**Vectors**

**Parse CSV into a Vector**

void loadCourses(string CSVPath) {

Define vector data structure CourseVector to hold courses

Initialize CSV parser

Create vector prereqNotFoundHolder

Loop through CSV while iterator less than size of column 0

Loop while there are more columns

Create newCourseStruct()

CourseId set to column 0

CourseDescription set to column 1

If no column 1:

Throw error “All courses need an ID and Description”

Append any other columns to Course Prerequisites

If prerequisite not found in CourseVector:

Add course to prereqNotFoundHolder and move to next item

Else

Push Course to end of CourseVector

If prereqNotFoundHolder size > 0

Loop while there are items in prereqNotFoundHolder

Loop through each course’s prereqs

If all prereqs found

Push to end of CourseVector

Else

Throw error “All prerequisites must exist in catalogue” and move to next course

Delete prereqNotFoundHolder

}

**Find and Print Course in a Vector**

void printCourseInformation(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

Return

Print “Course [courseNumber] not found.”

}

**Print All Courses**

Void printCourseCatalogue(Vector<Course> courses) {

Create vector sortVector

Sort courses into sortVector by CourseID

Loop through sortVector start to finish

Print course struct item with newline

Delete sortVector

}

**Hash Tables**

**Parse CSV into a Hash Table**

void loadCourses(string CSVPath) {

Define nodes struct for hash table usage with constructors and a destructor

Define hash function

Define insert function

Initialize CSV parser

Create vector prereqNotFoundHolder

Loop through CSV while iterator less than size of column 0

Loop while there are more columns

Create new Course struct

CourseId set to column 0

CourseDescription set to column 1

If no column 1:

Throw error “All courses need an ID and Description”

Append any other columns to Course Prerequisites

If prerequisite not found in hash table:

Add course to prereqNotFoundHolder and move to next item

Store newCourse in a node

Hash course’s courseId

Assign newCourse node to hash table with hashed courseId as key

If a collision, push to back of vector in increasing order

If prereqNotFoundHolder size > 0

Loop while there are items in prereqNotFoundHolder

Loop through each course’s prereqs

If all prereqs found

Hash course node into hash table

Else

Throw error “All prerequisites must exist in catalogue” and move to next course

Delete prereqNotFoundHolder

}

**Find and Print a Course in the Hash Table Using the Node’s CourseKey**

void printCourseInformation (String courseNumber) {

run courseNumber through hash function

for all courses in that hash 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

Return

Print “Course [courseNumber] not found.”

}

**Print All Courses**

void printCourseCatalogue() {

Declare Vector<Course> sortVector

Loop from start to end of hash table

If iterator != UINT\_Max

Push Course to sortVector

Move to next node

While Node != nullptr

Push Course to sortVector

Move to next node

.sort() sortVector

While i < sortVector.size

Print CourseID and CourseKey

While j < CoursePrerequisites size

Print CoursePrerequisites[j]

}

**Binary Tree**

**Parse CSV into Binary Tree**

void loadCourses(string CSVPath) {

Define nodes struct for binary tree usage with constructors and a destructor

Define binary tree class

Define insert function

Initialize CSV parser

Create vector prereqNotFoundHolder

Loop through CSV while iterator less than size of column 0

Loop while there are more columns

Create new Course struct

CourseId set to column 0

CourseDescription set to column 1

If no column 1:

Throw error “All courses need an ID and Description”

Append any other columns to Course Prerequisites

If prerequisite not found in binary tree:

Add course to prereqNotFoundHolder and move to next item

Store newCourse in a node

Pass node through insert function

If root = nullptr

New node is root

Else

If newCourse node Id is less than root, recurse down the left

Else if newCourse node id is greater than root, recurse down the right

Insert at end of appropriate branch

If prereqNotFoundHolder size > 0

Loop while there are items in prereqNotFoundHolder

Loop through each course’s prereqs

If all prereqs found

InsertNode() into Binary Table

Else

Throw error “All prerequisites must exist in catalogue” and move to next course

Delete prereqNotFoundHolder

}

**Find and Print a Course in Binary Tree using the Node’s CourseKey**

void printCourseInformation(String courseNumber) {

step through binary tree

recurse from bottom of the left side from root

if the course is the same as courseNumber

print out the course information

for each prerequisite of the course

print the prerequisite course information

Return

recurse down right side

Print “Course [courseNumber] not found.”

