# CS 300 Pseudocode Document

## Function Signatures

**// Vector pseudocode**

Vector createCourse(Vector<Course> courses) {

Create local vector

Csv::parser object = csv::parser(course)

For each row in passed vector

//create courses construct to hold file data

course courses

if object[I][0] && object[I][1] == null {

skip the line

Else

courses.courseNumber = object[I][0]

courses.courseName = object[I][1]

N = 3

while object[I][N] != null

courses.prerequisite = object[I][N]

++N

append to the local vector

for course in courses

Call numPrerequisiteCourses

for each prerequisite in course

If the prerequisite is != to courseNumber in courses

Remove attribute from object

Else

Print prerequisite found

Return course

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

if course number is found

for each prerequisite in course

for prerequisite course number in vector

if course has no prerequisites

print course information

else

search for prerequisite information in vector

print prerequisite course

information

Print 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

SortList(Vector<course>, int begin, int end)

Initialize midpoint, pivot and temp variable, boolean for done

midpoint = begin + (end - begin) / 2

Set pivot equal to midpoint

done = false

while (!done)

while (courses[begin].title < pivot.title)

begin++

while (pivot.title < courses[end].title)

end--

if (begin >= end)

done = true

else

temp = courses[begin]

courses[begin] = courses[end]

courses[end] = temp

begin++

end--

return end

**// Hashtable pseudocode**

populateTable(hashTable, csv path){

Create local hash table

Csv::parser object = csv::parser(course)

While (!= end of file)

If (csv[I] has at least 2 parameters)

place in table;

if (csv[I] != at least 2 parameters)

return;

int numPrerequisiteCourses(Hashtable<Course> courses) {

Int numPrerequisiteCourses = 0;

for (I=0; I < 2; ++I)

if (course-> prerequisite course != null)

++numPrerequisiteCourses;

else

return;

void printSampleSchedule(Hashtable<Course> courses)

For (I=0; I < Course.size(); ++I)

For (J=0; j < numPrerequisiteCourses(); ++j)

Return course->prerequisite courseNumber and courseName;

Return course->courseNumber and courseName;

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

For (I=0; I < Course.size(); ++I)

If (courses->courseNumber == courseNumber)

return courses->course;

For (I = 0; I < numPrerequisiteCourses();++I)

Return courses->prerequisite courses;

If (courses->courseNumber != courseNumber && I=course.size()-1)

Print “Course not found”;

SortTable(Vector<course>, int begin, int end)

Initialize midpoint, pivot and temp variable, boolean for done

midpoint = begin + (end - begin) / 2

Set pivot equal to midpoint

done = false

while (!done)

while (courses[begin].title < pivot.title)

begin++

while (pivot.title < courses[end].title)

end--

if (begin >= end)

done = true

else

temp = courses[begin]

courses[begin] = courses[end]

courses[end] = temp

begin++

end--

return end

**// Tree pseudocode**

Void CreateTree(string csvPath, BinarySearchTree bst){

Parse the file with csv parser object;

For each line in the file

If the line in the file has 2 inputs

if prerequisite exists

if prerequisites match courses in file

create courses object;

place course information in appropriate

variables, assign them to the position in

the file;

Use insert method to place them in the

tree;

Else

Create course object;

Assign appropriate information to

variables, omit non-qualifying

Prerequisites;

Use insert method to place in the tree;

Else

create courses object;

place course information in appropriate

variables, assign them to the position in the file;

Use insert method to place them in the tree;

Iterate to next line;

Else

Skip the line;

Void Insert(Tree<Course> courses)

If root is null

add node = root;

Else

call add function and pass root and course;

Void addNode(Node\* node, Tree<Course> courses){

If node course > course

If node->left = null

node->left = new Node(course)

Else

Call addNode function and recurse left nodes

Else

If node->right = null

node->right = new Node(course);

Else

Call addNode function and recurse right nodes

int numPrerequisiteCourses(Tree<Course> courses)

Create node for root course in courses

Create int variable to store numPrerequisites

While course node is not null

if course node prerequisite = 1

++numPrerequisites;

if course node prerequisite = 2

numPrerequisites = numPrerequisites+2;

else

go to the next node via in order traversal;

Go to the next node via in order traversal;

Return numPrerequisites;

void printSampleSchedule(Tree<Course> courses)

Create node and set it = root;

While node != null

if course prerequisites != null

print course and course prerequisite details;

go to the next node via in order traversal;

else

print course details;

go to the next node via in order traversal;

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

Create node and set it = root;

While node != null

print course details;

go to the next node via in order traversal;

InOrder()

This->inOrder(root)

InOrder(Node node)

