# 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 all courses

print course name

if course has prerequisites

for each prerequisite

print prerequisite

}

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) {

totalPrerequisites = Hashtable[c]

for each prerequisite p in totalPrerequisites

add prerequisites in Hashtable[p] to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Hashtable<Course> courses) {

for all key, value pair in courses

print key course name

if value has prerequisites

for each prerequisites

print prerequisites

}

void printCourseInformation(Hashtable<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 Hashtable[course]

print the prerequisite course information

}

// Tree pseudocode

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

totalPrerequisites = left and right child of Node c

for each prerequisite p in totalPrerequisites

add left and right Nodes of node p to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Tree<Course> courses) {

for all Nodes as courses

print course name

if course has left node

print left node as prerequisite

if course has right node

print right node as prerequisite

}

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

for all Nodes

if the course is the same as courseNumber

print out the node's information

if course has left node

print left node as prerequisite couse information

if course has right node

print right node as prerequisite couse information

end Function

else

if course has left node

goto left node

if course has right node

goto right node

}

//Menu

Create Schedule object to hold courses

While (choice != 9) {

Output: "Menu:"

Output: "1. Load Data Structure"

Output: "2. Print Course List"

Output: "3. Print Course"

Output: "9. Exit"

Wait for input and store choice

Switch (choice) {

Case 1: Load Course Data (fileName)

Break

Case 2: printSampleSchedule

Break

Case 3: Output "Enter course number:"

Wait for input

if (courseNumber is not found) {

output: "Course does not exist"

else printCourseInformation

}

break

}

}

Output "goodbye"

## Vector Runtime Analysis

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Initialize vector of strings contents to hold file contents** | 1 | 1 | 1 |
| **Initialize string line to hold single line** | 1 | 1 | 1 |
| **Initialize ifstream instream using fileName** | 1 | 1 | 1 |
| **Open file with instream using fileName** | 1 | 1 | 1 |
| **If instream doesn't open, output "error"** | 1 | 1 | 1 |
| **Pull line from instream until no lines left in file** | 1 | 1 | 1 |
| **Push line to back of contents** | 1 | n | n |
| **Close file using instream** | 1 | n | n |
| **Return contents** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

## Hash Table Runtime Analysis

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Initialize vector of strings contents to hold file contents** | 1 | 1 | 1 |
| **Initialize string line to hold single line** | 1 | 1 | 1 |
| **Initialize ifstream instream using fileName** | 1 | 1 | 1 |
| **Open file with instream using fileName** | 1 | 1 | 1 |
| **If instream doesn't open, output "error"** | 1 | 1 | 1 |
| **Pull line from instream until no lines left in file** | 1 | 1 | 1 |
| **Push line to back of contents** | 1 | n | n |
| **Close file using instream** | 1 | n | n |
| **Return contents** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

## Binary Tree Runtime Analysis

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Initialize vector of strings contents to hold file contents** | 1 | 1 | 1 |
| **Initialize string line to hold single line** | 1 | 1 | 1 |
| **Initialize ifstream instream using fileName** | 1 | 1 | 1 |
| **Open file with instream using fileName** | 1 | 1 | 1 |
| **If instream doesn't open, output "error"** | 1 | 1 | 1 |
| **Pull line from instream until no lines left in file** | 1 | 1 | 1 |
| **Push line to back of contents** | 1 | n | n |
| **Close file using instream** | 1 | n | n |
| **Return contents** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

1. Function OpenReadFile that takes parameters .

Created vector of the string keeps file data.

ifstream reads data and store in string variable

Add data into vector string

File close

Return file data

1. Course crs[size of vector]
2. Total course cost : O(n)
3. Total of the function = n\*1

Vector:

Advantage - Getting elements with an index.

Disadvantage - O(n) is for Insertion and deleting

Hash Table:

Advantage - Database indexing and caches are more efficient.

Disadvantage - It does not allow null values

Tree:

Advantage - Keep data in hierarchical way

Disadvantage - Instability of the data

Out of the previous data structures I would use a Hash Table because of the time complexity. Array's searching ,inserting element into array complexity is O(n). Binary tree, a searching ,inserting element into Binary tree complexity is O(log(n)). Hash Table's searching and inserting element into Hash table complexity is O(1). Also, Hash Table can be used in linear and non-linear structures.