**ABCU Advising Program: Pseudocode and Runtime Analysis**

**CS‑300 Project One**

**Ian Brown**

**Pseudocode Design**

The advising program must load course data from a file, create course objects, and support three main operations: printing all courses in alphanumeric order, printing the title and prerequisites for a given course, and exiting the program. The pseudocode has been designed for three data structures: vector, hash table, and binary search tree.

The course object contains a course number, title, and a list of prerequisites. A parsing function reads each line of the input file, validates formatting, and constructs a course object. The file loading function iterates through the file line by line, calling the parser and inserting each course into the chosen data structure.

For the vector implementation, courses are stored sequentially. Searching for a course requires scanning the vector until a match is found. Printing all courses in alphanumeric order requires sorting the vector before display. For the hash table implementation, courses are stored by key, allowing constant‑time lookups on average. Printing all courses requires extracting values and sorting them. For the binary search tree implementation, courses are inserted by course number, allowing logarithmic search on average and an in‑order traversal to print all courses in sorted order without additional sorting.

The menu system is consistent across all three implementations. Option 1 loads the file into the data structure. Option 2 prints the full course list in alphanumeric order. Option 3 prints the title and prerequisites for a given course. Option 9 exits the program.

**Runtime Analysis**

The runtime analysis considers file reading, course object creation, and insertion into the data structure. Assume there are *n* courses and each course has up to *k* prerequisites.

| **Data Structure** | **Build Time** | **Search Time** | **Print All Courses** | **Notes** |
| --- | --- | --- | --- | --- |
| Vector | O(n) load, O(1) insert | O(n) linear search | O(n log n) sort + O(n) print | Simple but inefficient for lookups |
| Hash Table | O(n) average load | O(1) average search | O(n log n) sort + O(n) print | Excellent for lookups, unordered by default |
| Binary Search Tree | O(n log n) average load, O(n²) worst | O(log n) average, O(n) worst | O(n) in‑order traversal | Naturally ordered, efficient if balanced |

**Evaluation**

The vector is the simplest to implement and memory efficient, but it requires linear searches and repeated sorting. The hash table provides the fastest single‑course lookups, but it does not maintain order, so printing all courses requires sorting. The binary search tree provides both efficient searches and naturally ordered output, though it is more complex to implement and can degrade if unbalanced.

**Recommendation**

The binary search tree is the best choice for the final implementation. It balances the advisor’s two requirements: efficient single‑course lookups and efficient ordered printing of the entire course list. A balanced tree such as an AVL or Red‑Black tree ensures that worst‑case performance remains acceptable. While the hash table is strong for lookups, the tree’s ability to produce a sorted list in O(n) time makes it the most appropriate structure for this application.

**Pseudocode for Menu and Course Printing**

function mainMenu()

print "1. Load data file"

print "2. Print course list (alphanumeric)"

print "3. Print course information"

print "9. Exit program"

while true

choice = getUserInput()

if choice == 1

path = prompt("Enter file path:")

loadFile(path) into chosen data structure

print "Data successfully loaded."

else if choice == 2

printSortedCourses()

else if choice == 3

courseNumber = prompt("Enter course number:")

printCourseInfo(courseNumber)

else if choice == 9

print "Exiting program."

break

else

print "Invalid selection. Please try again."

**Pseudocode: Printing Courses in Alphanumeric Order**

**Vector Implementation**

function printSortedCourses\_Vector(courses : Vector<Course>)

if courses is empty

print "No courses available."

return

sortedCourses = copy(courses)

sort(sortedCourses by courseNumber ascending)

for each course in sortedCourses

print course.courseNumber + ": " + course.title

**Hash Table Implementation**

function printSortedCourses\_Hash(coursesHT : HashTable<String, Course>)

if coursesHT is empty

print "No courses available."

return

tempVector = new Vector<Course>()

for each entry in coursesHT

tempVector.push(entry.value)

sort(tempVector by courseNumber ascending)

for each course in tempVector

print course.courseNumber + ": " + course.title

**Binary Search Tree Implementation**

function printSortedCourses\_Tree(coursesBST : Tree<String, Course>)

if coursesBST is empty

print "No courses available."

return

inOrderTraverse(coursesBST.root)

function inOrderTraverse(node)

if node is null

return

inOrderTraverse(node.left)

print node.key + ": " + node.value.title

inOrderTraverse(node.right)