Matthew Willhite

Professor Michael Susalla

CS-300

12 October 2025

Runtime Analysis

The runtime analysis below examines the worst-case running time for reading the file and creating course objects for each data structure. Menu operations and print/search functions are excluded. The analysis assumes there are n courses stored in the data structure, and that each course may have multiple prerequisites.

Line-by-Line Cost Table

| **Line of Code** | **Description** | **Cost per Execution** | **Executions** | **Total Cost** |
| --- | --- | --- | --- | --- |
| 1 | Read all lines from file | 1 | n | n |
| 2 | Split and parse tokens for each line | 1 | n | n |
| 3 | Create course object for each line | 1 | n | n |
| 4 | Validate prerequisite existence | 1 | n | n |
| 5 | Insert course into data structure | Depends on structure | n | varies |
| **Total Runtime** |  |  |  | **O(n)** (Vector & Hash Table avg) / **O(n log n)** (Tree balanced) |

Comparative Runtime Summary

| **Data Structure** | **Load & Create Cost** | **Search Cost** | **Memory Use** | **Advantages** | **Disadvantages** |
| --- | --- | --- | --- | --- | --- |
| **Vector** | **O(n)** | **O(n)** | **O(n)** | Simple, easy to code, predictable performance, good for small data sets | Slow for lookups, requires sorting for alphanumeric order |
| **Hash Table** | **O(n)** average / **O(n²)** worst | **O(1)** average | **O(n)** + overhead | Fast lookups and validation, minimal collision handling in practice | Must extract and sort data to print in order, higher memory use |
| **Binary Search Tree** | **O(n log n)** balanced / **O(n²)** worst | **O(log n)** balanced | **O(n)** | Naturally ordered by key, efficient search when balanced | Performance degrades if unbalanced, more complex to implement |

Evaluation

**Evaluation**

**Vector:**

The vector is simple and easy to understand. Loading the data and checking for errors both take O(n) time. Searching for a single course takes O(n) because each item must be checked one at a time. Sorting the list to display courses in order adds another O(n log n) step. This method works fine for smaller data sets, but it becomes slower as the dataset grows.

**Hash Table:**

The hash table is the fastest option overall. On average, loading and searching both take O(1) time. In the worst case, if too many items end up in the exact location, the time can increase to O(n). The hash table uses more memory due to the need for extra storage, but the faster lookups make that extra memory worth it.

**Binary Search Tree:**

A balanced binary search tree keeps insertion and search operations at O(log n), which makes it efficient for larger amounts of data. It also keeps the courses in order automatically. The problem is that if the tree is not balanced, it can slow down to O(n²). This can happen when the data is already sorted before being added. The tree is efficient when balanced but can lose speed quickly when it is not.

Recommendation

After reviewing all three data structures, the hash table stands out as the best option for the ABC University advising program. It handles searches very quickly, usually in constant time, and makes it easy to check course prerequisites. When the advisor needs to see all courses in order, the data from the hash table can be placed into a vector and sorted before it is printed.

This combination gives the program several advantages:

* Fast lookups with an average time of O(1) when finding specific courses
* Correct alphabetical order with O(n log n) sorting when displaying the full list
* Reasonable memory use without the extra work of balancing a tree

Overall, using a hash table for storage and a vector for output provides the best mix of speed, accuracy, and simplicity for the advising program.

Pseudocode for Menu

FUNCTION displayMenu():

PRINT "1 - Load Data Structure"

PRINT "2 - Print Course List"

PRINT "3 - Search for Course"

PRINT "9 - Exit"

FUNCTION main():

DECLARE userChoice AS INTEGER = 0

WHILE userChoice != 9:

displayMenu()

INPUT userChoice

IF userChoice == 1:

CALL loadCourseData()

ELSE IF userChoice == 2:

CALL printAllCourses()

ELSE IF userChoice == 3:

CALL searchCourse()

ELSE IF userChoice == 9:

PRINT "Goodbye!"

ELSE:

PRINT "Invalid selection, try again."

END FUNCTION

Vector

FUNCTION printVector(vector):

SORT vector BY courseNumber // quick sort or built-in sort

FOR EACH course IN vector:

PRINT course.number + ", " + course.title

END FUNCTION

Hash Table

FUNCTION printHashTable(hashTable):

DECLARE tempVector AS empty list

FOR EACH course IN hashTable:

ADD course TO tempVector

SORT tempVector BY courseNumber

FOR EACH course IN tempVector:

PRINT course.number + ", " + course.title

END FUNCTION

Binary Search Tree

FUNCTION printBST(node):

IF node IS NOT NULL:

printBST(node.left)

PRINT node.course.number + ", " + node.course.title

printBST(node.right)

END FUNCTION

Tree Data Structure Pseudocode

FUNCTION loadCourseData():

DECLARE courseTree AS BinarySearchTree

OPEN "course\_data.txt" FOR reading

IF file cannot be opened:

PRINT "Error: File not found"

RETURN

FOR EACH line IN file:

SPLIT line BY comma INTO tokens

IF number of tokens < 2:

PRINT "Error: Invalid line format"

CONTINUE

courseNumber = tokens[0]

courseTitle = tokens[1]

prereqs = tokens[2 to end]

CREATE courseObject(courseNumber, courseTitle, prereqs)

courseTree.insert(courseObject)

CLOSE file

PRINT "Courses loaded successfully!"

END FUNCTION