Let’s begin by setting up our pseudocode that can be used to create the menu for any of our three data structures:

**Pseudocode for a menu:**

// Main menu procedure

procedure displayMenu():

while true:

print "Menu Options:"

print "1. Load the file data into the data structure"

print "2. Print an alphanumerically ordered list of all the courses"

print "3. Print the course title and the prerequisites for a specific course"

print "9. Exit the program"

print "Enter your choice: "

read userChoice

if userChoice == 1:

print "Enter the filename: "

read filename

courseData = readFile(filename)

courseStructure = createCourseObjects(courseData)

print "Data loaded successfully."

elif userChoice == 2:

if courseStructure is not empty:

printAlphanumericCourses(courseStructure)

else:

print "Error: Load the course data first."

elif userChoice == 3:

if courseStructure is not empty:

print "Enter the course number: "

read courseNumber

searchCourse(courseStructure, courseNumber)

else:

print "Error: Load the course data first."

elif userChoice == 9:

print "Exiting program."

break

else:

print "Invalid choice. Please try again."

// Procedure to print an alphanumerically ordered list of all courses

procedure printAlphanumericCourses(courseStructure):

sortedCourses = sortCourses(courseStructure)

for each course in sortedCourses:

print "Course Number:", course.courseNumber, "Course Name:", course.courseName

// Procedure to sort courses alphanumerically

procedure sortCourses(courseStructure):

create empty list sortedCourses

if courseStructure is a list or vector:

sortedCourses = sort courseStructure by courseNumber

elif courseStructure is a hash table:

extract all values from hash table and sort by courseNumber

elif courseStructure is a binary search tree:

perform in-order traversal to get sorted list of courses

return sortedCourses

**Psuedocode for printing out a list of courses in alphanumeric order:**

// Procedure to print an alphanumerically ordered list of all courses

procedure printAlphanumericCourses(courseStructure):

sortedCourses = sortCourses(courseStructure)

for each course in sortedCourses:

print "Course Number:", course.courseNumber, "Course Name:", course.courseName

// Procedure to sort courses alphanumerically based on the data structure

procedure sortCourses(courseStructure):

create empty list sortedCourses

if courseStructure is a list or vector:

sortedCourses = sort courseStructure by courseNumber

elif courseStructure is a hash table:

allCourses = extract all values from hash table

sortedCourses = sort allCourses by courseNumber

elif courseStructure is a binary search tree:

sortedCourses = inOrderTraversal(courseStructure)

return sortedCourses

// Helper procedure to perform in-order traversal for binary search tree

procedure inOrderTraversal(treeNode):

create empty list result

if treeNode is not null:

result = inOrderTraversal(treeNode.left)

add treeNode.course to result

result += inOrderTraversal(treeNode.right)

return result

**Runtime Analysis of Vector Pseudocode:**

| Line of Code | Cost per Line | Number of Executions | Total Cost |
| --- | --- | --- | --- |
| courses = [] | O(1) | 1 | O(1) |
| with open(filename, 'r') as file: | O(1) | 1 | O(1) |
| for line in file: | O(1) | n | O(n) |
| tokens = line.strip().split(',') | O(1) | n | O(n) |
| if len(tokens) < 2: | O(1) | n | O(n) |
| print("Error: Line does not have enough parameters") | O(1) | k | O(k) |
| continue | O(1) | k | O(k) |
| course\_number = tokens[0] | O(1) | n | O(n) |
| course\_name = tokens[1] | O(1) | n | O(n) |
| prerequisites = tokens[2:] | O(1) | n | O(n) |
| courses.append((course\_number, course\_name, prerequisites)) | O(1) | n | O(n) |
| course\_numbers = {course[0] for course in courses} | O(n) | 1 | O(n) |
| for course in courses: | O(1) | n | O(n) |
| for prerequisite in course[2]: | O(1) | n \* m | O(nm) |
| if prerequisite not in course\_numbers: | O(1) | n \* m | O(nm) |
| print(f"Error: Prerequisite {prerequisite} not completed for course {course[0]}") | O(1) | l | O(l) |
| return courses | O(1) | 1 | O(1) |
| Total Cost Overall | O(1) | 1 | O(nm) |

