Andrea C. Sherry

Andreacarmen.sherry@snhu.edu

Project one

CS - 300

DSA: Analysis and Design

1. **Resubmit pseudocode from previous pseudocode assignments and update as necessary.**

**Pseudocode Vector Data Structure  
Reads data, Parses each line, and Checks for errors**

INITIALIZE Program

USE fstream to open file

IF return value -1 THEN

File not found

ELSE

Open file with course information

DISPLAY information

READ each line of the file

IF less than two values in each line THEN

Parse parameters and store them in the proper variable

ELSE

Read parameters in each line

IF prerequisites not found in line THEN

Move to next line

ELSE

Display error

FINALIZE program

**Create course objects and store them in the appropriate data structure.**

INITIALIZE program

OPEN file with course information

READ each line of the file

LOOP to process all lines in file

FIND parameters THEN

PARSE parameters

ADD parameters in variables

IF no course object exists THEN

ADD new course object with the information stored in the variables

ADD course object in the appropriate data structure

IF any still lines to be processed THEN

REPEAT process

ELSE

FINALIZE program

**Search data structure for a specific course and print out course information and prerequisites.**

INITIALIZE program

ASK user for courseNumber

INPUT courseNumber

IF courseNumber correct THEN

PRINT course information and prerequisites

IF courseNumber not correct THEN

DISPLAY error and ask to try again

IF courseNumber not correct again THEN

DISPLAY error and goodbye message

FINALIZE program

**Pseudocode Hash Table Data Structure  
Pseudocode open file, read data from the file, parse each line, check for file format errors.**

INITIALIZE empty directory

Store course data:

courseNumber

name

prerequisites

INITIALIZE empty list to store validation errors

OPEN input file

IF file does not open THEN

DISPLAY “error” message

END program

} ELSE {

OPEN file

FOR each line in the file DO:

READ each line from file

SPLIT line into parts

IF number of parts is less than 2 THEN

ADD validation error to list

CONTINUE to the next iteration

EXTRACT courseNumber and name from the first two parts

INITIALIZE empty list to store prerequisites

FOR each remaining part DO:

ADD each remaining part to prerequisite list

STORE course data in dictionary

Course\_data[courseNumber] = {

‘name’: course\_name, ‘prerequisites’: prerequisites

}

FOR each prerequisite in the list DO:

IF prerequisite not found in course\_data THEN

ADD validation “error” message

CLOSE file

IF validation errors exist THEN

FOR each error DO:

DISPLAY “error” message

END program

} ELSE {

DISPLAY file formatted correctly

END program

**Create course objects and store them in the appropriate data structure**

INITIALIZE empty hash table

ASSOCIATE course object with its courseNumber

OPEN input file

IF file does not open THEN

DISPLAY “error” message

END program

} ELSE {

OPEN file

WHILE lines remaining DO:

READ next line in the file

SPLIT line into parts

IF number of parts LESS than 2 THEN

DISPLAY “invalid” line message

} ELSE {

CONTINUE to next iteration

EXTRACT courseNumber, name, prerequisite FROM:

courseNumber = parts[0]

course\_name = parts[1]

IF prerequisites shown THEN

CREATE empty list named prerequisites\_list

FOR each part left from index 2 DO:

APPEND part to prerequisites\_list

CREATE new course object

SET (new course object).courseNumber = courseNumber

SET (new course object).name = course\_name

SET (new course object).prerequisites = prerequisites\_list

STORE course object in hash table:

ADD course object to hash table with courseNumber:

course\_hash\_table[courseNumber] = (new course object)

CLOSE file

IF errors when parsing or creating course THEN

DISPLAY error

END program

} ELSE {

DISPLAY successful message

CONTINUE processing

END program

**Pseudocode Print out course information and prerequisites**

FUNCTION PrintCourseInfo(course):

