Brianna Behrensprung   
October 7th,2025  
CS-300 Project One   
  
  
Vector Pseudocode:

STRUCT Course

courseNumber : STRING

courseTitle : STRING

prerequisites : LIST OF STRING

END STRUCT

FUNCTION LoadCourses\_Vector(fileName : STRING) RETURNS VECTOR OF Course

DECLARE courses AS EMPTY VECTOR OF Course

DECLARE courseNumbers AS EMPTY SET

OPEN file WITH fileName FOR READING

IF file NOT FOUND THEN

PRINT "Error: File not found."

RETURN EMPTY VECTOR

END IF

WHILE NOT END OF FILE

READ line FROM file

IF TRIM(line) == "" THEN CONTINUE

SPLIT line BY "," INTO tokens

IF SIZE(tokens) < 2 THEN

PRINT "Error: Invalid format on line -> " + line

CONTINUE

END IF

DECLARE newCourse AS Course

newCourse.courseNumber = TRIM(tokens[0])

newCourse.courseTitle = TRIM(tokens[1])

newCourse.prerequisites = EMPTY LIST

FOR i FROM 2 TO SIZE(tokens)-1

ADD TRIM(tokens[i]) TO newCourse.prerequisites

END FOR

APPEND newCourse TO courses

ADD newCourse.courseNumber TO courseNumbers

END WHILE

CLOSE file

FOR EACH course IN courses

FOR EACH prereq IN course.prerequisites

IF prereq NOT IN courseNumbers THEN

PRINT "Error: Prerequisite " + prereq + " for " + course.courseNumber + " not found."

END IF

END FOR

END FOR

RETURN courses

END FUNCTION

FUNCTION PrintCourseInfo\_Vector(courses : VECTOR OF Course, searchNumber : STRING)

FOR EACH course IN courses

IF course.courseNumber == searchNumber THEN

PRINT course.courseNumber + ", " + course.courseTitle

IF SIZE(course.prerequisites) == 0 THEN

PRINT "Prerequisites: None"

ELSE

PRINT "Prerequisites:"

FOR EACH p IN course.prerequisites

PRINT " - " + p

END FOR

END IF

RETURN

END IF

END FOR

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

END FUNCTION

FUNCTION PrintAllCourses\_Vector(courses : VECTOR OF Course)

SORT courses BY course.courseNumber

FOR EACH c IN courses

PRINT c.courseNumber + ", " + c.courseTitle

END FOR

END FUNCTION

FUNCTION Main\_Vector()

DECLARE courses AS EMPTY VECTOR OF Course

DO

PRINT "1: Load file"

PRINT "2: Print sorted course list"

PRINT "3: Print course info (enter course number)"

PRINT "9: Exit"

READ option

IF option == 1 THEN

PRINT "Enter filename:"

READ fileName

courses = LoadCourses\_Vector(fileName)

IF SIZE(courses) > 0 THEN PRINT "Loaded " + STR(SIZE(courses)) + " courses."

END IF

ELSE IF option == 2 THEN

IF SIZE(courses) == 0 THEN PRINT "No data. Load first."

ELSE PrintAllCourses\_Vector(courses)

END IF

ELSE IF option == 3 THEN

PRINT "Enter course number:"

READ searchNumber

IF SIZE(courses) == 0 THEN PRINT "No data. Load first."

ELSE PrintCourseInfo\_Vector(courses, searchNumber)

END IF

ELSE IF option == 9 THEN

PRINT "Exiting."

BREAK

ELSE

PRINT "Invalid choice."

END IF

LOOP

END FUNCTION

Hash Table Pseudocode:

STRUCT Course

courseNumber : STRING

courseTitle : STRING

prerequisites : LIST OF STRING

FUNCTION Constructor(number : STRING, title : STRING, prereqList : LIST OF STRING)

courseNumber = number

courseTitle = title

prerequisites = prereqList

END FUNCTION

END STRUCT

CLASS HashTable

table : ARRAY OF LIST OF Course

size : INTEGER

FUNCTION Constructor(capacity : INTEGER)

size = capacity

INITIALIZE table WITH size empty lists

END FUNCTION

FUNCTION HashFunction(key : STRING) RETURNS INTEGER

sum = 0

FOR EACH character IN key

sum = sum + ASCII(character)

