**Runtime Comparison For Inserting Courses in a Vector, Hash Table, and Binary Search Tree**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Vector Code** | **Line Cost** | **# Times Executes** | | **Total Cost** |
| Open file csvFile | 1 | 1 | | 1 |
| If file cannot be opened | 1 | 1 | | 1 |
| Print "Error: File could not be opened" | 1 | 1 | | 1 |
| Exit function | 1 | 1 | | 1 |
| Create a Courses object named courseList | 1 | 1 | | 1 |
| For each line in file | 1 | n | | n |
| If at least two fields exist | 1 | n | | n |
| Split line into courseId, courseName, prerequisites | k | n | | n\*k |
| Create new Course object | 1 | n | | n |
| Set courseId, courseName, and prerequisites from line data | 1 | n | | n |
| Append Course to courseList | 1 | n | | n |
| Print "Error: Invalid line format" | 1 | n | | n |
| For each Course in courseList | 1 | n | | n |
| Call validate method on Course | n\*m | n | | n2\*m |
| If validate returns false | 1 | n | | n |
| Remove Course from courseList | n | n | | n2 |
| Return courseList | 1 | 1 | | 1 |
| **Total Cost** | | | n\*k+7n+n2\*m+n2+6 | |
| **Runtime** | | | O(n2\*m) | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Hash Table Code** | **Line Cost** | **# Time Executes** | | **Total Cost** |
| Open file csvFile | 1 | 1 | | 1 |
| If file cannot be opened | 1 | 1 | | 1 |
| Print "Error: File could not be opened" | 1 | 1 | | 1 |
| Exit function | 1 | 1 | | 1 |
| For each line in file | 1 | n | | n |
| If at least two fields exist | 1 | n | | n |
| Split line into courseId, courseName, prerequisites | k | n | | n\*k |
| Create new Course object | 1 | n | | n |
| Set courseId, courseName, and prerequisites from line data | 1 | n | | n |
| Call hash(courseId) | k | n | | n\*k |
| Call insert(newNode, index) | 1 | n | | n |
| Increment courseTotal | 1 | n | | n |
| Print "Error: Invalid line format" | 1 | n | | n |
| For each Course in hash table | 1 | n | | n |
| Call validate(Course) | m | n | | n\*m |
| If validate returns false | 1 | n | | n |
| Print "Prerequisite was not found." | 1 | n | | n |
| Print "Removing the course" | 1 | n | | n |
| Remove Course from hash table | 1 | n | | n |
| Decrement courseTotal | 1 | n | | n |
| **Total Cost** | | | n\*k+n\*m+12n+4 | |
| **Runtime** | | | O(n\*(k+m)) | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **BST Code** | **Line Cost** | **# Times Executes** | | **Total Cost** |
| Declare and Initialize courseTotal | 1 | 1 | | 1 |
| Open file csvFile | 1 | 1 | | 1 |
| If file cannot be opened | 1 | 1 | | 1 |
| Print "Error: File could not be opened." | 1 | 1 | | 1 |
| Exit function | 1 | 1 | | 1 |
| For each line in file | 1 | n | | n |
| If at least two fields exist | 1 | n | | n |
| Split line into courseId, courseName, prerequisites | k | n | | n\*k |
| Create new Course object | 1 | n | | n |
| Set courseId, courseName, prerequisites from line data | 1 | n | | n |
| Call insert(root, newCourse) | h | n | | n\*h |
| Increment courseTotal | 1 | n | | n |
| Print "Error: Invalid line format." | 1 | n | | n |
| Close file | 1 | 1 | | 1 |
| Call validate(root, courseTotal) | n\*m\*h | 1 | | n\*m\*h |
| **Total Cost** | | | n\*(k+h+3+m\*h)+5 | |
| **Runtime** | | | O(n\*(k+m\*n)) | |

**Advantages and Disadvantages of Data Structures**

**Vector:**

The advantage of using a vector data structure is its ease of implementation. Courses can be loaded into the vector by simply appending. Also, iterating through the vector to print all courses is straightforward. However, given the requirement, the courses need to be printed in alphanumeric order, a separate sort algorithm needs to be used, which will affect runtime. Another disadvantage of using a vector has to do with searching them. Searching for a course in an unsorted vector requires starting at the beginning of the vector and comparing each course until finding the one searched for. The search issue also becomes a problem when loading and validating courses to see if their prerequisites exist. The vector must be linearly searched each time a course is validated, giving it the worst runtime for loading courses out of the three data structures. When vectors are sorted, algorithms such as binary search and lower bound and upper bound can be used to search more efficiently. However, having to use an algorithm to sort and then search makes the vector less efficient than other data structures.

**Hash Table:**

Hash tables have the advantage of constant time insertion, giving it the best runtime for loading courses. Since each course ID is hashed and corresponds to an index in the hash table, searching for and removing courses is also highly efficient. However, the nature of hash tables makes it so the courses are not stored in any particular order. Therefore, printing all courses in alphanumeric order requires them to be extracted and stored in a vector, and then a sort algorithm can be used to order them. The process needed to sort items in a hash table is its main disadvantage.

**Binary Search Tree:**

One advantage of binary search trees is everything is stored in order. Therefore, printing all courses in alphanumeric order does not require sorting beforehand. Another benefit is that searching for a course and printing its prerequisites is relatively efficient if the tree is balanced. However, implementing a balanced binary search tree can be complex. Additionally, each node in the tree requires pointers for its children, which creates overhead space.

**Recommendation**

I recommend using a hash table for the ABCU program. The hash table has the best complexity for loading and validating courses of the three data structures at O(n\*(k+m)). Additionally, the hash table can search for courses in constant-time (O(1)) which is faster than the vector and binary search tree. Although a vector is easy to implement, that trade-off is not worth the inefficiency for this use case. A binary search tree does have the advantage of being sorted by nature. However, the tree's inefficient loading and validation of courses makes it unsuitable for processing a potentially large list of courses. Although when using a hash table, the courses must be extracted and sorted to be printed alphanumerically, the process is still relatively efficient. In conclusion, the hash tables' ability to search, insert, and delete courses in O(1) time makes it the best data structure for the ABCU program.