## Evaluation:

The run-time of each implementation of the project was similar at O(n^3) for insertion and parsing due to the data validation function being run each time a new course was loaded into the data structure. In hindsight, I should have created a new function that performed the data validation all at once in the beginning to separate it from the insertion functions for each of the data structure implementations.The reason the time complexity was so high was due to a nested for loop causing the complexity to start at n^2 just by running the data validation function.

Starting with the vector, the advantages are that it is simple and does not use many pointers in its data structure explicitly. It is easy to read when implemented and easy to understand. It is largely slower than both the hashtable and binary search tree in searching for courses. When sorting courses, it is on the same level as the hashtable implementation, and slower than the binary search tree since it is “sorted” during implementation, and all that is needed is an in-order transversal. For the vector, insertion is fastest out of the three as no additional methods other than adding it to the vector need to be performed. Removal is also easy since it is just popping it out of the vector. Memory-wise the vector is probably the smallest as it does not include memory for extra pointers and is a standalone vector.

The HashTable’s advantages are a very fast search algorithm (O(1)). It does not provide an advantage when sorting, as the data points are not sorted by value when hashed. Insertion for the hashtable is still in the same order of magnitude as a vector but the data will need to be hashed prior to insertion. Removal of a element in a hash table is similar to a vector as no pointers need to be adjusted. Memory-wise the hashtable is worse than the vector as keys need to be saved.

The binary search tree has the advantage that it is sorted during insertion and so has a very quick sort or “print all sorted” function. Insertion and removal are a bit harder than both the hash table and the vector due to having to reconnect and create pointers to other elements. Searching within a binary search tree is better than a vector with a value of O(logn). Memory-wise the binary search tree is worse than a vector because it needs to contain two pointers for each node and data such as a root.

From the analysis of the three different data structures, I have decided that for the project I will utilize a hash-table. The ease and speed of searching will be useful for the PrintCoursePrerequisites function, where it would need to search for a course’s prerequisites to print the course information. I expect the searching function to be used more than things such as the “sort and print all” function by the end-user. For example, you could potentially sort the courses once during the parsing and loading and then would never need to utilize the sorting function again. However a search time of O(1) will be extremely useful if students are looking for specific courses and are repeatedly searching for courses and printing prerequisites.

## 

## Pseudocode for a menu:

**Menu Pseudocode:**

Int main(int argc, char\* argv[])

Define data structure to hold courses

Define a course

Set filePath

Set currentChoice to 0

While (choice is not 4):

Output “Menu”

Output “1. Load data structure”

Output “2. Print course list”

Output “3. Print Course”

Output “4. Exit”

Switch (currentChoice)

If 1:

Create new data structure

Call loadCourses on filePath

If 2:

Call printAllCourses(data structure of courses) function

If 3:

Call printCourses(string courseNumber) function

End loop

Output “goodbye”

## Pseudocode for printAllCourses():

### Vector:

**printAllCourses(courses)**

Create new vector<Course> named sortedCourses

For each course in Courses vector:

Copy the course and save a copy in sortedCourses

quickSort(sortedCourses, 0, length) //calls quicksort function

For each course in sortedCourses:

Print each course’s courseNumber, courseName

End function

**quickSort(data structure of courses, begin, end)**:

If there are less than or equal to 1 courses left

Return

Function call to partition(courses, begin, end) and set pivot equal to the index

Partition function:

Set low = begin

Set high = end

Select the midpoint between low and high and set to pivot

While low<high pointers:

While courses.at(low).courseNumber < courses.at(pivot).courseNumber

Increment low

While courses.at(pivot).courseNumber < courses.at(high).courseNumber

Decrement high

If low<high:

Swap courses at indices low and high

Increment low

Decrement high

When low>=high:

Return high

Call quicksort on items to left of pivot

Call quicksort on items to right of pivot

### HashTable:

HashTable will utilize a similar implementation for sorting as vector:

**printAllCourses(courses)**

Create new vector<Course> named sortedCourses

For each Node in vector of Nodes:

Copy the Node.course and save a copy in sortedCourses

quickSort(sortedCourses, 0, length) //calls quicksort function

For each course in sortedCourses:

Print each course’s courseNumber, courseName

End function

**quickSort(data structure of courses, begin, end)**:

If there are less than or equal to 1 courses left

Return

Function call to partition(courses, begin, end) and set pivot equal to the index

Partition function:

Set low = begin

Set high = end

Select the midpoint between low and high and set to pivot

While low<high pointers:

While courses.at(low).courseNumber < courses.at(pivot).courseNumber

Increment low

While courses.at(pivot).courseNumber < courses.at(high).courseNumber

Decrement high

If low<high:

Swap courses at indices low and high

Increment low

Decrement high

When low>=high:

Return high

Call quicksort on items to left of pivot

Call quicksort on items to right of pivot

### BinarySearchTree:

Since the Binary Search Tree is sorted during implementation, an In-Order transversal would print all courses in alphanumeric order.

