Megan Mitchell
Professor Muthukudage
CS 300 DSA: Analysis and Design
11 October 2025

Pseudocodes

Vector(Milestone 1)

```
FUNCTION loadCoursesVector(fileName)
  OPEN file
  IF file cannot open THEN PRINT "Error opening file" AND EXIT

  CREATE empty vector called courses
  FOR each line in file
    SPLIT line by commas into tokens
    SET number = tokens[0]
    SET title = tokens[1]
    SET prereqs = remaining tokens after index 1
    CREATE Course object with these values
    ADD Course to courses
  END FOR
  CLOSE file
  RETURN courses
END FUNCTION

FUNCTION printCourseVector(courses, courseNumber)
  FOR each course in courses
    IF course.number == courseNumber THEN
      PRINT course.number + ": " + course.title
      IF course.prereqs is empty THEN PRINT "Prerequisites: None"
      ELSE PRINT "Prerequisites: " + list of prereqs
    END IF
  END FOR
END FUNCTION

FUNCTION printAllCoursesVector(courses)
  SORT courses by course.number
  FOR each course in courses
    PRINT course.number + ": " + course.title
  END FOR
END FUNCTION

FUNCTION menuVector()
  DISPLAY menu options (1 Load, 2 List, 3 Search, 9 Exit)
  GET user choice
  CASE 1: courses = loadCoursesVector("courses.txt")
  CASE 2: printAllCoursesVector(courses)
  CASE 3: input courseNumber → printCourseVector(courses, courseNumber)
  CASE 9: PRINT "Goodbye"
END FUNCTION
```

Hash Table (Milestone 2)

```
FUNCTION loadCoursesHash(fileName)
    OPEN file
    CREATE empty hashTable
    FOR each line in file
        SPLIT by commas into tokens
        CREATE Course c = (tokens[0], tokens[1], remaining tokens)
        hashTable[c.number] = c
    END FOR
    CLOSE file
    RETURN hashTable
END FUNCTION

FUNCTION printCourseHash(table, courseNumber)
    IF courseNumber IN table THEN
        c = table[courseNumber]
        PRINT c.number + ": " + c.title
        IF c.prereqs empty THEN PRINT "Prerequisites: None"
        ELSE PRINT "Prerequisites: " + list of prereqs
    ELSE PRINT "Course not found"
END FUNCTION

FUNCTION printAllCoursesHash(table)
    keys = list of all keys in table
    SORT keys alphanumerically
    FOR each key IN keys
        c = table[key]
        PRINT c.number + ": " + c.title
```

```
   END FOR
END FUNCTION

FUNCTION menuHash()
   DISPLAY menu
   GET choice
   CASE 1: table = loadCoursesHash("courses.txt")
   CASE 2: printAllCoursesHash(table)
   CASE 3: input courseNumber → printCourseHash(table, courseNumber)
   CASE 9: PRINT "Goodbye"
END FUNCTION
```

Binary Search Tree (Milestone 3)

```
STRUCT Node
   Course course
   Node* left
   Node* right
END STRUCT

CLASS BST
   Node* root = NULL

   METHOD insert(Course c)
      IF root == NULL THEN root = new Node(c)
      ELSE addNode(root, c)

   METHOD addNode(Node* node, Course c)
      IF c.number < node.course.number THEN
         IF node.left == NULL THEN node.left = new Node(c)
         ELSE addNode(node.left, c)
      ELSE
         IF node.right == NULL THEN node.right = new Node(c)
         ELSE addNode(node.right, c)

   METHOD search(number)
      node = root
      WHILE node != NULL
         IF number == node.course.number THEN RETURN node.course
         ELSE IF number < node.course.number THEN node = node.left
         ELSE node = node.right
      RETURN NULL
   METHOD inOrder(Node* node)
      IF node == NULL RETURN
      inOrder(node.left)
      PRINT node.course.number + ": " + node.course.title
      inOrder(node.right)
END CLASS

FUNCTION loadCoursesBST(fileName)
   CREATE BST tree
   OPEN file
   FOR each line in file
      PARSE courseNumber, title, and prereqs
      tree.insert(Course(courseNumber, title, prereqs))
   END FOR
   RETURN tree
END FUNCTION

FUNCTION printCourseBST(tree, courseNumber)
   c = tree.search(courseNumber)
   IF c is NULL THEN PRINT "Course not found"
   ELSE PRINT c.number + ": " + c.title and prereqs
END FUNCTION

FUNCTION menuBST()
   DISPLAY menu
   GET choice
   CASE 1: tree = loadCoursesBST("courses.txt")
   CASE 2: tree.inOrder(tree.root)
   CASE 3: input courseNumber → printCourseBST(tree, courseNumber)
   CASE 9: PRINT "Goodbye"
END FUNCTION
```

