Mitchell McNaney – CS 300 Project 1 Data Structure Analysis

Southern New Hampshire University  
mitchell.mcnaney@snhu.edu

**Vector Implementation**

PROCEDURE LoadCourses\_Vector(fileName)

CREATE empty Vector<Course> called courses

OPEN file with name = fileName

IF file cannot be opened THEN

PRINT "Error: Could not open file."

EXIT program

END IF

FOR each line IN file

SPLIT line by commas INTO tokens

IF number of tokens < 2 THEN

PRINT "Error: Missing course number or title."

CONTINUE

END IF

CREATE newCourse as Course

newCourse.courseNumber = tokens[0]

newCourse.title = tokens[1]

FOR i FROM 2 TO end of tokens

ADD tokens[i] TO newCourse.prerequisites

END FOR

ADD newCourse TO courses

END FOR

CLOSE file

RETURN courses

END PROCEDURE

PROCEDURE ValidateCourses\_Vector(courses)

FOR each course IN courses

FOR each prereq IN course.prerequisites

IF prereq NOT FOUND IN any course.courseNumber IN courses THEN

PRINT "Error: prerequisite " + prereq + " does not exist."

END IF

END FOR

END FOR

END PROCEDURE

PROCEDURE SearchCourse\_Vector(courses, searchNumber)

FOR each course IN courses

IF course.courseNumber == searchNumber THEN

PRINT "Course Number: " + course.courseNumber

PRINT "Course Title: " + course.title

IF course.prerequisites IS EMPTY THEN

PRINT "Prerequisites: None"

ELSE

PRINT "Prerequisites:"

FOR each prereq IN course.prerequisites

PRINT " " + prereq

END FOR

END IF

RETURN

END IF

END FOR

PRINT "Course " + searchNumber + " not found."

END PROCEDURE

| **Line of Code** | **Description** | **Execution Count** | **Cost per Line** | **Total Cost** |
| --- | --- | --- | --- | --- |
| Open file | Initialize file reading | 1 | 1 | 1 |
| FOR each line | Loop through all courses | n | 1 | n |
| Split line | Separate by commas | n | O(k) | O(n) |
| Create new course | Instantiate object | n | 1 | n |
| Add prerequisites | Loop through m prerequisites | n \* m | 1 | O(nm) |
| Add course to vector | Append operation | n | O(1) | O(n) |
| **Total Runtime** | — | — | — | **O(n)** |
| **Memory Usage** | — | — | — | **O(n)** |

Total Runtime: O(n)  
Memory Usage: O(n)

**Hash Table Implementation**

CLASS HashTable

METHOD insert(key, Course)

METHOD search(key) RETURNS Course

METHOD printAll()

END CLASS

PROCEDURE LoadCourses\_HashTable(fileName, hashTable)

OPEN file with name = fileName

IF file cannot be opened THEN

PRINT "Error: File not found."

EXIT program

END IF

FOR each line IN file

SPLIT line by commas INTO tokens

IF number of tokens < 2 THEN

PRINT "Error: Missing course number or title."

CONTINUE

END IF

CREATE newCourse as Course

newCourse.courseNumber = tokens[0]

newCourse.title = tokens[1]

FOR i FROM 2 TO end of tokens

ADD tokens[i] TO newCourse.prerequisites

END FOR

hashTable.insert(newCourse.courseNumber, newCourse)

END FOR

CLOSE file

FOR each course IN hashTable

FOR each prereq IN course.prerequisites

IF prereq NOT FOUND IN hashTable THEN

PRINT "Error: prerequisite " + prereq + " not found."

END IF

END FOR

END FOR

END PROCEDURE

PROCEDURE PrintCourseInfo\_HashTable(hashTable, courseNumber)

course ← hashTable.search(courseNumber)

IF course IS NULL THEN

PRINT "Course not found."

RETURN

END IF

PRINT course.courseNumber + ": " + course.title

IF course.prerequisites IS EMPTY THEN

PRINT "No prerequisites."

ELSE

PRINT "Prerequisites:"

FOR each prereq IN course.prerequisites

PRINT prereq

END FOR

END IF

END PROCEDURE

| **Line of Code** | **Description** | **Execution Count** | **Cost per Line** | **Total Cost** |
| --- | --- | --- | --- | --- |
| Open file | Initialize file reading | 1 | 1 | 1 |
| FOR each line | Loop through all courses | n | 1 | n |
| Split line | Separate tokens | n | O(k) | O(n) |
| Create new course | Instantiate object | n | 1 | n |
| Insert into hash table | Keyed by course number | n | O(1) average / O(n) worst | O(n) avg / O(n²) worst |
| **Total Runtime** | — | — | — | **O(n)** average / **O(n²)** worst |
| **Memory Usage** | — | — | — | **O(n)** |

Total Runtime: Average O(n), Worst O(n²)  
Memory Usage: O(n)

**Binary Search Tree Implementation**

CLASS BinarySearchTree

METHOD insert(Course)

METHOD search(courseNumber) RETURNS Course

METHOD inOrderTraversal(node)

END CLASS

PROCEDURE LoadCourses\_BST(fileName, courseTree)

OPEN file with name = fileName

IF file cannot be opened THEN

PRINT "Error: cannot open file."

EXIT program

END IF

WHILE there are lines remaining IN file

READ currentLine

SPLIT currentLine by commas INTO tokens

IF number of tokens < 2 THEN

PRINT "Format error: missing course title."