**printAllCourses(courses)**

If (node is null):

Return

else :

Recursive function call to printAllCourses(node’s left ptr)

Print node’s courseNumber, courseName

Recursive function call to printAllCourses(node’s right ptr)

End function

## Resubmit pseudocode from previous pseudocode assignments and runtime analysis of each

**Vector Pseudocode:**

*This function validates the line of data that is being loaded, it is called by loadCourses():*

bool validateData(parser file, int line)

If there are two parameters on row line:

return true

Else if there are three or more parameters on row line:

For each parameter index, p, starting from the 3rd to the end of the line:

Set tempValid = false

For each row, k, in the file:

If file[line][p] == file[k][0]: //if the current prerequisite exists in file

Set tempValid = true

End Loop

If tempValid = false:

return false

Else:

return true

End Loop

Else: //if there are less than 2 parameters

return false

*This function loads all the data from the file*

vector<course> loadCourses(String filePath)

Define a vector, courses, to hold multiple Course items

Initialize file = parser(filePath)

For each row of data, i, in the file:

If validateData(file, i) == true:

Create a course data structure

Set course.courseNumber to the first parameter

Set course.courseName to the second parameter

If there are three or more parameters on row i:

Set course.hasPrerequisites = true

For each parameter index, p, starting from 3rd to the end of the line:

Add file[i][p] to the end of course.prerequisites

End Loop

Else:

Set course.hasPrerequisites = false

Add course data structure to the end of courses

End Loop

*This is what the structure of a Course will be, the method getInformation is called by printCourses():*

Struct Course {

String courseNumber

String courseName

bool hasPrerequisites

vector<string> prerequisites

String getInformation():

return courseNumber, courseName as a string

}

*This function prints the course information:*

void printCourses (Vector<Course> courses, String courseNumber)

For each course, i, in courses:

If i == courseNumber:

print i.getinformation()

If course.hasPrerequisites = true:

//this section searches each string for a match in courseNames in courses

For each string, s, in course.prerequisite:

For each course, j, in courses:

If s == j.courseName:

print j.getinformation

End Loop

End Loop

End Loop

If there are two parameters on row line:

return true

Else if there are three or more parameters on row line:

For each parameter index, p, starting from the 3rd to the end of the line:

Set tempValid = false

For each row, k, in the file:

If file[line][p] == file[k][0]: //if the current prerequisite exists in file

Set tempValid = true

End Loop

If tempValid = false:

return false

End Loop

Else: //if there are less than 2 parameters

return false

Runtime Analysis for validateData():

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| If there are two parameters on row line: | 1 | 1 | 1 |
| return true | 1 | 1 | 1 |
| Else if there are three or more parameters on row line: | 1 | 1 | 1 |
| For each parameter index, p, starting from the 3rd to the end of the line: | 1 | n | n |
| Set tempValid = false | 1 | n | n |
| For each row, k, in the file: | 1 | n^2 | n^2 |
| if the current prerequisite exists in file | 1 | n^2 | n^2 |
| Set tempValid = true | 1 | n^2 | n^2 |
| If tempValid = false | 1 | n | n |
| Return false | 1 | n | n |
| Else | 1 | n | n |
| Return true | 1 | n | n |
| Else: | 1 | 1 | 1 |
| Return false | 1 | 1 | 1 |
| **Total Cost** | | | 3n^2+6n+4 |
| **Runtime** | | | O(n^2) |

Runtime Analysis for loading the file and creating course objects:

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| for each row of data in the course | 1 | n | n |
| If validateData() is true | n^2 | n | n^3 |
| Create course data structure | 1 | n | n |
| Set course.courseNumber | 1 | n | n |
| Set course.courseName | 1 | n | n |
| If there are three or more prerequisites on row i | 1 | n | n |
| Set course.hasPrerequisites to true | 1 | n | n |
| for each parameter index, p starting from the third: | 1 | n^2 | n^2 |
| Add file[i][p] to end of course.prerequisites | 1 | n^2 | n^2 |
| Else: | 1 | n | n |
| Set course.hasPrerequisites = false | 1 | n | n |
| Add course data structure to end of courses. | 1 | n | n |
| **Total Cost** | | | n^3 + 2n^2 + 9n |
| **Runtime** | | | O(n^3) |

**HashTable pseudocode:**

*This function validates the line of data that is being loaded, it is called by loadCourses()*

bool validateData(parser file, int line)

If there are two parameters on row line:

return true

Else if there are three or more parameters on row line:

For each parameter index, p, starting from the 3rd to the end of the line:

Set tempValid = false

For each row, k, in the file:

