# CS 300 Pseudocode Document

## 

## Vector (Module 3)

// Course object data structure

Course {

string number

string name

Vector<string> prerequisites

Course(number, name, prerequisites) {

this.number = number

this.name = name

this.prerequisites = prerequisites

}

void printCourse {

print course number, name, and prerequisites

}

}

// File reading

Vector<Course> readFile(string filePath) {

File file

try

file = open filePath

catch

throw “File could not be opened”

Vector<Course> courses

Set<string> prereqCourses

for each line in file {

lineSplit = split line on ‘,’

// ensure at least two parameters on each line

if lineSplit.size < 2 {

throw “Invalid course”

}

number = lineSplit[0]

name = lineSplit[1]

prereqs = slice lineSplit from 2 to end

newCourse = new Course(number, name, prereqs)

add newCourse to courses

for each prereq in prereqs {

add prereq to prereqCourses

}

}

// Validate that there is no prereq not in courses

for each prereq in prereqCourses {

valid = false

for each course in courses {

if course.name = prereq {

valid = true

}

}

if !valid {

throw “Prerequisite not found in courses”

}

}

return courses

}

// Recursively print course and all its prerequisites

void printCourse (Vector<Course> courses, String courseNumber) {

for each course in courses {

if course.number = courseNumber {

course.printCourse()

for each prereq of course.prerequisites {

printCourse(courses, prereq)

}

}

}

}

## Hashtable (Module 4)

// Course object data structure

Course {

string number

string name

Vector<string> prerequisites

Course(number, name, prerequisites) {

this.number = number

this.name = name

this.prerequisites = prerequisites

}

void printCourse {

print course number, name, and prerequisites

}

}

// HashTable structure

class HashTable {

vector<Course\*> courses

int tableSize = 200

Course\* EMPTY\_SINCE\_START

Course\* EMPTY\_AFTER\_REMOVAL

HashTable(size) {

tableSize = size

courses.resize(tableSize)

}

private hash(key) {

return key % tableSize

}

public insert(Course course) {

key = course.number

bucket = hash(key)

bucketsProbed = 0

// Linearly probe until empty bucket is found

while (bucketsProbed < tableSize) {

// Insert item in next empty bucket

if (courses[bucket] is empty) {

courses[bucket] = course

return

}

// Increment bucket index

bucket = hash(bucket + 1)

++bucketsProbed

}

}

public printAll() {

for (int i = 0; i < tableSize; ++i) {

course = courses[i];

if (course is not empty) {

print course information

}

}

}

public remove(courseToRemove) {

key = courseToRemove.number

bucket = hash(key)

bucketsProbed = 0

// Linearly probe until empty bucket is found

while (courses[bucket] is not EMPTY\_SINCE\_START and bucketsProbed < tableSize) {

course = courses[bucket]

// Insert item in next empty bucket

if (course is not empty and course.number = key) {

courses[bucket] = EMPTY\_AFTER\_REMOVAL

return

}

// Increment bucket index

bucket = hash(bucket + 1)

++bucketsProbed

}

}

public search(key) {

bucket = hash(key)

bucketsProbed = 0

// Linearly probe until empty bucket is found

while (courses[bucket] is not EMPTY\_SINCE\_START and bucketsProbed < tableSize) {

course = courses[bucket]

// Insert item in next empty bucket

if (course is not empty and course.key = key) {

return course

}

// Increment bucket index

bucket = hash(bucket + 1)

++bucketsProbed

}

}

}

// File reading

Vector<Course> readFile(string filePath) {

File file

try

file = open filePath

catch

throw “File could not be opened”

HashMap<Course> courseMap

Set<string> prereqCourses

for each line in file {

lineSplit = split line on ‘,’

// ensure at least two parameters on each line

if lineSplit.size < 2 {

throw “Invalid course”

}

number = lineSplit[0]

name = lineSplit[1]

prereqs = slice lineSplit from 2 to end

newCourse = new Course(number, name, prereqs)

courseMap.insert(newCourse)

for each prereq in prereqs {

add prereq to prereqCourses

}

}

// Validate that there is no prereq not in courses

for each prereq in prereqCourses {

valid = false

for each course in courses {

if course.name = prereq {

valid = true

}

}

if !valid {

throw “Prerequisite not found in courses”

}

}

return courseMap

}

// Recursively print course and all its prerequisites

void printCourse (HashMap<Course> courseMap, int courseNumber) {

Course course = courseMap.search(courseNumber)

for prereq of course.prereqruisites {

printCourse(prereq)