PRINT “Course Number: “ + course.courseNumber

PRINT “Course Name: “ + course+name

IF course.prerequisites not empty THEN

PRINT “Prerequisites: “

FOR each prerequisite in course.prerequisites

PRINT “ – “ + prerequisite

FUNCTION PrintCourseData(courses\_hash\_table): //takes a hash table with a course object in it and calls ‘PrintCourseInfo’ to print the information of a course

FOR each courseNumber in courses\_hash\_table DO:

CALL PrintCourseInfo(courseInstance) //prints information from a course such as courseNumber, name, and prerequisites if applicable

**Pseudocode Tree Data Structure  
Opens the file, reads data from the file, parses each line, and checks for file format errors.**function readCourseDataFromFile(filename):

try:

**OPEN** file **WITH** filename **FOR** reading

**INITIALIZE** empty list of courses

**FOR** each line:

**IF** line empty **THEN**:

Continue to **NEXT** line

**IF** length of fields < 2 THEN:

**PRINT** “Error Insufficient parameters on line”

Continue to **NEXT** line

courseNumber = fields[0]

courseNumber = fields[1]

prerequisites = fields[2:] IF length fields > 2

**ELSE** empty list

**FOR** each prerequisite in prerequisite:

**IF NOT** courseExists(prerequisite, courses):

print “Error: Prerequisite”, prerequisite, “not found”

break

**ELSE**:

add (courseNumber, courseName, prerequisites) to courses list

**CLOSE** file

**RETURN** to courses

**IF** IOError **THEN**:

**PRINT** “Error: Unable to open file”

**RETURN** empty list

**FUNCTION** courseExists(courseNumber, courses):

**FOR** course **IN** courses:

**IF** course[0] == courseNumber:

return **TRUE**

**ELSE**

return **FALSE**

**Create course objects and store them in the appropriate data structure.   
FUNCTION** createCourseObjectsFromFile (fileName):

coursesVector = empty vector of Course objects

courseData = readCourseDataFromFile(fileName)

**FOR** each courseTuple in courseData:

courseNumber = courseTuple[0]

courseName = courseTuple[1]

prerequisites = courseTuple[2]

courseObject = Course(courseNumber, courseName, prerequisites)

coursesVector.push\_back(courseObject)

**RETURN** coursesVector

**Print out course information and prerequisites.   
FUNCTION** printCourseInformation(Tree<Course> courses, String courseNumber):

courseNode = courses.findNode(courseNumber) // Find corresponding node to given course number

**IF** courseNode is null **THEN**:

**PRINT** "Course not found"

RETURN

printCourseInfoAndPrerequisites(courseNode)

**FUNCTION** printCourseInfoAndPrerequisites(Node<Course> node):

**PRINT** node.course.courseNumber, node.course.courseName // Course Info Printed

**IF** node.left is **NOT** null **THEN**: //**IF** left child exists (prerequisite)

printCourseInfoAndPrerequisites(node.left) //**PRINT** left child (prerequisite)

**IF** node.right is **NOT** null **THEN**: //**IF** right child exists (prerequisite)

printCourseInfoAndPrerequisites(node.right) // **PRINT** right child (prerequisite)

1. **Create pseudocode for a menu.  
   a.** **Load Data Structure and Exit:  
   FUNCTION** main():

courses <- empty data structure

**loop**:

display\_menu()

choice <- get\_user\_input() **switch choice**:  
 **case 1**:   
 load\_data\_structure(courses)  
 **case 2**:  
 print\_course\_info(courses)  
 **case 3**:  
 print\_sorted\_course\_list(courses)  
 **case 4**:  
 exit\_program()  
 **default**:

display\_error\_message()  
**FUNCTION** display\_menu():  
 PRINT(“**Menu:** “)  
 PRINT("**Please, choose an option.**”)

PRINT("**1. Load Data Structure**.”)

PRINT("**2. Print Course Information**.”)

PRINT("**3. Print Sorted List of Courses**.”)

PRINT("**4. Exit**.”)