If file[line][p] == file[k][0]: //if the current prerequisite exists in file

Set tempValid = true

End Loop

If tempValid = false:

return false

End Loop

Else: //if there are less than 2 parameters

return false

End Function

*This function loads all the data from the file*

void loadCourses(String filePath, Hashtable\* hashTable)

Initialize file = parser(filePath)

For each row of data, i, in the file:

If validateData(file, i) == true:

Create a course data structure

Set course.courseNumber to the first parameter

Set course.courseName to the second parameter

If there are three or more parameters on row i:

Set course.hasPrerequisites = true

For each parameter index, p, starting from 3rd to the end of the line:

Add file[i][p] to the end of course.prerequisites

End Loop

Else:

Set course.hasPrerequisites = false

HashTable method call to Insert(course) into hashTable

End Loop

End Function

*This is the HashTable method Insert(Course course):*

void Insert(Course course)

Convert course.courseNumber to int

Set key to (int courseNumber % tableSize) //this is the hashed courseNumber

Create Node\* node and set to node at key

If Node at key is nullptr:

Insert node at vector of nodes within HashTable class to index key from beginning

Else if node is not used:

Set the current node’s key to key

Set the current node’s course to course

Set the current node’s next pointer to nullptr

Else:

While (current node is not nullptr):

Increment current node to next node

End Loop

Add node last node’s next

End Function

*This is what the structure of a Course will be, the method getInformation is called by printCourses():*

Struct Course {

String courseNumber

String courseName

bool hasPrerequisites

vector<string> prerequisites

String getInformation():

return courseNumber, courseName as a string

}

*To make the printCourses function easier, I will use a Search() method for HashTable:*

Course Search(string courseNumber)

Convert courseNumber to int

Set key to (int courseName % tableSize) //this is the hashed courseNumber

Create Node\* node and set to node at index key

If node’s key == UINT\_MAX

Return empty course object

If node is not a nullptr:

While (node is not a nullptr):

If node’s course.courseNumber is equal to courseNumber:

Return node’s course

Increment node to next node

End Loop

Return empty course object

End Function

*This function prints the course information:*

Void printCourses (HashTable hashTable, String courseNumber)

Set tempCourse to hashTable->Search(courseNumber)

If tempCourse is empty course:

Return //returns without printing if courseNumber not found

Else:

Print tempCourse.getinformation()

If tempCourse.hasPrerequisites == true:

//this section searches for each prerequisite and then prints it’s course information

For each string, s, in tempCourse.prerequisites:

Set pCourse to hashTable->Search(courseNumber)

If pCourse is empty course:

Return

Else:

Print pCourse.getinformation

End Loop

End Function

Runtime Analysis for Insert() function.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Convert course.courseNumber to int | 1 | 1 | 1 |
| Set key to (int courseNumber % tableSize) | 1 | 1 | 1 |
| Create Node\* node and set to node at key | 1 | 1 | 1 |
| If Node at key is nullptr: | 1 | 1 | 1 |
| Insert node at vector of nodes within HashTable class to index key from beginning | 1 | 1 | 1 |
| Else if node is not used: | 1 | 1 | 1 |
| Set the current node’s key to key | 1 | 1 | 1 |
| Set the current node’s course to course | 1 | 1 | 1 |
| Set the current node’s next pointer to nullptr | 1 | 1 | 1 |
| Else: | 1 | 1 | 1 |
| While (current node is not nullptr): | 1 | n | n |
| Increment current node to next node | 1 | n | n |
| Add node last node’s next | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 11 |
| **Runtime** | | | O(n) |

Runtime Analysis for loading the file and creating course objects:

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Initialize file = parser(filePath) | 1 | 1 | 1 |
| For each row of data, i, in the file: | 1 | n | n |
| If validateData(file, i) == true: | n^2 | n | n^3 |
| Create a course data structure | 1 | n | n |
| Set course.courseNumber to the first parameter | 1 | n | n |
| Set course.courseName to the second parameter | 1 | n | n |
| if there are three or more parameters on row i: | 1 | n | n |
| Set course.hasPrerequisites = true | 1 | n | n |
| For each parameter index, p, starting from 3rd to the end of the line: | 1 | n^2 | n^2 |
| Add file[i][p] to the end of course.prerequisites | 1 | n^2 | n^2 |
| Else: | 1 | n | n |
| Set course.hasPrerequisites = false | 1 | n | n |
| HashTable method call to Insert(course) into hashTable | n | 1 | n |
| **Total Cost** | | | n^3+2n^2+9n + 1 |
| **Runtime** | | | O(n^3) |

**Binary Search Tree Pseudocode:**

*This is what the structure of a Course will be, the method getInformation is called by printCourses():*