If the node is not null

traverse left

output information

traverse right

**//Menu psuedocode**

Int choice = 0

While choice != menu option for exit

print menu options: load courses, print course list, print course, exit

case 1 (load courses):

call method to load courses (already defined for each data structure)

break

case 2 (print course list):

print courses in order

break

case 3 (print course):

print course information methods

break

print “Good bye”

return 0

## Example Runtime Analysis

| **Code for loading file** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create local vector** | 1 | 1 | 1 |
| **Create parser object** | 1 | 1 | 1 |
| **For each row in passed vector** | 2 | n | 2n |
| **Create local object** | 1 | 1 | 1 |
| **if the line in file does not meet requirements** | 1 | n | n |
| **Skip the line** | 1 | n | n |
| **else** | 1 | n | n |
| **Initialize course number** | 1 | 1 | 1 |
| **Initialize course name** | 1 | 1 | 1 |
| **Initialize n variable at prerequisite index** | 1 | 1 | 1 |
| **While line at index n is not null** | 1 | n | n |
| **Set the course prerequisite to the position** | 1 | n | n |
| **Iterate N value** | 1 | n | n |
| **Append to the structure** | 1 | 1 | 1 |
|  |  |  |  |
| **Total Cost** | | | 8n + 7 |
| **Runtime** | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Code for appending vector/linked list** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **Create local node** | 1 | 1 | 1 |
| **If head is null** | 1 | 1 | 1 |
| **Initialize head with new node** | 1 | 1 | 1 |
| **Initialize tail with new node** | 1 | 1 | 1 |
| **else** | 1 | 1 | 1 |
| **Tail's next pointer is the new node** | 1 | 1 | 1 |
| **Tail is now the new node** | 1 | 1 | 1 |
| **Increase the size of the list** | 1 | 1 | 1 |
| **Total Cost** | | | 8 |
| **Runtime** | | | O(1) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Code for appending hash table** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **Assign key to hashed value** | 1 | 1 | 1 |
| **Create node at key value** | 1 | 1 | 1 |
| **If node is null** | 1 | 1 | 1 |
| **Create new node at key** | 1 | 1 | 1 |
| **Insert the node at the key** | 1 | 1 | 1 |
| **else** | 1 | 1 | 1 |
| **If node key is empty** | 1 | 1 | 1 |
| **Set the key to key** | 1 | 1 | 1 |
| **Set the course to course** | 1 | 1 | 1 |
| **Set the next node to null** | 1 | 1 | 1 |
| **else** | 1 | 1 | 1 |
| **While the next pointer is not null** | 1 | n | n |
| **Loop to the next node** | 1 | n | n |
| **Assign the next pointer to the new node** | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 12 |
| **Runtime** | | | O(N) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Code for appending tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **If the root is null** | 1 | 1 | 1 |
| **Root is the new node** | 1 | 1 | 1 |
| **else** | 1 | 1 | 1 |
| **Call addNode function and assign the course starting from the root** | 1 | 1 | 1 |
| **Total Cost** | | | 4 |
| **Runtime** | | | O(1) |

The three structures presented each have their advantages and disadvantages. The linked list consists of a series of nodes, each pointing to the next element in the list. Traversal can be expensive as far as memory is concerned because all the elements are linear and cannot be randomly accessed. On the other hand, the size of a linked list is dynamic and can be changed, making insertions and deletions easier than with other data structures. For example, in the chart above, the run time for appending an element to a linked list is O(1). This is because the algorithm locates the tail node directly and assigns the element to the end instead of traversing the whole list. A hash table involves creating a hash key from a piece of data and the size of the table, then storing that information in a bucket. A hash table can be easily and quickly accessed because the location of data is predictable. However, insertions may become difficult due to the number of collisions that are likely to happen as the table gets larger. Additionally, the size of the hash table is not dynamic and must be changed manually if more data is to be stored. Collisions in a hash table can occupy a lot of time if there is a lot of data already being stored in the buckets. For example, the run time for a hash table is O(N). This is because the algorithm must make many comparisons and traverse through the table until an empty bucket is found. Binary trees are data structures that follow the same structure as a tree. There is a root node at the beginning, inner nodes and leaf nodes at the end of a branch. Binary trees sort data in a vector by comparing the value presented to the existing values in a tree. It then sorts the data by determining whether that value is larger or smaller than the current value. Due to the binary nature of this structure, the processing times are very low because the computer only has to make 1 comparison. Additionally, since the data is already sorted, the search for a node can be quicker than other structures, such as a linked list. However, once a tree becomes larger the processing times can become longer. Searching for a node in a large tree can take longer than other structures such as hash tables because the program must start the search at the root. An append method for a binary tree has a run time of O(1). This is because the algorithm only has to make one comparison, and then it can keep moving down the tree until a space is found.

I intend to use a binary tree in my program. Binary trees tend to be one of the faster algorithms when appending an element, as proved by the charts above. They are faster than hash tables, which have a processing time of O(N) and comparable to a linked list processing time of O(1). Additionally, the traversal of a binary tree is faster than a linked list, since the algorithm does not have to review every node. Because of this, I believe a binary tree will be the quickest and most efficient way to store the course information required by the project.