# ABCU Advising Program – Final Pseudocode and Runtime Analysis

## Pseudocode

### Vector Implementation

Load Courses

FUNCTION LoadCoursesFromFile\_Vector(filename)  
 OPEN file with filename  
 IF file cannot be opened  
 PRINT "Error: File not found."  
 RETURN  
  
 WHILE NOT end of file  
 READ line  
 SPLIT line by "," into tokens  
 courseNumber = tokens[0]  
 courseTitle = tokens[1]  
  
 IF courseNumber is empty OR courseTitle is empty  
 PRINT "Error: Missing course number or title."  
 CONTINUE  
  
 CREATE new Course object  
 Course.number = courseNumber  
 Course.title = courseTitle  
 Course.prerequisites = EMPTY LIST  
  
 FOR i = 2 to length(tokens) - 1  
 ADD tokens[i] to Course.prerequisites  
  
 APPEND Course to coursesVector  
 CLOSE file  
END FUNCTION

Print All Courses (Sorted)

FUNCTION PrintCourses\_Vector()  
 SORT coursesVector by Course.number (ascending alphanumeric)  
 FOR each Course in coursesVector  
 PRINT Course.number + ", " + Course.title  
END FUNCTION

Print Specific Course Info

FUNCTION PrintCourseInfo\_Vector(courseNumber)  
 FOR each Course in coursesVector  
 IF Course.number == courseNumber  
 PRINT Course.number + ", " + Course.title  
 IF Course.prerequisites NOT empty  
 PRINT "Prerequisites: " + join(Course.prerequisites, ", ")  
 ELSE  
 PRINT "No prerequisites."  
 RETURN  
 PRINT "Course not found."  
END FUNCTION

### Hash Table Implementation

Load Courses

FUNCTION LoadCoursesFromFile\_Hash(filename)  
 OPEN file with filename  
 WHILE NOT end of file  
 READ line  
 SPLIT line by "," into tokens  
 courseNumber = tokens[0]  
 courseTitle = tokens[1]  
  
 CREATE new Course object  
 Course.number = courseNumber  
 Course.title = courseTitle  
 Course.prerequisites = EMPTY LIST  
  
 FOR i = 2 to length(tokens) - 1  
 ADD tokens[i] to Course.prerequisites  
  
 INSERT Course into hashTable with key = courseNumber  
 CLOSE file  
END FUNCTION

Print All Courses

FUNCTION PrintCourses\_Hash()  
 keys = GET all keys from hashTable  
 SORT keys alphanumerically  
 FOR each key in keys  
 PRINT hashTable[key].number + ", " + hashTable[key].title  
END FUNCTION

Print Specific Course Info

FUNCTION PrintCourseInfo\_Hash(courseNumber)  
 IF hashTable contains courseNumber  
 course = hashTable[courseNumber]  
 PRINT course.number + ", " + course.title  
 IF course.prerequisites NOT empty  
 PRINT "Prerequisites: " + join(course.prerequisites, ", ")  
 ELSE  
 PRINT "No prerequisites."  
 ELSE  
 PRINT "Course not found."  
END FUNCTION

### BST Implementation

Load Courses

FUNCTION LoadCoursesFromFile\_BST(filename)  
 OPEN file with filename  
 WHILE NOT end of file  
 READ line  
 SPLIT line by "," into tokens  
 courseNumber = tokens[0]  
 courseTitle = tokens[1]  
  
 CREATE new Course object  
 Course.number = courseNumber  
 Course.title = courseTitle  
 Course.prerequisites = EMPTY LIST  
  
 FOR i = 2 to length(tokens) - 1  
 ADD tokens[i] to Course.prerequisites  
  
 INSERT Course into BST using courseNumber as key  
 CLOSE file  
END FUNCTION

Print All Courses (In-Order Traversal)

FUNCTION PrintCourses\_BST()  
 CALL InOrderTraversal(BST.root)  
END FUNCTION  
  
FUNCTION InOrderTraversal(node)  
 IF node == NULL  
 RETURN  
 InOrderTraversal(node.left)  
 PRINT node.course.number + ", " + node.course.title  
 InOrderTraversal(node.right)  
END FUNCTION

Print Specific Course Info

FUNCTION PrintCourseInfo\_BST(courseNumber)  
 node = BST.Search(courseNumber)  
 IF node != NULL  
 PRINT node.course.number + ", " + node.course.title  
 IF node.course.prerequisites NOT empty  
 PRINT "Prerequisites: " + join(node.course.prerequisites, ", ")  
 ELSE  
 PRINT "No prerequisites."  
 ELSE  
 PRINT "Course not found."  
END FUNCTION

## Runtime Analysis

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data Structure | Load Time (n courses) | Search Time | Print Sorted List | Memory Usage |
| Vector | O(n) read + 0(1) amortized insert | O(n) linear search | O(n log n) (must sort) | Contiguous memory; minimal overhead |
| Hash Table | O(n) read + O(1) avg/ O(n) worst insert | O(1) avg, O(n) worst | O(n log n) (sort keys) | Higher: bucket array + pointers |
| BST | O(n log n) insert avg/ O(n²) worst unbalanced | O(log n) avg, O(n)worst | O(n) in-order traversal | Moderate: node pointers per element |

## Advantages & Disadvantages

Vector:  
- Simple implementation  
- Good cache performance  
- Search is O(n) unless sorted and binary search is used  
- Requires sorting before ordered output

Hash Table:  
- O(1) average lookups  
- No need to traverse for search  
- No inherent ordering — must sort keys for ordered list  
- More memory overhead due to buckets

BST:  
- Keeps courses in sorted order naturally  
- O(log n) search, insert, delete when balanced  
- Can degrade to O(n) if unbalanced  
- Slightly more complex code

## Recommendation

Because ABCU requires frequent sorted output and efficient course lookups, a balanced BST (AVL or Red-Black Tree) is the optimal choice:

* O(log n) search, insertion, and deletion
* O(n) ordered output without additional sorting
* Reasonable memory use

If balancing is not implemented, a hash table is the fastest alternative for search but will require O(n log n) key sorting for ordered output.