**Project One – Part Two:**

**Evaluating Run-time Complexity of Various Data Structures**

**Note to Instructor:** In my pseudocode portion of the project, indentation was done using tabs but, to keep cost per line analysis on the same line as the pseudocode being analyzed, four spaces per level of indentation are used in this portion of the project

**Opening and Parsing a File Complexity Analysis**

Each data structure utilizes the [loadFile function](https://snhu-my.sharepoint.com/personal/jonathan_thomas9_snhu_edu/Documents/CS-300/Project%20One%20-%20Pseudocode.docx#loadFile) to load each line of a given file into a vector, “lines”. This vector will later be used to create Course objects and insert them into their respective data structure.

**Pseudocode Cost Per Line**

n = number of lines in the opened file

Vec<string> loadFile(string FILEPATH):

Vec<string> lines = [ ] {Time: O(1)–Space: O(1)--# Executions: 1}

open file (FILEPATH) {Time: O(1)–Space: O(1)--# Executions: 1}

if (file is opened): {Time: O(1)–Space: O(1)--# Executions: 1}

for (i=0; i < file.length; i++): {Time: O(n)–Space: O(1)--# Executions: n}

lines.append(file[i]) {Time: O(n)–Space: O(1)--# Executions: n}

return lines; {Time: O(1)–Space: O(n)--# Executions: 1}

else: {Time: O(1)–Space: O(1)--# Executions: max. 1}

print(“Error: file could not be opened”) {Time: O(1)–Space: O(1)--# Executions: max. 1}

throw FileNotOpenedError {Time: O(1)–Space: O(1)--# Executions: max. 1}

**Worst Case Complexity Analysis**

Time Complexity: O(n)

Space Complexity: O(n)

**loadValidCourseIds Function Complexity Analysis**

While the [loadValidCourseIds function](https://snhu-my.sharepoint.com/personal/jonathan_thomas9_snhu_edu/Documents/CS-300/Project%20One%20-%20Pseudocode.docx#loadValidCourseIds) is defined under the Vector Data Structure section of my pseudocode, each data structure utilizes this function for various reasons and therefore, it receives its own analysis section.

**Pseudocode Cost Per Line**

n = number of elements in the lines vector

m = number of elements separated by commas

Vec<string> loadValidCourseIds():

Vec<string> validIds = [ ] {Time: O(1)–Space: O(1)--# Executions: 1}

Vec<string> lines = loadFile(FILEPATH) {Time: O(n)–Space: O(n)--# Executions: 1}

for (line : lines): {Time: O(n)–Space: O(1)--# Executions: n}

Vec<string> splitLine = line.split(“,”) {Time: O(m)–Space: O(m)--# Executions: n}

validIds.append(splitLine[0]) {Time: O(n)–Space: O(1)--# Executions: n}

return validIds; {Time: O(1)–Space: O(n)--# Executions: 1}

**Worst Case Complexity Analysis**

Space Complexity is not O(n + m) because m will always be less than or equal to n. This is because no valid course can exist in the prerequisites list (m) without also being included in the list of all valid courses (n). If there exist any courses in the lines vector that have no prerequisites, n is automatically larger than m. Time Complexity is O(n\*m) because line.split is iterative and within a for loop. Nested iteration increases time complexity multiplicatively.

Time Complexity: O(n\*m)

Space Complexity: O(n)

**Vector Data Structure Complexity Analysis**

[loadCourses function](https://snhu-my.sharepoint.com/personal/jonathan_thomas9_snhu_edu/Documents/CS-300/Project%20One%20-%20Pseudocode.docx#loadCourses)

**Pseudocode Cost Per Line**

n = number of lines in the opened file

m = number of elements separated by commas

Vec<Course> loadCourses():