END FOR

RETURN sum MOD size

END FUNCTION

FUNCTION Insert(course : Course)

index = HashFunction(course.courseNumber)

ADD course TO table[index]

END FUNCTION

FUNCTION Search(courseNumber : STRING) RETURNS Course OR NULL

index = HashFunction(courseNumber)

FOR EACH course IN table[index]

IF course.courseNumber == courseNumber THEN RETURN course

END FOR

RETURN NULL

END FUNCTION

FUNCTION GetAllCourses() RETURNS LIST OF Course

DECLARE list AS EMPTY LIST

FOR i FROM 0 TO size - 1

FOR EACH course IN table[i]

ADD course TO list

END FOR

END FOR

RETURN list

END FUN  
  
**Binary Search Tree Pseudocode:**

STRUCT Course

courseNumber : STRING

courseTitle : STRING

prerequisites : LIST OF STRING

END STRUCT

STRUCT Node

course : Course

left : Node

right : Node

END STRUCT

CLASS BST

root : Node

FUNCTION Insert(course)

IF root IS NULL THEN

root = NEW Node(course)

RETURN

END IF

current = root

WHILE TRUE

IF course.courseNumber < current.course.courseNumber THEN

IF current.left IS NULL THEN

current.left = NEW Node(course)

BREAK

ELSE

current = current.left

END IF

ELSE

IF current.right IS NULL THEN

current.right = NEW Node(course)

BREAK

ELSE

current = current.right

END IF

END IF

END WHILE

END FUNCTION

FUNCTION Search(key) RETURNS Node OR NULL

current = root

WHILE current IS NOT NULL

IF key == current.course.courseNumber THEN RETURN current

ELSE IF key < current.course.courseNumber THEN current = current.left

ELSE current = current.right

END IF

END WHILE

RETURN NULL

END FUNCTION

FUNCTION InOrderTraversal(node, outList)

IF node IS NULL THEN RETURN

InOrderTraversal(node.left, outList)

ADD node.course TO outList

InOrderTraversal(node.right, outList)

END FUNCTION

END CLASS

FUNCTION LoadCourses\_BST(fileName : STRING) RETURNS BST

OPEN file

IF file NOT FOUND THEN PRINT "File not found." AND RETURN EMPTY BST

DECLARE tempList AS EMPTY LIST

DECLARE courseNumbers AS EMPTY SET

WHILE NOT END OF FILE

READ line

IF TRIM(line) == "" THEN CONTINUE

SPLIT line BY "," INTO tokens

IF SIZE(tokens) < 2 THEN PRINT "Invalid line." AND CONTINUE

DECLARE c AS Course

c.courseNumber = TRIM(tokens[0])

c.courseTitle = TRIM(tokens[1])

c.prerequisites = EMPTY LIST

FOR i FROM 2 TO SIZE(tokens)-1

ADD TRIM(tokens[i]) TO c.prerequisites

END FOR

ADD c TO tempList

ADD c.courseNumber TO courseNumbers

END WHILE

CLOSE file

FOR EACH c IN tempList

FOR EACH p IN c.prerequisites

IF p NOT IN courseNumbers THEN

PRINT "Missing prerequisite " + p + " for " + c.courseNumber

END IF

END FOR

END FOR

DECLARE tree AS NEW BST

FOR EACH c IN tempList

tree.Insert(c)

END FOR

RETURN tree

END FUNCTION

FUNCTION PrintAllCourses\_BST(tree : BST)

DECLARE outList AS EMPTY LIST

tree.InOrderTraversal(tree.root, outList)

FOR EACH c IN outList

PRINT c.courseNumber + ", " + c.courseTitle

END FOR

END FUNCTION

FUNCTION PrintCourseInfo\_BST(tree : BST, searchNumber : STRING)

node = tree.Search(searchNumber)

IF node IS NULL THEN PRINT "Course not found."

