# CS300: DSA – Analysis and Design Project One

Riley Eichenour

Southern New Hampshire University

CS300: DSA – Analysis and Design

Professor Norman Lippincott

June 13, 2025

## CS300: DSA – Analysis and Design Project One

### Pseudocode

#### Common

struct Course {

String courseNumber

String courseTitle

Vector<String> prerequisites

}

Course processCourse(String courseString){

split courseString by token

if split string is invalid

print course malformed error

exit program

sanitize tokens

set first token to courseNumber

set second token to courseTitle

if there are 3 or more tokens

for each additional token

add token to prerequisites vector

return course

}

void promptCourseMenu() {

while the user does not provide the exit command

print menu

prompt choice

dispatch user choice

}

#### Vector Data Structure

void loadCourseFile(String filePath, Vector<Course>& courses){

open filePath

if filePath opened successfully

create fileLine variable

while line is not end of file

read file line into fileLine variable

call processCourse with fileLine

add course to courses

close filePath

else

print file open error

exit program

}

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

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

}

void printCourses(Vector<Course> courses) {

sort courses

for all courses

print the course

for all course prerequisites

print the prerequisite

}

#### Hash Table Data Structure

void loadCourseFile(String filePath, Vector<Course>& courses){

open filePath

if filePath opened successfully

create fileLine variable

while line is not end of file

read file line into fileLine variable

call processCourse with fileLine

hash courseNumber

add course to hashed course bucket in order

close filePath

else

print file open error

exit program

}

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

hash courseNumber

get hashed courseNumber bucket

for all courses in the bucket

if the course is the same as courseNumber

print out the course information

for each prerequisite of the course

print the prerequisite course information

}

void printCourses(const Vector<Course>& courses){  
 for each course bucket

while the current item is not null

print course information

for each prerequisite

print prerequisite

assign current item to next node

}

#### Binary Tree Data Structure

void loadCourseFile(String filePath, BinaryTree<Course>& courses){

open filePath

if filePath opened successfully

create fileLine variable

while line is not end of file

read file line into fileLine variable

call processCourse with fileLine

add course to courses

close filePath

else

print file open error

exit program

}

Class Tree {

private:

struct Node {

Course course

Node\* left

Node\* right

}

Node\* root

void addCourse(Node\* node, Course course)

Course findCourse(Node\* node, string courseNumber)

void inOrderMap(Node\* node, Map<string, Course>& prequisiteMap)

void inOrderPrint(Node\* node)

void inOrderValidate(Node\* node, const Map<string, Course>& prequisiteMap)

void getAllPrerequisites(Node\* node, Map<string, Course>& prequisiteMap)

public:

void AddCourse(Course course)

void FindCourse(string courseNumber)

void CreatePrequisitesMap(Map<string, Course>& prequisiteMap)

void PrintAllCourses()

}

void BinaryTree::addCourse(Node\* node, Course course) {

if node is null

create a new Node with course

set node to new node with course

if node course number greater than course number

call addCourse with node left child and course

else

call addCourse with node right child and course

}

void BinaryTree::findCourse(Node\* node, Course course) {

if node is null

return null

if node course number greater than course number

call addCourse with node left child and course

else

call addCourse with node right child and course

}

void BinaryTree::inOrderMap(Node\* node, Map<string, Course>& prequisiteMap) {

if node is null

return

call inOrderMap on node left child

for each coursePrequsite in node prerequisite

add prerequisite to map

call inOrderMap on node right child

}

void BinaryTree::inOrderPrint(Node\* node) {

if node is null

return

call inOrderPrint on node left child

print course number

for each coursePrequsite in node prerequisite

print prerequisite

call inOrderPrint on node right child

}

void BinaryTree::Insert(Course course) {

call addCourse on root and course

}

Course BinaryTree::FindCourse(string courseNumber) {

call findCourse with root and course number)

