**CS300 Project 1**

**Salah Bedoui**

**Runtime:**

**Vector**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| OPEN file | 1 | 1 | 1 |
| IF file not open | 1 | 1 | 1 |
| PRINT error message | 1 | 1 | 1 |
| READ line | 1 | n | n |
| WHILE (end of file not reached) | 1 | n | n |
| DECLARE course object | 1 | n | n |
| IF the input stream reads less than 2 values | 1 | n | n |
| PRINT error message about file format | 1 | n | n |
| ELSEIF  Input stream reads 2 values | 1 | n | n |
| CALL setCourseNumber for first value | 1 | n | n |
| CALL setcourseName for second value | 1 | n | n |
| ELSE  Input stream reads more than 2 values | 1 | n | n |
| CALL setCoursePrereq for values greater than 2 | 1 | n | n |
| PUSH\_BACK course object to the vector | 1 | n | n |
| CLOSE file | 1 | 1 | 1 |
| **Total Cost** | | | 11n + 4 |
| **Runtime** | | | O(n) |

**HashTable**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| OPEN file | 1 | 1 | 1 |
| IF file not open | 1 | 1 | 1 |
| PRINT error message | 1 | 1 | 1 |
| READ line | 1 | n | n |
| WHILE (end of file not reached) | 1 | n | n |
| DECLARE course object | 1 | n | n |
| IF the input stream reads less than 2 values | 1 | n | n |
| PRINT error message about file format | 1 | n | n |
| ELSEIF  Input stream reads 2 values | 1 | n | n |
| CALL setCourseNumber for first value | 1 | n | n |
| CALL setcourseName for second value | 1 | n | n |
| ELSE  Input stream reads more than 2 values | 1 | n | n |
| CALL setCoursePrereq for values greater than 2 | 1 | n | n |
| IF (HashSearch(course⇢key) == null) | 1 | n | n |
| bucketList = hashTable[Hash(course⇢key)] | 1 | n | n |
| node = Allocate new linked list node | 1 | n | n |
| node⇢next = null | 1 | n | n |
| node⇢data = course | 1 | n | n |
| ListAppend(bucketList, node) | 1 | n | n |
| CLOSE file | 1 | 1 | 1 |
| **Total Cost** | | | 16n + 4 |
| **Runtime** | | | O(n) |

**BinarySearchTree**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| OPEN file | 1 | 1 | 1 |
| IF file not open | 1 | 1 | 1 |
| PRINT error message | 1 | 1 | 1 |
| READ line | 1 | n | n |
| WHILE (end of file not reached) | 1 | n | n |
| DECLARE course object | 1 | n | n |
| IF the input stream reads less than 2 values | 1 | n | n |
| PRINT error message about file format | 1 | n | n |
| ELSEIF  Input stream reads 2 values | 1 | n | n |
| CALL setCourseNumber for first value | 1 | n | n |
| CALL setcourseName for second value | 1 | n | n |
| ELSE  Input stream reads more than 2 values | 1 | n | n |
| CALL setCoursePrereq for values greater than 2 | 1 | n | n |
| IF (root is null) | 1 | n | n |
| INSERT course in root node | 1 | n | n |
| SET left to null | 1 | n | n |
| SET right to null | 1 | n | n |
| ELSE  SET cur to root | 1 | n | n |
| WHILE (cur is not null) | 1 | n | n |
| IF (node⇢courseNumber < cur⇢courseNumber) | 1 | n | n |
| IF (cur⇢left is null) | 1 | n | n |
| SET cur⇢left to node | 1 | n | n |
| SET cur to null | 1 | n | n |
| ELSE  SET cur to cur⇢left | 1 | n | n |
| ELSE  IF (cur⇢right is null) | 1 | n | n |
| SET cur⇢right to node | 1 | n | n |
| SET cur to null | 1 | n | n |
| ELSE  SET cur to cur⇢right | 1 | n | n |
| SET node⇢left to null | 1 | n | n |
| SET node⇢right to null | 1 | n | n |
| CLOSE file | 1 | 1 | 1 |
| **Total Cost** | | | 27n + 4 |
| **Runtime** | | | O(n) |

All three data structures have a worst-case time complexity of O(n). While Hash Tables have O(1) average time complexity for search, insertion and deletion it is not the best choice for the ABCU Project. First, the size of the data (list of courses) is not expected to be too big. The advantages provided by Hash Tables in the case of large data sets will not be of an added value for this project. But most importantly, one of the main features of the project (which is an option in the software menu) is to alphanumerically sort the courses. In that case, the choice will be made between vectors and binary search trees. Both data structures offer similar advantages and complexity time. By design, BSTs help achieving the sorting goal requested by the project. An in-order traversal of the tree will print the list of course in alphanumerical order. This operation has an O(log(n)) time complexity. Vectors can also be sorted in O(nlog(n)) using the sort function provided by C++ standard library.

Worst-case complexity for common operations (search, insertion, deletion) is O(n) for both. BSTs have the advantage of having an O(log(n)) average time complexity.

While both vectors and BSTs are close in term of efficiency and responding to the program requirements, a Binary Search Tree has a slight advantage with the sorting operation and the average time complexity of searching (Insertion is a one-time operation using an input file and there is no deletion option in the project while searching for specific courses is the most important as it is a feature in the menu of the program). For those reasons, I am choosing a Binary Search Tree to implement my ABCU Software project.