**CS 300: Project One**

**Andrew Fenwick**

**1) Resubmit from previous assignments and update as necessary**

**// VECTOR PSEUDOCODE**

**// File pseudocode**

//OPENING randomFile

//file = open(randomFile)

//WHILE file is not blank

//READING randomFile

//line = readline(file)

//PARSING randomFile

//CHECK FOR ERRORS IN FILE FORMAT randomFile

//IF no errors present

//CHECK for any prerequisite that is provided on a line exists as a course

//anything at end of line must have another line with course

**//Create Course Object Pseudocode**

//Creation of courses variables

//Open and read required files for courses

//Find correct way of storage

//Determine file format to help with parsing

//While file is open

//Store in proper location

//Vector data structure

//VDS will have multiple course objects once entire file processed

**//Search Specific Course Pseudocode**

//Open required files pertaining to courses needed

//While file is open

//Print information present

//Store necessary data in vector data structure

**// HASH TABLE PSEUDOCODE**

**//File pseudocode**

//VALIDATE the sample file

//Ensure the format is correct

//CHECK all the following

//Ensure there are 2 parameters on each line

//Some courses may not have prerequisites

//Ensure any prerequisites present on a line exists as a course in the file

// Any prerequisites at the end of a line must have another line in that file starting with the course number

**//Create/Store Course Object pseudocode**

//SHOW how to create course objects

//One course object should hold data from a single line from the input file

//LOAD all the data into the hash table structure

//Know file format

//Helps with parsing and storing each token of data into appropriate location

**//Printing pseudocode**

//USING a vector data structure is provided and recommended

//PROPER techniques and code for printing hash table data structures provided aswell

//EXAMPLE for method

//implement logic to print all bids

    // for node begin to end iterate

    // if key not equal to UINT\_MAx

            // output key, bidID, title, amount and fund

            // node is equal to next iter

            // while node not equal to nullptr

                // output key, bidID, title, amount and fund

                // node is equal to next node

**// BINARY TREE PSEUDOCODE**

**//File pseudocode**

//OPEN the file

//Ensure the format is correct

//READ the data from the file

//Ensure there are 2 parameters on each line

//Some courses may not have prerequisites

//PARSE each line

//Ensure any prerequisites present on a line exists as a course in the file

// Any prerequisites at the end of a line must have another line in that file starting with the course number

//CHECK for file format errors

**//Create/Store Course Object pseudocode**

//SHOW how to create course objects

//One course object should hold data from a single line from the input file

//LOAD all the data into the tree data structure

//Know file format

//Helps with parsing and storing each token of data into appropriate location

//USE of a loop will be needed to process all lines

**//Printing pseudocode**

//USING a vector data structure is provided and recommended

//PROPER techniques and code for printing tree data structures provided as well

**2) Create pseudocode for a menu**

**Create a Switch for the menu**

//SET int choice equal to zero

//Create a while loop where choice is not equal (!=) to 4

//cout << “1. Load Data Structure”

//cout << “2. Print Course List”

//cout << “3. Print Course”

//cout << “4. Exit”

//cin >> choice

**Load Data Structure**

//Switch choice for case 1

//Initialize the CSV Parser

//Create a loop that will read rows of a CSV file

//Initialize a course using data from current row(i)

//cout<<course.courseId<<": "<<course.name<<endl;

//Add this course to the end

//Run a sort function to alphabetize all the courses

//add catch statement for potential error

**Print Course List**

//Switch choice for case 2

//Implement print logic

//Start at the head

//While loop over each node looking for a match

//cout<<course.courseId<<": "<<course.name<<endl;

//Set current equal to next

**Print Course**

//Switch choice for case 3

//Implement search logic first

//Special case if matching node is head

//Make head point to the next node in the list

//Decrease size count

//Return

//Start at the head of the list

//Keep searching until end reached with while loop (next != nullptr)

//If the current node matches, return it

//Else current node is equal to next node

//Return course

//Output current coursed and courseName

**Exit**

//Switch choice for case 4

//cout << “Good bye” << endl;

//Return 0;

**3) Design pseudocode that prints out the list of courses in the CS program alphabetically**

//CREATE nested loops to receive the course names and then print them in order

//RECEIVE course name

//string str[#ofcourses], temp;

//OUTPUT << “Enter course name” << endl

//FOR (int i equals 0, i < # of courses, ++i

//getline(cin,str[i]

//ORDER them properly

//FOR (int i equals 0, i < # of courses, ++i

//FOR(int j = i+1, j < # of courses, ++j

//IF (str[i]> str[j]