ELSE

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

IF SIZE(node.course.prerequisites) == 0 THEN

PRINT "Prerequisites: None"

ELSE

PRINT "Prerequisites:"

FOR EACH p IN node.course.prerequisites

PRINT " - " + p

END FOR

END IF

END IF

END FUNCTION

**Runtime Analysis**   
Vector:

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # times executes | Total Cost |
| Read line From File | 1 | n | n |
| Split and Trim Tokens | 1 | n | n |
| Create and append course object | 1 | n | n |
| Validate prerequisites | 1 | n | n |
| Print results | 1 | 1 | 1 |
| Total Cost | | | 4n+1 |
| Runtime | | | O(n) |

Hash Table:

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # times executes | Total Cost |
| Read line From File | 1 | n | n |
| Split and Trim Tokens | 1 | n | n |
| Create Course object | 1 | n | n |
| Insert into hash table | O(1) | n | n |
| Validate prerequisites | 1 | n | n |
| Total Cost | | | 5n+1 |
| Runtime | | | O(n) |

Binary Search Tree:

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # times executes | Total Cost |
| Read line From File | 1 | n | n |
| Split and Trim Tokens | 1 | n | n |
| Create course Object | 1 | n | n log n |
| Insert into tree | log n | n | n |
| Validate prerequisites | 1 | n | n |
| Total Cost | | | 3n + n log n + 1 |
| Runtime | | | O(n log n) |

**Analysis – Advantages and Disadvantages:**   
**Vectors:**   
 A vector data structure shows a linear performance for file reading and object creation. Each course line from the file is read once, split into tokens and then converted into a course object. These operations occur in this order, resulting in a total runtime cost equivalent to the number of courses, which is represented by O(n). The process of appending each object to the vector has a returned constant cost per insertion. Therefore, keeping the performance of the dataset stable as it increases. Printing of all courses requires sorting them alphanumerically, which adds an additional runtime of O(n log n) when displaying the list. However, when searching for an individual course, a vector requires a full scan that leads to O(n) runtime. Memory usage of a vector is efficient, requiring space for course object and minimal indexing overheads. Overall, vectors perform best when dataset is smaller and when the operations are in order but can be inefficient for looking up data frequently.

**Hash Tables:**   
 Hash tables approach provides the most efficient average runtime for lookups. File reading and object creation still require O(n) time, as each course line must be parsed. Inserting data into the hash table averages constant time per course, O(1) because of the direct indexing by hash of the course number. In the rare case of excessive collisions or poor hash distributions, the performance might degrade to O(n^2). Although degrading is unlikely if the table is the correct size and has good hash functioning. A large advantage of the hash table is that it has almost instant access to individual courses. Searching for a course number is about O(1) on average, which allows the user to retrieve course titles and prerequisites with minimal delay. Even though memory usage is higher than that of a vector, efficiency is justified by faster speed. To produce an alphanumeric list of all courses, the progam must extract all the values and sort them, adding O(n log n) time for that operation. A hash table offers strong performances of insertions and search, therefore is perfect for frequent lookups with effective retrieval. The most common drawback or disadvantage is the complexity of managing collisions and sorting for an ordered output.

**Binary Search Tree:**

Binary search tree structure provides an ordered organization of courses based on their alphanumeric identifiers. File ready and course creation remain O(n), while insertions into the binary search tree are dependent on the balance of the structure. Most commonly each insertion is O(log n), resulting in a total build time of O(n log n). However, if the data is already sorted, the tree can degrade into a linear chain, increasing the runtime to O(n^2). A benefit to a binary search tree is the ability to produce a sorted list directly through an in-order traversal, which operates in O(n) time after the system is built. Searching for a course is O(log n) which makes it faster than a vector but slower than a hash table. The memory usage is moderate but includes additional storage for left and right pointers in each node that there is not a child node present already. The disadvantage of a binary search tree is that the unbalanced nature of a standard tree opens a risk of poor performance, even though it provides predictable sorting and relatively fast lookup.   
**Final Recommendation:**  
 Based on runtime efficiency, memory considerations, and practical use for the system, I recommend using a hash table for the data structure. It provides the best performance balance by having quick course retrieval for advisory queries and efficient data loading. The ability to use the course number as a direct key ensures constant time lookups, giving immediate access to the courses and their prerequisites. Although it requires additional sorting steps for the full course listing, I feel the overall speed and reliability outweigh the cost. Therefore, the final recommendation I have for the ABCU’s advising team should implement a hash table to serve as the foundation for the next project phase.