CS 300 Project One: Pseudocode & Runtime Analysis

This document contains finalized pseudocode for three data structures—Vector, Hash Table, and Binary Search Tree (BST)—to power ABCU Advising's course tool. It also includes a Big-O runtime and memory analysis and a data-structure recommendation. The input file uses comma-separated values: courseNumber, name, prerequisite1..N.

Course Object (common across all designs):

```
STRUCT Course:
   courseNumber : STRING
   name         : STRING
   prerequisites: LIST<STRING>
END STRUCT
```

File Open, Read, Parse (common logic used by each structure):

```
FUNCTION LoadFromFile(path: STRING) -> LIST<Course>:
   courses = EMPTY LIST<Course>
   file = OPEN(path, "r")
   FOR EACH line IN file:
      IF line IS EMPTY: CONTINUE
      fields = SPLIT(line, ",")
      IF LENGTH(fields) < 2:
         PRINT "Format error: missing course number or name"; CONTINUE
      course = NEW Course
      course.courseNumber = TRIM(fields[0])
```

```
    course.name         = TRIM(fields[1])
    course.prerequisites = EMPTY LIST
    FOR i FROM 2 TO LENGTH(fields)-1:
      prereq = TRIM(fields[i])
      IF prereq != "": APPEND(course.prerequisites, prereq)
    APPEND(courses, course)
  CLOSE(file)
  RETURN courses
END FUNCTION
```

Vector Design:

```
GLOBAL vecCourses : VECTOR<Course>


PROCEDURE BuildVector(path):
  vecCourses = LoadFromFile(path)


FUNCTION FindCourseVector(courseNum: STRING) -> Course|NULL:
  FOR EACH c IN vecCourses:
    IF c.courseNumber == courseNum: RETURN c
  RETURN NULL


PROCEDURE PrintCourseInfoVector(courseNum: STRING):
  c = FindCourseVector(courseNum)
  IF c == NULL: PRINT "Course not found"; RETURN
```

```
    PRINT c.courseNumber + ": " + c.name
    IF LENGTH(c.prerequisites) == 0:
        PRINT "Prerequisites: None"
    ELSE:
        PRINT "Prerequisites: " + JOIN(c.prerequisites, ", ")

PROCEDURE PrintSortedListVector():
    temp = COPY(vecCourses)
    SORT temp BY courseNumber ASC  // O(n log n)
    FOR EACH c IN temp: PRINT c.courseNumber + ", " + c.name
```

Hash Table Design (keyed by courseNumber):

```
GLOBAL htCourses : HASH_TABLE<STRING, Course>

PROCEDURE BuildHash(path):
    htCourses = NEW HASH_TABLE
    list = LoadFromFile(path)
    FOR EACH c IN list:
        INSERT htCourses[c.courseNumber] = c

FUNCTION FindCourseHash(courseNum: STRING) -> Course|NULL:
    IF EXISTS htCourses[courseNum]: RETURN htCourses[courseNum]
    RETURN NULL
```

```
PROCEDURE PrintCourseInfoHash(courseNum: STRING):
   c = FindCourseHash(courseNum)
   IF c == NULL: PRINT "Course not found"; RETURN
   PRINT c.courseNumber + ": " + c.name
   IF LENGTH(c.prerequisites) == 0:
      PRINT "Prerequisites: None"
   ELSE:
      PRINT "Prerequisites: " + JOIN(c.prerequisites, ", ")


PROCEDURE PrintSortedListHash():
   keys = KEYS(htCourses)
   SORT keys ASC              // O(n log n)
   FOR EACH k IN keys:
      c = htCourses[k]
      PRINT c.courseNumber + ", " + c.name
```