**Runtime Analysis of Hash Table Pseudocode:**

| Line of Code | Cost per Line | Number of Executions | Total Cost |
| --- | --- | --- | --- |
| courses = {} | O(1) | 1 | O(1) |
| with open(filename, 'r') as file: | O(1) | 1 | O(1) |
| for line in file: | O(1) | n | O(n) |
| tokens = line.strip().split(',') | O(1) | n | O(n) |
| if len(tokens) < 2: | O(1) | n | O(n) |
| print("Error: Line does not have enough parameters") | O(1) | k | O(k) |
| continue | O(1) | k | O(k) |
| course\_number = tokens[0] | O(1) | n | O(n) |
| course\_name = tokens[1] | O(1) | n | O(n) |
| prerequisites = tokens[2:] | O(1) | n | O(n) |
| courses[course\_number] = (course\_name, prerequisites) | O(1) | n | O(n) |
| for course\_number, (course\_name, prerequisites) in courses.items(): | O(1) | n | O(n) |
| for prerequisite in prerequisites: | O(1) | n \* m | O(nm) |
| if prerequisite not in courses: | O(1) | n \* m | O(nm) |
| print(f"Error: Prerequisite {prerequisite} not found for course {course\_number}") | O(1) | l | O(l) |
| return courses | O(1) | 1 | O(1) |
| Total Cost Overall | O(1) | 1 | O(nm) |

**Runtime Analysis of Tree Pseudocode:**

| Line of Code | Cost per Line | Number of Executions | Total Cost |
| --- | --- | --- | --- |
| courses = {} | O(1) | 1 | O(1) |
| with open(filename, 'r') as file: | O(1) | 1 | O(1) |
| for line in file: | O(1) | n | O(n) |
| tokens = line.strip().split(',') | O(1) | n | O(n) |
| if len(tokens) < 2: | O(1) | n | O(n) |
| print("Error: Line does not have enough parameters") | O(1) | k | O(k) |
| continue | O(1) | k | O(k) |
| courseNumber = tokens[0] | O(1) | n | O(n) |
| courseName = tokens[1] | O(1) | n | O(n) |
| prerequisites = tokens[2:] | O(1) | n | O(n) |
| courses[courseNumber] = (courseName, prerequisites) | O(1) | n | O(n) |
| for courseNumber in courses: | O(1) | n | O(n) |
| for prerequisite in courses[courseNumber].prerequisites: | O(1) | n \* m | O(nm) |
| if prerequisite not in courses: | O(1) | n \* m | O(nm) |
| print("Error: Prerequisite", prerequisite, "not found for course", courseNumber) | O(1) | l | O(l) |
| return courses | O(1) | 1 | O(1) |
| course\_dict = {} | O(1) | 1 | O(1) |
| for each courseNumber in courseData: | O(1) | n | O(n) |
| create new Course object course | O(1) | n | O(n) |
| course.courseNumber = courseNumber | O(1) | n | O(n) |
| course.courseName = courseData[courseNumber][0] | O(1) | n | O(n) |
| course.prerequisites = courseData[courseNumber][1] | O(1) | n | O(n) |
| courseTree.insert(course) | O(n) | n | O(n^2) |
| return courseTree | O(1) | 1 | O(1) |
| searchTree(courseTree.root, courseNumber) | O(n) | 1 | O(n) |
| if course is not null: | O(1) | 1 | O(1) |
| print "Course Number:", course.courseNumber | O(1) | 1 | O(1) |
| print "Course Name:", course.courseName | O(1) | 1 | O(1) |
| print "Prerequisites:" | O(1) | 1 | O(1) |
| if length of course.prerequisites == 0: | O(1) | 1 | O(1) |
| print "None" | O(1) | 1 | O(1) |
| for each prerequisite in course.prerequisites: | O(1) | m | O(m) |
| print prerequisite | O(1) | m | O(m) |
| print "Course not found" | O(1) | 1 | O(1) |
| if courseTree.root is not null: | O(1) | 1 | O(1) |
| printInOrder(courseTree.root) | O(n) | 1 | O(n) |
| printInOrder(node.leftChild) | O(1) | n | O(n) |
| print "Course Number:", node.course.courseNumber | O(1) | n | O(n) |
| print "Course Name:", node.course.courseName | O(1) | n | O(n) |
| print "Prerequisites:" | O(1) | n | O(n) |
| if length of node.course.prerequisites == 0: | O(1) | n | O(n) |
| print "None" | O(1) | n | O(n) |
| for each prerequisite in node.course.prerequisites: | O(1) | n \* m | O(nm) |
| courses = {} | O(1) | 1 | O(1) |
| with open(filename, 'r') as file: | O(1) | 1 | O(1) |
| for line in file: | O(1) | n | O(n) |
| tokens = line.strip().split(',') | O(1) | n | O(n) |
| if len(tokens) < 2: | O(1) | n | O(n) |
| print("Error: Line does not have enough parameters") | O(1) | k | O(k) |
| continue | O(1) | k | O(k) |
| courseNumber = tokens[0] | O(1) | n | O(n) |
| courseName = tokens[1] | O(1) | n | O(n) |
| prerequisites = tokens[2:] | O(1) | n | O(n) |
| courses[courseNumber] = (courseName, prerequisites) | O(1) | n | O(n) |
| for courseNumber in courses: | O(1) | n | O(n) |
| for prerequisite in courses[courseNumber].prerequisites: | O(1) | n \* m | O(nm) |
| if prerequisite not in courses: | O(1) | n \* m | O(nm) |
| print("Error: Prerequisite", prerequisite, "not found for course", courseNumber) | O(1) | l | O(l) |
| return courses | O(1) | 1 | O(1) |
| course\_dict = {} | O(1) | 1 | O(1) |
| for each courseNumber in courseData: | O(1) | n | O(n) |
| create new Course object course | O(1) | n | O(n) |
| course.courseNumber = courseNumber | O(1) | n | O(n) |
| course.courseName = courseData[courseNumber][0] | O(1) | n | O(n) |
| course.prerequisites = courseData[courseNumber][1] | O(1) | n | O(n) |
| courseTree.insert(course) | O(n) | n | O(n^2) |
| return courseTree | O(1) | 1 | O(1) |
| searchTree(courseTree.root, courseNumber) | O(n) | 1 | O(n) |
| if course is not null: | O(1) | 1 | O(1) |
| print "Course Number:", course.courseNumber | O(1) | 1 | O(1) |
| print "Course Name:", course.courseName | O(1) | 1 | O(1) |
| print "Prerequisites:" | O(1) | 1 | O(1) |
| if length of course.prerequisites == 0: | O(1) | 1 | O(1) |
| print "None" | O(1) | 1 | O(1) |
| for each prerequisite in course.prerequisites: | O(1) | m | O(m) |
| print prerequisite | O(1) | m | O(m) |
| print "Course not found" | O(1) | 1 | O(1) |
| if courseTree.root is not null: | O(1) | 1 | O(1) |
| printInOrder(courseTree.root) | O(n) | 1 | O(n) |
| printInOrder(node.leftChild) | O(1) | n | O(n) |
| print "Course Number:", node.course.courseNumber | O(1) | n | O(n) |
| print "Course Name:", node.course.courseName | O(1) | n | O(n) |
| print "Prerequisites:" | O(1) | n | O(n) |
| if length of node.course.prerequisites == 0: | O(1) | n | O(n) |
| print "None" | O(1) | n | O(n) |
| for each prerequisite in node.course.prerequisites: | O(1) | n \* m | O(nm) |
| print prerequisite | O(1) | n \* m | O(nm) |
| printInOrder(node.rightChild) | O(1) | n | O(n) |
| Total Cost Overall | O(1) | 1 | O(n^2 + nm) |

