# CS 300 Pseudocode Document

## Pseudocode

CLASS Course {

id: string with course ID number

title: STRING with course title

prerequisites: LIST of STRING

}

**//Pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for file format errors.**

ifStream file(courseInformationFile);

string line;

vector<Course> courses;

While(getLine(file, line)){

Try{

parseLine(line)// function that parses each comma when it is encountered.

if each parse line has at least two parameters

if prerequisites are found as a course on the file

iss >> Course Number(id) >> Course Title;

else print “Prerequisite x not found as a course.

else print “Line x has less than 2 parameters”

Check if prerequisites are found as a course on the file else print “Prerequisite x not found as a course.

string prerequisite;

while (iss >> prerequisite) {

prerequisites.push\_back(prerequisite);

}

courses.push\_back(course); // Store the parsed course in the courses vector

}catch(const std::exception& e){

Print a statement to user about the file format error.

}

}

**// Vector pseudocode**

int number of Prerequisite Courses(Vector<Course> courses, Course course) {

totalPrerequisites = prerequisites of course

for each prerequisite i in totalPrerequisites

add prerequisites of i to totalPrerequisites

return number of totalPrerequisites

}

void printSampleSchedule(Vector<Course> courses) {

Initialize an empty vector to store the sorted order of courses (sample schedule)

Initialize an empty map to store the number of prerequisites for each course

Populate the “in-degree” map by iterating through each course in the vector

If the course is not already in the in-degree map, set its in-degree to 0

Iterate through the prerequisites of the current course

increment the in-degree of each prerequisite course

Perform the topological sort

Find a course with in-degree 0 (no prerequisites)

If no such course is found, there is a cycle in the prerequisites (error)

Print "Error: Cycle detected in prerequisites"

Return

Add the course with no prerequisites to the sorted list of courses

Remove the course with no prerequisites from the inDegree map

Update the in-degree of the remaining courses in the map

Print the sample schedule (sortedCourses)

}

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

**for all courses**

**if the course is the same as courseNumber**

**print out the course information**

**for each prerequisite of the course**

**print the prerequisite course information**

}

**//Hashtable pseudocode**

int numPrerequisiteCourses(Hashtable<Course> courses) {

Declare a new variable to keep track of the total number of prerequisite courses

For each course object in the Hashtable

If course object contains a prerequisite

add the number of prerequisite courses to the total

Return the total number of prerequisite courses

}

void printSampleSchedule(Hashtable<Course> courses) {

List<Course> sortedCourses

Map<Course, int> inDegree

For each Course in courses:

If course not in inDegree:

inDegree[course] = 0

For each Course prerequisite:

inDegree[prerequisite] = inDegree[prerequisite] + 1

While inDegree is not empty:

courseWithNoPrerequisite = course in inDegree with value 0

If courseWithNoPrerequisites is None:

Print "Error: Cycle detected in prerequisites."

Return

sortedCourses.append(courseWithNoPrerequisites)

inDegree.remove(courseWithNoPrerequisites)

For each Course nextCourse in courseWithNoPrerequisites.prerequisites:

inDegree[nextCourse] = inDegree[nextCourse] - 1

For each Course in sortedCourses:

Print courseNumber, courseTitle

}

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

Check if the courseNumber exists in the Hashtable

Retrieve the Course object associated with the courseNumber from the Hashtable

Iterate through courses and print each prerequisite course.

Print prerequisite

Else:

Print "No Prerequisites for this course"

Else:

Print an error message if the courseNumber is not found in the Hashtable

}

**// Tree pseudocode**

Class TreeNode:

Public:

Course course

List<TreeNode\*> children

Function TreeNode(Course aCourse):

course = aCourse

Class Tree:

Public:

TreeNode\* root

int numPrerequisiteCourses(Tree<Course> courses) {

// Variable to count total prerequisites

int totalPrerequisites = 0

Function DFS(TreeNode\* node):

If node is null:

Return;

For each child in node.children:

totalPrerequisites += 1

DFS(child)

DFS(courses.root)

Return totalPrerequisites}

void printSampleSchedule(Tree<Course> courses) {

DFS(TreeNode\* node):

If node is null:

Return

For each child in node.children:

DFS(child)

print node.course.courseNumber, node.course.title

DFS(courses.root)

}

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

Function findCourse(TreeNode\* node, String courseNumber):

If node is null:

Return null

If node.course.courseNumber == courseNumber:

Return node

For each child in node.children:

TreeNode\* result = findCourse(child, courseNumber)

If result is not null:

Else:

Return result

Else:

Return null

}

TreeNode\* courseNode = findCourse(courses.root, courseNumber)

If courseNode is not null:

Print "Course Number:", courseNode.course.courseNumber

Print "Title:", courseNode.course.title

If courseNode.children.length > 0:

Print "Prerequisites:"

For each child in courseNode.children:

Print child.course.courseNumber, child.course.title

Else:

Print "No Prerequisites for this course"

Else:

Print "Error: Course not found!"

}

## Menu pseudocode:

//While choice is not 9 run switch statement

int choice = 0;

while (choice != 9) {

cout <<"Menu:" << endl;

cout << " 1. Load Bids" << endl;

cout << " 2. Print Course List" << endl;

cout << " 3. Print Course" << endl;

cout << " 9. Exit" << endl;

cout << "Enter choice: ";

cin >> choice;

switch (choice):

case 1:

// Complete the method call to load the course information file

loadBids(csvPath);

break;

case 2:

PrintCourseList();

break;

case 3:

PrintCourse(courseId)

break;

## Runtime Analysis

When you are ready to begin analyzing the runtime for the data structures that you have created pseudocode for, use the chart below to support your work. This example is for printing course information when using the vector data structure. As a reminder, this is the same pairing that was bolded in the pseudocode from the first part of this document.

| **Vector Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Space Complexity** | | | O(1) |
| **Time Complexity** | | | O(n) |

| **Hash table Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Space Complexity** | | | O(1) |
| **Time Complexity** | | | O(n) |

| **Binary Tree Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Space Complexity** | | | O(n) |
| **Time Complexity** | | | O(logn) |

**Evaluation:**

* **HashTable**: The printCourseInformation function standsout due to its average-case time complexity O(1). The key-based lookup property of a hash table makes retrieving a specific course extremely efficient on average. However, hash tables have a potential worst-case of O(n) when there are many collisions, but with a good hash function and rehashing strategy, this can often be mitigated.
* **Vector**: The operations are linear, making it straightforward but not optimal for operations like retrieving a specific course.
* **Tree** (Binary Search Tree): While the tree can be efficient for sorted data and if maintained as balanced, it doesn't offer advantages over hash tables for this use case, especially when direct lookup is a primary operation.

Since the main concern is the quick retrieval of course information the **HashTable** would be the best choice among the three. Since the application is not required to frequently perform operations that require sorting or structured hierarchical traversal, a balanced binary search tree is not the most appropriate.