# CS300 – Project One

Contents

[CS300 – Project One 1](#_Toc110530860)

[Pseudocode – Reading File Contents 1](#_Toc110530861)

[Pseudocode – Create a “Course” object 2](#_Toc110530862)

[Pseudocode – Printing Course Info (Vector) 3](#_Toc110530863)

[Pseudocode – Printing Course Info (Hash Table) 3](#_Toc110530864)

[Pseudocode – Printing Course Info (Tree) 3](#_Toc110530865)

[Pseudocode – Creating a Menu 4](#_Toc110530866)

[Pseudocode – Printing a Sorted List of Courses (Vector) 4](#_Toc110530867)

[Pseudocode – Printing a Sorted List of Courses (Hash Table) 4](#_Toc110530868)

[Pseudocode – Printing a Sorted List of Courses (Tree) 4](#_Toc110530869)

[Runtime and Memory Evaluation 5](#_Toc110530870)

[Advantages/Disadvantages 5](#_Toc110530871)

[Recommendation 5](#_Toc110530872)

## Pseudocode – Reading File Contents

Vector<string> readFileContents(string filename) {

vecCourseInfo = new Vector<string>

courseFile = open\_file(filename)

while courseFile still has lines to read {

strCourseInfo = get\_current\_line(courseFile)

if number of commas in strCourseInfo <= 1 {

print message “Formatting error. Missing parameters”

return null // early return from function

}

vecCourseInfo.append(strCourseInfo)

}

close\_file(courseFile)

return vecCourseInfo

}

bool validatePrerequisites(Vector<string> vecCourseInfo) {

for each line in vecCourseInfo {

if line has >= 2 commas {

lineData = split(line, ‘,’)

// skip first two, since those are the ID and name

// and we want the prerequisites.

for data in lineData (skip first two iterations) {

bool foundPrereq = false

for each innerLine in vecCourseInfo {

innerLineData = split(innerLine, ‘,’)

if innerLineData[0] == data {

foundPrereq = true

break

}

}

}

// If no prereq is found for the current course ID,

// exit with false

if (!foundPrereq) {

return false

}

}

}

// If we make it through each line in the file, that means

// all prereqs were found, so the file is valid

return true

}

## Pseudocode – Create a “Course” object

Course createCourse(string strCourseInfo, Vector<string> strCourseInfoList) {

strCourseData = split(strCourseInfo, ‘,’)

Course newCourse = new Course

newCourse.ID = strCourseData[0]

newCourse.Name = strCourseData[1]

newCourse.Prerequisites = new Vector<Course>

for i = 2 to length of strCourseData - 1 {

prereqID = strCourseData[i]

for each courseInfo in strCourseInfoList {

if prereqID == substring(courseInfo, 0, len(prereqID)) {

prereqCourse = createCourse(courseInfo, strCourseInfoList)

newCourse.Prerequisites.Append(prereqCourse)

}

}

}

}

## Pseudocode – Printing Course Info (Vector)

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

}

## Pseudocode – Printing Course Info (Hash Table)

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

Get requested course from hashtable by key using “courseNumber”

print out the course information

for each prerequisite of the course

print the prerequisite course information

}

## Pseudocode – Printing Course Info (Tree)

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

Start at the root of the tree

If the root node is the same as the courseNumber, return that node

Else

If the courseNumber is less than the root node, check the left child

Else, check the right child

print out the course information

for each prerequisite of the course

print the prerequisite course information

}

## Pseudocode – Creating a Menu

void displayUserMenu() {

set choice = 0

while choice != 9

display menu

get choice from user input

perform action based on user choice

if 1, Load course information from file

if 2, Print Course List (sorted)

if 3, Print Course Title & Prereqs (course provided)

if 9, exit the program

else display "invalid choice" message

}

## Pseudocode – Printing a Sorted List of Courses (Vector)

void sortCourses(Vector<Course> courses) {  
 set sortedCourseList = quicksort(courses)

for each course in sortedCourseList

print out the course information

for each prerequisite of the course

print the prerequisite course information   
}

## Pseudocode – Printing a Sorted List of Courses (Hash Table)

void sortCourses(HashTable<Course> courses) {  
 // the HashTable will contain the vector of nodes  
 set sortedCourseList = quicksort(courses.nodes)

for each course in sortedCourseList

print out the course information

for each prerequisite of the course

print the prerequisite course information   
}

## Pseudocode – Printing a Sorted List of Courses (Tree)

void sortCourses(Tree<Course> courses) {  
 if courses.root is null {  
 return // no courses  
 }  
  
 inOrderTraversal of left side of tree

print out the course information

for each prerequisite of the course

print the prerequisite course information   
  
 inOrderTraversal of right side of tree  
}

## Runtime and Memory Evaluation

**Vector**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **sortedList = quicksort(courses)** | n\*log(n) | 1 | n\*log(n) |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 3n\*log(n) |
| **Runtime** | | | O(n\*log(n)) |

**Hash Table**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **sortedList = quicksort(courses.nodes)** | n\*log(n) | 1 | n\*log(n) |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 3n\*log(n) |
| **Runtime** | | | O(n\*log(n)) |

**Tree**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **If courses.root is null** | 1 | 1 | 1 |
| **In order traversal (left)** | 1 | n/2 | n/2 |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **In order traversal (right)** | 1 | n/2 | n/2 |
| **Total Cost** | | | 3n + 2 |
| **Runtime** | | | O(n) |

## Advantages/Disadvantages

For using a vector data structure, one of the main advantages is the low number of operations that would need to be implemented to satisfy the requirements of the project. Appending to the end of the vector is easy enough and allows for quick insertion. Searching, however, could take up to N searches, where N is the number of courses in our vector, so as the list of courses grows, searching could take longer. For sorting, the naïve approach would lead to very slow results (O(n^2)), but we could achieve at least O(N\*log(N)) by utilizing the quicksort algorithm.

For the hash table, in the underlying data structure, we still have a list or vector of nodes representing our data. However, because of the hashing algorithm which decides which “bucket” to place the data into the hash table, we can’t guarantee that the vector containing our nodes is already in order. This would also only apply for a single item per bucket and would contain more work if we had a linked list in each bucket, needing to iterate through each of those items, as well. As a result, we’ll still need to make use of the quicksort algorithm to sort the buckets/nodes before looping again to display the information. The advantages, however, would be insertion, as we can insert and search into the Hash Table in constant time O(1).

For the tree data structure, the main disadvantages are the amount of operations that need to be implemented to make the tree functional. However, once these operations are implemented, the benefits of doing so greatly outweigh the disadvantages. For insertion, we’re able to insert into the tree in an efficient manner, and the nature of the tree structure allows for some natural ordering, since half of the values are found on one side, and the other half on the other side, allowing us to cut in half the number of operations to perform when searching/inserting. Lastly, we don’t have to take the additional step to sort the data in the tree, and we can simply use an “in-order traversal” over the tree to display the course data in a sorted fashion.

## Recommendation

After going through the pseudocode for the different data structures and reviewing the evaluation and advantages/disadvantages of each, I believe the best data structure to move forward with would be the Binary Search Tree. For the purposes of loading the data into the structures, BSTs can handle inserts very efficiently, while also having the additional benefit of having some implicit ordering to it, since all values less than a particular key will be on one side of the tree, and values greater than that key will be on the other side. Using an in-order traversal of the tree after loading the course data will allow us to display the data in a sorted fashion, satisfying the requirements for the project.