In each of the above Runtime Analysis Charts: n = the number of lines in the file, m = the average number of prerequisites that each course has, k = the number of lines that have errors, and l = the number of prerequisite errors.

**Advantages and Disadvantages of each data structure:**

Vector

Advantages:

1. Dynamic Size – Vectors can resize easily making it easier to manage when elements can change in size.
2. Random Access – Efficient to regain something or update elements.
3. Cache-Friendly – Elements are stored in a manner that is great for cache performance.

Disadvantages:

1. Insertion/Deletion Overhead – Inserting and deleting can be very impactful to performance and require moving other elements.
2. Memory Reallocation – When resizing, the memory may need to be reallocated, again affecting performance.
3. Lack of Sorted Data – Data isn’t sorted unless done directly by the programmer.

Hash Table

Advantages:

1. Average Constant Time Operations – The hashing makes inserting, deleting, and searching constant.
2. Efficient Lookups – Great for fast needs to recollect an element or insert/delete.
3. Flexibility – Very versatile for different key types and applications.

Disadvantages:

1. Worst-Case Performance – Upon collisions, elements are stored in a linked list.
2. Non-Sequential Access – Cannot iterate in a sorted order.
3. Space Overhead – Requores extra room for hash table and collision handling mechanisms.

Binary Tree

Advantages:

1. Sorted Data – Maintains elements in a sorted order.
2. Balanced Trees – Ensures a balanced time complexity to insert, delete, and search.
3. Versatile Traversals – Supports many different traversal methods making it very versatile.

Disadvantages:

1. Complexity – More complex and difficult to implement than vectors or hash tables.
2. Space Overhead – Requires extra memory and space consumption.
3. Average Performance – In an unbalanced tree, the performance can fall similarly to a linked list.

**Recommendation**

Based on my Big O analysis and the requirements of the advisors, I believe that the Binary Tree data structure is the best choice to move forward with. The binary search tree is the best option for several reasons. First, the BST can eliminate the separate sorting step for vectors and hash tables by using an in-order traversal to print alphanumerically. The BST will also have efficient insertions, lookups, and deletions compared to that of hash tables and vector’s time complexities. It will also use memory in an efficient manner. Finally, the BST would easily be scaled to a larger system or potentially smaller, if necessary, in an efficient manner. In conclusion, the binary search tree is the most suitable data structure for ABCU’s Academic Advisors and their needs due to its efficiency and different functionalities over vectors or hash tables.