//SET temp equal to str[i]

//SET str[i] equal to str[j]

//SET str[j] equal to temp

//PRINT sorted list to a display

//FOR (int i equals 0, i < # of courses, ++i

//OUTPUT<< str[i] << endl

|  |  |  |  |
| --- | --- | --- | --- |
| **VECTOR** | LINE COST | # TIMES EXECUTED | TOTAL COST |
| CREATE vector for holding all courses | 1 | 1 | 1 |
| LOOP for each line provided | 1 | n | n |
| ADD line to create vector for course item | 1 | n | n |
| IF prereq exists with course | 1 | n | n |
| Append prereq to vector | 1 | n | n |
| RETURN course | 1 | n | n |
|  |  | Total Cost | 5n + 1 |
|  |  | Runtime | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **HASH TABLE** | LINE COST | # TIMES EXECUTED | TOTAL COST |
| CREATE hash table holding courses | 1 | 1 | 1 |
| ADD method | 0 | 0 | 0 |
| MAKE each course a key | 1 | n | n |
| IF no entry found for key | 1 | n | n |
| Pair node with key | 1 | n | n |
| ELSE | 1 | n | n |
| Pair node with UNIT\_MAX | 1 | n | n |
| SET to key | 1 | n | n |
| SET older node to course | 1 | n | n |
| SET old node to null | 1 | n | n |
| ELSE | 1 | n | n |
| Search for next free node | 1 | n | n |
| Add new node to the end | 1 | n | n |
| FOR every course in file | 1 | n | n |
| CREATE vector for course | 1 | n | n |
| IF prereq exists with course | 1 | n | n |
| Append prereq to vector | 1 | n | n |
| RETURN course | 1 | n | n |
|  |  | Total Cost | 16n+1 |
|  |  | Runtime | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **TREE** | LINE COST | # TIMES EXECUTED | TOTAL COST |
| BST add method (tree,node) | 0 | 0 | 0 |
| IF tree🡪root is null | 1 | 1 | 1 |
| Tree🡪root = node | 1 | n | n |
| Node🡪left= null | 1 | n | n |
| Node🡪right = null | 1 | n | n |
| ELSE | 1 | n | n |
| Curr = tree🡪root | 1 | n | n |
| While (cur is not null) | 1 | n | n |
| IF (node🡪key < cur🡪key) | 1 | n | n |
| IF (cur🡪left is null) | 1 | n | n |
| Cur🡪left = node | 1 | n | n |
| Cur = null | 1 | n | n |
| ELSE | 1 | n | n |
| Cur = cur🡪left | 1 | n | n |
| ELSE | 1 | n | n |
| If (cur🡪right is null) | 1 | n | n |
| Cur🡪right = node | 1 | n | n |
| Cur = null | 1 | n | n |
| ELSE | 1 | n | n |
| Cur = cur🡪right | 1 | n | n |
| Node🡪left = null | 1 | n | n |
| Node🡪right = null | 1 | n | n |
| FOR every course in file | 1 | n | n |
| CREATE vector for course | 1 | n | n |
| IF prereq exists with course | 1 | n | n |
| Append prereq to vector | 1 | n | n |
|  |  | Total Cost | 24n+1 |
|  |  | Runtime | O(log(n)) |

Tree, Hash Table and Vector have many similarities but also many differences. Figuring out which is the best option depends on what type of data you are dealing with and the quantity of that data.

A vector is much like an array, but it has additional abilities. You can resize them, they are easy to use and they keep track of its size well. They can be slow though when dealing with excessive amounts of data.

A hash table can take data and store it in the proper location. The way nodes connect are arbitrary and data values are not ordered. Inserting, searching or deleting are completed in O(1) time. They can be implemented as either linear or non-linear structures. A downside is that the input data size must be specified before the map is created.

The binary tree have nodes that can only connect to an additional 2 nodes. It has a non linear data structure and . inserting, searching or deleting are completed in O(log(n)) time. The data values are usually sorted and ordered. Unlike hash maps, the size of the data being imported does not need to be specified.

For ABCU’s computer science department, I believe that the hash table would be the best option. Although trees are have an expectional run time, being that is is O(log(n)), the fact it can only connect to a max of 2 nodes makes it a tough choice. If there becomes a case where 3 classes are a prerequisite for a course, it would not work. A hash table could handle that quantity. Although a vector could work, the amount the data ABCU will use would make runtimes too slow for the standards the Computer Science department.