**FUNCTION** get\_user\_input():

input <- read\_user\_input

**RETURN** input

**FUNCTION** load\_data\_structure(courses):

file\_path <- get\_file\_path\_from\_user()

**IF** file\_path not empty **THEN**:

read\_data\_from\_file(file\_path, courses)

**PRINT**(“**Data was loaded**.”)

**ELSE**:

**PRINT**(“**Error in file path**.”)

**FUNCTION** read\_data\_from\_file(file\_path, courses):

file <- open(file\_path)

**FOR** each line **THEN**:

course\_data <- parse\_line(line)

add\_course\_to\_data\_structure(course\_data, courses)

**CLOSE**(file)

**FUNCTION** parse\_line(line):

course\_data <- split(line, “,”)

**RETURN** course\_data

**FUNCTION** add\_course\_to\_data\_structure(course\_data, courses):

course\_num <- course\_data[0]

course\_name <- course\_data[1]

prerequisites <- course\_data[2:]

courses[course\_num] <- (course\_name, prerequisites)

**FUNCTION** print\_course\_info(courses):

**IF** data\_structure\_not\_empty(courses) **THEN**:

**FOR** each course\_num, (course\_name, prerequisites) in courses:

**PRINT**(“**Course:** “, course\_num)

**PRINT**(“**Name:** ”, course\_name)

**PRINT**(“**Prerequisites:** “, prerequisites)

**ELSE**:

**PRINT**(“**Data structure empty. Try again**.”)

**FUNCTION** print\_sorted\_course\_list(courses):

**IF** data\_structure\_not\_empty(courses) **THEN**:

sorted\_courses <- sort\_courses(courses)

**FOR** each course\_num in sorted\_courses:

**PRINT**(course\_num)

**ELSE**:

**PRINT**(“**Data structure empty. Try again**.”)

**FUNCTION** sort\_courses(courses):

sorted\_courses <- sort(courses)

**RETURN** sorted\_courses

**FUNCTION** exit\_program():

**PRINT**(“**Exiting program. Goodbye**!”)

**EXIT**()

**FUNCTION** display\_invalid\_choice\_message():

**PRINT**(“**Error, try again**.”)

**FUNCTION** data\_structure\_not\_empty(courses):

**RETURN** courses NOT empty

**b. Print Course List**

**FUNCTION** print\_course\_list(courses):

cs\_courses <- empty list

**FOR** each course\_num, (course\_name, \_) in courses:

**IF** course\_is\_from\_cs\_dep(course\_num) **THEN**:

cs\_courses.append(course\_num)

**IF NOT** cs\_courses **THEN**:

**PRINT**(“**No course found in this department. Try again**.”)

**ELSE**:

sorted\_cs\_courses <- sort(cs\_courses)

**FOR** each course\_num in sorted\_cs\_courses:

**PRINT**(course\_num)

**FUNCTION** course\_is\_from\_cs\_dep(course\_num):

**RETURN** course\_num.startwith(“**CSCI**”)

c. **Print Course**

**FUNCTION** print\_course(courses):

course\_num <- get\_input\_user\_course\_num()

**IF** course\_exist(course\_num, courses) **THEN**:

course\_name, prerequisites <- get\_course\_info(course\_num, courses)

**PRINT**(“**Course Title:** “, course\_name)

**PRINT**(“**Prerequisites:** “, prerequisites)

**ELSE:**

**PRINT**(“**Course not found. Try again**.”)

**FUNCTION** get\_input\_user\_course\_num():

course\_num <- read\_user\_input(“**Enter course number:** “)

**RETURN** course\_num

**FUNCTION** course\_exist(course\_num, courses):

**RETURN** course\_num in courses

**FUNCTION** get\_course\_info(course\_num, courses):

course\_name, prerequisites <- courses[course\_num]

**RETURN** course\_name, prerequisites

1. **Design pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order.**

**FUNCTION** main():

courses <- empty data structure

**LOOP**:

display\_menu()

choice <- get\_user\_input()