}

**Print All Courses**

void printCourseCatalogue() {

If Node != nullptr

Recurse down the left side until the bottom

Print CourseID and CourseKey

For Each Prerequisite

Print Prerequisite

Recurse down the right side

Work up the tree until end

}

**Create Course Struct**

Create Course struct {

Create struct members

CourseKey -> Unsigned Int

CourseId -> string

CourseDescription -> string

CoursePrerequisites -> vector

}

**Menu**

int main() {

Process command line arguments

Define a timer

Define courseContainer as chosen data structure

Define userInput -> int

While userInput != 9

Print visual menu

Switch Statement:

Case 1 Load Data Structure

Initialize Timer

loadCourses(csvPath, courseContainer)

Print “Courses Loaded”

Calculate and Display time elapsed

Break

Case 2 Print Course List

printCourseCatalogue(courseContainer)

Case 3 Print Course

Initialize timer

printCourseInformation()

Calculate and Display Elapsed Time

Case 9 Exit

Close

}

**Evaluation**

Looking at this pseudocode, we can tell that from a big overall picture, the speed efficiencies are similar. Below is a table to display my findings.

|  |  |  |  |
| --- | --- | --- | --- |
| Process | Vector | Hash Table | Binary Search Tree |
| loadCourses() | O(n3) | O(n3) | O(n3) |
| printCourseInformation() | O(n) | O(n2) | O(n) |
| printCourseCatalogue() | O(n) | O(n2) | O(n) |

**loadCourses()**

All three data structures utilize a very similar method for parsing the data into their respective data types. A max of a loop with two nested loops is present in each, raising each time to O(n3). The Binary Search Tree might take longer as it sorts as it goes, and the Vector does a fairly quick push to add to its data type, so it could be a touch quicker than the rest. Still, the big picture is that the data set needs to be properly iterated through, and that will occupy the brunt of the time for each of the three data types in question. It should be roughly equivalent across the board.

**printCourseInformation()**

This method once again sees a very similar method of data retrieval through the three data types. Vectors and Binary Search Trees both require a single iteration through the data set in order to locate a specific key, however a vector will need to potentially go through each node to find the desired key or to return that there is no such key in the vector. A Binary Search Tree is presorted, and much fewer nodes will need to be checked to find the desired key. Essentially, even though both are O(n), a Binary Tree’s n is likely to be significantly smaller and it will almost certainly see quicker search and return times than a vector in a data set of any significant size. A Hash Table is likely quicker than a vector, even if its worst case could be O(n2). However, because I am using vectors to handle collisions, we technically have a higher O value. With a well designed hashing function and a dynamic table size, a hash table will still be visiting less pieces of data than a vector likely will, and so I think it is likely a second place contender.

**printCourseCatralogue()**

This method sees the biggest difference in speed. A vector’s default sort method has a guaranteed max timing of O(N log(n)), but that needs to get called in addition to a loop of O(n). So a vector can print relatively quickly, but must be sorted since we did not sort upon insert into the list, which likely would have been more resource intensive. A hash table is sorted by key, but there is no guarantee—nor any reasonably efficient manner I can think to do so—that it is in any way ordered. So, the best way I can think to accomplish a sorted print is to load the entire table into a vector then sort that, which adds steps and essentially makes it slower than a vector.

A Binary Search Tree, however, comes presorted. It is a true O(n), with no reliance on any other sorting or processes. Therefore, it will be the quickest and will only require a single iteration through the list from left to right.

**Conclusion**

As far as I can gather, every single aspect of the program will be benefitted most by using a Binary Search Tree as the default data structure. Though parsing could potentially take a longer, the benefits found elsewhere seem to heavily outweigh any possible draw backs. It seems to me a superior data structure for this application.