# Matthew R. Leclerc

Southern New Hampshire University

CS-300 | Project One Assignment

**2:**

**MENU PSEUDOCODE:**

PRINT "1. Load Data Structure"

PRINT "2. Print Course List"

PRINT "3. Print Course"

PRINT "4. Exit"

INPUT choice

SWITCH choice:

CASE 1:

LoadDataStructure()

CASE 2:

PrintCourseList()

CASE 3:

INPUT "Enter the course number: " courseNumber

PrintCourse(courseNumber)

CASE 4:

EXIT

DEFAULT:

PRINT "Invalid choice!"

FUNCTION LoadDataStructure():

OPEN file for reading

FOR each line in file:

PARSE line to get courseNumber, name, and list of prerequisites

ADD to DS

CLOSE file

FUNCTION PrintCourseList():

LIST allCourses = SORTED(DS based on courseNumber)

FOR course in allCourses:

PRINT course.courseNumber, course.name

FUNCTION PrintCourse(courseNumber):

COURSE course = FIND course in DS with courseNumber

IF course exists:

PRINT course.courseNumber, course.name

PRINT "Prerequisites:"

FOR prerequisite in course.prerequisites:

PRINT prerequisite

ELSE:

PRINT "Course not found!"

**3:**

**SORTING & PRINTING COURSE LIST (VECTOR):**

VECTOR courses

FUNCTION LoadDataStructure():

OPEN file for reading

FOR each line in file:

PARSE line to get courseNumber, name, and list of prerequisites

ADD to courses

CLOSE file

FUNCTION PrintCourseList():

SORT courses based on courseNumber

FOR course in courses:

PRINT course.courseNumber, course.name

**SORTING & PRINTING COURSE LIST (HASH TABLE):**

HASHTABLE courses

FUNCTION LoadDataStructure():

OPEN file for reading

FOR each line in file:

PARSE line to get courseNumber, name, and list of prerequisites

INSERT course into courses with courseNumber as key

CLOSE file

FUNCTION PrintCourseList():

LIST allCourses = EXTRACT all values from courses

SORT allCourses based on courseNumber

FOR course in allCourses:

PRINT course.courseNumber, course.name

**SORTING & PRINTING COURSE LIST (TREE BST):**

TREE courses

FUNCTION LoadDataStructure():

OPEN file for reading

FOR each line in file:

PARSE line to get courseNumber, name, and list of prerequisites

INSERT course into courses based on courseNumber

CLOSE file

FUNCTION PrintCourseList():

InOrderTraversal(courses.root)

FUNCTION InOrderTraversal(node):

IF node is not NULL:

InOrderTraversal(node.left)

PRINT node.course.courseNumber, node.course.name

InOrderTraversal(node.right)

**4:**

**EVALUATION (RUNTIME ANALYSIS)**

**VECTOR:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | m | m |
| **print the prerequisite course information** | 1 | m | m |
| **Total Cost** | | | 2n+2m+1 |
| **Runtime** | | | O(n+m) |

**HASHTABLE:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | 1 | 1 |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | m | m |
| **print the prerequisite course information** | 1 | m | m |
| **Total Cost** | | | n+2m+3 |
| **Runtime** | | | O(n+m) |

**TREE (BST):**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | Log (n) | Log(n) |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | m | m |
| **print the prerequisite course information** | 1 | m | m |
| **Total Cost** | | | n+2m+log(n)+2 |
| **Runtime** | | | O(n+m) |

Note: ( m = max number of pre-reqs, n = # of courses)

**5:** Based on the advisor’s requirements, analyze each data structure (vector, hash table, and tree). Explain the advantages and disadvantages of each structure in your evaluation.

**Vectors**, also known as dynamic arrays, offer a high degree of simplicity in their implementation and usage, with many programming languages offering built-in support for them. However, one of their key limitations is in searching. Without any supplementary indexing, a search operation within a vector can take up to O(n) time, making it inefficient for larger datasets.

**HashTables**, when equipped with an effective hashing mechanism, can provide constant time operations, allowing actions like insertion, deletion, and search to be executed in average O(1) time. However, they come with the drawback of memory overhead, primarily due to their methods for managing collisions and the storage of keys.

**Binary search trees (BSTs)** inherently uphold the order of their elements, providing an advantage in scenarios where data sequencing is essential. However, a notable drawback is their complexity, particularly when there's a need for them to be self-balancing, making their implementation more intricate.

**6:** When looking at how quickly different systems work and considering the tasks like finding courses, adding them, and checking their prerequisites, the **HashTable** appears to be the best choice for this job. Why? Because HashTables are really fast for common tasks like adding, deleting, or searching. This speed is essential when we have lots of course data. Plus, if you have a course number, you can quickly get its details with a HashTable, without having to go through a long list or tree.