Nicholas Boodoo

Project 1 Runtime Evaluation

CS 300

27 February 2022

Evaluation

| Vector  Load and Validate Courses | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| INITIALIZE fstream fileStream to GET elements of file | 1 | 1 | 1 |
| INITIALIZE string line to store a single line in file | 1 | 1 | 1 |
| INITIALIZE stringstream lineStream to GET elements of each line | 1 | 1 | 1 |
| INITIALIZE string token to store a single word in line | 1 | 1 | 1 |
| Open fileName with fileStream | 1 | 1 | 1 |
| INITIALIZE int count to store the token count per line in file | 1 | 1 | 1 |
| GET line from fileStream | 1 | n | N |
| Fill lineStream with current line | 1 | n | N |
| SET count to 1 | 1 | n | N |
| Create Course aCourse for each line in file | 1 | n | N |
| GET token from lineStream up to comma | 1 | 2n | N |
| IF count is equal to 1 | 1 | N | N |
| SET aCourse courseNumber to token | 1 | N | N |
| i++ | 1 | N | N |
| ELSE IF count is equal to 2 | 1 | N | N |
| SET aCourse courseName to token | 1 | N | N |
| i++ | 1 | N | N |
| IF token exists in courses as a course | 1 | N | N |
| ADD token to aCourse PreReqs | 1 | N | N |
| ELSE OUTPUT file format error | 1 | 1 | 1 |
| i++ | 1 | N | N |
| IF count is less than 2 | 1 | N | 1 |
| OUTPUT "format error” | 1 | 1 | 1 |
| push aCourse to back of courses | 1 | N | N |
| CLEAR lineStream for next line | 1 | n | N |
| Total Cost | | | 17n + 6 |
| Runtime | | | O(n) |

| Hash Table  Load and Validate Courses | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| INITIALIZE fstream fileStream to GET elements of file | 1 | 1 | 1 |
| INITIALIZE string line to store a single line in file | 1 | 1 | 1 |
| INITIALIZE stringstream lineStream to GET elements of each line | 1 | 1 | 1 |
| INITIALIZE string token to store a single word in line | 1 | 1 | 1 |
| Open fileName with fileStream | 1 | 1 | 1 |
| INITIALIZE int count to store the token count per line in file | 1 | 1 | 1 |
| GET line from fileStream until none left | 1 | n | n |
| Fill lineStream with current line | 1 | n | n |
| vector<string> parts = lineStream.split(","); | 1 | n | n |
| if (parts.size() < 2) { | 1 | 1 | 1 |
| output file format error | 1 | 1 | 1 |
| aCourse.courseNumber = parts[0]; | 1 | 1 | 1 |
| aCourse.courseName = parts[1]; | 1 | 1 | 1 |
| for (int i = 2; i < parts.size(); ++i) { | 1 | n | n |
| if parts[i] exists in Hashtable as a course | 1 | n | n |
| aCourse.preReqs.insert(parts[i]); | 1 | n | n |
| else  output file format error | 1 | 1 | 1 |
| Total Cost | | | 7n + 11 |
| Runtime | | | O(n) |

| Load Course Obj | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| Create key for aCourse by hashing aCourse courseNumber | 1 | 1 | 1 |
| Create Node\* node to obtain node using key | 1 | 1 | 1 |
| IF node is equal to nullptr | 1 | 1 | 1 |
| Create new node newCourse with aCourse and key | 1 | 1 | 1 |
| Insert elements of newCourse into table at position[key] | 1 | 1 | 1 |
| ELSE IF node key is equal to UINT\_MAX | 1 | 1 | 1 |
| Update node key to key | 1 | 1 | 1 |
| Update node course to aCourse | 1 | 1 | 1 |
| Update node next to nullptr | 1 | 1 | 1 |
| WHILE node next is not equal to nullptr | 1 | N | N |
| SET node to node next | 1 | 1 | 1 |
| Create new node newCourse with aCourse and key | 1 | 1 | 1 |
| SET node next to newCourse | 1 | 1 | 1 |
| Total Cost | | | N + 3 |
| Runtime | | | O(n) |

| Validate Course Obj | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| If (aCourse’d courseNumber < current node’d courseNumber) | 1 | 1 | 1 |
| If (node’d left child is empty) | 1 | 1 | 1 |
| Add new node with course at node’s left child | 1 | 1 | 1 |
| Else recursively traverse node’s left sub-tree | 1 | n | n |
| If (node’s right child is empty) | 1 | 1 | 1 |
| Add new Node with course at node’s right child | 1 | 1 | 1 |
| Else recursively traverse node’s right sub-tree | 1 | n | n |
| If (root is empty) set root to new node with aCourse | 1 | 1 | 1 |
| Else addNode(root, aCourse) | N+3 | 1 | N+3 |
| Total Cost | | | N + 3 |
| Runtime | | | O(n) |

