Aleksandr Donovan

Professor Nathan Lebel

6/11/2025

Project 1

Vector:

Pseudocode for a menu - Vector:

| FUNCTION displayMenu()  OUTPUT "====== ABCU Course Planner ======"  OUTPUT "1. Load Data Structure"  OUTPUT "2. Print Course List"  OUTPUT "3. Print Course"  OUTPUT "9. Exit"  END FUNCTION  FUNCTION runProgram()  DECLARE courses AS empty vector  DECLARE userChoice AS INTEGER  WHILE TRUE DO  CALL displayMenu()  INPUT userChoice  IF userChoice EQUALS 1 THEN  INPUT "Enter file name to load: " INTO fileName  SET courses TO loadCoursesFromFile(fileName)  OUTPUT "Data loaded successfully."  ELSE IF userChoice EQUALS 2 THEN  CALL printCourseList(courses)  ELSE IF userChoice EQUALS 3 THEN  INPUT "Enter course number: " INTO courseNumber  CALL searchCourse(courses, courseNumber)  ELSE IF userChoice EQUALS 9 THEN  OUTPUT "Goodbye!"  BREAK  ELSE  OUTPUT "Invalid option. Please try again."  END IF  END WHILE  END FUNCTION |
| --- |

Pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order - Vector:

| FUNCTION printCourseList(courses)  // Sort the courses vector by course.number using bubble sort or any preferred sorting method  FOR i FROM 0 TO SIZE OF courses - 1 DO  FOR j FROM 0 TO SIZE OF courses - i - 2 DO  IF courses[j].number > courses[j + 1].number THEN  SWAP courses[j] AND courses[j + 1]  END IF  END FOR  END FOR  // After sorting, print course number and title  FOR EACH course IN courses DO  OUTPUT course.number, ": ", course.title  END FOR  END FUNCTION |
| --- |

Hash Table:

Pseudocode for a menu - Hash Table:

| FUNCTION displayMenu()  OUTPUT "====== ABCU Course Planner ======"  OUTPUT "1. Load Data Structure"  OUTPUT "2. Print Course List"  OUTPUT "3. Print Course"  OUTPUT "9. Exit"  END FUNCTION  FUNCTION runProgram()  DECLARE courses AS empty hash table (key: STRING, value: Course)  DECLARE userChoice AS INTEGER  WHILE TRUE DO  CALL displayMenu()  INPUT userChoice  IF userChoice EQUALS 1 THEN  INPUT "Enter file name to load: " INTO fileName  SET courses TO loadCoursesFromFile(fileName)  OUTPUT "Data loaded successfully."  ELSE IF userChoice EQUALS 2 THEN  CALL printCourseList(courses)  ELSE IF userChoice EQUALS 3 THEN  INPUT "Enter course number: " INTO courseNumber  CALL searchCourse(courses, courseNumber)  ELSE IF userChoice EQUALS 9 THEN  OUTPUT "Goodbye!"  BREAK  ELSE  OUTPUT "Invalid option. Please try again."  END IF  END WHILE  END FUNCTION |
| --- |

Pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order - Hash Table:

| FUNCTION printCourseList(courses)  DECLARE courseList AS empty vector  // Extract values from hash table  FOR EACH key-value pair IN courses DO  ADD value TO courseList  END FOR  // Sort courseList by course.number  FOR i FROM 0 TO SIZE OF courseList - 1 DO  FOR j FROM 0 TO SIZE OF courseList - i - 2 DO  IF courseList[j].number > courseList[j + 1].number THEN  SWAP courseList[j] AND courseList[j + 1]  END IF  END FOR  END FOR  // Print sorted course list  FOR EACH course IN courseList DO  OUTPUT course.number, ": ", course.title  END FOR  END FUNCTION |
| --- |

Binary Search Tree:

Pseudocode for a menu - Binary Search Tree:

| FUNCTION displayMenu()  OUTPUT "====== ABCU Course Planner ======"  OUTPUT "1. Load Data Structure"  OUTPUT "2. Print Course List"  OUTPUT "3. Print Course"  OUTPUT "9. Exit"  END FUNCTION  FUNCTION runProgram()  DECLARE courseTree AS empty binary search tree  DECLARE userChoice AS INTEGER  WHILE TRUE DO  CALL displayMenu()  INPUT userChoice  IF userChoice EQUALS 1 THEN  INPUT "Enter file name to load: " INTO fileName  SET courseTree TO loadCoursesFromFile(fileName)  OUTPUT "Data loaded successfully."  ELSE IF userChoice EQUALS 2 THEN  CALL printCourseList(courseTree.root)  ELSE IF userChoice EQUALS 3 THEN  INPUT "Enter course number: " INTO courseNumber  CALL searchCourse(courseTree, courseNumber)  ELSE IF userChoice EQUALS 9 THEN  OUTPUT "Goodbye!"  BREAK  ELSE  OUTPUT "Invalid option. Please try again."  END IF  END WHILE  END FUNCTION |
| --- |

Pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order - Binary Search Tree:

| FUNCTION printCourseList(node)  IF node IS null THEN  RETURN  END IF  CALL printCourseList(node.left)  OUTPUT node.number, ": ", node.title  CALL printCourseList(node.right)  END FUNCTION |
| --- |

Evaluation:

Runtime analysis:

Vector:

| WHILE NOT end of file DO  READ line FROM file // O(1)  SPLIT line INTO tokens // O(1)  IF SIZE OF tokens < 2 // O(1)  EXTRACT course data // O(1)  CREATE Course object // O(1)  ADD course TO vector // O(1) amortized  END WHILE  FOR EACH course IN vector DO // O(n)  FOR EACH prereq IN course.prerequisites DO // O(p)  CHECK if prereq IN courseNumbers (set) // O(1)  END FOR  END FOR |
| --- |

Total Runtime:

* O(n + n·p) → O(n·p)

Memory:

* O(n)

Search Time:

* O(n) (linear search)

Advantages:

* Simple to implement
* Maintains insertion order

Disadvantages:

* Linear-time course search (O(n))
* Must manually sort for alphanumeric course list
* Slower for dynamic insertions/removals compared to hash tables or trees
* Doesn’t scale efficiently for large datasets if frequent searches are needed

Hash Table:

| WHILE NOT end of file DO  READ line FROM file // O(1)  SPLIT line INTO tokens // O(1)  EXTRACT course data // O(1)  CREATE Course object // O(1)  INSERT into hashTable // O(1) average  ADD courseNumber TO courseNumbers (set) // O(1)  END WHILE  FOR EACH key-value pair IN hashTable DO // O(n)  FOR EACH prereq IN course.prerequisites DO // O(p)  CHECK if prereq IN courseNumbers // O(1)  END FOR  END FOR |
| --- |

Total Runtime:

* O(n + n·p) → O(n·p)

Memory:

* O(n)

Search Time:

* O(1) average

Advantages:

* Fastest course lookup (O(1))
* Good for random access
* Automatically prevents duplicate keys

Disadvantages:

* Not inherently sorted — extra step needed to print in order
* Slightly more complex to implement and manage collisions

BST:

| WHILE NOT end of file DO  READ line FROM file // O(1)  SPLIT line INTO tokens // O(1)  EXTRACT course data // O(1)  CREATE Course object // O(1)  INSERT into BST // O(log n) average, O(n) worst  ADD courseNumber TO courseNumbers (set) // O(1)  END WHILE  FOR EACH course IN BST (in-order traversal) DO // O(n)  FOR EACH prereq IN course.prerequisites DO // O(p)  CHECK if prereq IN courseNumbers // O(1)  END FOR  END FOR |
| --- |

Total Runtime:

* O(n log n + n·p) average,
* O(n² + n·p) worst

Memory:

* O(n)

Search Time:

* O(log n) average,
* O(n) worst

Advantages:

* Naturally keeps data sorted (perfect for printing course list)
* Balanced trees support fast search, insert, and delete (O(log n))
* More memory-efficient than hash tables (no key overhead)
* In-order traversal returns a sorted list without extra sorting step

Disadvantages:

* Must implement logic for tree structure and node links
* Can degrade to linear performance if unbalanced (e.g., inserting sorted data)
* Requires recursion or stack for in-order traversal
* Harder to debug and implement than vector or hash table

Run time analysis chart:

| Operation | Vector | Hash Table | BST |
| --- | --- | --- | --- |
| Insert: | O(1) amortized | O(1) Average | Average: O(log n)  Worst:O(n) |
| Search for course: | O(n) | O(1) Average | Average: O(log n)  Worst:O(n) |
| Prerequisite validation: | O(n·p) | O(n·p) | O(n·p) |
| Sorted course list: | O(n log n) | O(n log n) | O(n) (in-order traversal) |
| Memory usage: | O(n) | O(n) | O(n) |
| Best for: | Simplicity | Speed and lookup | Naturally sorted data |
| Worst-case: | Slow lookup | Hash collisions → O(n) | Unbalanced tree → O(n²) |

Final recommendation:

Based on the runtime analysis and the structural behavior of the data structures, the hash table would be the most appropriate data structure for the ABCU advising system. Although it does not maintain order, it provides the best balance of speed, efficiency, and scalability while being the fastest at average case searching for specific courses.

It also handles validation and insertion efficiently and does not pose risks like BST for potential unbalancing and higher implementation complexity. Therefore, based on Big O analysis, it is the optimal data structure for this application.