# CS 300 Pseudocode Document

//Hash Table Puseodocode

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

// Generate hash index for given course number

Set index = hashFunction(courseNumber)

// Search for course in the list at the hash index

for each course in courses.table[index] {

if course.courseNumber is equal to courseNumber {

// Print course information

Print "Course Number: " + course.courseNumber

Print "Course Title: " + course.courseTitle

// Check for any prerequisites

if length of course.prerequisites > 0 {

Print "Prerequisites: "

// Print each prerequisite

for each prerequisite in course.prerequisites {

Print prerequisite

}

} else {

Print “No prerequisites.”

}

Return

}

// If course is not found, output error message

Print "Course " + courseNumber + " not found."

}

void loadFileData(HashTable<Course> courses, String filename) {

// Open the file

OPEN file with name filename

// Read each line from the file

for each line in file {

// Parse line to extract course data

Split line into courseNumber, courseTitle, prerequisites

// Create Course object

Course course = new Course(courseNumber, courseTitle, prerequisites)

// Generate hash index for course number

Set index = hashFunction(courseNumber)

// Add Course to hash table at the created index

courses.table[index].append(course)

}

CLOSE file

}

void printSortedCourses(HashTable<Course> courses) {

// Create a list to store courses

List<Course> courseList

// Iterate over hash table to collect all courses

for each bucket in courses.table {

for each course in bucket {

courseList.append(course)

}

}

// Sort the list of courses alphanumerically by course number

QuickSort(courseList)

// Print each course in sorted order

for each course in courseList {

Print "Course Number: " + course.courseNumber + ", Course Title: " + course.courseTitle

}

}

void menu() {

HashTable<Course> courses

while true {

// Display menu options

Print "1. Load Data"

Print "2. Print Sorted Courses"

Print "3. Print Course Details"

Print "9. Exit"

// Read user option

int option = READ user input

if option == 1 {

// Load file data into the data structure

Print "Enter filename:"

String filename = READ user input

loadFileData(courses, filename)

} else if option == 2 {

// Print an alphanumerically ordered list of all the courses

printSortedCourses(courses)

} else if option == 3 {

// Print the course title and the prerequisites for any individual course

Print "Enter course number:"

String courseNumber = READ user input

searchCourse(courses, courseNumber)

} else if option == 9 {

// Exit the program

EXIT

} else {

Print "Invalid option. Please try again."

}

}

}

}

## Example Runtime Analysis

When you are ready to analyze the runtime for the Project One data structures for which you created the pseudocode, use the example chart below to support your work. This particular example is for printing course information when using the vector data structure. As a reminder, this is the same pairing that was bolded in the pseudocode from the first part of this document. The example only covers the search function for the vector structure. You do not have to complete your runtime analysis until Project One. However, working on your analysis now may help you understand the changes as you complete the milestones. Don’t forget to include your charts in Project One. You will submit Project One in Module Six.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **for each prerequisite of the course** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

Starting with vectors, vectors offer the advantage of simplicity and contiguous memory allocation leading to O(1) access time for elements. Though, when inserting or deleting elements, the performance can degrade to O(n) in worst case. Reading data and creating objects with vectors generally run in O(n) time due to each element being accessed sequentially. This is efficient but could lead to increased memory usage if the vector needs to dynamically resize. Looking at the hash table, it provides an average-case time complexity of o(1) for insertions. Deletions, and lookups that make it extremely efficient for handling large datasets. However, the worse-case scenario, often caused by collisions, can degrade the performance to O(n). Hash tables require more memory overhead due to their need to handle collisions and maintain efficient hashing.

Lastly, the Binary Search Tree (BST) offers a balanced approach, with search, insertion, and deletion operations generally run in O(log n) time if the tree is balanced. However, the worst-case such as when a skewed tree, the time complexity can downgrade to O(n). Binary Search Trees provide an ordered structure which can have its advantages for certain operations but require careful balancing to maintain their efficiency. Given these evaluations, I would recommend using the hash table for this project. The hash table’s average-case O(1) time complexity for most operations aligns well with the need for efficient data retrieval and storage on an application. Despite its higher memory overhead, the benefits of fast access times and robustness against large datasets make it the most suitable choice based on the Big O analysis and project requirements.