HASH TABLE BASED PSEUDOCODE

Load Courses

FUNCTION LoadCourses(fileName)

CREATE empty vector called courses

OPEN file with name fileName

IF file cannot be opened THEN

PRINT "Error: Unable to open file"

RETURN

FOR each line in the file DO

SPLIT line by comma into tokens

IF number of tokens < 2 THEN

PRINT "Format Error"

CONTINUE

CREATE new Course object

SET course.courseNumber = tokens[0]

SET course.courseTitle = tokens[1]

FOR each token from index 2 to end DO

ADD token to course.prerequisites

ADD course to courses vector

CLOSE file

CALL ValidatePrerequisites(courses)

RETURN courses

END FUNCTION

Validate Prerequisites

FUNCTION ValidatePrerequisites(courses)

FOR each course in courses DO

FOR each prerequisite in course.prerequisites DO

SET found = FALSE

FOR each otherCourse in courses DO

IF otherCourse.courseNumber == prerequisite THEN

SET found = TRUE

BREAK

IF found == FALSE THEN

PRINT "Error: Missing prerequisite"

Search Course

FUNCTION SearchCourse(courses, courseNumber)

FOR each course in courses DO

IF course.courseNumber == courseNumber THEN

PRINT course info and prerequisites

RETURN

PRINT "Course not found."

END FUNCTION

Sort and Print All Courses

FUNCTION PrintAllCourses(courses)

SORT courses by courseNumber (alphanumeric)

FOR each course in sorted courses DO

PRINT course.courseNumber + ": " + course.courseTitle

TREE BASED PSEUDOCODE

Load Courses

FUNCTION LoadCourses(fileName)

CREATE list allCourseLines

CREATE courseTree

OPEN file

FOR each line DO

SPLIT by comma

IF invalid THEN PRINT error CONTINUE

ADD line to allCourseLines

FOR each line in list DO

CREATE Course object

SET courseNumber, title

ADD prerequisites

INSERT course into courseTree

CALL ValidatePrerequisites(courseTree)

RETURN courseTree

Validate Prerequisites

FUNCTION ValidatePrerequisites(courseTree)

FOR each course in in-order traversal DO

FOR each prerequisite DO

IF not found in tree THEN PRINT error

Search Course

FUNCTION SearchCourse(courseTree, courseNumber)

SET node = root

WHILE node != NULL DO

IF match THEN PRINT info RETURN

ELSE IF smaller THEN go left

ELSE go right

PRINT "Not found"

Print All Courses

FUNCTION PrintAllCourses(node)

IF node is NULL THEN RETURN

CALL PrintAllCourses(node.left)

PRINT courseNumber + ": " + courseTitle

CALL PrintAllCourses(node.right)

MENU PSEUDOCODE

DISPLAY Menu:

1 - Load course data

2 - Print all courses

3 - Search for a course

9 - Exit

LOOP UNTIL choice is 9

IF choice is 1 THEN

PROMPT for filename

CALL LoadCourses()

ELSE IF choice is 2 THEN

CALL PrintAllCourses()

ELSE IF choice is 3 THEN

PROMPT for courseNumber

CALL SearchCourse()

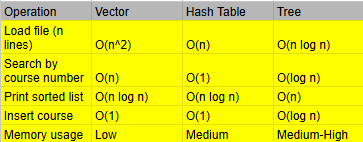
ELSE IF choice is 9 THEN

EXIT

ELSE

PRINT "Invalid choice"

Runtime Analysis



Analysis

Vectors are convenient and appropriate for small datasets. They support dynamic storing and appending of the data in course load time with efficiency. They are inefficient, however, for finding one course from the vector since they do linear scan (O(n) time). Printing the courses in terms of their names also requires an explicit sort like quicksort or selection sort with added O(n log n) of time complexity. Therefore, the bigger the dataset, the poorer the performance. Even though vectors are space-efficient and convenient, they are inappropriate for programs with heavy lookups or sorted views of the information.

Hash tables are well suited for fast lookup with constant-time (O(1)) average-case performance for insertions and lookups of course data. This makes it well suited for fast lookup and recovery of course data but doesn't recommend it for storing data in any sort of order. Printing out an alphanumeric course listing would require retrieving all values and sorting in O(n log n) time. This forfeits some of the performance gain. Hash tables will be more space-intensive than vectors or trees due to internal overhead and the threat of collision. Hash tables will be perfect for fast lookup but will need added logic for ordered printing.

A binary search tree is also a trade-off between order and performance. Proper implementation (as in the case of a balanced BST) search, insert, and delete are all O(log n), which is extremely fast for loading data and querying it. The biggest advantage of a BST in this application is that it serializes data into sorted (alphanumeric) order with in-order traversal, without having to sort whatsoever. BSTs therefore are well-suited for applications like ABCU’s advising tool, where it is not only necessary for fast lookups, but ordered lists too. BSTs can be more difficult to implement than vectors or hashtables, and will require slightly more space for node pointers.