# 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).

**// Menu Pseudocode**

int menuInput = 0;

while (menuInput != 9) {

Output menu information/options:

1. Load Data
2. Print Course List
3. Print Course

9. Exit

switch (menuInput) {

case 1:

validateCourseData();

importCourseData();

break;

case 2:

sortCourses();

printSortedCourseList();

break;

case 3:

printCourseInformation();

break;

}

return 0;

}

**// Sort Pseudocode**

// For binary search tree and hash tables, we’ll import the original data into a vector and use a quick sort // algorithm instead of trying to sort the other data structures directly.

void sortCourses(vector courseData, int left, int right) {

if (left < right) {

int pivot = partition(courseData, left, right);

sortCourses(courseData, left, pivot – 1);

sortCourses(courseData, pivot, right);

}

}

void printSortedCourseList(vector courseList) {

for all course in vector courseList:

print out the course information

}

**// Vector pseudocode**

void validateCourseData(String fileName) {

Open file fileName

While EOF not reached:

Read Line

Separate Elements

Save first element to a list of courses used for data validation

If number of elements < 2:

Error

Else if prerequisite course doesn’t exist in course list:

Error

Close file fileName

}

void importCourseData(String fileName) {

Open file fileName

Iterate through the file line by line until EOF reached:

Read Line

Separate Elements

Create Course Object using parameters from the line elements (first element being courseNumber, second being courseName, any additional elements being a prerequisiteCourse)

Add Course Object to Vector

Close file

}

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

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites

add prerequisites of p to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Vector<Course> courses) {

Initialize currentNode to track position

While currentNode does not equal null:

Print course information for currentNode

currentNode++

}

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**

void validateCourseData(String fileName) {

Open file fileName

While EOF not reached:

Read Line

Separate Elements

Save first element to a list of courses used for data validation

If number of elements < 2:

Error

Else if prerequisite course doesn’t exist in course list:

Error

Close file fileName

}

void importCourseData(String fileName) {

Open file fileName

Create a vector to store Course objects

Iterate through the file line by line until EOF reached:

Read Line

Separate Elements

Create Course Object using parameters from the line elements (first element being courseNumber, second being courseName, any additional elements being a prerequisiteCourse)

Create a hashed key based on courseNumber

If the key location in the vector is empty:

Add course object to vector[key]

If the key location is occupied:

Find the next open location in the vector

Add course object to that location

Close File

}

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

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites

add perquisites of p to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Hashtable<Course> courses) {

Initialize currentNode to track position

While currentNode does not equal null:

Print course information for currentNode

currentNode = currentNode->next

}

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

Initialize currentNode to track position

Iterate through nodes from beginning to end:

If currentNode’s courseNumber matches input courseNumber:

Print out the course information

For each prerequisite:

Print course information

}

**// Tree pseudocode**

void validateCourseData(String fileName) {

Open file fileName

While EOF not reached:

Read Line

Separate Elements

Save first element to a list of courses used for data validation

If number of elements < 2:

Error

Else if prerequisite course doesn’t exist in course list:

Error

Close file fileName

}

Void importCourseData(String fileName) {

Open File fileName

Create tree to store course objects

Until reaching EOF, iterate through the file line by line:

Read the line

Separate the elements

Create Course Object using parameters from the line elements (first element being courseNumber, second being courseName, any additional elements being a prerequisiteCourse)

Insert the Course at the correct location:

Initialize currentNode to track position

If root null: insert course at root

Else while currentNode != null:

If courseNumber < currentNode’s courseNumber:

If currentNode->left is empty: Insert Course there

Else: Progress down left branch and continue searching

If courseNumber > currentNode’s courseNumber:

If currentNode->right is empty: Insert Course there

Else: Progress down right branch and continue searching

Close File

}

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

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites:

add prerequisites of p to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Tree<Course> courses) {

Initialize currentNode to track position

While currentNode does not equal null:

printSampleSchedule(currentNode->left)

Print information about currentNode’s course

printSampleSchedule(currentNode->right)

}

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

Initialize currentNode to track position

While currentNode does not equal null:

If currentNode’s Course matches the input courseNumber:

Print information about currentNode->Course

Print information about currentNode->Course perquisites

Return

Else if input courseNumber is less than that of currentNode’s Course:

Progress down the left branch by setting currentNode = currentNode->left

Else if input courseNumber is greater than that of currentNode’s Course:

Progress down the right branch by setting currentNode = currentNode->right

If no match is found:

Return message informing user that the course was not found

}

**Runtime Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Vector | Hash Table | Binary Tree |
| Import Course Information | O(1) | Between O(1) and O(n) depending on number of collisions. | O(log(n)) to O(n) depending on data |
| Print Course Information (Search) | O(n) | Between O(1) and O(n) depending on number of collisions. | O(log(n)) to O(n) depending on data |
| Sort | O(n log(n)) | O(n log(n)) + O(1) | O(n log(n)) + O(1) |
| Print Course Information  (All Courses) | O(n) | O(n) | O(n) |

**Evaluation**

Each of the three data structures have advantages and disadvantages. For example, Vectors are faster at importing course information than both hash tables and binary trees. However, searching for data in a vector is slower than searching for data in a hash table as in the worst-case scenario, the entire vector must be searched. Sorting the data and printing all the data are relatively equal among the three data structures. The performance of the binary tree can depend on the data being input. If sorted data or semi-sorted data is input, the binary tree can become unbalanced and less performant.

It's also important to consider the use cases for this application. Importing data is likely to occur infrequently, likely only when first running the application. Printing a list of all the courses is similarly likely to occur infrequently. However, searching for and printing specific course information is likely to occur more frequently. Therefore, we should weigh the efficiency of searching for a specific course over the other functionality. In this category, hash tables are clearly the best option. With a sufficiently sized table, many searches would perform at O(1).

**Recommendation**

Given the use case for the application and the performance metrics of the three data structures being analyzed, I believe a Hash Table would be the best data structure to use for this application. Given an adequate table size, hash tables are more performant at importing and searching for courses, and equally performant at sorting and printing all course information than the other two data structures.