Task: analyze worst-case running time of reading the file and creating course objects for Vector, Hash Table, and BST. Line cost of running a function is the run time of that function

Vector:

| **Code** | **Line Cost** | **# Times Exe** | **Total Cost** |
| --- | --- | --- | --- |
| While lines to read in file: | 1 | n | n |
| Declare vector courseAttributes | 1 | 1 | n |
| Declare string attribute | 1 | 1 | 1 |
| While there are comma sep. att’s | 1 | 4 | 4 |
| Append each attribute to courseAttributes | 1 | 1 | 1 |
| If size courseAtt < 2, print error msg | 1 | n | n |
| Declare tempCourse | 1 | n | n |
| Populate tempCourse with courseNumber | 1 | n | n |
| Populate tempCourse with courseName | 1 | n | n |
| Populate tempCourse with prereqCourses vector | 1 | n | n |
| For each prereq course in prereqCourses vector: | 1 | 2 | 2 |
| Search() Courses for prereq course, | n | n | n2 |
| Print error message if prereq not found by end | 1 | 1 | 1 |
| Append tempCourse to Courses | 1 | n | n |
| Close the file | 1 | 1 | 1 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| **Total Cost** | | | n2 + 8n + 10 |
| **Runtime** | | | O(n2) |

**Advantages of Vector:**

Vector data structures are fairly simple to code, and can implement many built-in methods in C++. They are also very efficient at accessing elements. Looking up elements by a known index yields constant-time random access. Also, as the size of the input grows over time, vectors can be dynamically resized to accommodate the additional data. This data is all stored in contiguous memory and is not prone to memory leaks or inappropriate lookups.

**Disadvantages of Vector:**

My method for validating prerequisites is very costly. As the input size gets large, I would be searching a growing database once for each prereq listed for a course. Based on the data we were given, I assumed the max number of prereqs for a course was 2. However, I think it represent O(n2) behavior. For each course we add to the input of size n, we would have to make up to n comparisons to validate the existence of a single prerequisite course. As the input size grows, the number of comparisons that must be made for each course also grows at the same rate, thus exponential growth. In fact, if every course had, say, 4 prerequisites, it would more accurately be 4n2. I know this would still be categorized as O(n2), but in practice it would execute much more slowly than other O(n2) family programs. Other, more general disadvantages of Vectors are that it is inefficient to remove elements in the middle of the vector, because all other elements must be bumped down one to preserve the continuous index.

**Hash Table:**

| **Code** | **Line Cost** | **# Times Exe** | **Total Cost** |
| --- | --- | --- | --- |
| While lines to read in file: | 1 | n | n |
| Declare vector courseAttributes | 1 | 1 | n |
| Declare string attribute | 1 | 1 | 1 |
| While there are comma sep. att’s | 1 | 4 | 4 |
| Append each attribute to courseAttributes | 1 | 1 | 1 |
| If size courseAtt < 2, print error msg | 1 | n | n |
| Declare tempCourse | 1 | n | n |
| Populate tempCourse with courseNumber | 1 | n | n |
| Populate tempCourse with courseName | 1 | n | n |
| Populate tempCourse with prereqCourses vector | 1 | n | n |
| For each prereq course in prereqCourses vector: | 1 | 2 | 2 |
| Search() the Hash Table by key for prereq course | 2 | n | 2n |
| Print error message if prereq not found by end | 1 | 1 | 1 |
| Get hashKey based on courseNumber | 1 | n | n |
| Check if node at hashKey location is default | 1 | n | n |
| If default, replace with tempCourse | 1 | n | n |
| If not default, (either already inserted, or collision): |  |  |  |
| Declare currentCourse, prevCourse | 1 | n | n |
| Access headnode at currentCourse | 1 | n | n |
| While current !null AND currentNum != tempNum: | 1 | n | n |
| Traverse the linked list, update current and prev ptrs | 1 | n | n |
| Either append tempCourse to final node, or print error that tempCourse was already added | 1 | n | n |
|  |  |  |  |
| **Total Cost** | | | 17n + 9 |
| **Runtime** | | | O(n) |

**Advantages of Hash Table:** The greatest advantage of hash tables is probably the fast lookup time, with constant-time lookup on average. Also, thanks to their ability to dynamically resize, hash tables are very adaptable to different applications. One of the most common applications is password storage, which enables servers to accurately assess whether a user has entered the correct password without actually storing all those passwords in a single location. Such a cache would be a great target for criminal hackers. Hashing the passwords allows administrators to verify that the user typed the correct password without knowing the literal password. Hash functions are also used in cryptography and cryptocurrency like Bitcoin. They enable computers to very quickly verify the integrity of a certain calculation, without having to expend all of the computer power to actually find the solution to that calculation.

