# CS 300 Pseudocode Document

## Function Signatures

Below are the function signatures that you can fill in to address each of the three program requirements using each of the data structures. The pseudocode for printing course information, if a vector is the data structure, is also given to you below (depicted in bold).

// Vector pseudocode

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

totalPrerequisites = prerequisites of course c 1

for each prerequisite p in totalPrerequisites n

add prerequisites of p to totalPrerequisites 1

print number of totalPrerequisites 1

}

2n +3

void printSampleSchedule(Vector<Course> courses) {

while courses is not empty: n

foundEligibleCourse = false 1

for each course in courses: n

if NoUnscheduledPrerequisites(course, schedule): n

add course to schedule n

remove course from courses list n

foundEligibleCourse = true n

break the loop when course requirements are met and it is in the schedule

if not foundEligibleCourse: 1

1 print "Unable to schedule remaining courses due unsatisfied prerequisites."

break

print "Sample Schedule:" 1

for each course in schedule n

print "Course Number: " + course.courseNumber n

print "Course Title: " + course.courseTitle n

}

9n+3

bool NoUnscheduledPrerequisites(Course course, Schedule schedule) { 1

for each prerequisite in course.prerequisites: n

if prerequisite not in schedule: 1

return false if course has unscheduled prerequisites 1

return true if all prerequisites are scheduled 1

}

n+4

}

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**

}

4n+1

**total 16n+11**

**runtime O(n)**

**// Hashtable pseudocode**

int numPrerequisiteCourses(Hashtable<Course> courses) { 1

totalPrerequisites = 0 1

for each course in courses: n

if courseNumber in course.prerequisites: n

// Count prerequisites

totalPrerequisites = totalPrerequisites + 1 n

print "Total Prerequisites for " + c.courseNumber + ": " + 1 totalPrerequisites

} 3n+2

}

void printSampleSchedule(Hashtable<Course> courses) { 1

schedule = createEmptySchedule() 1

while courses is not empty: n

foundEligibleCourse = false n

for each course in courses: n

if canScheduleCourse(course, schedule): n

add course to schedule n

remove course from courses list n

foundEligibleCourse = true n

break //Exit the loop once a course is found 1

if not foundEligibleCourse: 1

print "Unable to schedule remaining courses due to 1 unsatisfied prerequisites."

break 1

print "Sample Schedule:" 1

for each course in schedule n

print "Course Number: " + course.courseNumber n

print "Course Title: " + course.courseTitle n

} 10n+7

bool canScheduleCourse(Course course, Schedule schedule) { 1

for each prerequisite in course.prerequisites: n

if prerequisite not in schedule: n

n return false if this course has unscheduled prerequisites

return true if prerequisites are scheduled n

}

} 4n+1

void printCourseInformation(Hashtable<Course> courses, String 1 courseNumber) {

course = getCourseFromHashtable(courses, courseNumber) 1

if course is not null: 1

print "Course Number: " + course.courseNumber 1

print "Course Title: " + course.courseTitle 1

print "Prerequisites:" 1

for each prerequisite in course.prerequisites: n

print "- " + prerequisite n

else: 1

print "Course not found."

}

2n+7

Course getCourseFromHashtable(Hashtable<Course> courses, String 1 courseNumber) {

if courseNumber in courses: 1

return courses[courseNumber] 1

else:

