n). However, since the project requires the course information to be sorted, it will be more efficient store the information using Binary Search Trees (BSTs). Because the data in BSTs is already sorted, they do not need to be sorted before being displayed. Also, search and deletion operations for BSTs are faster than for linked lists. For this reason, I will use BSTs when writing the code.