# CS 300 Pseudocode Document

Oscar Rosa

CS 300

Southern New Hampshire university

6-2 Project One

10/07/23

**// Vector pseudocode**

**// Creating Course Objects, Storage, Reading, Parsing, Check Data**

START

VECTOR courses

FUNCTION loadData(fileName)

OPEN fileName

FOR each line within file

Check if line has at least 2 parameters

IF 2 or more parameters

Parse and store each parameter in variables

Check if prerequisites provided exists within file

IF all (prerequisites in courses for prerequisite)

Create new course object and store in vector data structure

newCourse = Course(courseNum, coursed, prerequisite)

courses.append(newCourse)

CLOSE file

**// Printing Course List in ABC order**

FUNCTION printCourseList()

SORT courses BY courseNUM

FOR each course

PRINT course.courseNUM, course.courseID

**// Searching Data Structure and Printing Course Information and Prerequisites**

FUNCTION printCourse(courseNum)

For each course

IF course.courseNum == courseNum

PRINT “Course Number:”, course.courseNum

PRINT “Course Title:”, course.courseID

IF course.prerequisites:

PRINT “Prerequisites:”,”,”.join(course.prerequisites)

ELSE:

PRINT “Prerequisites: None”

RETURN

PRINT “ERROR”

**// MENU**

FUNCTION main()

DISPLAY “Welcome”

WHILE True

DISPLAY “Please select an option:”

DISPLAY “1. Load Data Structure”

DISPLAY “2. Print Course List”

DISPLAY “3. Print Course Information and Prerequisites”

DISPLAY “4. Exit”

READ userInput

IF userInput == 1

LOAD DATA “courseData.txt”

PRINT “Data loaded”

ELSE IF userInput == 2

PRINT Course List()

ELSE IF userInput == 3

READ courseNum

PRINT course(courseNum)

ELSE IF userInput == 4

PRINT “Thank you, exiting program.”

BREAK

ELSE

PRINT “Error”

END

// **Hashtable pseudocode**

**// Creating Course Objects, Storage, Reading, Parsing, Check Data**

START

HASH courses

FUNCTION loadData(fileName)

OPEN fileName

FOR each line within file

Check if line has at least 2 parameters

IF 2 or more parameters

Parse and store each parameter in variables

Check if prerequisites provided exists within file

IF all (prerequisites in courses for prerequisite)

Create new course object and store in vector data structure

newCourse = Course(courseNum, coursed, prerequisite)

courses.append(newCourse)

CLOSE file

**// Printing Course List in ABC order**

FUNCTION printCourseList()

SORT keys of courses

FOR each key in stored keys of courses

Course = course[key]

PRINT course.courseNum, course.courseID

**// Searching Data Structure and Printing Course Information and Prerequisites**

FUNCTION printCourse(courseNum)

IF courseNum in courses

Course = courses[courseNum]

PRINT “Course Number:”, course.courseNum

PRINT “Course Title:”, course.courseID

IF course.prerequisites:

PRINT “Prerequisites:”,”,”.join(course.prerequisites)

ELSE:

PRINT “Prerequisites: None”

ELSE:

PRINT “Error”

**// MENU**

FUNCTION main()

DISPLAY “Welcome”

WHILE True

DISPLAY “Please select an option:”

DISPLAY “1. Load Data Structure”

DISPLAY “2. Print Course List”

DISPLAY “3. Print Course Information and Prerequisites”

DISPLAY “4. Exit”

READ userInput

IF userInput == 1

LOAD DATA “courseData.txt”

PRINT “Data loaded”

ELSE IF userInput == 2

PRINT Course List()

ELSE IF userInput == 3

READ courseNum

PRINT course(courseNum)

ELSE IF userInput == 4

PRINT “Thank you, exiting program.”

BREAK

ELSE

PRINT “Error”

END

// **Tree pseudocode**

**// Creating Course Objects, Storage, Reading, Parsing, Check Data**

FUNCTION addCourseToTree(root, course)

IF tree is empty, set course as root

IF root IS None

Root = course

ELSE

Traverse tree to find correct position for course

currentNode = root

WHILE True

IF course number is less than current node, move to left child

IF course.courseNum < currentNode.courseNum

IF currentNode.leftChild is none

currentNode.leftChild = course

BREAK

ELSE

currentNode = currentNode.leftChild

ELSE

IF course number is greater than or equal to current node, move to right child

IF currentNode.rightChild is none

currentNode.rightChild = course

BREAK

ELSE

currentNode = currentNode.rightChild

**// Printing Course List in ABC order**

FUNCTION printCourseList(node)

Traverse tree in order and print course information

PRINT node.courseNum, node.courseID

**// Searching Data Structure and Printing Course Information and Prerequisites**

FUNCTION searchCourse(node, courseNum)