return null if Course not found in the hashtable 1

}

}

4

**total 19n+21**

**runtime O(n)**

**// Tree pseudocode**

int numPrerequisiteCourses(Tree<Course> courses) { 1

return countPrerequisiteCourses(courses.getRoot()) 1

}

2

int countPrerequisiteCourses(TreeNode<Course> node) { 1

if node is null: 1

return 0 1

course = node.getData() 1

totalPrerequisites = length(course.prerequisites) 1

leftCount = countPrerequisiteCourses(node.getLeft()) 1

rightCount = countPrerequisiteCourses(node.getRight()) 1

return totalPrerequisites + leftCount + rightCount 1

}

8

void printSampleSchedule(Tree<Course> courses) { 1

schedule = createEmptySchedule() 1

inOrderTraversal(courses.getRoot(), schedule) 1

print "Sample Schedule:" 1

for each course in schedule n

print "Course Number: " + course.courseNumber n

print "Course Title: " + course.courseTitle n

}

3n+14

void inOrderTraversal(TreeNode<Course> node, Schedule schedule) { 1

if node is not null: 1

// Try to schedule the left subtree

inOrderTraversal(node.getLeft(), schedule) 1

// Check if the current course can be scheduled

if canScheduleCourse(node.getData(), schedule): 1

add node.getData() to schedule 1

// Attempt to schedule the right subtree

inOrderTraversal(node.getRight(), schedule) 1

}

bool canScheduleCourse(Course course, Schedule schedule) { 1

for each prerequisite in course.prerequisites: n

if prerequisite not in schedule: 1

return false if this course has unscheduled prerequisites 1

return true if all prerequisites are scheduled 1

}

N+10

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

TreeNode<Course> courseNode = findCourseNode(courses.getRoot(), courseNumber)

if courseNode is not null: 3

Course course = courseNode.getData() 1

print "Course Number: " + course.courseNumber 1

print "Course Title: " + course.courseTitle 1

print "Prerequisites:" 1

for each prerequisite in course.prerequisites: n

print "- " + prerequisite

else: n

print "Course not found." 1

}

2n+8

TreeNode<Course> findCourseNode(TreeNode<Course> currentNode, String courseNumber) {

if currentNode is null: 2

return null 1

Course currentCourse = currentNode.getData() 1

if currentCourse.courseNumber is equal to courseNumber: 1

return currentNode 1

// Search in the left and right subtrees

leftResult = findCourseNode(currentNode.getLeft(), courseNumber) 1

rightResult = findCourseNode(currentNode.getRight(), courseNumber)

if leftResult is not null: 2

return leftResult 1

return rightResult 1

}

11

}

**total 7n+43**

**runtime O(n)**

## Example 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.

| **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** | | | 4n + 1 |
| **Runtime** | | | O(n) |

Vectors are an efficient data structure that offer simplicity and ease of implementation, which can make more sense for basic data storage tasks. They are good at providing efficient access to objects via index-based retrieval, making them good for tasks like retrieving course information by course number. Vectors can also be memory-efficient since they use contiguous memory. However, there are disadvantages to vectors. Vectors lack built-in sorting capabilities, making the task of sorting a vector computationally expensive, which is shown by the high number of n variables in the runtime of the pseudocode. Vectors do not provide built-in hashing capabilities either, which could be useful for quick lookups, especially when dealing with a large dataset of courses.

Hashtable data structuresare a good tool for data retrieval and lookups. They are good at providing fast lookups with an average-case time complexity of O(1), which is highly advantageous for finding course information by course number. Their built-in hashing capabilities facilitate quick lookups. However, hashtables do come with some trade-offs. They may consume more memory due to the need for hashing and collision resolution, potentially leading to increased memory overhead. Hashtables do not maintain order, which could pose challenges when printing sorted course lists, requiring additional effort for sorting.

Tree data structures, especially binary search trees, provide an ordered structure for various applications, including course management systems like in this case. Trees maintain a sorted order of elements, making them advantageous for printing sorted course lists efficiently.. Trees also naturally represent hierarchies, making it efficient to check prerequisites. However, trees also have their limitations. In the worst case, unbalanced trees can result in O(n) lookup times, so balancing techniques are essential for maintaining efficiency. They may consume more memory due to node structures and pointers, leading to increased memory overhead compared to simpler data structures. Implementing a balanced binary search tree can be complex, requiring careful balancing to maintain efficiency, which adds to the development effort.

Based on the Big O analysis results and the specific requirements of the advisor's program, I recommend utilizing the Hashtable data structure for managing course information. The Hashtables have very good average runtimes ensuring rapid retrieval of course details by course number, regardless of the dataset's size. Additionally, the Hashtable's built-in hashing capabilities simplify implementation and provide fast lookups, crucial for efficient course management. While Vectors and Trees are viable options, the Hashtable's superior efficiency in lookups and streamlined prerequisite checks make it the most suitable choice for this program, balancing performance and ease of implementation.