Project One Module: 6-2

Oliver Rodriguez

CS 300: Pseudocode and Runtime Analysis

Pseudocode (common course object & menu)

```
// Course record used by all structures
STRUCT Course
   courseNumber : STRING
   courseTitle  : STRING
   prerequisites : LIST of STRING
END STRUCT
// Program Menu (shared)
FUNCTION Menu()
   WHILE true
      PRINT "1) Load file"
      PRINT "2) Print all courses (alphanumeric)"
      PRINT "3) Print course title & prerequisites"
      PRINT "9) Exit"
      input choice
      IF choice == 1 THEN LoadCourses(filename)        // delegates to chosen DS
implementation
      ELSE IF choice == 2 THEN PrintAllCourses()
      ELSE IF choice == 3 THEN
         input courseNum
         PrintCourse(courseNum)
      ELSE IF choice == 9 THEN EXIT
      END IF
   END WHILE
END FUNCTION
```

Vector Implementation (Array / dynamic array)

```
// Load file into vector
FUNCTION LoadCourses_Vector(fileName) -> Vector<Course>
   courses ← empty vector
   OPEN file
   IF cannot open THEN print error and return empty vector
   FOR each line in file
      tokens ← SPLIT(line, ',')
      IF tokens.size < 2 THEN print error; CONTINUE
      c ← NEW Course
      c.courseNumber ← TRIM(tokens[0])
      c.courseTitle  ← TRIM(tokens[1])
      FOR i FROM 2 TO tokens.size-1 DO
         APPEND TRIM(tokens[i]) TO c.prerequisites
      END FOR
      APPEND c TO courses        // amortized O(1)
   END FOR
   CLOSE file

   // Validate prerequisites exist
```

```
    FOR each course IN courses
        FOR each prereq IN course.prerequisites
            IF prereq NOT IN any courses.courseNumber THEN
                PRINT "Error: prereq " + prereq + " not found"
            END IF
        END FOR
    END FOR

    RETURN courses
END FUNCTION

FUNCTION PrintAllCourses()
    // Sort vector by courseNumber then print
    SORT courses BY courseNumber       // O(n log n)
    FOR each course IN courses DO PrintCourseInfo(course)
END FUNCTION

FUNCTION SearchCourse_Vector(courseNum)
    FOR each course IN courses
        IF course.courseNumber == courseNum THEN PrintCourseInfo(course); RETURN
    PRINT "Course not found"
END FUNCTION
```

Hash Table Implementation

```
// Assume HashTable supports insert(key, value), find(key), keys()
FUNCTION LoadCourses_Hash(fileName)
    table ← new HashTable
    tempList ← empty vector   // to validate prereqs easily
    OPEN file
    FOR each line in file
        parse tokens as above --> c
        INSERT c INTO table WITH KEY c.courseNumber
        APPEND c.courseNumber TO tempList
    END FOR
    CLOSE file

    // Validate prerequisites
    FOR each key IN table.keys()
        course ← table.find(key)
        FOR each prereq IN course.prerequisites
            IF table.find(prereq) IS NULL THEN PRINT error
        END FOR
    END FOR

    RETURN table
END FUNCTION
```

```
FUNCTION PrintAllCourses_Hash()
   keys ← table.keys()            // O(n)
   SORT keys BY alphanumeric      // O(n log n)
   FOR each k IN keys DO PrintCourseInfo(table.find(k))
END FUNCTION

FUNCTION SearchCourse_Hash(courseNum)
   course ← table.find(courseNum)  // O(1) average
   IF course IS NULL THEN PRINT "not found" ELSE PrintCourseInfo(course)
END FUNCTION
```

Binary Search Tree (BST) Implementation

```
// BST stores Course keyed by courseNumber
STRUCT Node
   course : Course
   left, right : Node
END STRUCT

FUNCTION InsertNode(root, course)
   IF root IS NULL THEN root ← NEW Node(course); RETURN root
   IF course.courseNumber < root.course.courseNumber THEN
      root.left ← InsertNode(root.left, course)
   ELSE
      root.right ← InsertNode(root.right, course)
   END IF
   RETURN root
END FUNCTION
FUNCTION LoadCourses_BST(fileName)
   root ← NULL
   tempList ← empty vector
   OPEN file
   FOR each line in file
      parse tokens as above -> c
      APPEND c.courseNumber TO tempList
      root ← InsertNode(root, c)       // O(h) per insert, h = tree height
   END FOR
   CLOSE file

   // Validate prerequisites by searching tempList or traversing tree
   FOR each courseNum IN tempList
      course ← BST_Search(root, courseNum)
      FOR each prereq IN course.prerequisites
         IF BST_Search(root, prereq) IS NULL THEN PRINT error
      END FOR
   END FOR
```

```
    RETURN root
END FUNCTION

FUNCTION PrintAllCourses_BST()
    InOrderTraversal(root)   // prints in sorted order — O(n)
END FUNCTION

FUNCTION SearchCourse_BST(courseNum)
    RETURN BST_Search(root, courseNum)   // O(h) where h = tree height
END FUNCTION
```

