**CS300 Project 1 Runtime Analysis**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Code | | Line Cost | | # Times Executes | Total Cost |
| **Vector** | Initialize CSV parser | | 1 | | 1 | 1 |
| For each row in file | | 1 | | N | N |
| Init a temp vector & place each string of text separated by comma into it | | 1 | | N | N |
| If vector size less than 2 | | 1 | | N | N |
| Throw inadequate info error | | 1 | | N | N |
| Create new course | | 1 | | N | N |
| For each index in temp vector | | 1 | | C(N) | C(N) |
| Place course info into new course | | 1 | | N | N |
| Append course to courses | | 1 | | N | N |
| Total Cost | | | | | 5n(2n) + 1 |
| Runtime | | | | | O(n2) |
|  |  | | | | |  |
|  | Code | Line Cost | | # Times Executes | | Total Cost |
| **Hashtable** | Initialize CSV parser | 1 | | 1 | | 1 |
| For each row in file | 1 | | N | | N |
| Init a temp vector & place each string of text separated by comma into it | 1 | | N | | N |
| If vector size less than 2 | 1 | | N | | N |
| Throw inadequate info error | 1 | | N | | N |
| Create new course | 1 | | N | | N |
| For each index in temp vector | 1 | | C(N) | | C(N) |
| Place course info into new course | 1 | | N | | N |
| Create new key equal to hash of course number | 1 | | N | | N |
| If key is null | 1 | | N | | N |
| Create new node and place it into the vector | 1 | | N | | N |
| If node at key is null, load in information | 1 | | N | | N |
| If node at key is not null | 1 | | N | | N |
| Loop until next node is null and load in new node and course info | 1 | | C(N) | | C(N) |
| Total Cost | | | | | 4n(6n)(n) + 1 |
| Runtime | | | | | O(n3) |
| **BinarySearchTree** |  | | | | |  |
| Code | Line Cost | | # Times Executes | | Total Cost |
| Initialize CSV parser | 1 | | 1 | | 1 |
| For each row in file | 1 | | N | | N |
| Init a temp vector & place each string of text separated by comma into it | 1 | | N | | N |
| If vector size less than 2 | 1 | | N | | N |
| Throw inadequate info error | 1 | | N | | N |
| Create new course | 1 | | N | | N |
| For each index in temp vector | 1 | | C(N) | | C(N) |
| Place course info into new course | 1 | | N | | N |
| For each course in temp vector | 1 | | C(N) | | C(N) |
| If root of tree is null | 1 | | 1 | | 1 |
| Create new node with course info and made it a root | 1 | | 1 | | 1 |
| Else | 1 | | N | | N |
| Create new node point | 1 | | N | | N |
| While pointer does not equal null | 1 | | C(log n) | | C(log n) |
|  | Move down tree until correct spot for new node and add new node | 1 | | N | | N |
|  | Total Cost | | | | | 5n(2n)(3n+2)(log n \* n) + 1 |
|  | Runtime | | | | | O(n2 log n) |

For vector data structures, the advantages is that it fairly simply to construct and debug. Additionally, vectors make visiting numerous elements a rapid procedure as they are next to each other in storage. However, runtime of the necessary operations for vectors are lengthy as most operations will have to traverse the entire vector if it is not already sorted. Thus, before each critical operation is run, a sorting algorithm will need to be completed to ensure that other operations can finish quickly. Furthermore, vectors need to have a size set when they are created. Thus, when adding a new element exceeds the already established vector size, another operation will be needed to create a new vector and then transfer all the elements over.

For hashtable data structures, they allow quick creation and storage of elements through creating a key and storing the elements there. With the addition of procedures like chaining and probing, elements can easily be stored and located when collisions occur. However, the same problem of vector sizes needing to be preestablished and cannot be dynamically altered exits with hashtables as well. If a hashtable exceeds its preeastalbished size, then another method needs to be called to create a new hashtable and then rehash and store the courses back into new hashtable.

For BinarySearchTrees (BSTs), they take large amounts of elements and effectively stores them in an order method with quick addition, removal, and search capabilities. Unlike the previous two data structures, they do not have a preset size restriction and allow for dynamic size alterations to the data structure. However, if the data being used for the data structure is already in a sorted state, the binary search tree becomes a lost less effective and can result in predominantly one-sided tree. Furthermore, since the nodes within the BST are not next to each and require traverse across different memory addresses, BSTs do not give way to quick traversal between elements.

When deciding between the three previous data structures, the runtime analysis of BianrySearch Trees proved to be the quickest with vectors second fastest. Additionally, further sorting of the data does not need to be completed as the BST incorporates sorting of the data within the initialization of the BST and the addition and removal of nodes.