**SWITCH** choice:

**CASE 1:**

load\_data\_structure(courses)

**CASE 2:**

print\_course\_info(courses)

**CASE 3:**

print\_sorted\_course\_list(courses)

**CASE 4:**

print\_sorted\_courses(courses)

**CASE 5:**

print\_course(courses)

**CASE 6:**

exit\_program()

**DEFAULT:**

display\_invalid\_message()

**FUNCTION** print\_sorted\_course\_list(courses):

cpp\_courses <- empty list

**FOR** each course\_num, (course\_name, prerequisites) in courses:

**IF** course\_num.startswitch(“CSCI”) AND “C++” in course\_name:

cpp\_courses.append((course\_num, course\_name, prerequisites))

**IF** no cpp\_courses **THEN**:

**PRINT**(“**No course found in this department. Try again**.”)

**ELSE**:

sorted\_courses <- sort(cpp\_courses, key=lambda x: x[0])

**FOR** course\_num, course\_name, prerequisites in sorted\_courses:

**PRINT**(“**Courses:** “, course\_num)

**PRINT**(“**Name:** “, course\_name)

**PRINT**(“**Prerequisites:** “, prerequisites)

1. **Evaluation**
   1. **Define how the program opens the file, reads the data from the file, parses each line, and checks for formatting errors.**

**Pseudocode Vector**

|  |  |  |  |
| --- | --- | --- | --- |
| **Pseudocode** | **Cost Per Line of Code** | **Num. of Times Line Will Execute** | **TOTAL** |
| **Initialize File** | **1** | **1** | **1** |
| **IF not End-Of-File THEN continue reading lines** | **1** | **n** | **n** |
| **IF < 2 THEN store values** | **1** | **n** | **n** |
| **ELSE continue reading each line** | **1** | **1** | **1** |
| **IF prerequisites not found move to next** | **1** | **n** | **n** |
| **ELSE error** | **1** | **1** | **1** |

**Pseudocode Hash Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **Pseudocode** | **Cost Per Line of Code** | **Num. of Times Line Will Execute** | **TOTAL** |
| **Initialize File** | **1** | **1** | **1** |
| **IF not End-Of-File THEN continue reading lines** | **1** | **n** | **n** |
| **IF < 2 THEN store values** | **1** | **n** | **n** |
| **ELSE continue reading each line** | **1** | **1** | **1** |
| **IF prerequisites not found move to next** | **1** | **n** | **n** |
| **ELSE error** | **1** | **1** | **1** |

**Pseudocode Binary Search Tree**

|  |  |  |  |
| --- | --- | --- | --- |
| **Pseudocode** | **Cost Per Line of Code** | **Num. of Times Line Will Execute** | **Total** |
| **Initialize File** | **1** | **1** | **1** |
| **IF not End-Of-File THEN continue reading lines** | **1** | **n** | **n** |
| **ELSE go next line** | **1** | **n** | **n** |
| **IF line empty THEN continue** | **1** | **n** | **n** |
| **IF length field < 2 THEN error** | **1** | **n** | **n** |
| **ELSE go next line** | **1** | **n** | **n** |
| **ADD bids to binary search tree** | **1** | **n** | **n** |
| **FOR value 1 and 2 ADD course num, course name, prerequisites** | **1** | **n** | **n** |
| **IF values is > 2 THEN ADD prerequisites** | **1** | **n** | **n** |

**5. Explain the advantages and disadvantages of each structure in your evaluation.  
  
Vector (Advantages):**

There are several advantages to using vectors. Firstly, vectors are fairly easy to implement and are commonly available in most programming languages. Next, the elements in a vector are stored in contiguous memory locations, facilitating efficient access through pointer arithmetic. This feature, commonly found in programming languages like C and C++, enables fast and direct access to elements. Finally, elements in a vector can be accessed in constant time using their indices. This makes vectors ideal for scenarios where random access to elements is necessary, as accessing any element by its index is quick and straightforward.  
  