IF node is none OR node.courseNum == courseNum

RETURN node

IF course number is less than current node, move to left child

IF courseNum < node.courseNum

RETURN searchCourse(node.leftChild, courseNum)

IF course number is greater than current node, move to right child

ELSE

RETURN searchCourse(node.rightChild, courseNum)

**// MENU**

FUNCTION main()

DISPLAY “Welcome”

WHILE True

DISPLAY “Please select an option:”

DISPLAY “1. Load Data Structure”

DISPLAY “2. Print Course List”

DISPLAY “3. Print Course Information and Prerequisites”

DISPLAY “4. Exit”

READ userInput

IF userInput == 1

LOAD DATA “courseData.txt”

PRINT “Data loaded”

ELSE IF userInput == 2

PRINT Course List(root)

ELSE IF userInput == 3

READ courseNum

IF course is NOT None

PRINT “Course Number:”, course.courseNum

PRINT “Course Title:”, course.courseID

ELSE IF userInput == 4

PRINT “Thank you, exiting program.”

BREAK

ELSE

PRINT “Error”

END

## Runtime Analysis

| **Code**  **Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **All courses** | 1 | n | n |
| **Course is the same as courseNum** | 1 | n | n |
| **Print course information** | 2 | 1 | 1 |
| **Prerequisite of courses exists** | 1 | n | n |
| **Print prerequisite course information** | 2 | n | n |
| **Total Cost** | | | 5n + 1 |
| **Runtime** | | | O(n) |

| **Code**  **Hashtable** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **All courses** | 2 | n | n |
| **Course is the same as courseNum** | 1 | n | n |
| **Print course information** | 1 | 1 | 1 |
| **Prerequisite of courses exists** | 2 | n | n |
| **Print prerequisite course information** | 4 | n | n |
| **Total Cost** | | | 9n + 1 |
| **Runtime** | | | O(n) |

| **Code**  **Tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **All courses** | 1 | n | n |
| **Course is the same as courseNum** | 1 | n | n |
| **Print course information** | 2 | 1 | 1 |
| **Prerequisite of courses exists** | 1 | n | n |
| **Print prerequisite course information** | 4 | n | n |
| **Total Cost** | | | 7n + 1 |
| **Runtime** | | | O(n) |

While working on this project and the assignments before it, this experience has taught me that each type of data structure has its own strengths and weaknesses for the requirements of the program. To start, the vector data structure has the advantage of being the fastest method to use for reading a file and adding course objects compared to the others. Vectors are a very straightforward method in which as the file is parsed, each item within the file is then appended to the end of the vector. In addition, from my runtime analysis the vector had the shortest runtime of the three being at 5n + 1, even though all three data structures shared the same O(n) notation. However, despite having the advantage of being the fastest method of reading a file and adding course objects, vectors also have a disadvantage as well. The disadvantage that a vector has is that it has a hard time searching for a specific item in a list as it requires the program to check each item on the list until a match is found, in which case the disadvantage is that it takes time to do so.

The next data structure that is being looked at for this project was hashtables, and like vectors and other data structures also has its own advantages and disadvantages. The advantages that a hashtable has is the ability to search through a list more quickly than the other data structures can. By creating a key for the hashtable, the locations of a given item or course in this case for the project will also be known and can be easily searched and print the output for. However, the disadvantage that a hashtable has is that it has a slower implementation when creating an initial list because each item within the list needs a key to be created for those items as well as finding a spot to insert each course. In addition, hashtables also have issues when it comes to sorting as the table itself cannot be sorted. In order for hashtables to print an alphanumeric list of all the courses, each value must be extracted, sorted, and then printed. Because a hashtable needs to be able to handle all this information the runtime for this data structure is longer and in this case with my run time I got 9n + 1 which is higher than the one I had for the vector data structure being at 5n + 1.

The last data structure is the binary search tree data structure, and like the previous data structure also has its advantages and disadvantages. The advantage of the tree is that it is great for searching, insertion and deletion, as well as being able to handle large data sets. The disadvantage of using a tree however is that it is more complex compared to a vector data structure and a hashtable data structure. In addition, a binary tree would have to search every element if the tree ended up with only having left leaves. This would change the search time to O(h) instead of O(n) in which h would be the height of the tree.

After working through the pseudocode and runtime analysis for this project, my recommendation would be to use a vector data structure for this program. I think the ability to quick sort to print the entire catalogue of courses would prove valuable to the client. Furthermore, the loss of time for the search can be overlooked for the utility of the sort function that the vector data structure has and the runtime it has compared to the other data structures being at 5n + 1 which is the shortest compared to the others being at 9n + 1 for the hashtable data structure and 7n + 1 for the binary search tree data structure respectively. Overall, I think the vector data structure would be the appropriate approach for the program because it is easier to implement and does not require much memory which can be an important factor to consider especially if working in a limited environment.