# 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**

CREATE Course Struct {

Course() {

Course Number;

Course Name;

Course Prerequisites;

}

}

CREATE file path variable;

Vector<Course> loadCourses(string CSVpath) {

CREATE Course VECTOR;

//Open file

OPEN Course CSV file;

ASSIGN CSV file to path variable;

//Read and store file values in Course Vector

WHILE not end of CSV file {

CREATE new Course Struct;

FIND next NEWLINE;

SAVE CSV elements before newline in new Course Struct instance variables;

//The instance variable for the prerequisites of each course object will be a vector

//Input Validation and error checking

IF new Course Struct does not contain Number and Name {

PRINT ERROR;

}

SAVE new Course Struct as next element in Course VECTOR;

} //End WHILE loop

//Input Validation and error checking

FOR each Course Struct in Course Vector {

FOR each Prerequisite Course Number {

IF Course number not found in any Course Struct Course Number {

PRINT ERROR;

}

}

}

}

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**

}

**// Hashtable pseudocode**

CREATE Course Struct {

Course() {

Course Number;

Course Name;

Course Prerequisites;

}

}

CREATE file path variable;

Vector<Course> loadCourses(string CSVpath) {

CREATE Course VECTOR;

//Open file

OPEN Course CSV file;

ASSIGN CSV file to path variable;

//Read and store file values in Course Vector

WHILE not end of CSV file {

CREATE new Course Struct;

FIND next NEWLINE;

SAVE CSV elements before newline in new Course Struct instance variables;

//The instance variable for the prerequisites of each course object will be a vector

//Input Validation and error checking

IF new Course Struct does not contain Number and Name {

PRINT ERROR;

}

HashTable::Insert(Course, courses);

} //End WHILE loop

//Input Validation and error checking

FOR each Course Struct in Course Vector {

FOR each Prerequisite Course Number {

IF Course number not found in any Course Struct Course Number {

PRINT ERROR;

}

}

}

}

HashTable::Insert(Course, courses) {

CREATE key for given course using hash();

Retrieve node using key;

Assign this node to the retrieved key position;

IF (no entry found for the key) {

CREATE new node;

INSERT node at location;

}

ELSE IF (node is not used) {

Assign old node key to UINT\_MAX;

Set old node key to key;

Set old node course to course;

Set node next to nullptr;

}

ELSE {

Find the next open node;

Add new node to the end;

}

}

int numPrerequisiteCourses(Hashtable<Course> courses) {

}

void printSampleSchedule(Hashtable<Course> courses) {

}

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

HashTable::Search(String courseNumber);

Print course

}

**// Tree pseudocode**

**CREATE Course Struct {**

**Course() {**

**Course Number;**

**Course Name;**

**Course Prerequisites;**

**}**

**}**

**CREATE file path variable;**

**Vector<Course> loadCourses(string CSVpath) {**

**CREATE Course VECTOR;**

**//Open file**

**OPEN Course CSV file;**

**ASSIGN CSV file to path variable;**

**//Read and store file values in Course Vector**

**WHILE not end of CSV file {**

**CREATE new Course Struct;**

**FIND next NEWLINE;**

**SAVE CSV elements before newline in new Course Struct instance variables;**

**//The instance variable for the prerequisites of each course object will be a vector**

**//Input Validation and error checking**

**IF new Course Struct does not contain Number and Name {**

**PRINT ERROR;**

**}**

**SAVE new Course Struct as next element in Course VECTOR;**

**} //End WHILE loop**

**//Input Validation and error checking**

**FOR each Course Struct in Course Vector {**

**FOR each Prerequisite Course Number {**

**IF Course number not found in any Course Struct Course Number {**

**PRINT ERROR;**

**}**

**}**

**}**

**}**

**void BinarySearchTree::Insert(<Course>, courses) {**

**// if root equal to null ptr**

**if (root == nullptr) {**

**// root is equal to new node course**

**root = new Node(course);**

**}**

**// else**

**else {**

**// add Node root and course**

**this->addNode(root, course);**

**}**

**}**

**Course BinarySearchTree::Search(string courseNumber) {**

**// set current node equal to root**

**Node\* current = root;**

**// keep looping downwards until bottom reached or matching course number found**

**while (current != nullptr) {**

**// if match found, return current course number**

**if (current->course.number.compare(courseNumber) == 0) {**

**return current->course;**

**}**

**// if course number is smaller than current node traverse left**

**else if (courseNumber.compare(current->course.courseNumber) < 0) {**

**current = current->left;**

**}**

**// else larger so traverse right**

**else {**

**current = current->right;**

**}**

**}**

**Course courses;**

**return courses;**

**}**

**void BinarySearchTree::addNode(Node\* node, Course courses) {**

**// if node is larger then add to left**

**if (node->courseNumber.compare(courses.courseNumber) > 0) {**

**// if no left node**

**if (node->left == nullptr) {**

**// this node becomes left**

**node->left = new Node(courses);**

**}**

**// else recurse down the left node**

**else {**

**this->addNode(node->left, courses);**

**}**

**}**

**// else**

**else {**

**// if no right node**

**if (node->right == nullptr) {**

**// this node becomes right**

**node->right = new Node(courses);**

**}**

**//else**

**else {**

**// recurse down the left node**

**this->addNode(node->right, courses);**

**}**

**}**

**}**

**void displayCourse(Course courses) {**

**cout << courses.courseNumber << ": " << courses.courseName << " | " << courses.prerequisites << endl;**

**return;**

**}**

int numPrerequisiteCourses(Tree<Course> courses) {

}

void printSampleSchedule(Tree<Course> courses) {

}

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

}

| **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) |

The worst-case run time for a hash table is O(n) and occurs when all the keys are mapped to one index, and one has to traverse through each value in that index. The worst case for a binary search tree is O(n) and occurs if the tree is skewed all to the left or right, causing the tree to have one branch equal to its height. Hash tables and binary search trees are faster than traversing linearly through a vector, but hash tables potentially have a greater space complexity than binary search trees. Therefore, I would choose a binary search tree data structure because it is faster than a linear search through a vector and as fast as a hash table with less memory utilized.