# CS 300 Pseudocode Document

Brian Fulton

CS300

6-2

Begin

// Vector pseudocode

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 parseCSV(“csvPath”){

Implement library “Fast C++ CSV Parser” credit: [Fast-CPP-CSV-Parser](https://github.com/ben-strasser/fast-cpp-csv-parser)

Input/Output CSVReader<3> in("Courses.csv")

Read header(ignore\_extra\_column, "NULL", “NULL”)

string Course; int date; double cost;

try{

while reading row (course, date, cost){

write row to vector

End while

add\_default\_value()

Catch(exception for failed loop){

Do\_while reading row (course, date, cost){

End while

}

}

}

}

Void validateCourses(Vector<Course> courses, String courseNumber){

For all courses in (“Courses.csv”)

If course in row has prerequisite

Prerequisite exists in (“Courses.csv”)

Else Print “Course” courseNumber “Has a prerequisite that

does not exist”

End if

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**

}

// Hashtable pseudocode

int numPrerequisiteCourses(Hashtable<Course> courses, Course c) {

totalPrerequisite = Hashtable [c]

for each prerequisite p in totalPrerequisites

add prerequisites of p to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Hashtable <courses>) {

for int i = 0 while i > hashtablesize iterate i

print course at i

if course at i has prerequisite

print prerequisite course

continue

}

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

for all courses

if the course is the same as courseNumber

print course information

if course.has prerequisite

print prerequisite information

End if

End if

}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses, Node c) {

totalPrerequisites = left and right child of Node c

for each prerequisite p in total prerequisites

add left and right Nodes of p to total prerequisites

print number of total prerequisites

end for

}

void printSampleSchedule(Tree<Course> courses) {

for all Nodes

print course name

if course has left node

print left as prerequisite

if course has right node

print right as prerequisite

}

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

for all Nodes

if course matches courseNumber

print node

if course has left node

print left as prerequisite

if course has right node

print right as prerequisite

end if

else if course has left node

traverse to left

else if couse has right node

traverse to right

end if

}

// Menu Pseudocode

**Begin**

While true

Print "Menu:"

Print " 1. Load Bids"

Print < " 2. Display All Bids"

Print < " 3. Find Bid"

Print < " 4. Remove Bid"

Comment out Print " 5. TEST CASE"

cout << " 9. Exit"

cout << "Enter choice: “

read user input

Case 1:

Read data file

Break

Case 2:

If data file Open

Print vector <courses>

Else

Continue

Break

Case 3:

If data file Open

Print “ Find Bid”

Else

Continue

Break

Case 4:

If data file Open

Print “ Remove Bid”

Else

Continue

Break

Case 5:

If data file Open

Comment out Print “ Test case”

Else

Continue

Break

Case 9:

End While

Print “Good bye”

**End**

// Alphanumeric courses Pseudocode

**Begin**

int numCourses(vector<Course> courses, Course c) {

totalCourses = vector [c]

for each course p in totalcourses

compare course p with next pointer

}

void printAlphanumeric(vector <courses>) {

for int i = 0 while i < vectorsize iterate i

if course at I > course at i+1

push course to back

return

for int i = 0 while i < vectorsize){

print course at i

**END**

**Hash Table Analysis Vector**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Initialize vector <courses>** | 1 | 1 | 1 |
| **Initialize reading lines** | 1 | n | n |
| **Open data file for N value** | 1 | 1 | 1 |
| **If file opens** | 1 | n | n |
| **Get line of data as string** | 1 | n | n |
| **If string is not null** | 1 | 1 | 1 |
| **Add to vector <courses>** | 1 | n | n |
| **Throw error for null vector** | 1 | n | n |
| **Catch exception** | 1 | n | 1 |
| **Write line of data to vector** | 1 | n | n |
| **Close file** | 1 | n | n |
| **Total Cost** | | | 7n + 4 |
| **Runtime** | | | O(n) |

**Runtime Analysis Binary Search Tree**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Initialize vector <courses>** | 1 | 1 | 1 |
| **Initialize reading lines** | 1 | n | n |
| **Open data file for N value** | 1 | 1 | 1 |
| **If file opens** |  |  |  |
| **Read line of data while not null** | 1 | n | n |
| **Add line to vector <courses>** | 1 | n | n |
| **Break if last line of data** | 1 | 1 | 1 |
| **Error if file not opened** | 1 | n | n |
| **Total Cost** | | | 4n + 3 |
| **Runtime** | | | O(n) |

**Runtime Analysis Vector**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Initialize vector <courses>** | 1 | 1 | 1 |
| **Initialize reading lines** | 1 | n | n |
| **Open data file for N value** | 1 | 1 | 1 |
| **Read line of data** | 1 | n | n |
| **Add line to vector <courses>** | 1 | n | n |
| **Break if last line of data** | 1 | 1 | 1 |
| **Error if file not opened** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 5n + 3 |
| **Runtime** | | | O(n) |

**Vector**

A Vector is a simple data structure to hold data. When course data from the master list is parsed and sorted how the user desires, any course is easily accessible using the index

**Hash Table**

A hash table doesn’t need to sort its data to reference a specific piece of data or node. When creating the hash table, it is difficult to visualize exactly what will happen with the keys. Data collisions can occur if data already exists in a bucket when trying to write new data.

**Binary Search Tree**

The binary search tree data structure uses a parent/child relationship where the first node is the parent that can have two or fewer child nodes. The primary advantage is that when searching for a node, the algorithm quickly eliminates the need to look at entire branches of nodes due to the left right depth construction of the tree.

Recommendation:

For this application the Vector data structure is sufficient with such a small data set the O(n) complexity is not slow enough to be noticed. With the data being school courses, it is unlikely that the number of courses needed to be handled would ever scale to a point making the vector approach sluggish.

If however the end user had other intentions with the algorithm, looking at all courses offered at any school in the US for instance, I would recommend the Binary Search Tree be implemented to take advantage of the O(log(n)) complexity for insertion/deletion of data. That would give the user unlimited scale-ability with the only disadvantage being that the data has the chance to throw errors when the tree is being built, that can be minimized and negated with careful design and maintenance.