**Runtime complexity evaluation**

|  |  |
| --- | --- |
| **Functionality** | **Worst-case Runtime Complexity** |
| Csv Validation | O(N2) |
| Vector loading (without sorting) | O(N) |
| Vector loading (with sorting) | O(N2) |
| Hash table loading | O(N2) |
| Search tree loading | O(N2) |

**a. Csv validation**

valCourseNum() has a constant runtime of 7 in the worst case, meaning that it does not increase the order of runtime complexity when called by valCsv().

Prior to its first for-loop, valCsv() makes two function calls. The first for-loop makes one call to assign i = 0, N calls to compare i < file.rowCount, and N + 1 to increment ++i. Within this first for-loop, the runtime complexity is again constant in the worst case, rendering (*c* × N) calls for a constant *c*, until the nested for-loop iterating j. The j for-loop iterates through the file’s columns, each of which contains one artifact of course data including the name, number, and one prerequisite per column. Because this is a variable number, it might be treated as N, but practically speaking no course will have more than a small handful of prerequisites, so j will actually be bounded somewhere around 5, giving up to about 4 iterations of the j for-loop. We will therefore treat it as constant. Within this j for-loop, the worst-case complexity is again constant outside of its nested k for-loop which, like the i for-loop, iterates through the csv’s rows, giving it N runtime complexity. Thus, the worst-case runtime complexity of valCsv() can be calculated by taking N calls to *c* calls to N calls, giving O(N2).

**b. Creating and storing course objects in a vector**

The vector’s LoadCourses() method makes a constant number of function calls outside of its for-loop iterating on i. (These constant calls include the creation of Course() struct objects.) Within this for-loop are constant calls until its nested j for-loop, which iterates N times. Similarly to the j for-loop discussed in 4a, this j for-loop iterates over the csv’s columns and so is most reasonably treated as constant. Each iteration of it makes constant function calls. Thus, the worst-case runtime complexity of loading courses into a vector is O(N).

However, all functions accessing the vector of course objects necessitate sorting the vector as an intermediary step between loading and accessing the vector, whereas the other two data structures sort while loading. Thus, to calculate a runtime complexity more reasonably comparable to the other structures, we must add the complexity of the QuickSort() function. In the worst case, the Partition() function, will create N – 1 partitions giving it a runtime complexity of O(N), and QuickSort() will call it N/2 times, giving the function a runtime complexity of O(N2). Therefore, the overall worst-case runtime of loading course data into a vector and sorting it to make it available for further use is O(N + N2) = O(N2).

**c. Creating and storing course objects in a hash table**

Insertion in a hash table uses the hash() function, called by Insert(), which has a constant runtime complexity.

Insert() has a constant runtime until its while loop, which if a node’s insertion bucket is already occupied, finds the end of that bucket’s chain. It iterates up to N times; while the number of nodes in a chain is not likely to become very large, it will vary unpredictably depending on the course numbers’ ASCII values, so it is not appropriate to treat this function’s worst-case runtime complexity as constant. Rather, it is O(N).

The LoadCourses() method, which calls Insert(), makes a constant number of function calls including those involved in creating a Course() object. It iterates through two for-loops; as in previous cases, the for-loop on i iterates across the csv’s rows and the for-loop on j iterates across the csv’s columns, so their complexities are respectively O(N) and O(*c*). The i for-loop calls Insert(), giving the hash table’s insertion functionality an overall worst-case runtime of O(N2).

**d. Creating and storing course objects in a binary search tree**

The binary search tree structure, in the worst case, makes a call from Insert() to the recursive AddNode() method. AddNode() makes several constant function calls, and recurs at most once for each level of the tree (if the node to be inserted will be a leaf added after the highest level), giving it a runtime complexity of O(N).

As in the hash table class, the binary search tree’s LoadCourses() method contains two for-loops, one iterating N times over the csv’s rows and one iterating *c* times over its columns. The outer loop with N iterations calls Insert(), giving binary search trees’ course-loading functionality a worst-case runtime complexity of O(N2).