# CS 300 Project One Pseudocode Document

## Pseudocode from previous assignments

**// Vector pseudocode**

int loadCourses(Vector<Course> courses, String filename) {

If file exists on disc and is readable

For each line in file

Read the Line

Call createCourseObject(lineString)

add course id to a hash for easy lookup

add course object to courses vector

// Read of all lines complete

For each course in vector

For each prereq in course

does prereq exist in courses hash

If not, throw exception

Else

throw exception and return

}

Course createCourseObject(String lineFromFile) {

Break lineFromFile by commas

Create a new Course object

Assign first part of lineFromFile to course.courseId

Assign second part of lineFromFile to course.courseName

If either of the above does not exist, throw an exception

Assign any other comma separated strings in file to an array of prereqs in the course object

}

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

for each course in courses

if course.courseId = courseNumber

print details about the course

for each prerequisite in course

Recursively call printCourseInformation

}

**// Hashtable pseudocode**

int loadCourses(HashTable courses, String filename) {

If file exists on disc and is readable

For each line in file

Read the Line

Call createCourseObject(lineString)

// Read of all lines complete

For each course in courses HashTable

For each prereq in course

does prereq exist in temporary hash

If not, throw exception

Else

throw exception and return

}

// This is literally the same function as in Module 3.

Course createCourseObject(String lineFromFile) {

Break lineFromFile by commas

Create a new Course object

Assign first part of lineFromFile to course.courseId

Assign second part of lineFromFile to course.courseName

If either of the above does not exist, throw an exception

Assign any other comma separated strings in file to an array of prereqs in the course object

// add course id to a tempory hash for easy lookup

Identify bucket to insert by hashing courseId

If location does not have a valid node in the backing vector, replace node

Else

Iterate to end of linked list in the hashtable bucket and insert current course as a node

add course object to courses HashTable

}

void printCourseInformation(HashTable courses, String courseNumber) {

Identify the bucket by hashing the courseNumber

Find the Node at the bucket location in the hashTable

If node is not a course node

return

For each valid node

if node.course.courseId = courseNumber

print details about the course

for each prerequisite in course

Recursively call printCourseInformation

Return since you found a match

}

**// BST pseudocode**

int loadCourses(BST<Course> courses, String filename) {

If file exists on disc and is readable

For each line in file

Read the Line

Call createCourseObject(lineString)

// Read of all lines complete

For each course in inorder(courses BST)

For each prereq in course

does prereq exist in temporary hash

If not, throw exception

Else

throw exception and return

}

// This is literally the same function as in Module 3.

Course createCourseObject(String lineFromFile) {

Break lineFromFile by commas

Create a new Course Node

Assign first part of lineFromFile to course.courseId

Assign second part of lineFromFile to course.courseName

If either of the above does not exist, throw an exception

Assign any other comma separated strings in file to an array of prereqs in the course object

// add course id to a tempory hash for easy lookup

Identify bucket to insert by hashing courseId

If location does not have a valid node in the backing vector, replace node

Else

Iterate to end of linked list in the hashtable bucket and insert current course as a node

add course object to courses BST

}

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

Traverse the BST to find the Course

If course is not found

return

print details about the course

for each prerequisite in course

Recursively call printCourseInformation

## Pseudocode for Menu

void printMenu() {

filename = “./courseData” // default filename

Vector<Course> vCourses;

HashTable hCourses;

BST<Course> bCourses;

cin << input

switch input

Case Load Data Structure:

Get file name from input

if input is empty, use default filename

loadCourses(vCourses, filename)

loadCourses(hCourses, filename)

loadCourses(bCourses, filename)

break

Case Print Course List:

printCourseList(vCourses)

printCourseList(hCourses)

printCourseList(bCourses)

Break

Case Print Course:

Get course number from input

printCourseInformation(vCourses, courseId)

printCourseInformation(hCourses, courseId)

printCourseInformation(bCourses, courseId)

break

Any other Input

exit(0)

}

## Print Courses in Alphanumeric Order

void printCourses(Vector<Course> courses)

// Insertion sort

for i=0; i<courses.length-1; i++){

for(j=i+1; j<courses.length; j++){

if courses[i] > courses[j]

temp = courses[i]

courses[i] = courses[j]

courses[j] = temp

for i=0; i<courses.length; i++){

printCourseInformation(courses[i]);

}

}

void printCourses(HashTable<Course> courses)

previousPrinted = 0 //some arbitrary value which cannot be a course number

while true { //infinite loop

minValue = previousPrinted

for each bucket in courses as currentCourse:

if currentCourse > previousPrinted and currentCourse < minValue

minValue = currentCourse

if minValue > previousPrinted // found the next smallest value

printCourseInformation(courses, minValue)

previousPrinted = minValue

else

exit // already printed largest value in previous iteration

}

}

void printCourses(BST<Course> courses){

// Print Inorder

inOrder(courses, courses.root);

}

void inOrder(courses, nodeptr){

return if nodeptr is null

inOrder(nodeptr.left)

printCourseInformation(nodeptr)

inOrder(nodeptr.right)

