# CS 300 Pseudocode Document

Nicholas Ciesla CS300 H7201 22EW4

// Vector pseudocode

Class Course {

Public members

Course()

Course –|

v

(string courseNumber, string courseName, Vector<String> prerequisites)

Getters/setters for all private members

Private members

String courseNumber

String courseName

Vector<String> prerequisites

}

Main(){

Vector<Course> courses

int choice = 0

file = get filepath from user

while choice is not 9

print “Menu:”

print “1. Load Courses”

print “2. Display Course List”

print “3. Display Course Info”

print “9. Exit”

Switch on choice

Case 1:

Courses = readCourses(file)

break

Case 2:

sortCourses(address of courses)

printCourseList(courses)

break

Case 3:

courseNumber = Get course number from user

printCourseInformation(courses, courseNumber)

break

print “Goodbye”

}

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**

}

void printCourseList(Vector<Course> courses) {

**for all courses**

**print out the course information**

}

void sortCourses(Vector<Course> \*courses){

for i = 1, to size of courses, i++

for j = 0, to size of courses – 1 – i; j++

if courses at j courseNum > courses at j + 1 courseNum

swap courses at j and courses at j + 1

}

Vector<Course> readCourses(file){

open file

declare vector<Course> courses

for all lines in file

read line

declare vector<string> pieces

declare vector<string> preReq

while line length is greater than 0

if a comma exists in line

push back pieces substring before comma

line = substring after comma

else

push back pieces rest of line

if size of pieces is not less than two

for all indexes of pieces greater than 1

push back preReq

courseNumber = pieces at 0

courseName = pieces at 1

else

print message regarding incorrect formatting

continue loop

Course course(courseNumber, courseName, preReq)

addCourse(course)

return courses

}

Int addCourse(Course course, vector<Course> \*courses){

For all courses in course

If course.getcourseNumber == courseNumber

Return -1 on duplicate value

Push back courses course

Return 1 on success

}

// Hashtable pseudocode

Class Course {

Public members

Course()

Course –|

v

(string courseNumber, string courseName, Vector<String> prerequisites)

Getters/setters for all private members

Private members

String courseNumber

String courseName

Vector<String> prerequisites

}

Main(){

Hashtable<Course> courses

int choice = 0

file = get filepath from user

while choice is not 9

print “Menu:”

print “1. Load Courses”

print “2. Display Course List”

print “3. Display Course Info”

print “9. Exit”

Switch on choice

Case 1:

Courses = readCourses(file)

break

Case 2:

printAllCourses(courses)

break

Case 3:

courseNumber = Get course number from user

printCourseInformation(courses, courseNumber)

break

print “Goodbye”

}

int hash(string courseNumber, int N){

int hashVal

coursId = Convert last three chars of course Number to int

hashVal = coursId % N

return hashVal

}

Course hashSearch(Hashtable<Course> courses, String courseNumber){

N = size of courses

pos = hash(courseNumber, N)

for i equal to zero; while i < N; increment i

if courses at pos has been null since beginning

return null

if coursesNumber of course at pos is equal to courseNumber

return courses at pos

else

pos = (pos + 1) & N

}

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

Course temp

Course c = hashSearch(courseNumber)

Print c’s information

**for each prerequisite of the course**

temp = hashSearch(prerequisite courseNumber)

print temp’s information

}

void printAllCourses(Hashtable<Course> courses) {

Vector<course> courseVector

for each entry in courses

if not null

append course to vector

sortCourses(courseVector)

for each entry in courseVector

print course info

}

Hashtable<Course> readCourses(file){

open file

declare Hashtable<Course> courses

for all lines in file

read line

declare vector<string> pieces

declare vector<string> preReq

while line length is greater than 0

if a comma exists in line

push back pieces substring before comma

line = substring after comma

else

push back pieces rest of line

if size of pieces is not less than two

for all indexes of pieces greater than 1

push back preReq

courseNumber = pieces at 0

courseName = pieces at 1

else

print message regarding incorrect formatting

continue loop

Course course(courseNumber, courseName, preReq)

addCourse(course)

return courses

}

void sortCourses(Vector<Course> \*courses){

for i = 1, to size of courses, i++

for j = 0, to size of courses – 1 – i; j++

if courses at j courseNum > courses at j + 1 courseNum

swap courses at j and courses at j + 1

}

Int addCourse(Course course, Hashtable<Course> \*courses){

Pos = hash(course courseNumber)

for i equal to zero; while i < N; increment i

if courses at pos is null

courses at pos = course

return 1

else if courses at pos courseNumber = course courseNumber

return -1 on duplicate value

else

pos = (pos + 1) & N

}

// Tree pseudocode

Class Course {

Public members

Course()