Vec<Course> courses = [ ] {Time: O(1)–Space: O(1)--# Executions: 1}

Vec<string> lines = loadFile(FILEPATH) {Time: O(n)–Space: O(n)--# Executions: 1}

Vec<string> validIds = loadValidCourseIds(lines) {Time: O(n\*m)–Space: O(n\*m)--# Executions: 1}

for (line : lines): {Time: O(n)–Space: O(1)--# Executions: n}

Vec<string> splitLine = line.split(“,”) {Time: O(m)–Space: O(m)--# Executions: n}

if splitLine.size == 2: {Time: O(1)–Space: O(1)--# Executions: max. n }

Course newCourse = Course() {Time: O(1)–Space: O(1)--# Executions: max. n }

newCourse.courseId = splitLine[0] {Time: O(1)–Space: O(1)--# Executions: max. n }

newCourse.courseName = splitLine[1] {Time: O(1)–Space: O(1)--# Executions: max. n }

courses.append(newCourse) {Time: O(n)–Space: O(1)--# Executions: max. n }

else if (splitLine.size > 2): {Time: O(1)–Space: O(1)--# Executions: max. n}

Course newCourse = Course() {Time: O(1)–Space: O(1)--# Executions: max. n}

newCourse.courseId = splitLine[0] {Time: O(1)–Space: O(1)--# Executions: max. n}

newCourse.courseName = splitLine[1] {Time: O(1)–Space: O(1)--# Executions: max. n}

for (i = 2; i < splitLine.size; i++): {Time: O(m)–Space: O(m)--# Executions: max. n\*m}

for (id : validIds): {Time: O(n)–Space: O(n)--# Executions: max. n\*m\*n}

if (splitLine[i] == id): {Time: O(1)–Space: O(1)--# Executions: max. n\*m\*n}

newCourse.prereqCourses.append(splitLine[i]) {Time: O(n)–Space: O(1)--# Executions: max. n\*m\*n}

courses.append(newCourse) {Time: O(n)–Space: O(1)--# Executions: max. n\*m\*n }

return courses; {Time: O(1)–Space: O(1)--# Executions: max. n}

**Worst Case Complexity Analysis**

Time Complexity: O(m\*n^2)

Space Complexity: O(n)

**Hash Table Data Structure Complexity Analysis**

While the Hash Table data structure does not have an explicitly defined function to load file information into Course objects, the logic defined in menu option 1 of the Hash Table menu display exactly matches the logic of the [loadCourses function](https://snhu-my.sharepoint.com/personal/jonathan_thomas9_snhu_edu/Documents/CS-300/Project%20One%20-%20Pseudocode.docx#loadCourses) defined in the Vector data structure section with the exception that Course objects are inserted to the Hash Table using “courses.Insert(newCourse)” as opposed to “courses.append(newCourse)”. Because of this, we can derive the time and space complexity of the Hash Table equivalent of the loadCourses function by finding the time and space complexity of the Hash Table’s [Insert function](https://snhu-my.sharepoint.com/personal/jonathan_thomas9_snhu_edu/Documents/CS-300/Project%20One%20-%20Pseudocode.docx#htInsert).

**Pseudocode Cost Per Line**

n = number of Nodes in a given bucket (worst case – the fullest bucket)

uint HashTable::hash(uint key):

return key % tableSize; {Time: O(1)–Space: O(1)--# Executions: 1}

void HashTable::Insert(Course course):

Node newNode = Node(course) {Time: O(1)–Space: O(1)--# Executions: 1}

// The time complexity of the atoi function is O(m) where m is the number of characters in the string being converted because in its worst case, it iterates over every character in a string to convert it to an integer

uint key = hash(atoi(course.courseId)) {Time: O(m)–Space: O(1)--# Executions: 1}

newNode.key = key {Time: O(1)–Space: O(1)--# Executions: 1}

existingNode = nodes[key] {Time: O(1)–Space: O(1)--# Executions: 1}

if (existingNode.key == UINT\_MAX): {Time: O(1)–Space: O(1)--# Executions: 1}

existingNode = newNode {Time: O(1)–Space: O(1)--# Executions: 1}

else: {Time: O(1)–Space: O(1)--# Executions: 1}

while (existingNode.next != nullptr): {Time: O(n)–Space: O(1)--# Executions: n}

existingNode = existingNode.next {Time: O(1)–Space: O(1)--# Executions: n}

existingNode.next = newNode {Time: O(1)–Space: O(1)--# Executions: 1}

**Worst Case Complexity Analysis – HashTable::Insert**

Time Complexity: O(n)

