# 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**

struct Course {

String storing courseNumber

String storing courseName

Vector<String> storing coursePrerequisites

}

// reads file contents and stores it in a course object

void openReadCourseList (file courseList) {

Vector<Courses> newCourse will store new course information

While there is a line to read {

if there is a comma-separated string value{

Read comma-separated string and store in courseNumber String

}

if there is a comma-separated string value{

Read comma-separated value and store in courseName

}

if line contains more comma separated values{

Read next value and store in Vector coursePrerequisites

}

Create a node vector to store each node

}

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

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites

add prerequisites of p to totalPrerequisites

return number of totalPrerequisites

}

void printSampleSchedule(Vector<Course> courses) {

for all courses

print out the course information

for each prerequisite of the course

print the prerequisite course information

}

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**

struct Course {

String storing courseNumber

String storing courseName

Vector<String> storing coursePrerequisites

}

// reads file contents and stores it in a course object

void openReadCourseList (file courseList) {

Vector<Courses> newCourse will store new course information

While there is a line to read {

if there is a comma-separated string value{

Read comma-separated string and store in courseNumber String

}

if there is a comma-separated string value{

Read comma-separated value and store in courseName

}

if line contains more comma separated values{

Read next value and store in Vector coursePrerequisites

}

Create a node vector to store to be added to the hash table.

}

int numPrerequisiteCourses(Hashtable<Course> courses) {

get key from the course by hashing the course id number

for every course key

add 1 to prerequisiteCourseNum

return prerequisiteCourseNum

}

void printSampleSchedule(Hashtable<Course> courses) {

get key from the course by hashing the course id number

for every course key

print course information

for every prerequisite

print course prerequisite information

}

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

get key from the course by hashing the course id number

print course information

for every course prerequisite

print prerequisites

}

**// Tree pseudocode**

struct Course {

String storing courseNumber

String storing courseName

Vector<String> storing coursePrerequisites

}

// reads file contents and stores it in a course object

void openReadCourseList (file courseList) {

Vector<Courses> newCourse will store new course information

While there is a line to read {

if there is a comma-separated string value{

Read comma-separated string and store in courseNumber String

}

if there is a comma-separated string value{

Read comma-separated value and store in courseName

}

if line contains more comma separated values{

Read next value and store in Vector coursePrerequisites

}

Create a node vector to be added to binary tree.

}

int numPrerequisiteCourses(Tree<Course> courses) {

create a new node and add the bid information to it

if the root of the tree is null (empty)

save new node with bid information in the root of the tree

else

traverse the tree recursively to find an empty place

if a bid is smaller than the root

insert node as a left child

if a bid is larger than the root

insert node as a right child

}

void printSampleSchedule(Tree<Course> courses) {

if root node is not empty

call printSampleSchedule with the left node as the root

print node information

call printSampleSchedule with the right node as the root

}

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

search for course number by traversing through tree

if node with course number is found

print course number

if node with course number has prerequisites

print prerequisites

}

**// Menu pseudocode**

void displayMenu(){

display menu prompting user for input

if loadDataStructure is input

call loadDataStructure

if printCourseAlphaNum is input

call printCourseAlphaNum

if printCourseInformation is input

prompt user for a specific course number

call printCourseInformation

if exit is input

exit program

}

void loadDataStructure(DataStructure<Course> courses){

for every course

find the correct alphanumeric location to insert new course

redirect pointers to make space for new item

insert course into data structure

redirect pointers to finalize insertion

}

Void printCourseListAlphaNum(DataStructure<Course> courses){

sort data structure from lowest to highest

while courses is not null traverse the data structure

for every course

print courses in alphanumeric order

}

## 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 pseudocode** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Vector<Courses> newCourse will store new course information | 1 | n | n |
| While there is a line to read | 1 | n | n |
| if there is a comma-separated string value | 1 | n | n |
| Read comma-separated string and store in courseNumber String | 1 | n | n |
| Read comma-separated value and store in courseName | 1 | n | n |
| Read next value and store in Vector coursePrerequisites | 1 | n | n |
| Create a node vector to store each node to be added to Vector. | 1 | 1 | 1 |
| **Total Cost** | | | 7n + 1 |
| **Runtime** | | | O(n) |

| **Hashtable pseudocode** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Vector<Courses> newCourse will store new course information | 1 | n | n |
| While there is a line to read | 1 | n | n |
| if there is a comma-separated string value | 1 | n | n |
| Read comma-separated string and store in courseNumber String | 1 | n | n |
| Read comma-separated value and store in courseName | 1 | n | n |
| Read next value and store in Vector coursePrerequisites | 1 | n | n |
| Create a node vector to store each node to be added to Hashtable. | 1 | 1 | 1 |
| **Total Cost** | | | 7n + 1 |
| **Runtime** | | | O(n) |

| **Tree pseudocode** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Vector<Courses> newCourse will store new course information | 1 | n | n |
| While there is a line to read | 1 | n | n |
| if there is a comma-separated string value | 1 | n | n |
| Read comma-separated string and store in courseNumber String | 1 | n | n |
| Read comma-separated value and store in courseName | 1 | n | n |
| Read next value and store in Vector coursePrerequisites | 1 | n | n |
| Create a node vector to store each node to be added to Tree. | 1 | 1 | 1 |
| **Total Cost** | | | 7n + 1 |
| **Runtime** | | | O(n) |

## Data structures advantages and disadvantages

**Vector:** A vector has the advantage of direct accessibility of an item with an index, has some simple algorithm implementations, and is light on memory in comparison with other data structures. The disadvantages are that vector size must be declared when instantiated, so the size of the application must be known, and iterating through all elements when inserting and deleting items is resource intensive.

**Hash table**: A hash table has the advantages of constant time operation, reading faster than trees, and searching a data base efficiently. The disadvantages are that items are not in order, and creating a hash derived from an item specific ID can create collisions, adding to the complexity of the algorithms used.

**Tree**: A binary tree has the advantage of being memory efficient, and because items in the tree are sorted, finding items, traversing the tree, finding an adjacent item, and finding items greater or less than a specific item can be accomplished very efficiently due to its binary structure. A disadvantage is that the implementation and troubleshooting of a binary tree can be complicated due to recursion.

## Project data structure recommendation

The binary tree data structure is a good candidate for the database required by ABCU’s computer science department.

Even though there is no notable difference in the O-notation between the pseudocodes, there is a clear advantage in the user of a binary tree because its items are sorted, quickly accessible, and resizing should be efficient in comparison to the other two options. Therefore, it is the data structure I intend to use in the implementation of the ABCU’s computer science program.