# 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) {

for course in vector\_courses:

print(course)

}

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) {

function print\_course\_list(hash\_table\_courses):

courses = hash\_table\_courses.values()

}

void printSampleSchedule(Hashtable<Course> courses) {

sort(courses)

}

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

for course in courses:

print(course)

print\_course\_list(courses\_hash\_table)

}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {

function print\_course\_list(tree\_courses):

// Traverse the tree to get the sorted list of courses

sorted\_courses = inorder\_traversal(tree\_courses)

}

void printSampleSchedule(Tree<Course> courses) {

}

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

// Print the sorted list of courses

for course in sorted\_courses:

print(course)

print\_course\_list(courses\_tree)

}

## 

## Menu Options Pseudocode

function load\_data\_structure(file):

// Load the data from the file into the data structure

function print\_course\_list():

// Sort the list alphanumerically

// Retrieve the courses

function print\_course(course\_code):

// Print the course title and prerequisites

function main():

data\_structure\_loaded = false

while true:

print("Menu:")

print("1. Load Data Structure")

print("2. Print Course List")

print("3. Print Course")

print("4. Exit")

choice = read\_input("Enter your choice:")

if choice == 1:

file = read\_input("Enter the file name:")

load\_data\_structure(file)

data\_structure\_loaded = true

print("Data structure loaded successfully.")

else if choice == 2:

if data\_structure\_loaded:

print\_course\_list()

else:

print("Data structure not loaded. Please load the data structure first.")

else if choice == 3:

if data\_structure\_loaded:

course\_code = read\_input("Enter the course code:")

print\_course(course\_code)

else:

print("Data structure not loaded. Please load the data structure first.")

else if choice == 4:

print("Exiting the program.")

break

else:

print("Invalid choice. Please enter a valid option.")

main()

## 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) |

**Vector:**

Pros: Vector allows for sequential access and easy indexing of elements and allows for resizing if necessary.

Cons: Finding specific elements might also take longer as the entire vector needs to be searched.

**Hash Table:**

Pros: Hash tables provide efficient lookup and insertion operations on average (O(1)) when properly implemented.

Cons: They do not guarantee the order of insertion, so additional steps may be needed to sort the data.

**Tree:**

Pros: They also allow efficient searching, insertion, and deletion operations when balanced.

Cons: Self-balancing trees can have higher memory overhead due to additional pointers and cache.

**Data structure recommendation**

Based on the Big O analysis and the advantages/disadvantages, the recommendation for this scenario would be to use the Vector-based data structure. The Vector offers a total worst-case running time of O(n), which is comparable to the Hash Table. However, the Vector has the advantage of maintaining the order of insertion, allowing for easy sorting of elements when printing the list of courses in alphanumeric order. This is crucial for fulfilling the menu requirement of displaying the sorted course list. Both the Vector and Hash Table have similar time complexities, but Vectors have advantages in terms of sequential access and maintaining the order of insertion. Additionally, the Vector data structure is simpler to implement compared to a self-balancing Tree structure, reducing the complexity of the codebase. Considering these factors, the Vector-based data structure aligns well with the requirements of the ABCU's advising program. Since the menu requires printing a sorted list of courses, the Vector's ability to maintain insertion order while easily sorting the elements makes it a suitable choice. However, the final decision should also consider other factors such as the size of the dataset, specific memory requirements, and any additional operations or constraints not mentioned in the requirements.