Struct Course {

String courseNumber

String courseName

bool hasPrerequisites

vector<string> prerequisites

String getInformation():

return courseNumber, courseName as a string

}

*This function validates the line of data that is being loaded, it is called by loadCourses()*

bool validateData(parser file, int line)

If there are two parameters on row line:

return true

Else if there are three or more parameters on row line:

For each parameter index, p, starting from the 3rd to the end of the line:

Set tempValid = false

For each row, k, in the file:

If file[line][p] == file[k][0]: //if the current prerequisite exists in file

Set tempValid = true

End Loop

If tempValid = false:

return false

End Loop

Else: //if there are less than 2 parameters

return false

End Function

*These function loads all the data from the file by calling Insert():*

void loadCourses(String filePath, BinarySearchTree\* bst)

Initialize file = parser(filePath)

For each row of data, i, in the file:

If validateData(file, i) == true:

Create a course data structure

Set course.courseNumber to the first parameter

Set course.courseName to the second parameter

If there are three or more parameters on row i:

Set course.hasPrerequisites = true

For each parameter index, p, starting from 3rd to the end of the line:

Add file[i][p] to the end of course.prerequisites

End Loop

Else:

Set course.hasPrerequisites = false

If bst’s root is null:

Set root to new Node(course)

else:

BinarySearchTree method call to bst-> Insert(bst->root, course)

End Loop

End Function

*This is the BinarySearchTree method Insert(Node\* node, Course course):*

void BinarySearchTree::Insert(Node\* node, Course course)

If (node’s courseNumber is greater than the course’s courseNumber):

If node has no left node:

Create new node with course, and set node’s left ptr to new node

Else:

Call addNode(node->left, bid)

Else:

If node has no right node:

Create new node with course, and set node’s right ptr to new node

Else:

Call addNode(node->right, bid)

End function

*This function prints the course information by recursively transversing the tree in order and printing the course info. It utilizes the Search() function to find course prerequisite’s:*

Void BinarySearchTree::printCourses (Node\* node)

If (node is null):

Return

else :

Recursive function call to printCourses(node’s left ptr)

Print node’s course.getInformation()

If node’s course.hasPrerequisites:

For each string in node’s course.prerequisites:

Set tempCourse = Function call to Search(prerequisite)

If tempCourse is empty:

Return

Else:

Print tempCourse’s course.getinformation

End Loop

Recursive function call to printCourses(node’s right ptr)

End function

*This function searches the tree for a specific courseNumber and is used by printCourses():*

Course BinarySearchTree::Search (string courseNumber)

Set Node\* currentNode to root

While currentNode is not null:

If currentNode’s courseNumber is equal to searched courseNumber:

Return currentNode’s Course:

Else if currentNode’s courseNumber is larger than searched courseNumber:

Set currentNode to currentNode’s right ptr

Else:

Set currentNode to currentNode’s left ptr

End Loop

Return empty Course

End Function

Runtime Analysis for Insert():

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| If (node’s courseNumber is greater than the course’s courseNumber): | 1 | 1 | 1 |
| If node has no left node: | 1 | 1 | 1 |
| Create new node with course, and set node’s left ptr to new node | 1 | 1 | 1 |
| Else: | 1 | 1 | 1 |
| Call addNode(node->left, bid) | n | 1 | n |
| Else: | 1 | 1 | 1 |
| If node has no right node: | 1 | 1 | 1 |
| Create new node with course, and set node’s right ptr to new node | 1 | 1 | 1 |
| Else: | 1 | 1 | 1 |
| Call addNode(node->right, bid) | n | 1 | n |
| **Total Cost** | | | 2n + 8 |
| **Runtime** | | | O(n) |

Runtime Analysis for loading the file and creating course objects:

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Initialize file = parser(filePath) | 1 | 1 | 1 |
| For each row of data, i, in the file: | n | n | n^2 |
| If validateData(file, i) == true: | n^2 | n | n^3 |
| Create a course data structure | 1 | n | n |
| Set course.courseNumber to the first parameter | 1 | n | n |
| Set course.courseName to the second parameter | 1 | n | n |
| If there are three or more parameters on row i: | 1 | n | n |
| Set course.hasPrerequisites = true | 1 | n | n |
| For each parameter index, p, starting from 3rd to the end of the line: | 1 | n^2 | n^2 |
| Add file[i][p] to the end of course.prerequisites | 1 | n^2 | n^2 |
| Else: | 1 | n | n |
| Set course.hasPrerequisites = false | 1 | n | n |
| If bst’s root is null: | 1 | n | n |
| Set root to new Node(course) | 1 | n | n |
| else: | 1 | n | n |
| BinarySearchTree method call to bst-> Insert(bst->root, course) | n | n | n^2 |
| **Total Cost** | | | n^3+4n^2 + 10n + 1 |
| **Runtime** | | | O(n^3) |