**Disadvantages of Hash Table:**

For my code, the worst case scenario is that the HashTable only contains one bucket, and thus will essentially behave as a linked list when it comes to analyzing run times. That’s why I assigned time costs of “n” for traversing the linked list that handles collision in a given bucket. In practice, selecting an appropriate number of buckets based on the input size will greatly improve the run times, because it will quickly distribute that one monster linked list into many shorter linked lists in each bucket. More generally, Hash Tables do not lend themselves well to sorting and ordering. The bucket that elements are stored in are based on a hash function, which could be as simple as modulo division. Furthermore, the number on which the hash is performed might be seem random, like the middle bunch of characters in a part’s serial number, and thus the bucket in which the element lands in might seem random. Poorly designed hash functions can lead to more collisions, which both slows down future operations and does not take advantage of all the memory afforded to the hash table’s other buckets. Finally, hash tables are more complex to implement compared to vectors. Attempting to parse collisions can easily lead to memory leaks, dereferenced null pointers, or memory access violations.

**Binary Search Tree**

| **Code** | **Line Cost** | **# Times Exe** | **Total Cost** |
| --- | --- | --- | --- |
| While lines to read in file: | 1 | n | n |
| Declare vector courseAttributes | 1 | 1 | n |
| Declare string attribute | 1 | 1 | 1 |
| While there are comma sep. att’s | 1 | 4 | 4 |
| Append each attribute to courseAttributes | 1 | 1 | 1 |
| If size courseAtt < 2, print error msg | 1 | n | n |
| Declare tempCourse | 1 | n | n |
| Populate tempCourse with courseNumber | 1 | n | n |
| Populate tempCourse with courseName | 1 | n | n |
| Populate tempCourse with prereqCourses vector | 1 | n | n |
| For each prereq course in prereqCourses vector: | 1 | 2 | 2 |
| Search() the BST using Search() from assignment 5 | 2 | log(n) | 2log(n) |
| Print error message if prereq not found by end | 1 | 1 | 1 |
| Insert(tempCourse) into BST | 1 | n | n |
| Insert() calls addNode() if BST is not null | 1 | n | n |
| addNode() recursively traverses the BST until finding appropriately-located null node… continually halves the amount of data it has to parse | 1 | log(n) | log(n) |
| Assign tempCourse into that null node | 1 | n | n |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| **Total Cost** | | | 3 log(n) + 10n + 9 |
| **Runtime** | | | O(n) |

**Advantages of BST:** Because the BST uses recursion for many common functions like Search(), it’s runtime should be faster than Vector and HashTable. The runtime can be thought of as O(n + log(n)), but evaluates to O(n) since the linear side will grow much faster than the logarithmic operations as the input size grows. Binary Search Trees facilitate sorted traversals, since they are sorted while being added to the tree. This makes it much easier to deal with the sorting and printing task compared to the Hash Table structure. Also, BST can be accessed in a variety of ways, such as in-order, pre-order, and post-order, depending on the desired data access and context. Adding or removing a node from the BST is much easier, and less of a performance penalty, than other data structures. I think that my BST program will be easier to program than Hash Tables, and faster than both Hash Tables and Vectors.

**Disadvantages of BST:**

One of the obvious disadvantages of BSTs is that unlike Vectors and Hash Tables, we do not have the luxury of constant time lookup. Searching for a particular node will take log(n) time, since we potentially have to traverse all the way to a leaf node. Hash Tables and Vectors both allow fast access via a known position (index, or hash key), but BST is structured in such a way that does not allow quick indexing. Something we could look into in order to improve the performance of the BST is to balance the tree, ensuring that the heights of child subtrees stay relatively equal. Otherwise, we could theoretically have a tree that consisted of a single linear branch. Searching that branch would take O(n) time, since we don’t have the benefit of halving the data size as we go. Thus we would have the drawback of Vector runtime without the simplicity of Vector location lookup.

**Conclusion:**

In a perfect world, I think a directed graph would be the most suitable structure for this assignment. However, given the three choices, BST seems to be the clear winner. Although my assessment of O(n) runtime matches the Hash Table time, I think that in practice the BST will allow much faster access and will be closer to log(n) on average. O(n) represents the worst-case scenarios, as discussed above. The fact that the flowchart for this data already resembles a “tree” actually made me think that the BST would be the most suitable data structure from the start. The course diagram seems to flow from root nodes (having no prerequisites) to leaf nodes (the final courses, with no follow-on activities). It feels intuitive to translate this structure into a BST in a coding environment.