# CS 300 Pseudocode Document

// Vector pseudocode

**struct course** {

string Num

string Name

Vector<string> prereqs

}

**void loadCourses(Vector<Course> courses){**

if checkFileLines() = 0

print course file not complete

open course information csv file

if file is not open

print file failed to open

else

for each line in file

courses[current line num – 1].Num = first val

courses[current line num – 1].Name = next val

for each value remaining in current line

courses[current line num–1].prereqs[] = next val

close course file

if checkPrereqs() = false

print Not all prerequisites are offered

}

**int checkFileLines**() {

open course information file

if file is not open

print file failed to open

else

for each line in file

if number of values before /n is < 2

return 0

close course file

return 1

}

**bool checkPrereqs(Vector<Course> courses)**{

bool prereqsExist = true

for each course in Courses

for each prereq in current course

for each course

if current prereq = Num

prereqsExist = true

break

else

prereqsExist = false

return prereqsExist

}

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

bool hasPrinted = false

**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**

hasPrinted = true

break

if hasPrinted is false

print No course matches found.

}

**void printAllCourses(vector<Course> courses) {**

quickSort(courses)

for each course

print out the course information

for each prerequisite of the course

print prerequisite

}

**vector quicksort(vector<Course> courses, int begin, int end) {**

int mid = 0

if begin >= end

return

mid = partition(courses, begin, end)

quicksort(courses, begin, mid)

quicksort(courses, mid+1, end)

}

**int partition(vector<Course> courses, int highindex = size of courses-1, int lowindex = 0)**

midpoint = lowIndex + (highindex – lowIndex) / 2

pivot = courses[midpoint]

bool done = false

while(not done)

while(numbers[lowindex] < pivot)

lowIndex +=1

while(numbers[highIndex] > pivot)

highIndex -= 1

if lowIndex >= highindex

done = true

else

temp = numbers[lowindex]

numbers[lowindex] = numbers[highindex]

numbers[highindex] = temp

lowindex += 1

highindex -= 1

**}**

// Hashtable pseudocode

**struct course** {

string Num

string Name

Vector<string> prereqs

}

**class HashTable {**

**Private:**

**struct Node {**

Course course

unsigned int key

Node \*next

**Node()** {

key = MAX

next = null

}

}

**vector<Node> nodes;**

**unsigned int tablesize =** default

**unsigned int hash(**int key)

**Private:**

**HashTable()**

**void Insert(course** course)

**}**

**void HashTable::Insert(Course couse) {**

key = hash of courseNum

Node\* oldNode = pointer to nodes key

if no entry found for key

assign this node to the key position

else if key is not used

assign old node key to MAX

set node to key

set old node to bid

set old noted to next node

set next node to null

else

while next node is not null

oldnode = node after old node

next node = new Node(course, key)

}

**void loadCourses(Hashtable<Course> courses){**

if checkFileLines()= 0

print course file not complete

else

open course info csv

if file isn’t open

print file open failed

else

new course

for each line in file

course.num = first val

course.Name = next val

for each value remaining in current line

course.prereqs[] = next val

HashTable::insert (course)

if checkPrereqs() = false

print not all prereqs are offered

}

**int checkFileLines()** {

open course info file

if file is not open

print file failed to open

else

for each line in file

if number of values before new line is <2

close course info file

return 0

close course info file

return 1

}

**bool checkPrereqs(Hashtable<Course> courses) {**

bool exists = true

for each course in courses

for each prereq in current course

for each course

if current prereq = course.Num

exists = true

break

else

exists = false

return exists

}

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

key = hash of courseNumber

if courseNum of course at key = courseNumber

print courseNum and Name

for each prereq for the course

print prereq

else

print no course matches found

}

**void printAllCourses(Hashtable<Course> courses) {**

for all courses in courses

if node is not null

print course info

for each prereq for the course

print prereq

**}**

// Tree pseudocode

**struct course** {

string Num

string Name

Vector<string> prereqs

}

**struct Node {**

Bid bid

Node\* left

Node\* right

Node() {

set left and right to null

}

Node(Bid bid):Node() {

assign this node’s bid = to bid

}

**void loadCourses(BinarySearchTree<Course> courses)**{

if checkFileLines() = 0

print course file not complete

new course

open course information csv file

if file is not open

print file failed to open

else

for each line in file

course.Num = first val

course.Name = next val

for each value remaining in current line

course.prereqs[] = next val

addNode(course)