Vector Data Structure

Line of Code	Cost per Line	Times Executed
Open fileName as file	1	1
If file cannot be opened THEN	1	1
Print “Error: Could not open file”	1	0
Create empty vector called courses	1	1
While there are more lines to read in file	1	n
Read a line from file	1	n

Split the line by commas	1	n
If number of tokens < 2 THEN	1	n
Display “Error: Invalid line format”	1	0
Continue to next iteration	1	0
Set courseNumber = first token	1	n
Set courseTitle = second token	1	n
Store remaining tokens as prerequisites	1	n
Create Course object with these values	O(1)	n
Add Course object to vector	O(1)	n
Close the file	1	1

Overall Runtime:

- The overall runtime of loading data into this vector is O(n), since every line of data in this vector is read one at a time.
- After sorting the vector, the addition will be O(n log n), and searching for a particular course will be O(n).
- In the worst-case scenario, the time that would be taken to perform frequent sorting and searching would be O(n^2) if they were combined.

Hash Table Data Structure

Line of Code	Cost per Line	Times Executed
Attempt to Open File “course.txt” for reading	1	1

If File cannot be opened THEN	1	1
Display “Error: File cannot be opened”	1	0
Exit program	1	0
Declare hash table courseTable	1	1
While there are more lines in the file DO	1	n
Read a line from the File	1	n
Split the line into tokens using comma	1	n
If number of tokens < 2 THEN	1	n
Display “Error: Invalid line format”	1	0
Set courseNumber = first token	1	n
Set courseTitle = second token	1	n
For each prerequisite in the line DO	1	n x m
If prerequisite not found in table THEN	1	n x m
Display “Error: Prerequisite not found”	1	0
Create Course object with all data	O(1)	n

Insert Course object into Hash Table	O(1)	n
Close the File	1	1

My Overall Runtime:

- In the case of a hash table, each insert and lookup is performed in O(1) on average.
- Considering that I process n lines at a time, the total runtime for loading is O(n).
- When I run the prerequisite validation process, it runs O(nm) (where m = average number of prerequisites per course).
- In the worst case scenario, the process takes O(nm) when collisions occur or if validation checks every prerequisite.

Binary Search Tree (BST) Data Structure

Line of Code	Cost per Line	Times Executed
Open File “courses.txt” for reading	1	1
If File cannot be opened	1	1

THEN		
Display “Error: Could not Open File”	1	0
Create empty Binary Search Tree courseTree	1	1
While there are more lines in File DO	1	n
Read a line from the file	1	n
Split the line by commas	1	n
If number of parts < 2 THEN	1	n
Display “Error: Invalid format”	1	0
Continue to next iteration	1	0
Extract course number and title	1	n
Store remaining tokens as prerequisites	1	n
For each prerequisite	1	n x m

token DO		
If prerequisite not found THEN	1	n x m
Display “Error: Prerequisite missing”	1	0
Create new Course object	O(1)	n
Insert Course int BST (by course number)	O(log n)	n
Close the file	1	1

Overall Runtime:

- There is an average of O(log n) for each insertion into the BST on average (for a balanced tree).
- The total reading time for n courses is O(n log n)..
- There are m prerequisites for each course, so checking them all adds up to O(nm**)**.
- A worst case scenario is that it will take O(n2) if the BST becomes unbalanced (for example, if the data is already sorted).

Final Overall Summary

Data Structure	Load File	Search Course	Print All Courses	Average Runtime	Worst Case
Vector	O(n)	O(n)	O(n log n)	O(n log n)	O(n^2)

Hash Table	O(n)	O(1)	O(n log n)	O(n log n)	O(nm)
Binary Search Tree	O(n log n)	O(log n)	O(n)	O(n log n)	O(n^2)

When I thought about how many times each line runs in this pseudocode, I realized the hash table is by far the most efficient. I recommend this tool if you want to find one course, because that is what advisors do most often, which is to search for a single course. As I mentioned before, the vector is fine for small lists, but when it comes to sorting or searching, it gets slow, whereas the binary search tree is ideal for keeping courses sorted, but if it becomes unbalanced it can become very slow. It would be easy for me to build a hash table if I were to choose one to build in code. If I had to choose one, it's simple, fast, and reliable for what ABCU needs.