}

}

## Binary Search Tree (Module 5)

// Course object data structure

Course {

string number

string name

Vector<string> prerequisites

Course(number, name, prerequisites) {

this.number = number

this.name = name

this.prerequisites = prerequisites

}

void printCourse {

print course number, name, and prerequisites

}

}

// Node object data structure

struct Node {

Course course

Node\* left

Node\* right

// default constructor

Node() {

left = null

right = null

}

// initialize with a course

Node(Course course) : Node() {

this.course = course

}

};

// Binary Search Tree structure

class BinarySearchTree {

Node root

BST() {

root = null

}

search(name) {

Node current = root

while (current is not null) {

if (current courseName is name) {

// Course found so return it

return current.course

} else if (name < current.name) {

current = current.left

} else {

current = current.right

}

}

}

addCourse(course) {

node = Node(course)

if (root is null) {

root = node

node.left = null

node.right = null

} else {

Node current = root

while (current is not null) {

if (node.course.name < current.course.name) {

if (current.left is null) {

current.left = node

current = null

} else {

current = current.left

}

} else {

if (current.right is null) {

current.right = node

current = null

} else {

current = current.right

}

}

}

node.left = null

node.right = null

}

}

}

// File reading

BinarySearchTree readFile(string filePath) {

File file

try

file = open filePath

catch

throw “File could not be opened”

BinarySearchTree courseTree

Set<string> prereqCourses

for each line in file {

lineSplit = split line on ‘,’

// ensure at least two parameters on each line

if lineSplit.size < 2 {

throw “Invalid course”

}

number = lineSplit[0]

name = lineSplit[1]

prereqs = slice lineSplit from 2 to end

newCourse = new Course(number, name, prereqs)

bst.addCourse(newCourse)

for each prereq in prereqs {

add prereq to prereqCourses

}

}

// Validate that there is no prereq not in courses

for each prereq in prereqCourses {

valid = false

for each course in courses {

if course.name = prereq {

valid = true

}

}

if !valid {

throw “Prerequisite not found in courses”

}

}

return courseTree

}

void printCourse (BinarySearchTree courseTree, Course course) {

course.printCourse()

for prereqName of course.prerequisites {

Course prereqCourse = courseTree.search(prereqName)

if (prereqCourse is not null) {

printCourse(courseTree, prereqCourse)

}

}

}

## Menu Pseudocode

// Course object data structure

Course {

string number

string name

Vector<string> prerequisites

Course(number, name, prerequisites) {

this.number = number

this.name = name

this.prerequisites = prerequisites

}

void printCourse {

print course number, name, and prerequisites

}

}

void menu () {

// This could be either vector, hash, or bst

dst = new DataStructure()

int choice = 0

while (choice is not 9) {

output “Menu”

output “1. Load courses from file”

output “2. Print course list”

output “3. Print course”

output “9. Exit”

input choice

// Load data from file into data structure

if (choice is 1) {

File file

try {

path = “./data.csv”

file = open file(path)

}

catch (e) {

print error and exit

}

for each line in file {

lineSplit = split line on ‘,’

if lineSplit.size < 2 {

throw “Invalid course”

}

number = lineSplit[0]

name = lineSplit[1]

prereqs = slice lineSplit from 2 to end

newCourse = new Course(number, name, prereqs)

dst.addCourse(newCourse)

}

}

// Print all course list

if (choice is 2) {

for each course in dst {

output course.number

output course.name

}

}

// Print course

if (choice is 3) {

output “Enter course number”

input courseNum

courseMatch = dst.search(courseNum)

if (courseMatch is not null) {

courseMatch.printCourse()

}

}

// Exit

if (choice is 9) {

output “goodbye”

return

}

// Invalid option

else {

output “Invalid option entered”

}

}

}

## Print list of alphabetical courses

// Vector

def printList = (Vector<Course> courses) {

swapped = false

while !swapped {

for i = 1; i < courses.length – 1; i++ {

currentCourse = courses i]

prevCourse = courses[i – 1];

if (currentCourse.name < prevCourse.name) {

swap currentCourse and prevCourse

swapped = true

print course name

print course number

} else {

swapped = false

}

}

}

}

// Hashmap

def printList (HashMap<Course> courseMap) {

swapped = false

prevCourse = first course in courseMap

while !swapped {

for course in courseMap {

if (course.name < prevCourse.name) {

swap currentCourse and prevCourse

swapped = true

print course name

print course number

} else {

swapped = false

}

prevCourse = course

}

}

}

// Binary Search Tree

def printList (BinarySearchTree<Course> courseTree) {

// Because it is a binary search tree, it is already sorted

// we just need to traverse it recursively

def printTree(node) {

if node is null return

printTree(node.left)

output node.course.number

output node.course.name

printTree(node.right)