Milestone One: Vector

Code	Line Cost	#Times Executes	Total Cost
Open file	1	1	1
For each line in file	1	n	n
Split line into tokens	1	n	n
Create new Course object	1	n	n
Assign courseNumber and title	2	n	2n
For each prerequisite in line	1	k (average prerequisites per course)	nk
Append course to vector	1	n	n
Validate prerequisites (nested loops)	1	n^2	n^2
Close file	1	1	1
Total Cost			n^2 + (nk + 5n + 2)
Runtime			O(n^2)

Vector Analysis

The vector (or dynamic array) allows easy sequential reading and appending since each append has an amortized O(1) cost. However, validation requires checking whether each prerequisite exists in the list, which leads to $O(n^2)$ time in the worst case because every course may need to be compared against all others. Sorting or searching through an unsorted vector is linear time, so large datasets reduce efficiency.

- **Advantages:**

  Easy to implement and iterate through.

  Great for small datasets.

  Preserves order naturally.

- **Disadvantages:**

  Searching and validation are O(n) per lookup.

  Insertion/removal in the middle of the list is expensive.

  Large memory shifts if array resizes.

Milestone Two: Hash Table

Code Step	Line Cost	#Times Executes	Total Cost
Open file	1	1	1
For each line in file	1	n	n
Split line into tokens	1	n	n
Create new Course object	1	n	n
Insert course into hash table	1	n	n

Append CourseNumber to temp list	1	n	n
Validate prerequisites	1	n*k	nk
Each hash lookup	O(1) avg	nk	nk
Close file	1	1	
Total Cost			4n + 2nk + 2
Runtime			O(nk)->O(n) if k is small

Hash Table Analysis

The hash table offers O(1) average-case time for insertions and lookups, making it much more efficient for validation and retrieval than the vector. Each course and its prerequisites can be inserted and verified quickly. The total runtime scales linearly with the number of courses, assuming hash collisions are minimal.

- **Advantages:**

  Fast lookups and insertions (O(1)) on average.

  Excellent for checking prerequisites and direct searches.

  Ideal for large datasets.

- **Disadvantages:**

  No inherent ordering (must sort keys for alphabetical output).

  Requires more memory for hash buckets.

  Poor performance if hash collisions are frequent.

Milestone Three: Binary Search Tree

Code Step	Line Cost	#Times Executes	Total Cost
Open file	1	1	1
For each line in file	1	n	n
Split line into tokens	1	n	n
Create new Course object	1	n	n
Insert course into BST	O(h)	n	n*h
Append courseNumber to temp list	1	n	n
Validate prerequisites	O(h)	n*k	nk*h
Close file	1	1	1
Total Cost			3n + n*h + nk*h + 2
Runtime			O(nh)

BST Analysis

A Binary Search Tree organizes data so that retrievals and insertions can be efficient if the tree remains balanced. This allows course validation and sorted output naturally via an in-order traversal.

- **Advantages:**

  Naturally maintains sorted order.

Efficient searches and insertions in balanced trees (O(log n)).

Easy traversal for ordered printing.

- **Disadvantages:**

  Can degrade to $O(n^2)$ if unbalanced (e.g., inserting sorted data).

  More complex to implement.

  Requires recursive structure, increasing overhead slightly.

Based on the runtime and functionality required by the advisor system, the Hash Table is the most suitable data structure. It provides fast lookups (O(1)), allowing advisors to quickly retrieve courses and verify prerequisites, even with a large number of courses. While the BST is ideal for sorted output, it risks inefficiency if unbalanced. The vector, though easy to implement, is the least efficient due to its $O(n^2)$ runtime when validating or searching. Therefore, the Hash Table offers the best balance of speed, scalability, and efficient data validation, meeting the system's needs for quick access and robust data management.