}

void BinaryTree::PrintAllCourses() {

call inOrderPrint on root with prerequisite map

}

void BinaryTree::CreatePrequisitesMap(Map<string, Course>& prerequisiteMap) {

call inOrderMap with tree root and prerequisiteMap

}

### Runtime Analysis

|  |  |  |
| --- | --- | --- |
| processCourse (Common) | Time Complexity (Worst Case) | Number of calls |
| split courseString by token | O(1) | 1 |
| if split string is invalid | O(1) | 1 |
| print course malformed error | O(1) | 1 |
| exit program | O(1) | 1 |
| sanitize tokens | O(1) | 1 |
| set first token to courseNumber | O(1) | 1 |
| set second token to courseTitle | O(1) | 1 |
| if there are 3 or more tokens | O(1) | 1 |
| for each additional token | O(1) | 1 |
| add token to prerequisites vector | O(n) | n |
| return course | O(1) | 1 |

|  |  |  |
| --- | --- | --- |
| loadCourseFile (Vector) | Time Complexity (Worst Case) | Number of calls |
| open filePath | O(1) | 1 |
| if filePath opened successfully | O(1) | 1 |
| create fileLine variable | O(1) | 1 |
| while line is not end of file | O(1) | n |
| read file line into fileLine variable | O(1) | n |
| call processCourse with fileLine | O(n) | n |
| add course to courses | O(n) | n |
| close filePath | O(1) | 1 |
| else print file open error | O(1) | 1 |
| exit program | O(1) | 1 |

|  |  |  |
| --- | --- | --- |
| loadCourseFile (Hash Table) | Time Complexity (Worst Case) | Number of calls |
| open filePath | O(1) | 1 |
| if filePath opened successfully | O(1) | 1 |
| create fileLine variable | O(1) | 1 |
| while line is not end of file | O(1) | n |
| read file line into fileLine variable | O(1) | n |
| call processCourse with fileLine | O(n) | n |
| hash courseNumber | O(1) | n |
| add course to hashed course bucket in order | O(n) | n |
| close filePath | O(1) | 1 |
| else  print file open error | O(1) | 1 |
| exit program | O(1) | 1 |

|  |  |  |
| --- | --- | --- |
| loadCourseFile (BinaryTree) | Time Complexity (Worst Case) | Number of calls |
| open filePath | O(1) | 1 |
| if filePath opened successfully | O(1) | 1 |
| create fileLine variable | O(1) | n |
| while line is not end of file | O(1) | n |
| read file line into fileLine variable | O(n) | n |
| call processCourse with fileLine | O(n) | n |
| add course to courses | O(n) | n |
| close filePath | O(1) | 1 |
| print file open error | O(1) | 1 |
| exit program | O(1) | 1 |

### Advantages and Disadvantages

|  |  |  |
| --- | --- | --- |
| **Data Structure** | **Advantages** | **Disadvantages** |
| Vector | * Fast element access if the index of the element is known * Simple access methods * Traversal is simple | * Insertion can be expensive when inserting in earlier indices * Increasing the size is expensive requiring a full copy of the previous object * Non-index based access is slow |
| Hash Table | * Fast access for reasonable hash function * Fast element access with hash function * Accessing elements can be based on arbitrary attributes | * Frequent collisions can decrease element access performance * Unsorted ordering of elements without inserting the element in the sorted position * Empty buckets lead to wasted memory space |
| Binary Tree | * Elements are sorted for an in order traversal by design * Insertion at worst is O(log n) in a balanced tree | * Without balancing element access and insertion can be slower or on par with simpler structures * Logic can be more difficult to understand than index and linear data structures |

### Recommendation

With consideration for the frequency of searching the data and prerequisite validations, my recommended data structure for the program would be the Binary Tree. Validating the course information while loading the file will frequently be searching the structure of the courses, the vector does not offer a key based access solution and searching would take an increasing amount of time based on the input size. The Hash Table has fast access given a suitable hash function, however not knowing enough information on the courses prior to loading them makes determining a suitable bucket distribution difficult, potentially requiring multiple reallocations of the data structure negatively impacting overall performance. The searching and insertion time for random data loosely assuming that the data will be inserted in random order leading to at most the height of the tree leads to the best data structure for this program. Additionally, the tree data structure can prepare for transitioning into a graph data structure if the application has the need to do so for improved prerequisite traversal. The worst case analysis does indicate very similar performance, however average cases tend to be better in the Hash Table and Binary Tree data structures.

## References

Eichenour, R. (n.d.). *CS 300 3-1 Project One Milestone One*.

Eichenour, R. (n.d.). *CS 300 4-1 Project One Milestone Two*.

Eichenour, R. (n.d.). *CS 300 5-1 Project One Milestone Three*.

Southern New Hampshire University. (n.d.). *CS 300 Pseudocode Document*. Snhu.edu.