Course –|

v

(string courseNumber, string courseName, Vector<String> prerequisites)

Getters/setters for all private members

Private members

String courseNumber

String courseName

Vector<String> prerequisites

}

Main(){

Tree<Course> courses

int choice = 0

file = get filepath from user

while choice is not 9

print “Menu:”

print “1. Load Courses”

print “2. Display Course List”

print “3. Display Course Info”

print “9. Exit”

Switch on choice

Case 1:

Courses = readCourses(file)

break

Case 2:

printCourseTree(courses.Root)

break

Case 3:

courseNumber = Get course number from user

printCourseInformation(courses, courseNumber)

break

print “Goodbye”

}

addCourse(course){

create Node node with course as data

create Node curr for traversal

if Tree->root is null

set root = node

else

while curr is not a null pointer

if course.courseNumber < curr->course.courseNumber

if curr->left is null

set curr->left to node

else

set curr to curr->left

else

if curr->right is null

set curr->right to node

else

set curr to curr->right

}

Course Search(courseNumber){

Create Node curr for traversal

Set curr to root

While curr is not a null pointer

If curr->course.courseNumber = courseNumber

Return curr->course

Else if courseNumber < curr->course.courseNumber

Advance curr left

Else

Advance curr right

Return null if loop exits without returning

}

void printCourseTree(Node \*node) {

if node is a null pointer

return

printCourseTree(node->left)

print course information

printCourseTree(node->right)

}

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

Course temp

Course c = Search(courseNumber)

Print c’s information

**for each prerequisite of the course**

temp = Search(prerequisite courseNumber)

print temp’s information

}

Tree<Course> readCourses(file){

open file

declare Hashtable<Course> courses

for all lines in file

read line

declare vector<string> pieces

declare vector<string> preReq

while line length is greater than 0

if a comma exists in line

push back pieces substring before comma

line = substring after comma

else

push back pieces rest of line

if size of pieces is not less than two

for all indexes of pieces greater than 1

push back preReq

courseNumber = pieces at 0

courseName = pieces at 1

else

print message regarding incorrect formatting

continue loop

Course course(courseNumber, courseName, preReq)

addCourse(course)

return courses

}

**File Reading (same for every data structure)**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| open file | 1 | 1 | 1 |
| declare /DATA STRUCTURE/ courses | 1 | 1 | 1 |
| for all lines in file | 1 | n | n |
| read line | 1 | n | n |
| declare vector<string> pieces | 1 | n | n |
| declare vector<string> preReq | 1 | n | n |
| while line length is greater than 0 | 1 | 4n | 4n |
| if a comma exists in line | 1 | 4n | 4n |
| push back pieces substring before comma | 1 | 4n | 4n |
| line = substring after comma | 1 | 4n | 4n |
| else | 1 | n | n |
| push back pieces rest of line | 1 | n | n |
| if size of pieces is not less than two | 1 | n | n |
| for all indexes of pieces greater than 1 | 1 | 2n | 2n |
| push back preReq | 1 | 2n | 2n |
| courseNumber = pieces at 0 | 1 | n | n |
| courseName = pieces at 1 | 1 | n | n |
| Else | 1 | n | n |
| print message regarding incorrect formatting | 1 | n | n |
| continue loop | 1 | n | n |
| Course course(courseNumber, courseName, preReq) | 1 | n | n |
| addCourse(course) | DIFFERENT FOR EVERY DATA STRUCTURE | | |
| return courses | 1 | 1 | 1 |
| **Total Cost (Without addCourse)** | | | 33n + 3 |
| **Runtime (Without addCourse)** | | | O(n) |

**Vector** **addCourse Analysis**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| For all courses in course | 1 | n | n |
| If course.getcourseNumber == courseNumber | 1 | 1 | 1 |
| Return -1 on duplicate value | 1 | 1 | 1 |
| Push back courses course | 1 | 1 | 1 |
| Return 1 on success | 1 | 1 | 1 |
| **Total Cost** | | | n + 4 |
| **Runtime** | | | O(n) |
| **Total Runtime with readCourse** | | | O(n^2) |

**Advantages**

Using a vector in this application is easy to understand from a non programmer’s perspective. It would be rather easy to explain the inner workings of this data structure to the leperson. It also does not use an excessive amount of memory. However, that is where the benefits of using a vector end.

