# 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) {

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) {

}

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**

}

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

Initialize an empty vector "courseData" to store course information. //O(n)

Open the Course Information document. //O(1)

If the file cannot be opened, display an error message. //O(1)

Read each line from the file until the end of the file is reached. //O(1)

Check if there are at least two parameters on the line. //O(1)

Create a new Course object. //O(n)

Extract courseNumber, name, and prerequisites. //O(1)

Create vector to store prerequisites. //O(1)

Iterate through the remaining prerequisites and add them to the prerequisites vector. //O(n)

Add the Course object with the extracted data to the "courseData" vector. //O(n)

//Pseudocode to show how to create course objects and store them in the appropriate data structure

Define a class Course with attributes courseNumber, name, and prerequisites in a vector. //O(n)

Initialize an empty vector "courseData" to store course information. //O(n)

Open the Course Information document. //1

If the file cannot be opened, display an error message.

Read each line from the file until the end of the file is reached. //1

Extract courseNumber, name, and prerequisites. //O(n)

Create a new Course object. //O(n)

Create a list or vector to store prerequisites. //O(n)

Iterate through the remaining prerequisites and add them to the prerequisites vector. //O(n)

Set the Course object's attributes with the extracted data. //O(n)

Add the Course object to the "courseData" vector. //O(n)

//Pseudocode that will search the data structure for a specific course and print out course information and prerequisites.

Define a function findCourse(courseData, courseNumber): //O(n)

Initialize a variable "foundCourse" to None. //O(n)

Iterate through each Course object in "courseData": //O(n)

If the courseNumber matches the courseNumber of the current Course object: //O(n)

Set "foundCourse" to the current Course object. //O(n)

End If

If "foundCourse" is not None: //O(n)

Print the course information (courseNumber and name). //O(n)

Print the prerequisites (if any). //O(n)

End If

If "foundCourse" is null, print an error message indicating the course was not found. //O(n)

End if

Prompt the user to enter a courseNumber to search for. //O(n)

Call the findCourse function with "courseData" and the user's input. //O(n)

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

// Hashtable pseudocode

int numPrerequisiteCourses(Hashtable<Course> courses) {

}

void printSampleSchedule(Hashtable<Course> courses) {

}

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

}

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

For each line in the file: // O(n)

Read the line //1

Split the line into 2 keys // O(n)

If the number of keys is less than 2: // O(n)

Print an error message for the line 1

Continue to the next line // O(n)

//Pseudocode to show how to create course objects and store them in the appropriate data structure.

Else: // O(n)

Extract courseNumber, courseTitle, and prerequisites from the keys // O(n)

Create a new Course object with courseNumber, course Title, and prerequisites // O(n)

Add the Course object to the hash table using courseNumber as the key // O(n)

//Pseudocode that will print out course information and prerequisite

For each Course object in the hash table: // O(n)

Print courseNumber //1

Print courseTitle //1

If there are prerequisites: // O(n)

Print "Prerequisites:" //1

For each prerequisite in the prerequisites list of the Course: // O(n)

Print prerequisite //1

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

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {

}

void printSampleSchedule(Tree<Course> courses) {

}

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

}

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

Try to open the Course Information document //(O(log(n))

If the file cannot be opened: //(O(log(n))

Print an error message //1

//Design pseudocode to show how to create course objects and store them in the appropriate data structure.

Initialize an empty binary search tree (BST) to store course objects //(O(log(n))

While not end of file: //(O(log(n))

Read a line from the file //(O(log(n))

Split the line into keys ///(O(log(n))

If the number of keys is less than 2: //(O(log(n))

Print an error message for the line and continue to the next line //1

Else: //(O(log(n))

Extract course Number, course Title, and prerequisites from the keys //(O(log(n))

Create a new Course object with course Number, course Title, and prerequisites //(O(log(n))

Insert the Course object into the BST //(O(log(n))

For each Course object in the BST: //(O(log(n))

For each prerequisite in the prerequisites list of the Course: //(O(log(n))

If there is no Course object with the prerequisite course Number in the BST: //(O(log(n))

Print an error message indicating missing prerequisite //1

//Design pseudocode that will print out course information and prerequisites.

Define a function to perform an in-order traversal of the BST:

If the current node is not null: //(O(log(n))

Recursively traverse the left subtree //(O(log(n))

Print course Number //1

Print course Title //1

If there are prerequisites: //(O(log(n))

Print "Prerequisites:" //1

For each prerequisite in the prerequisites list of the Course: //(O(log(n))

Print prerequisite //1

Print a separator line //1

Recursively traverse the right subtree //(O(log(n))

Call the in-order traversal function starting from the root of the BST //(O(log(n))

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

Among the three options, the vector is straightforward but struggles with searching and large datasets. Hash tables are quick at finding things but can be memory-intensive and require extra work for sorting. Binary search trees (BSTs) excel at finding and sorting things, though they might be a bit slower when adding or removing data. With the runtime analysis charts, I choose the BST for my code. It seems to be the quickest to access courses, sorting them alphabetically, and efficient memory use.

//Pseudocode for Menu

Initialize Menu

Display Menu:

Print "1. Load Data Structure"

Print "2. Print Course List"

Print "3. Print Course"

Print "4. Exit"

Get User Input:

Print "Select from Menu”

Read input

Process User Input:

If input is 1:

Load Data Structure:

Try to open the Course Information file

If the file cannot be opened:

Print error opening file

Else:

Initialize the data structure

While not end of file:

Read a line from the file

Split the line into keys

Extract courseNumber, courseTitle, and prerequisites

Create a new Course object

Insert the Course object into the data structure

Close the file

If input is 2:

Print Course List:

Print an alphanumerically ordered list of all the courses in the Computer Science department

If input is 3:

Print Course:

Print "Enter the course number: "

Read courseNumber

Search the data structure for a course with the given courseNumber

If a matching course is found:

Print courseTitle

If there are prerequisites:

Print prerequisites

Else:

Print "Course not found."

If input is 4:

Exit the program