}

// Start recursive inner function

printTree(courseTree.root)

}

## Vector Runtime Analysis

The vector code loops over every line in the file and adds it to the list as well as adding the prerequisites to a list. Adding an item to the end of a vector is O(1). The read file and add items is O(n) + 1 and the validate prerequisites is O(n^2).

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| for each line in file | 1 | n | n |
| lineSplit = split line on | 1 | 1 | 1 |
| if lineSplit.size < 2 | 1 | 1 | 1 |
| ...other O(1) operations | 1 | 1 | 1 |
| add newCourse to courses | 1 | 1 | 1 |
| for each prereq in prereqs | 1 | n | n |
| for each prereq in prereqCourses | 1 | n | n |
| for each course in courses | 1 | n | n |
| **Total Cost** | | | n + 1 + n\*n |
| **Runtime** | | | O(n^2) |

## Hashtable Runtime Analysis

The hashtable has the same runtime complexity as the vector because we are doing the same operation of looping over the file and then looping over the prerequisites while looping over the courses. So the entire read file operation will be n^2. The process of inserting an item into a hashmap is usually o(1) but in worse-case scenario it can take multiple tries to insert, but this does not qualify as o(n). So the read file and add items is O(n) + 1 and the validate prerequisites is O(n^2).

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| for each line in file | 1 | n | n |
| key = course.number | 1 | 1 | 1 |
| bucket = hash(key) | 1 | 1 | 1 |
| while (bucketsProbed < tableSize) | 1 | 1 | 1 |
| if (courses[bucket] is empty) | 1 | 1 | 1 |
| bucket = hash(bucket + 1) | 1 | 1 | 1 |
| for each prereq in prereqCourses | 1 | n | n |
| for each course in courses | 1 | n | n |
| **Total Cost** | | | n + 1 + n\*n |
| **Runtime** | | | O(n^2) |

## Tree Runtime Analysis

The tree has the same runtime complexity as the vector and hashtable because we are doing the same operation of looping over the file and then looping over the prerequisites while looping over the courses. So the entire read file operation will be n^2. The process of inserting an item into a tree is worse than a vector or hashmap because we traverse the tree to add it which is O(n) inside the O(n). So the read file and add items is O(n^2) and the validate prerequisites is O(n^2) which is O(n^2)+O(n^2) or just O(n^2).

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| for each line in file | 1 | n | n |
| if (root is null) | 1 | 1 | 1 |
| root = node  node.left = null  node.right = null | 1 | 1 | 1 |
| else | 1 | 1 | 1 |
| while (current is not null) | n | n | n |
| Node assignments | 1 | 1 | 1 |
| for each prereq in prereqCourses | 1 | n | n |
| for each course in courses | 1 | n | n |
| **Total Cost** | | | n\*n + 1 + n\*n |
| **Runtime** | | | O(n^2) |

## Advantages and Disadvantages

Each of the three data structures have their own advantages and disadvantages. For vectors, the major advantages are that items can be read from the vector in constant time, items can be added to the end in constant time, and they are simple to implement. The major disadvantages for vectors are that it is slower to insert and delete elements in random access and searching a vector for element requires linear time. For hashtables, the major advantage is near constant time searching, insert, and delete. The disadvantages to hashtables are the difficulty to implement, collisions if a poor hash function is used, and getting items in order. Finally, the advantages for binary search trees are that they are ordered making sorting unnecessary; insert, search, and delete are in log n; and they are simple to implement. The disadvantage to binary search trees is that they have slightly worse memory performance due to additional pointers and constructs needed for the structure itself.

## Recommendation

Based on the analysis above as well as my own practice implementing these various structures throughout the course, I would recommend using a binary search tree for this project. I found that this data structure was more straightforward to create than a hashtable, and it has more benefits than a vector. To recap the analysis above, binary search trees have very good performance (log n) for common operations such as searching, inserting, and deleting. Additionally, because binary search trees are ordered, there is no need to sort items later which was one of the requirements for the project.