**Miguel Hilario**

**CS-300**

**Module 6 Project One**

**CS 300 Pseudocode Document**

// Vector pseudocode

int numPrerequisiteCourses(Vector<Course> courses, Course c) {

int totalPrerequisites = c.prerequisites.length

for each prerequisite p in c.prerequisites:

totalPrerequisites += p.prerequisites.length

PRINT totalPrerequisites

}

void printSampleSchedule(Vector<Course> courses) {

courses.sortByCourseNumber() // Sort courses by course number

for each Course c in courses:

PRINT c.courseTitle

}

void printCourseInformation(Vector<Course> courses, String courseNumber) {

for each Course c in courses:

if c.courseNumber is equal to courseNumber:

PRINT c.courseTitle

for each prerequisite p in c.prerequisites:

PRINT p.courseTitle

}

// Hashtable pseudocode

int numPrerequisiteCourses(Hashtable<Course> courses) {

int totalPrerequisites = 0

for each Course c in courses:

totalPrerequisites += c.prerequisites.length

PRINT totalPrerequisites

}

void printSampleSchedule(Hashtable<Course> courses) {

// Hashtable does not guarantee order; printing all courses may not be in alphanumeric order

for each Course c in courses:

PRINT c.courseTitle

}

void printCourseInformation(Hashtable<Course> courses, String courseNumber) {

if courses.contains(courseNumber):

Course c = courses.get(courseNumber)

PRINT c.courseTitle

for each prerequisite p in c.prerequisites:

PRINT p.courseTitle

}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {

// Traverse the tree and count all prerequisites

int totalPrerequisites = 0

for each Course c in courses:

totalPrerequisites += c.prerequisites.length

PRINT totalPrerequisites

}

void printSampleSchedule(Tree<Course> courses) {

courses.inOrderTraversal() // Sort courses in alphanumeric order

for each Course c in courses:

PRINT c.courseTitle

}

void printCourseInformation(Tree<Course> courses, String courseNumber) {

Course c = courses.find(courseNumber)

if c is not null:

PRINT c.courseTitle

for each prerequisite p in c.prerequisites:

PRINT p.courseTitle

}

**Runtime Analysis:**

**Vector:**

**Pros:** Allows quick access to elements using indices. Good for sequential access. Simple to use and understand.

**Cons**: Insertions and deletions can be slow, especially in large datasets. Sorting large vectors can be costly.

|  |  |  |  |
| --- | --- | --- | --- |
| **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 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 3n+1 |
| **Runtime** | | | O(n) |

**Hashtable:**

**Pros:** operations in large datasets.

**Cons:** Might have collisions which impact performance. Not suitable if the exact order of items matters.

|  |  |  |  |
| --- | --- | --- | --- |
| **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 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 3n+1 |
| **Runtime** | | | O(n) |

**Tree:**

**Pros:** Provides efficient search, insert, and delete operations, especially in sorted data. Preserves order in terms of traversals.

**Cons:** More complex to implement. Memory overhead can be higher. Slower performance for unbalanced trees.

|  |  |  |  |
| --- | --- | --- | --- |
| **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 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 3n+1 |
| **Runtime** | | | O(n) |

**Recommendation:**

Based on the analysis of runtime complexities and the requirements presented by the advisor, the Hashtable appears as a recommended data structure. Its average case for operations involving reading and creating course objects aligns well with the requirements. It offers efficient insertion and retrieval times and, in this context, might provide better performance compared to the other structures, especially considering the practical aspects of storing and accessing course-related information.