## Evaluation of data structures

### Vector

Based on the analysis below, insertion of data into a vector based store is cheap. Using a hashtable for verification makes the insertion cost of new items approximately O(1). There are a few details worth noting

1. Without the hashtable verification optimization, the cost of insertion if O(n^2) *(Sorry, I did’t realize while writing out the original code that I needn’t optimize)*
2. Vector insertion amortizes to O(1) but initial data structure resizes may be expensive
3. Search and ordering of data stored in vectors is expensive and an optimized algorithm might give at best nlog n access. (though with some creativity we could use radix sort and bring it down to nk)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **Create Course Object Method** |  |  | **6** |
| Break lineFromFile by commas | 1 | 1 | 1 |
| Create a new Course object | 1 | 1 | 1 |
| Assign first part of lineFromFile to course.courseId  Assign second part of lineFromFile to course.courseName | 2 | 1 | 2 |
| If either of the above does not exist, throw an exception | 1 | 1 | 1 |
| Assign any other comma separated strings in file to an array of prereqs in the course object | 1 | 1 | 1 |
| **loadCourses Method** |  |  |  |
| If file exists on disc and is readable | 1 | 1 | 1 |
| For each line in file | 1 | n | n |
| Read the Line | 1 | n | 1 |
| Call createCourseObject(lineString) | 6 | n | 6n |
| add course id to a hash for easy lookup  **Append course object to courses vector** | 2 | n | 2n |
| For each course in vector | 1 | n | n |
| For each prereq in course | k | kn | kn |
| does prereq exist in courses hash | 1 | kn | kn |
| If not, throw exception | 1 | kn | kn |
| Else  throw exception and return | 1 | 1 | 1 |
| **Total Cost**  \* *k is max number of prereqs* | | | (10+3k)n + 3 |
| **Runtime** | | | O(n) |

### Hashtable

Based on the analysis below, insertion of data into a hashtable based store is cheap. There are a few details worth noting

1. I have assumed a fairly large hashtable with minimal conflicts. Most sdk provided hashtables have minimum collisions due to highly efficient hashing functions. The complexity increases to n^2 if the assumption is untrue
2. Ordering of data stored in hashtables is expensive. I ended up doing an n^2 loop to identify the next large object each outer loop. The other option to use another data structure to hold the sorted data.
3. Search in hashtables when searching with the key is cheap. In fact it is called a lookup.

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **Create Course Object Method (Same as initial table)** |  |  | **6** |
| **loadCourses Method** |  |  |  |
| If file exists on disc and is readable | 1 | 1 | 1 |
| For each line in file | 1 | n | n |
| Read the Line | 1 | n | 1 |
| Call createCourseObject(lineString) | 6 | n | 6n |
| add course id to a hash for easy lookup  **add course object to courses hashtable** | 2 | n | 2n |
| For each course in vector | 1 | n | n |
| For each prereq in course | k | kn | kn |
| does prereq exist in courses hash | 1 | kn | kn |
| If not, throw exception | 1 | kn | kn |
| Else  throw exception and return | 1 | 1 | 1 |
| **Total Cost**  \* *k is max number of prereqs* | | | (10+3k)n + 3 |
| **Runtime** | | | O(n) |

### BST

Based on the analysis below, insertion of data into a BST based store is expensive. There are a few details worth noting

1. I am assuming worst cases where the courses were already sorted. It would create a one sided tree
2. Needless to say sorting (BSTs are already sorted) and search (binary search) is cheap.

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **Create Course Object Method (Same as initial table)** |  |  | **6** |
| **loadCourses Method** |  |  |  |
| If file exists on disc and is readable | 1 | 1 | 1 |
| For each line in file | 1 | n | n |
| Read the Line | 1 | n | 1 |
| Call createCourseObject(lineString) | 6 | n | 6n |
| add course id to a hash for easy lookup  **add course object to courses BST** | 1 + n | n | n^2 + n |
| For each course in vector | 1 | n | n |
| For each prereq in course | k | kn | kn |
| does prereq exist in courses hash | 1 | kn | kn |
| If not, throw exception | 1 | kn | kn |
| Else  throw exception and return | 1 | 1 | 1 |
| **Total Cost**  \* *k is max number of prereqs* | | | n^2 + (9+3k)n + 3 |
| **Runtime** | | | O(n^2) |

## Final Recommendation

I would recommend using a mix of data structure to optimize performance at the cost of storage.

1. Use a hashtable for easy lookup (O(1))
2. Use a BST for ordered list of courses

If that is not a possibility (maybe because the system is operating on an embedded device with limited memory), use a BST. It provides the best compromise – average log(n) insert, log(n) search and since it is already sorted, cost of printing is n.

The vector store is a red herring. While it is cheaper to insert, most dbms systems load data from file infrequently so other access patterns are more frequent. These access patterns would be search, sort which this data structure is quite bad at. (Although if the data is sorted in place, its print cost is n after the initial nlogn sort).