Project\_One\_Analysis

The pseudocode for reading and parsing the files for all data structures differs only in how the new element course is added to the data structure. The general read and parse file function runtime:

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **open file** | 1 | 1 | 1 |
| **if file is open** | 1 | 1 | 1 |
| **if file is empty** | 1 | 1 | 1 |
| **throw “Error: no data in the file”** | 1 | 1 | 1 |
| **while file.good()** | 1 | n | n |
| **read lines from file** | 1 | n | n |
| **for each line** | 1 | n | n |
| **if line is not empty** | 1 | n | n |
| **split line into elements separated by commas** | m | n | nm |
| **if the number of elements is less than 2** | 1 | n | n |
| **print out “Line # is skipped due insufficient number of elements”** | 1 | n | n |
| **continue** | 1 | n | n |
| **else create a new course structure** | 1 | n | n |
| **add trimmed element[0] as courseNumber** | 1 | n | n |
| **add trimmed element[1] as courseName** | 1 | n | n |
| **if there are more elements** | 1 | n | n |
| **add each next element using addPrerequisite()** | 1 per element | n(m-2) | nm |
| **\*\*\*\*(depends on the data structure)** |  |  |  |
| **LinkedList Append(course)** | 1 | n | n |
| **HashTable hash courseNumber and use as a key** | 1 | n | n |
| **add key and course object to HashMap** | 1 to n | n | n to n^2 |
| **BinarySearch insertCourse(root, course)**  **Tree** | n | n | LogN to n^2 |
| **else continue** | 1 | n | n |
| **close file** | 1 | 1 | 1 |
| **else throw ‘Error: failed to open the file’** | 1 | 1 | 1 |
| **Total cost** | | | |
| **LinkedList** | 6+13n+2nm | | |
| **HashTable** | 6+14n+2nm to 6+13n+2nm+n^2 | | |
| **BinarySearchTree** | 6+12n+2nm+logn to 6+12n+2nm+n^2 | | |
| **Runtime** | | | |
| **LinkedList** | **O(nm)** | | |
| **HashTable** | **O(nm)to O(n^2)** | | |
| **BinarySearchTree** | **O(nm) to O(n^2)** | | |

The pseudocode and the runtime analysis highlight the advantages and disadvantages of the three data structures suggested for the project implementation.  
 Based on the runtime analysis, the most appealing data structure to use is a linked list since the worst case runtime is O(nm), where *n* is the number of lines in the file and *m* is the number of prerequisites in each line when other data structures have the same runtime as the best case. The hash table might have a runtime of O(n^2) if the data structure is implemented poorly and many collisions occur. The binary search tree might have a runtime of O(n^2) if the tree is unbalanced, and adding a new element might become closer to traversing a linked list. Since balancing is not implemented for the binary search tree, loading a file might get closer to O(n^2).  
 Printing an alphanumerical order of all the courses will be the easiest to implement in the binary search tree since it’s already ordered, and only traversing is required, which doesn’t require extra memory storage. The implementation of a linked list requires an extra sorting technique. The merge sort I used in my pseudocode has a runtime complexity of O(nlogn). Temporary extra storage is needed since a list is divided and merged in order. Assuming that sorting will be required only once and the sorted list is stored, the result won’t be drastically influenced. The data in a hash table must be added to a vector for sorting and printing the ordered list, which also takes extra memory storage.  
 Although the linked list seems the most suitable for reading a file and storing objects based on the analysis, I suggest using a hash table to implement the project. With adequate hash table sizing and sufficient handling of the collision, we’ll reach the same O(nm) runtime. Considering the fact that files will be loaded rarely, and the search function will be used more often. The search function for a binary search tree runtime complexity is between O(logn) and O(n) if the tree is unbalanced. From this perspective, the hash table has its advantages. With the proper collision handling, the search function has a runtime of O(1) rather than O(n) in linked lists, where the entire list has to be traversed to find a match. Considering all the facts, advantages, and disadvantages of each data structure, I recommend using a hash table for the project implementation.