**Vector (Disadvantages)**:  
  
Some of the disadvantages associated with the use of vectors are related to inflexible memory allocation, fixed size, and inefficient random insertions within the array. First, dealing with large data can become problematic as arrays require contiguous memory allocations. This requirement can lead to issues with memory fragmentation and inefficient memory usage. Next, since arrays have a fixed size when they are created, resizing them can be challenging. When attempting to resize, one must allocate a new array and copy its elements, which can be time-consuming and memory intensive. Finally, insertions and deletions at random positions within the array are inefficient. This inefficiency arises because moving elements to accommodate new ones is necessary, which can result in a significant performance overhead, particularly for large arrays. Overall, these limitations make vectors less suitable for scenarios where dynamic resizing and efficient random insertions are required.  
  
**Hash Table (Advantages)**:  
  
There are several advantages to using hash tables. First, hash tables offer quick lookups, providing constant-time average-case lookup complexity. This feature makes them optimal for scenarios prioritizing swift retrieval of key-value pairs. Next, hash tables demonstrate flexibility in size, allowing dynamic resizing to include additional elements effortlessly, without incurring significant performance overhead. Finally, hash tables exhibit efficient insertion and deletion operations. These operations are usually streamlined, especially when maintaining a low load factor, ensuring optimal performance.  
  
**Hash Table (Disadvantages)**:  
  
Firstly, hash tables can encounter hash collisions, where two distinct keys hash to the same index. Addressing these collisions often requires additional overhead, such as using techniques like chaining. Next, hash tables lack ordering, meaning that elements stored inside are not arranged in any specific order. This absence of order can pose a challenge if an ordered traversal of elements is necessary. Finally, hash tables can require a higher memory overhead compared to other data structures. This increased overhead occurs due to the need for additional space to manage collisions effectively and maintain the underlying array structure.  
  
**Binary Search Tree (Advantages):**Firstly, trees feature a tiered structure that captures hierarchical relationships. This characteristic renders trees adept at modeling hierarchical data, including file systems and binary search trees. Next, binary search trees facilitate ordered traversal through various traversal algorithms such as pre-order and post-order traversals. These algorithms allow efficient ordered access to elements, improving their utility for tasks like sorting. Finally, binary search trees offer dynamic sizing abilities, allowing them to adapt to changing data sets by growing or shrinking as elements are inserted or deleted. This resizing feature makes trees suitable for situations where the size of the data set changes over time. **Binary Search Tree (Disadvantages):**Firstly, binary search trees require a complex implementation process, which can pose challenges in terms of both development and maintenance. Next, the search complexity of trees can vary based on their balance. While balanced trees, like binary search trees, offer average-case search complexity of O(log n), unbalanced trees can degenerate into linear search times of O(n). Finally, binary search trees may incur a higher memory overhead compared to hash tables. This increased overhead arises from the need to store additional pointers for parent-child relationships, which contribute to the overall memory consumption of the data structure. **6. Make a recommendation for which data structure you will plan to use in your code.**Based on the advantages and disadvantages of each data structure, I recommend using vectors instead of binary search trees or hash tables. Firstly, I recommend using vectors because of the reduced memory overhead. Vectors typically have lower memory overhead than, for example, hash tables, due to them storing only the actual elements without additional overhead for hashing. Due to this, reduced memory usage is beneficial, especially in memory-constrained environments. Next, vectors are easier to implement compared to other data structures like hash tables. Due to vectors being a built-in data structure in many programming languages like C++, they provide a solution for managing lists of courses without the need for extra data structure implementation. Finally, vectors tend to maintain a predictable order, something that cannot be said about other data structures. When required to maintain a specific order of courses like alphabetical order or order based on course numbers, vectors can easily traverse the list of courses in a predictable manner. Since ordering the courses alphabetically or by course numbers has been required in past assignments, I believe that using vectors is the best data structure to be used in my code.