Space Complexity: O(1)

**Worst Case Complexity Analysis**

Because the Hash Table Insert function has a worst-case time complexity of O(n) and a space complexity of O(1), the worst-case time and space complexity of the HashTable equivalent of the [loadCourses function](https://snhu-my.sharepoint.com/personal/jonathan_thomas9_snhu_edu/Documents/CS-300/Project%20One%20-%20Pseudocode.docx#loadCourses) is:

Time Complexity: O(m\*n^2)

Space Complexity: O(n)

Where *m* is the number of elements in the largest splitLine vector (in this case, this translates to the course with the most prerequisite courses), and *n* is the number of lines in the file containing course information.

**Binary Search Tree Data Structure Complexity Analysis**

Like the Hash Table data structure, the Binary Search Tree data structure does not have an explicitly defined function to load file data into Course objects. Also like the Hash Table data structure, the logic defined in menu option 1 in the Binary Search Tree menu mirrors that of the [loadCourses function](https://snhu-my.sharepoint.com/personal/jonathan_thomas9_snhu_edu/Documents/CS-300/Project%20One%20-%20Pseudocode.docx#loadCourses) from the Vector data structure with the only change being that “courses.Insert(newCourse)” is used to insert Course objects into the data structure rather than “courses.append(newCourse)”. Because of this, we can, like above, derive the time and space complexity of the Binary Search Tree equivalent of the loadCourses function by first finding the time and space complexity of the Binary Search Tree’s [Insert function](https://snhu-my.sharepoint.com/personal/jonathan_thomas9_snhu_edu/Documents/CS-300/Project%20One%20-%20Pseudocode.docx#bstInsert).

**Pseudocode Cost per Line**

The Insert function makes extensive use of the Node pointers defined within the [Node struct](https://snhu-my.sharepoint.com/personal/jonathan_thomas9_snhu_edu/Documents/CS-300/Project%20One%20-%20Pseudocode.docx#bstNode) which itself is defined within the Binary Search Tree class.

void Insert(Course newCourse):

Node\* newNode = Node(newCourse) {Time: O(1)–Space: O(1)--# Executions: 1}

if (root == nullptr): {Time: O(1)–Space: O(1)--# Executions: 1}

root = newNode {Time: O(1)–Space: O(1)--# Executions: 1}

else: {Time: O(1)–Space: O(1)--# Executions: 1}

Node\* cur = root {Time: O(1)–Space: O(1)--# Executions: 1}

while (cur != nullptr): {Time: O(n)–Space: O(1)--# Executions: n}

if (newCourse.courseId < cur.course.courseId): {Time: O(1)–Space: O(1)--# Executions: max. n}

if (cur.left == nullptr): {Time: O(1)–Space: O(1)--# Executions: 1}

cur.left = newNode; {Time: O(1)–Space: O(1)--# Executions: 1}

cur = nullptr; {Time: O(1)–Space: O(1)--# Executions: 1}

else: {Time: O(1)–Space: O(1)--# Executions: max. n}

cur = cur.left; {Time: O(1)–Space: O(1)--# Executions: max. n}

else: {Time: O(1)–Space: O(1)--# Executions: max. n}

if (cur.right == nullptr): {Time: O(1)–Space: O(1)--# Executions: 1}

cur.right = newNode {Time: O(1)–Space: O(1)--# Executions: 1}

cur = nullptr {Time: O(1)–Space: O(1)--# Executions: 1}

else: {Time: O(1)–Space: O(1)--# Executions: max. n}

cur = cur.right {Time: O(1)–Space: O(1)--# Executions: max. n}

**Worst Case Complexity Analysis – BinarySearchTree::Insert**

Time Complexity: O(n)

Space Complexity: O(1)

**Worst Case Complexity Analysis**

Because the Binary Search Tree Insert function has a worst-case time complexity of O(n) and a space complexity of O(1), the worst-case time and space complexity of the Binary Search Tree equivalent of the [loadCourses function](https://snhu-my.sharepoint.com/personal/jonathan_thomas9_snhu_edu/Documents/CS-300/Project%20One%20-%20Pseudocode.docx#loadCourses) is:

Time Complexity: O(m\*n^2)

Space Complexity: O(n)

Where *m* is the number of elements in the largest splitLine vector (in this case, this translates to the course with the most prerequisite courses), and *n* is the number of lines in the file containing course information.