| Binary Search Tree  Load and Validate Courses | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| INITIALIZE fstream fileStream to GET elements of file | 1 | 1 | 1 |
| INITIALIZE string line to store a single line in file | 1 | 1 | 1 |
| INITIALIZE stringstream lineStream to GET elements of each line | 1 | 1 | 1 |
| INITIALIZE string token to store a single word in line | 1 | 1 | 1 |
| Open fileName with fileStream | 1 | 1 | 1 |
| INITIALIZE int count to store the token count per line in file | 1 | 1 | 1 |
| GET line from fileStream until none left | 1 | N | N |
| Fill lineStream with current line | 1 | N | N |
| SET count to 1 | 1 | N | N |
| Create Course aCourse for each line in file | 1 | N | N |
| GET token from lineStream up to comma | 1 | 2n | 2n |
| IF count is equal to 1 | 1 | N |  |
| SET aCourse courseNumber to token | 1 | N | N |
| i++ | 1 | N | N |
| ELSE IF count is equal to 2 | 1 | N | N |
| SET aCourse courseName to token | 1 | N | N |
| i++ | 1 | N | N |
| IF token exists in bst as a course | 1 | n | n |
| ADD token to aCourse PreReqs | 1 | N | N |
| ELSE OUTPUT file format error | 1 | 1 | 1 |
| i++ | 1 | N | N |
| IF count is less than 2 | 1 | 1 | 1 |
| OUTPUT "format error” | 1 | 1 | 1 |
| insert aCourse into bst | N | N | N2 |
| CLEAR lineStream for next line | 1 | N | N |
| Total Cost | | | N2 + 16n + 6 |
| Runtime | | | O(n) |

Benefits of each Data Structure

There are many advantages of using the vector data structure. Some of these advantages include but are not limited to the ease of its implementation, if your vector is sorted, then search is only O(log n), and the ability to insert data points at the end of the vector in constant time, and insert for sorted vectors is O(log n). However, this data structure does have its disadvantages. This data structure must be sorted prior in order to take full advantage of its search capabilities. Also, removing data points creates a frame shift, which is O(n). Also, if a vector increases in size enough, then it is reallocated to another place in memory, which is O(n). Lastly, vectors may take up more memory space than is actually needed. Also, even though initial insertion is fast when more courses would be inserted, the vector would need to be resorted. In our case, we would be loading courses initially, so this may not be an issue.

The Hash table data structure allows the direct access of its items and the ability to insert and delete data points in constant time, regardless of table size. When implemented correctly, hash tables are the best data structures for speed. Some disadvantages of using the hash table are that , retrieving elements from the data structure does not preserve their original order, and elements from the table are randomly stored in memory which can cause issues with the cached memory and may cause delays or lags. A disadvantage is that hash tables have O(n) insert, search, and delete in the worst case, if there are hash collisions. A hash table has an array of indexes underneath the hash function. However, that array is not guaranteed to be sorted, so we have to iterate over all indexes and put them into a vector, which is O(n) in space. Then we sort that vector, which is O(n log n) and print which is O(n). Thus, the overall runtime is O(n log n) and also O(n) space. TO find a course in the hash table, we will retrieve it in O(1) time best case and O(n) in worst case. The hash table will be advantageous if we need to find a lot of individual courses or add many new courses. Searches and insertions are also O(1). However we would still need to be very careful when implementing a hash function.

The binary search tree is very useful because it will allows the retrieval of elements in the same order, the insertion and deletion of elements in O(log n) time. Search is O(log n), which isn't the fastest but is still relatively fast, which is relative to other worst-case searches for other data structures. However, this data structure can run into some stack overflow issues when large and implemented recursively, and its shape is heavily dependent on the first element created, meaning that care must be taken when creating this data structure. In order to print the list of courses, we insert them into a binary search tree, which is O(n log n). Then, to print the list, we traverse the tree in-order, which is O(n). Thus, the overall runtime will be O(n log n). To find a course in the binary search tree, we can use binary search, which is O(log n). The advantage is that if we had to insert more courses, insertion is faster relative to the vector, however for our project, we will not be inserting more courses, instead we will just be loading our data structure.

Recommendation

I recommend the vector because it has the same runtime as the binary search tree but does not have the included risk of choosing a bad root, and does not have an extra memory requirement, like the hash table does. Although the hash table is faster at finding a course with O(log n) speed for the sorted vector is still pretty fast, especially if the total number of courses is small. Also, we may run the risk of implementing a bad function for the hash table and may have to consider hash collisions if we go that route. I think that the vector will do what we need it to accomplish, and it will be pretty efficient at it for the above reasons.