close course file

if checkPrereqs() = false

print Not all prerequisites are offered

}

**int checkFileLines()** {

open course information file

if file is not open

print file failed to open

else

for each line in file

if number of values before /n is < 2

return 0

close course file

return 1

}

**bool checkPrereqs(BinarySearchTree<Course> courses){**

bool prereqsExist = true

for each course in Courses

for each prereq in current course

for each course

if BinarySearchTree::Search provides match

prereqsExist = true

break

else

prereqsExist = false

return prereqsExist

}

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

bool hasPrinted = false

**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**

hasPrinted = true

break

if hasPrinted is false

print No course matches found.

}

**void BinarySearchTree::addNode(Node\* node, course)** {

if node course Num is larger than supplied course num

if left node is null

assign left node with the supplied node

else

recurse addNode with the left node

else

if right node is null

assign right node with supplied node

else

recurse addNode with right node

}

**course BinarySearchTree::Search(string courseNum)** {

node current = root

while current isn’t null

if current course num matches supplied course num

return current node

else if current course num is smaller than supplied course num

current = left child node

else

current = right child node

}

**void printAllCourses(Node courses root)**

if node is null

return

printAllCourses(node->left)

print course info

for each prereq

print prereq

printAllCourses(node->right)

}

//Menu

void printMenu() {

print line of –

Print Menu: Enter one of the following:

print 1. Load data structure

Print 2. Print Course List

Print 3. Print Course

Print 4. Exit

Print line or –

print Your Input:

}

int takeMenuInput() {

take input

if input is less not 1, 2, 3, or 4

display error

else

return input

}

//Main

while input not 4

printMenu()

switch(takeMenuInput):

1. LoadCourses()
2. PrintAllCourses()
3. PrintCourse(take input for couseNum)
4. break

**Runtime Analysis:**

//Vector

Table

Description automatically generatedOne of the benefits of the vector method is that it is relatively simple to implement. The most difficult part of using a vector is sorting it. Accessing all of the information is relatively easy to code. What makes using vectors to store the information less beneficial is the fact that the methods involved can take a while to run and can use a lot of computer resources in the process.

Table

Description automatically generated//HashTable

A hash table is a great way to store information. One of the benefits that comes from using a hash table to store the data is that the way we go about organizing everything can make it very easy to search. The key system minimizes the resources and time it would take to find a specific node. The downside that I have found with hash tables is that they seem impossible for me to create a sorting method. So, if somehow, when adding the classes to the table is not in a sorted manner the way we would want, I have not found a way to sort otherwise. Since I have not included a sort algorithm, trying to understand how printing all courses would go is not accurate, and therefore hard to compare to the other data types. Also, the hash table still uses a decent amount of computer resources and can be a little difficult to implement. I will point out that I did not give pseudocode for some hash algorithms that may have made the implementation of the overall course concept a little easier, specifically, the search algorithm, which would help when trying to print course information.

Graphical user interface, table

Description automatically generated//Tree

The binary search tree is a wonderful data structure. What I find most beneficial is the fact the entire tree is organized as new nodes are added. This makes implementing a print all courses method much simpler since no sorting is necessary, just knowledge of implementing an in-order traversal. The overall resources spent doing the various tasks required for this assignment seem to be less with this data structure than the rest, although there are some parts that I am not sure I analyzed correctly, I think that the Binary search tree is the overall fastest and most efficient of the three data structures used in this assignment. The only drawback I have noticed is that search is not as perfectly efficient as it is with a hash table, but it is still faster than with a vector, and it is still overall the fastest of the three data structures examined here.

**Recommendation:**

Upon implementing and analyzing pseudocode for the three data structures, vectors, hash tables, and binary search trees, I would recommend, for loading and organizing a course list, a binary search tree would be the best option. It is an easy to traverse, well organized, fast data structure. It is not overly complicated and can be implemented relatively easily. Therefore, we should use them for a project of this sort.