CONTINUE

END IF

CREATE newCourse as Course

newCourse.courseNumber = tokens[0]

newCourse.title = tokens[1]

FOR i FROM 2 TO end of tokens

ADD tokens[i] TO newCourse.prerequisites

END FOR

IF courseTree.root IS NULL THEN

courseTree.root ← newCourse

ELSE

current ← courseTree.root

WHILE TRUE

IF newCourse.courseNumber < current.course.courseNumber THEN

IF current.left IS NULL THEN

current.left ← newCourse

BREAK

ELSE

current ← current.left

END IF

ELSE

IF current.right IS NULL THEN

current.right ← newCourse

BREAK

ELSE

current ← current.right

END IF

END IF

END WHILE

END IF

END WHILE

CLOSE file

PRINT "All courses successfully loaded into tree."

END PROCEDURE

FUNCTION SearchCourse\_BST(courseTree, courseNumber)

node ← courseTree.root

WHILE node IS NOT NULL

IF courseNumber == node.course.courseNumber THEN

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

IF node.course.prerequisites IS NOT EMPTY THEN

PRINT "Prerequisites:"

FOR each prereq IN node.course.prerequisites

PRINT prereq

END FOR

ELSE

PRINT "No prerequisites."

END IF

RETURN

ELSE IF courseNumber < node.course.courseNumber THEN

node ← node.left

ELSE

node ← node.right

END IF

END WHILE

PRINT "Course not found."

END FUNCTION

FUNCTION PrintAllCourses\_BST(courseTree)

PRINT "Course List (alphanumeric order):"

CALL InOrderTraversal(courseTree.root)

END FUNCTION

FUNCTION InOrderTraversal(node)

IF node IS NULL THEN RETURN

CALL InOrderTraversal(node.left)

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

CALL InOrderTraversal(node.right)

END FUNCTION  
Pseudocode Section:  
FOR each line IN file  
 CREATE newCourse  
 INSERT course INTO tree (ordered by courseNumber)  
END FOR

| **Line of Code** | **Description** | **Execution Count** | **Cost per Line** | **Total Cost** |
| --- | --- | --- | --- | --- |
| Open file | Initialize file reading | 1 | 1 | 1 |
| FOR each line | Loop through all courses | n | 1 | n |
| Split line | Separate tokens | n | O(k) | O(n) |
| Create new course | Instantiate object | n | 1 | n |
| Insert into BST | Ordered by course number | n | O(log n) avg / O(n) worst | O(n log n) avg / O(n²) worst |
| **Total Runtime** | — | — | — | **O(n log n)** avg / **O(n²)** worst |
| **Memory Usage** | — | — | — | **O(n)** |

Total Runtime: Average O(n log n), Worst O(n²)  
Memory Usage: O(n)

**Summary**

| **Data Structure** | **Average Runtime** | **Worst Case** | **Memory Usage** | **Key Advantage** | **Key Drawback** |
| --- | --- | --- | --- | --- | --- |
| **Vector** | O(n) | O(n) | O(n) | Simple, predictable | Slow lookups |
| **Hash Table** | O(n) | O(n²) | O(n) | Very fast average performance | No ordering |
| **BST** | O(n log n) | O(n²) | O(n) | Maintains sorted order | Complex/unbalanced |
| **Analysis:** |  |  |  |  |  |

Each data structure offers distinct trade-offs in performance and functionality. The **vector** is simple and efficient for sequential access with predictable memory use, but it performs poorly for searches because every lookup requires scanning all elements. The **hash table** provides the fastest average insert and search times, operating in O(1) on average, making it ideal for direct lookups by course number; however, it lacks any inherent ordering of data and can suffer from degraded performance if many hash collisions occur. The **binary search tree (BST)** maintains courses in sorted order and supports efficient range queries with O(log n) average operations, but it is more complex to implement, uses extra memory for pointers, and can degrade to O(n²) if it becomes unbalanced.

**Recommendation**  
Chosen Data Structure: Hash Table  
Based on the advisor’s requirements for quick course lookups by course number,   
the hash table is the most efficient choice. It provides O(1) average performance   
for insertion and search operations, while maintaining manageable memory usage. While a BST maintains order and a vector offers simplicity, neither matches   
the average-case efficiency of a hash table for frequent lookup operations.   
All structures share O(n) file-reading complexity, but hash tables provide   
the fastest access once data is loaded.  
Final Recommendation:  
Use a Hash Table to store and retrieve course data efficiently.

MAIN

PRINT "Select option:"

PRINT "1. Load course data"

PRINT "2. Print all courses in alphanumeric order"

PRINT "3. Print specific course details"

PRINT "9. Exit"

READ userChoice

IF userChoice == 1 THEN

CREATE new HashTable

CREATE new BinarySearchTree

LoadCourses\_HashTable("course\_information.txt", hashTable)

LoadCourses\_BST("course\_information.txt", courseTree)

PRINT "Data successfully loaded into both structures."

ELSE IF userChoice == 2 THEN

PRINT "Computer Science Courses (alphanumeric order):"

PrintAllCourses\_BST(courseTree)

ELSE IF userChoice == 3 THEN

PRINT "Enter course number to view details:"

READ inputCourse

PrintCourseInfo\_HashTable(hashTable, inputCourse)

ELSE IF userChoice == 9 THEN

PRINT "Exiting program..."

EXIT

ELSE

PRINT "Invalid choice. Please try again."

END MAIN