**Vector and Hash Data Structure Pseudocode**

//adds all valid courses in a file into a vector or courses

void readFile(String fileName, Vector<Course> courses) {

Open file filename for reading

While you have a next line in the file

If the line has at least 2 strings

Read the line and store in a temporary course object

If all the courses prereqs are in the courses vector

Store the course in courses

Close file

}

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Open file filename for reading | 1 | 1 | `1 |
| While you have a next line in the file | 1 | n | n |
| If the line has at least 2 strings | 1 | n | n |
| Read the line and store in a temporary course object | 1 | n | n |
| If all the courses prereqs are in the courses vector | n | n | n^2 |
| Store the course in courses | 1 | n | n |
| Close file | 1 | 1 | 1 |
| **Total Cost** | | | n^2 + 4n + 2 |
| **Runtime** | | | O(n^2) |

struct Course {

String name

String description

Vector<String> prereqs

}

//Adds a course to a vector or courses

void addCourse(Course course, Vector<Course> courses) {

If courses does not contain course

courses.add(course)

}

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| If courses does not contain course | n | 1 | `n |
| courses.add(course) | 1 | 1 | 1 |
| **Total Cost** | | | n + 1 |
| **Runtime** | | | O(n) |

//prints courses that has a specific preRequisite

void printCoursesWithPrereq(Vector<Course> courses, String prereqCourseNumber) {

For all courses

For all prereqs

If prereq matches prereqCourseNumber

Print course

Break out of prereq for loop

}

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| For all courses | 1 | n | `n |
| For all prereqs | 1 | n^2 | n^2 |
| If prereq matches prereqCourseNumber | 1 | n^2 | n^2 |
| Print course | 1 | n | n |
| Break out of prereq for loop | n | n | n |
| **Total Cost** | | | 2n^2 + 3n |
| **Runtime** | | | O(n^2) |

// Hashtable pseudocode

Struct HashTable{

vector<Node> hashVector

unsigned int size

}

Struct Node{

\*Course

\*Node

}

//adds all valid courses in a file into a Hash Table or courses

void readFile(String fileName, HashTable courses) {

Open file filename for reading

While you have a next line in the file

If the line has at least 2 strings

Read the line and store in a temporary course object

If all the courses prereqs are in the courses vector

Store the course in courses

Close file

}

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Open file filename for reading | 1 | 1 | `1 |
| While you have a next line in the file | 1 | n | n |
| If the line has at least 2 strings | 1 | n | n |
| Read the line and store in a temporary course object | 1 | n | n |
| If all the courses prereqs are in the courses vector | n | n | n^2 |
| Store the course in courses | 1 | n | n |
| Close file | 1 | 1 | 1 |
| **Total Cost** | | | n^2 + 4n + 2 |
| **Runtime** | | | O(n^2) |

//Adds a course to a hash table of courses

void addCourse(Course course, HashTable courses) {

If courses does not contain course

courses.add(course)

}

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| If courses does not contain course | 1 | 1 | `1 |
| courses.add(course) | 1 | 1 | 1 |
| **Total Cost** | | | n + 1 |
| **Runtime** | | | O(n) |

//prints courses and all prereqs

void printCoursesWithPrereq(hashtable courses, String CourseNumber) {

For all courses in the bucket hash(CourseNumber)

If Course Number in bucket matches CourseNumber

Print course

For each prereq in course

Print prereq

return

}

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| For all courses in the bucket hash(CourseNumber) | 1 | n | n |
| If Course Number in bucket matches CourseNumber | 1 | n | n |
| Print course | 1 | 1 | 1 |
| For each prereq in course | 1 | n | n |
| Print prereq | 1 | n | n |
| **Total Cost** | | | 4n + 2 |
| **Runtime** | | | O(n) |

//Data structure for binary tree

Node{

Course course

Int key

Node\* left

Node\* right

}

Node root = null

//pulls data from file and stores in binary tree

void readFile(String fileName, Node root) {

Open file filename for reading

While you have a next line in the file

If the line has at least 2 strings

Read the line and store in a temporary course object

If all the courses prereqs are in the courses vector

Store the course using root

Close file

}

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Open file filename for reading | 1 | 1 | `1 |
| While you have a next line in the file | 1 | n | n |
| If the line has at least 2 strings | 1 | n | n |
| Read the line and store in a temporary course object | 1 | n | n |
| If all the courses prereqs are in the courses vector | n | n | n^2 |
| Store the course using root | 1 | n | n |
| Close file | 1 | 1 | 1 |
| **Total Cost** | | | n^2 + 4n + 2 |
| **Runtime** | | | O(n^2) |

//create course objects and stores them in binary tree

void addCourse(Course course, Node node) {

If node is null

Store course in new node here

Else if Node key = course key

Course already exists return

Else if node key > course key

addCourse( course, node->left )

Else

addCourse( course, node->right )

