CS 300

Jake DeMuesy

Project 1

4/20/2025

Pseudocode for a menu

Initial implementation (vector data structure) -

PRINT "Welcome to the ABCU Course Information Access Program"

DECLARE courses AS VECTOR of COURSE objects

DECLARE dataLoaded AS BOOLEAN = FALSE

PRINT "1. Load Data Structure"

PRINT "2. Print Course List"

PRINT "3. Print Course"

PRINT "9. Exit"

PRINT "Please enter your selection: " (userSelection)

SWITCH userSelection

CASE 1:

CALL loadData(courses)

FOR EACH line IN file (std::ifstream)

SPLIT line BY (",")

IF SIZE of splitLine < 2

PRINT "Error: Invalid line format – " + line

CONTINUE to next line

DECLARE newCourse as COURSE

SET newCourse.courseNumber = splitLine[0]

SET newCourse.courseName = splitLine[1]

FOR i FROM 2 TO SIZE(splitLine) - 1

ADD splitLine [i] TO newCourse.prerequisites

ADD newCourse TO courses

CLOSE file

SET dataLoaded = TRUE

PRINT "Course data successfully loaded."

BREAK

CASE 2:

IF dataLoaded = FALSE

PRINT " Error: No data loaded, please load data first (option 1)."

ELSE

CALL printAllCourses(courses)

SORT courses BY course.courseNumber (alphanumerically)

FOR EACH course IN courses

PRINT course.courseNumber + ": " + course.courseName

BREAK

CASE 3:

IF dataLoaded = FALSE

PRINT "Error: No data loaded, please load data first (option 1)."

ELSE

PRINT "Enter course number:" (searchCourse)

CALL printCourseInfo(courses, searchCourse)

FOR EACH course IN courses

IF course.courseNumber == searchCourse

PRINT "Course Number: " + course.courseNumber

PRINT "Course Name: " + course.courseName

IF course.prerequisites IS NOT EMPTY

PRINT "Prerequisites: "

FOR EACH prereq IN course.prerequisites

PRINT " - " + prereq

ELSE

PRINT "No prerequisites"

RETURN

BREAK

CASE 9:

PRINT "Exiting program. Goodbye!"

EXIT LOOP

DEFAULT:

PRINT "Invalid option. Please try again."

From completing a run time analysis on the current pseudocode, Each operation has a distinct time complexity based on the number of courses (n), the number of prerequisites per course (m), and the number of lines in the input file (l). I can see that when the data loading process reads from the file, it splits each line by commas, then constructs course objects, and stores them in a vector. This operation runs in O(n \* m + l \* k) time, where k is the average number of tokens (prerequisites) per line.

When printing all of the courses, the program first checks if data has been loaded and then sorts the list of courses alphanumerically by course number before printing. The sorting operation is the one that dominates the time complexity, making this case O(n log n). Each course is then printed in a loop (which adds an O(n) component) though this ends up being overshadowed by the sorting. This ensures a clean and user-friendly listing of the course data. In the case of printing a specific course, the system prompts the user for a course number and then performs a linear search through the course list. This results in a worst-case time complexity of O(n) for searching, with an additional O(m) if prerequisites need to be printed. Overall, this operation has a complexity of O(n + m). For larger datasets, this could be optimized further by using a hash table or similar structure that will reduce lookup time.

Lastly, both the exit operation (user input) and the default case (handling invalid input) have constant time complexity, O(1), since they involve simple print and control operations. Overall, the system performs efficiently for small to moderate datasets, but scalability could always be improved by optimizing data lookups and file parsing.

If we were to worry about scalability with larger data sets, by comparison when analyzing hash tables, if we were to apply the same logic but with the hash table data structure, we would see that a natural strength the hash table brings to the table is it’s searching/ accessing capabilities. With single course lookups (printing a single course) we can see that the memory improvement occurs from O(n + m) to O(1 + m) because the course can be accessed directly by its key (the course number)! There are some negatives though when wanting to work with hash tables, such as that they do require extra memory for internal bookkeeping (buckets, potential collisions). This memory overhead does trade off for faster access times, if the system were to get large enough, implementing a hash table would help with lookup times.

Finally, in also comparing the binary search tree (BST) to the vector for storing courses, the runtime and memory behavior will differ in several key areas, especially when it comes to insertion, search and ordered traversal. While initially loading data into a vector is done in linear time, inserting into a tree does require O(log n) time per course, which would make total insertion time (O n log n). Trees offer more though, in efficient search and lookup operations (again a good case for scalability), reducing time complexity of finding a course from O(n) with a vector to O(log n). Additionally, since trees naturally store elements in sorted order, printing all courses no longer requires sorting — an in-order traversal yields an O(n) runtime compared to O(n log n) for sorting a vector.

In conclusion, with analyzing the runtime and memory complexities of vectors, hash tables and binary search trees, the most beneficial data structure I would recommend moving forward would be the binary search tree. After the initial implementation of vectors, it is certain that the binary search tree offers efficient O(log n) search times for specific courses, which is significantly better than the O(n) search time of vectors. It also eliminates the need for separate sorting when printing all courses, as the tree maintains courses in sorted order naturally with O(n) traversal.

Runtime Comparisons

| **Code Operation** | **Vector** | **Binary Search Tree** | **Hash Table** |
| --- | --- | --- | --- |
| **Load Data** | O(n) | O(n log n) | O(n) |
| **Print All Courses** | O(n log n) | O(n) | O(n log n) |
| **Find Course by ID** | O(n) | O(log n) | O(1) |
| **Print Course Details** | O(n + m) | O(log n + m) | O(1 + m) |