**Advantages and Disadvantages of Each Data Structure**

**Vector Data Structure**

Advantages:

* simple
* easy to understand
* accessing the data of any given element has a worst-case time complexity of O(1)

Disadvantages:

* search, insert, and delete methods all have average time complexity of O(n)

**Hash Table Data Structure**

Advantages:

* search, insert, and delete methods all have an average time complexity of O(1)

Disadvantages:

* complex data structure that may be hard to understand for the inexperienced
* no consistent method for accessing the data of a given element – custom function must be implemented depending on how the hash key is computed
* no easy method for accessing each element in the data structure in a sorted order

**Binary Search Tree Data Structure**

Advantages:

* data is organized in an intuitive way
* accessing the data of a given element and the search, insert, and delete methods all have an average time complexity of O(log(n))

Disadvantages:

* search, insert, and delete methods are not as fast as a Hash Table’s methods

**Recommendation**

To begin considering a recommendation, the full requirements of the program must be thoroughly examined.

This program must:

1. Load data from a file containing information about courses and their prerequisites, create Course objects from this data, and finally insert these Course objects into their respective data structure
2. Print all courses in alphanumeric order based on the course IDs
3. Print information about a specific course given a course ID

On point one, every data structure that has been analyzed in this project has a worst-case time complexity of *O(m\*n^2)* where *m* is the number of elements in the largest splitLine vector (in this case, this translates to the course with the most prerequisite courses), and *n* is the number of lines in the file containing course information. This time complexity arises from the nested for-loops used to ensure that all courses are inserted and that all prerequisites for any given course are valid and is unrelated to the methods used to insert an element into the data structure. Because of this, this time complexity will be unchanged even when the various insertion methods are not performing with their worst-case time complexities.

For point two, the three data structures vary significantly with regards to their worst-case time complexities.

* For the Vector data structure, the vector of courses must first be sorted according to course ID which is done using Quicksort which has a worst-case time complexity of *O(n^2)*.
* The Hash Table data structure, which does not store data in a way that can be easily sorted, utilizes the vector of valid course IDs created by the [loadValidCourseIds function](https://snhu-my.sharepoint.com/personal/jonathan_thomas9_snhu_edu/Documents/CS-300/Project%20One%20-%20Pseudocode.docx#loadValidCourseIds) which is sorted with Quicksort before passing the now sorted course IDs in order to the Hash Table PrintCourse function which itself calls the Search function. Overall, the worst-case time complexity for the Hash Table to print all course information in order is *O(n^2)* due to the use of Quicksort.
* The Binary Search Tree excels in this metric with its own established method called InOrderTraversal, which, due to the way data is organized in this data structure, has a worst-case time complexity of *O(n)* which is the best-case scenario when accessing every element in a data structure.

On point three, each of the three data structures utilizes its own implemented search method to match a Course object to a course ID before printing information about the course. This means that in each of the three data structures, the search function is the determining factor for the time complexity of printing information about a single course. The average time complexity of the search method in each of the three data structures is:

* Vector: O(n)
* Hash Table: O(1)
* Binary Search Tree: O(log(n))

Given the above analysis, my recommendation for the data structure that should be used in my code is the Binary Search Tree.

While the Hash Table has a search function with an impressive average time complexity, the determining factor was that the worst-case time complexity to print all course information is *O(n^2)* which, with a large enough data set, will slow execution times to a crawl while the worst-case time complexity for the same operation using a Binary Search Tree is *O(n)*, the best performance that could be asked for when visiting every available element in a data structure.