}

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| If node is null | 1 | 1 | `1 |
| Store course in new node here | 1 | 1 | 1 |
| Else if Node key = course key | 1 | 1 | 1 |
| Course already exists return | 1 | 1 | 1 |
| Else if node key > course key | 1 | 1 | logn |
| addCourse( course, node->left) | 1 | logn | 1 |
| Else | 1 | 1 | 1 |
| addCourse( course,node->right) | 1 | logn | logn |
| **Total Cost** | | | 2logn +6 |
| **Runtime** | | | O(logn) |

//Print out course info in prereqs from binary tree

//prints courses and all prereqs

void printCoursesWithPrereq(Node root, String CourseNumber) {

If Course Number matches CourseNumber in node

Print course

For each prereq in course

Print prereq

Return

If Course Number < CourseNumber in node

printCoursesWithPrereq( root->left, CourseNumber )

else

printCoursesWithPrereq( root->right, CourseNumber )

}

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| For all courses in the bucket hash(CourseNumber) | 1 | 1 | 1 |
| If Course Number in bucket matches CourseNumber | 1 | n | n |
| Print course | 1 | 1 | 1 |
| For each prereq in course | 1 | n | n |
| Print prereq | 1 | n | n |
| return | 1 | 1 | 1 |
| If Course Number < CourseNumber in node | 1 | 1 | 1 |
| printCoursesWithPrereq( root->left, CourseNumber ) | logn | 1 | logn |
| else | 1 | 1 | 1 |
| printCoursesWithPrereq( root->right, CourseNumber ) | logn | 1 | logn |
| **Total Cost** | | | 3n + 2logn + 3 |
| **Runtime** | | | O(n) |

//loads the menu with one of the different types of lists

loadMenu(dataStructure list) {

Initialize list structure

While program is running

Print menu

// will print out a menu where 1 = load

// 2 = display 3 = print a course and 4 = quit

C = Take input as character

If c = 1

ReadFile(fileName, list)

If c = 2

printSorted (dataStructure)

If c = 3

courseID = take in a string

Print course(dataStructure, courseID)

If c = 4

Exit while loop.

}

//prints all courses alphabetically from a vector

printSorted(vector courses) {

Vector = quicksort(vector)

For all courses in vector

Print course

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Vector = quicksort(vector) | logn | 1 | logn |
| For all courses in vector | 1 | n | n |
| Print course | 1 | n | n |
| **Total Cost** | | | 2n + logn |
| **Runtime** | | | O(n) |

//prints all courses alphabetically from a hash table

printSorted(hashTable courses) {

For all buckets in hashtable

For all courses in linked list

Print course

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| For all buckets in hashtable | 1 | m | m |
| For all courses in vector | 1 | n | n |
| Print course | 1 | n | n |
| **Total Cost** | | | 2n + m |
| **Runtime** | | | O(n) |

//prints all courses alphabetically from a binary tree

printSorted(Node node) {

Print sorted(left node)

If not null

Print course in node

Print sorted(right node)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Print sorted(left node) | 1/2n | 1 | 1/2n |
| If not null | 1 | 1 | 1 |
| Print course in node | 1 | 1 | 1 |
| Print sorted(right node) | 1/2n | 1 | 1/2n |
| **Total Cost** | | | n + 2 |
| **Runtime** | | | O(n) |

**Pseudocode Evaluation**

I have written pseudocode to accomplish the goal of storing the list of courses in code, and being able to print them out. The pseudocode also can use three different data structures to accomplish this task: a vector, a hash list, and a binary tree. A vector is a good data structure to use if you can take advantage of being able to instantly access any point, but for this project any search will take O(n) time to traverse through it. Sorting for the printout with the vector will also take O(n) time because of the need to sort the list, this sort will also take up an extra list worth of space if you use any quick sorting algorithms. If the hash table is used, it provides almost instant access to any data that is inserted because we can access the buckets directly. So the search function will be close to a O(k) search time most of the time as long as the hash table is big enough. The downside to a hash table is that printing it out into an ordered list could be difficult depending on how the table is designed. If it was hashed in such a way that order is maintained, printing the whole list could take O(mn) where m is the size of the hash table. If the order is not maintained, as it is with most hash tables, a sorted print could take up to O(mnlogn) time which you would not want for a common operation. The binary tree could be a good choice because it is usually fast for most operations. It will take a little longer than the hash list to create, insertion on average takes a little longer in a binary tree if the hash list is created correctly, but it makes up for this by being sorted upon insertion O(n) to iterate through in order and a relatively quick lookup of O(logn).

After considering all the options I have decided that the binary list is the best data structure to use for this problem. Printing a single course and printing the ordered list of courses are likely going to be more common operations than the loading of the list of courses and the binary tree has relatively quick ways to accomplish these tasks (O(logn)). If I valued searching individual values and creating a data set more highly I would’ve chosen a hash table but that hash table will potentially take a much longer time to print out a list of ordered data than a binary tree(O(mn) or (O(mnlogn)). An unsorted vector will always take longer to search through (O(n)) and print an ordered list of (O(nlogn)). A binary tree will offer a fast runtime for common operations and a fairly quick insertion time for a sorted list and should be used for this program.