**Project One Pseudocode and Evaluation**

Yaumel Betancourt

Southern New Hampshire University

CS 300

Prof. Lebel

June 9, 2025

**Pseudocode for menu**:

PRECONDITION: choice is equal to 0

WHILE choice != 9

PRINT menu options

INPUT choice

CASE choice of

1:

call: loadCourses

2:

call: PrintAll

3:

call: Search

IF courseId is empty THEN

PRINT “Course not found.”

ELSE

call: displayCourseDetails

ENDIF

9:

PRINT “Good bye.”

ENDCASE

ENDWHILE

**Pseudocode for sorting and printing courses using Vector**:

Sort coursesVector using mergesort algorithm

FOR course IN coursesVector

PRINT course.courseNumber and course.title

FOR prerequisite in course.prerequisites

PRINT prerequisite

ENDFOR

ENDFOR

**Pseudocode for sorting and printing courses using Hash Table**:

INIT coursesVectorSorted

INIT currNode

FOR course IN coursesVector

SET currNode to course

IF currentNode.key equals MAX\_INT THEN

continue

ENDIF

APPEND currentNode to coursesVectorSorted

SET currNode to currNode.next

WHILE currNode not equal to null pointer

APPEND currNode to courseVectorSorted

SET currNode to currNode.next

ENDWHILE

Sort coursesVectorSorted using merge sort algorithm

FOR course IN coursesVectorSorted

PRINT course.courseNumber and course.title

FOR prerequisite in course.prerequisites

PRINT prerequisite

ENDFOR

ENDFOR

ENDFOR

**Pseudocode for sorting and printing courses using Binary Tree:**

SET currNode to rootNode

WHILE currNode is not equal to null pointer

CALL print method recursively for left subtree of currNode

PRINT currNode.course.courseNumber and currNode.course.title

FOR prerequisites in currNode.course.prerequisites

PRINT prerequisite

ENDFOR

CALL print method recursively for right subtree of currNode

ENDWHILE

**Time complexity analysis for Vector**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executed | Total Cost |
| Reading | 1 | n | n |
| Creating Object (for loop consolidated not including append) | 4 | n | 4n |
| Appending to vector | 1 | n | n |
| Total Cost | | | 6n |
| Runtime | | | O(n) |

**Time complexity analysis for Hash Table**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executed | Total Cost |
| Reading | 1 | n | n |
| Creating Object (for loop consolidated not including append) | 6 | n | 6n |
| Hashing function | 1 | n | n |
| Appending Node (worst case with collision) | 1 | n | n |
| Total Cost | | | 7n |
| Runtime | | | O(n) |

**Time complexity analysis for Binary Tree**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executed | Total Cost |
| Reading | 1 | n | n |
| Creating Object (for loop consolidated not including append) | 6 | n | 6n |
| Appending Node | log n | n | n log n |
| Total Cost | | | n log n + 7n |
| Runtime | | | O(n log n) |

**Evaluation**

Each data structure has its advantages and disadvantages. Starting with vectors, this is the simplest approach since the process of loading the courses is simply inserting each course object into the vector as they are processed. The issue arises when it comes to sorting the courses. Since they are most likely unsorted after being processed, a sorting algorithm needs to be used to sort the vector. Fortunately, a merge sort can be used which has a time complexity of O(n log n). A hash table is similar when it comes to processing and inserting courses into a vector. The difference is a hashing function is used where in the vector the course is inserted. In addition, a linked list is used in case there are collisions where two courses can occupy the same location in the vector. Sorting will also be similar, the only difference being all elements of the vector must be traversed and inserted into a new vector which will then be sorted using merge sort. The sorting will also be O(n log n). Lastly, there is the binary tree. Processing courses involve traversing the tree and finding a spot to place the node based on the value of the course number which has a time complexity of O(log n). There is no need for a sorting algorithm since in order traversal will return the courses in alphanumerical order with a time complexity of O(n). It is important to note that it is possible to have a worse time complexity of O(n^2) when inserting and O(n) when searching an unbalanced tree that degenerates into a linked list, but in most cases, this will not be an issue.

**Conclusion**

Since this program is able to load, sort, print, and search for courses, the binary tree is the ideal data structure. As mentioned previously, the insertion process has a time complexity of O(log n) which is efficient. Since in order traversal offers an inherent benefit of a binary tree, that being that in order traversal will return the courses in alphanumerical order, there is no need for a sorting algorithm. Printing all courses has a time complexity of O(n), and searching for a particular has a time complexity of O(log n). Overall, this data structure scales best for this application.