Binary Search Tree Design (ordered by courseNumber):

```
STRUCT Node:
   key   : STRING       // courseNumber
   value : Course
   left  : Node
   right : Node
END STRUCT
```

```
GLOBAL root : Node = NULL

FUNCTION InsertBST(root, key, value) -> Node:
   IF root == NULL:
      node = NEW Node; node.key = key; node.value = value; RETURN node
   IF key < root.key:  root.left  = InsertBST(root.left, key, value)
   ELSE IF key > root.key: root.right = InsertBST(root.right, key, value)
   ELSE: root.value = value
   RETURN root

PROCEDURE BuildBST(path):
   root = NULL
   list = LoadFromFile(path)
   FOR EACH c IN list: root = InsertBST(root, c.courseNumber, c)

FUNCTION FindBST(root, key) -> Course|NULL:
   WHILE root != NULL:
      IF key == root.key: RETURN root.value
      IF key < root.key:  root = root.left
      ELSE:               root = root.right
   RETURN NULL

PROCEDURE PrintCourseInfoBST(courseNum: STRING):
```

```
    c = FindBST(root, courseNum)
    IF c == NULL: PRINT "Course not found"; RETURN
    PRINT c.courseNumber + ": " + c.name
    IF LENGTH(c.prerequisites) == 0:
        PRINT "Prerequisites: None"
    ELSE:
        PRINT "Prerequisites: " + JOIN(c.prerequisites, ", ")

PROCEDURE InOrder(node):
    IF node == NULL: RETURN
    InOrder(node.left)
    PRINT node.value.courseNumber + ", " + node.value.name
    InOrder(node.right)

PROCEDURE PrintSortedListBST():
    InOrder(root)           // O(n) once built
```

Menu (shared UX over any backing store):

```
PROCEDURE Main():
    dsType = PROMPT("Choose data structure: 1=Vector, 2=Hash, 3=BST")
    path   = PROMPT("Enter input file path")
    IF dsType == 1: BuildVector(path)
    ELSE IF dsType == 2: BuildHash(path)
    ELSE: BuildBST(path)
```

```
REPEAT:
    PRINT "1) Load file   2) Print course list   3) Print course info   9) Exit"
    option = PROMPT("Select:")
    IF option == 1:
        IF dsType == 1: BuildVector(path)
        ELSE IF dsType == 2: BuildHash(path)
        ELSE: BuildBST(path)
        PRINT "Data loaded."
    ELSE IF option == 2:
        IF dsType == 1: PrintSortedListVector()
        ELSE IF dsType == 2: PrintSortedListHash()
        ELSE: PrintSortedListBST()
    ELSE IF option == 3:
        key = PROMPT("Enter course number (e.g., CSCI200):")
        IF dsType == 1: PrintCourseInfoVector(key)
        ELSE IF dsType == 2: PrintCourseInfoHash(key)
        ELSE: PrintCourseInfoBST(key)
    ELSE IF option == 9:
        BREAK
    ELSE:
        PRINT "Invalid option."
```

UNTIL FALSE

END PROCEDURE

Runtime & Memory Analysis (worst-case, n courses, p avg prereqs):

Load & Parse (common): $O(n \cdot p)$ to split lines and collect prerequisites.

Vector:

• Build: $O(n)$ (copy from parsed list). Memory: $O(n)$.

• Find course: $O(n)$ linear scan.

• Print sorted list: $O(n \log n)$ (sort by courseNumber) each time.

• Strengths: simple, cache-friendly; easy to implement.

• Weaknesses: linear search; must re-sort or maintain order for option 2.

Hash Table:

• Build: $O(n)$ average inserts. Memory: $O(n)$ + overhead for buckets.

• Find course: $O(1)$ average, $O(n)$ worst under heavy collisions.

• Print sorted list: extract keys then $O(n \log n)$ sort each time.

• Strengths: fastest lookups for option 3.

• Weaknesses: requires separate sort for option 2; tuning load factor; non-deterministic order.

Binary Search Tree (unbalanced vs. balanced):

• Build: $O(n \log n)$ average inserts; $O(n^2)$ worst if unbalanced (sorted input).

• Find course: $O(\log n)$ average; $O(n)$ worst if unbalanced.

• Print sorted list: $O(n)$ via in-order traversal (no extra sort).

• Memory: $O(n)$ nodes + pointers.

• Strengths: naturally ordered; very fast repeated sorted listings.

• Weaknesses: worst-case degradation unless self-balancing (AVL/Red-Black).

Recommendation:

Use a self-balancing BST (e.g., AVL or Red-Black). Rationale: advisors routinely need an alphanumerically sorted list (option 2), which is O(n) to output from a BST after O(log n) inserts/lookups. While a hash table gives O(1) average lookups, it still needs an O(n log n) sort each time the full list is printed. If sorted listing is infrequent and random single-course queries dominate, a hash table is a strong alternative.