**Disadvantages**

The main disadvantage of using a vector is time complexity. Almost every operation on a vector has a worst case rating of o(n). That means, when two or more operations are needed in conjunction, time complexity can get as bad as o(n^2) as we see in the file reading/insert case. Annother operation outside of this example that will require n^2 operations is printing the list of courses. Because it must be in alphanumeric order, we must sort the vector beforehand. There are faster sorting algorithings, but I have chosen to use bubble sort for simplicity.

## Hash Table addCourse Analysis

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Pos = hash(course courseNumber) | 1 | 1 | 1 |
| for i equal to zero; while i < N; increment i | 1 | n | n |
| **if courses at pos is null** | 1 | n | n |
| courses at pos = course | 1 | 1 | 1 |
| return 1 | 1 | 1 | 1 |
| else if courses at pos courseNumber = course courseNumber | 1 | n | n |
| return -1 on duplicate value | 1 | 1 | 1 |
| else | 1 | n | n |
| pos = (pos + 1) & N | 1 | n | n |
| **Total Cost** | | | 3n + 4 |
| **Runtime** | | | O(n) |
| **Total Runtime with readCourse** | | | O(n^2) |

**Advantages**

A hash table has many advantages. Number one being efficiency. The worst case for a hash table insertion is o(n^2), but the best is o(1). The best case is o(1) for many different data structures, but for hastables, this best case happens much more frequently because of the hash function. If implemented correctly, the hash table can also be sorted on insertion. The pseudo code above uses linear probing to place values which has a worst case of o(n). However, if we use the course number’s numeric portion as a hash and use chaining in case there are two courses with the same numeric section, we can achieve o(1) insertions ever time.

**Disadvantages**

This disadvantage to the described hash table implementation would be that it would require much more memory than we *really* need to store the course information. We would need a vector with 500 or 600 nodes to effectively use the course number as a hash. This would greatly the increase the required time to traverse the table in order as well. This could be improved with a more complex hash function, but we would lose the automatic sorting.

**Binary Search Tree addCourse Analysis**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| create Node node with course as data | 1 | 1 | 1 |
| create Node curr for traversal | 1 | 1 | 1 |
| **if Tree->root is null** | 1 | 1 | 1 |
| set root = node | 1 | 1 | 1 |
| else | 1 | 1 | 1 |
| while curr is not a null pointer | 1 | log2n | log2n |
| if course.courseNumber < curr->course.courseNumber | 1 | log2n | log2n |
| if curr->left is null | 1 | log2n | log2n |
| set curr->left to node | 1 | 1 | 1 |
| else | 1 | log2n | log2n |
| set curr to curr->left | 1 | log2n | log2n |
| else | 1 | log2n | log2n |
| if curr->right is null | 1 | log2n | log2n |
| set curr->right to node | 1 | 1 | 1 |
| else | 1 | log2n | log2n |
| set curr to curr->right | 1 | log2n | log2n |
| **Total Cost** | | | 9(log2n) + 7 |
| **Runtime** | | | O(log2n) |
| **Total Runtime with readCourse** | | | O(nlog2n) |

**Advantages**

While being the most un-inuitive data structure, the Binary Search Tree is the most consistently fast option. It also uses only slightly more memory than abosloutley necessary (pointers). Also, most algorithms to insert, traverse, and remove are much simpler than one would expect from looking at it.

**Disadvantages**

I would say the only real disadvantage to using the Binary search tree would be the advanced nature of its concept. If someone not very experienced had to edit the code for any reason, they may find themselves lost. However, this is a computer science department, so I doubt that would cause any issue.

**RECOMMENDATION**

After some consideration, I have decided that the best data source for this application is the Binary Search Tree. It is the most effecenint in memory, consistent in its effeceiency across all operations, and rather simple to implement. The o(1) insert time of the Hash Table was tempting, but the possibility of ineffeciney in other regards and the tradeoff in memory you would have to make to achieve that efficiency is not necessarily worth it in this case.

Going with consistency over the chance of extreme speed in specific instances is the better tradeoff. This will cause the program to be more predictable and space efficient.