**File Check Functions**

Boolean checkFormatting(line)

DEFINE array parts = SPLIT line by “,”

IF length of parts NOT equal to 3

return false

IF parts[0[ is empty or NOT alphanumerical

return false

IF parts[1] is empty

return false

IF parts[2] is NOT empty AND NOT validPreqs(parts[2])

return false

return true

Boolean validRereqs(prereqs)

DEFINE array list = SPLIT prereqs by delimiter

FOR EACH req in list

IF req NOT alphanumeric

return false

return true

Boolean fileCheck(file)

OPEN file as fileName

FOR EACH line in filename

IF NOT checkFormatting(line)

return false

return true

**Create Course Object Function (Vector Data Type)**

void createVectorObjects(file)

DEFINE struct for course object (course number, course title, course prereqs)

DECLARE vector data structure to hold course structs courses

IF fileCheck(file) returns false

PRINT Error

return

ELSE

OPEN file as fileName

FOR EACH line in fileName

INITIALIZE new course struct

SPLIT line by “,”

ASSIGN component 1 to number

ASSIGN component 2 to title

ASSIGN components 3+ to prereqs

APPEND course struct to courses

CLOSE file

return

**Print Course Objects Function (Vector Data Type)**

void searchVector(Vector<Course> courses, String courseNumber)

DECLARE struct object as struct

FOR all struct in courses

IF the course->number is the same as courseNumber

PRINT out the course information

FOR EACH prerequisite of the course

PRINT the prerequisite course information

**Create Course Objects Function (HashTable Data Type)**

void createHashObjects(file)

DEFINE struct for course object (course number, course title, course prereqs, ID number)

DECLARE hash table data structure to hold course struct courses

IF fileCheck(file) returns false

PRINT Error

return

ELSE

OPEN file as fileName

DECLARE int object idCounter equal to 1

FOR EACH line in fileName

INITIALIZE new struct object - course

SPLIT line by “,”

ASSIGN idCounter value to idNumber

ASSIGN component 1 to number

ASSIGN component 2 to title

ASSIGN components 3+ to prereqs

INSERT course object using ID into hash table courses

ITERATE idNumber

CLOSE fileName

return

**Print Course Objects Function (HashTable Data Type)**

void searchHash(HashTable<Course> courses, String courseNumber)

DECLARE struct object as struct

FOR all struct in courses

IF the course->number is the same as courseNumber

PRINT out the course information

FOR EACH prerequisite of the course

PRINT the prerequisite course information

**Create Course Objects Function (Binary Search Tree)**

void insertNode(currentNode, course)

IF course->courseId is less than currentNode.courseId

IF currentNode->left is NULL

currentNode->left = Node(course)

ELSE insertNode(currentNode->left, course)

ELSE

IF currentNode->right is NULL

currentNode->right = Node(course)

ELSE

insertNode(currentNode->right, course)

void createTreeObjects(file)

DEFINE struct for course object (course number, course title, course prereqs)

DEFINE node class for BST

INITIALIZE new binary search tree data structure

IF fileCheck(file) returns false

PRINT Error

return

ELSE

OPEN file as fileName

FOR EACH line in filename

SPLIT line by “,” into coursed, courseName, prereqs

CREATE new course struct object(coursed, courseName, prereqs)

IF tree root is NULL

tree root = Node(course)

ELSE

insertNode(tree root, course)

return

**Print Course Objects Function (Binary Search Tree Data Type)**

void searchTree(BST<Course> courses, String courseNumber)

DECLARE pointer object for tree root of data type Node

WHILE pointer is not null

IF courseNumber is less than pointer->courseNumber

ASSIGN pointer object to node root->left

IF courseNumber is greater than pointer->courseNumber

ASSIGN pointer object to node root->right

ELSE

PRINT course information in pointer object

FOR EACH prerequisite listed in pointer object

PRINT the prerequisite course information

PRINT courseNumber not found

**Print Menu Function**

void menuPrint()

DECLARE int choice object equal to 0

WHILE choice does not equal 9

DISPLAY menu options below

Menu Options:

1 - Load file data into data structure

2 - Print Alphanumerically sorted list of course objects from CS department

3 - Print the course title and the prerequisites for any individual course

9 - Exit the program

READ user input into choice variable

IF choice equals 1

EXECUTE loadData

DISPLAY “Data loaded successfully”

ELSE IF choice equals 2

EXECUTE printSorted function based on data structure used

DISPLAY “Courses printed in alphanumerical order”

ELSE IF choice equals 3

DISPLAY “Enter the course ID you want to look up”

READ user input into courseID

EXECUTE printDetails(data structure, courseID) function based on data structure used

ELSE IF choice equals 9

DISPLAY “Exiting now..”

ELSE

DISPLAY “Invalid choice”

**Alphanumeric Printing Functions(All Three Data Structures)**

void selectionSort(list)

DECLARE int min = 0

DECLARE int size = size(list)

FOR (i = 0) iterate size number of times

min = i

FOR (j = 0) iterate size number of times

IF list[j] is less than list[min]

min = j

IF i does NOT equal min

SWAP list’s item (list[i], list[min])

return

void printSortedVector(courses)

CALL selectionSort(courses)

FOR EACH course in courses

PRINT course

void printSortedHash(courses)

DEFINE new vector list as courseList

FOR EACH item in courses

APPEND item to courseList

CALL selectionSort(courseList)

FOR EACH course in courseList

PRINT course

void printSortTree(node)

IF node is NOT NULL

CALL printSortTree(node->left)

PRINT node->course

CALL printSortTree(node->right)

**Print Details Functions(All Three Data Structures)**

void printDetailsVector(vector, courseID)

FOR EACH course in vector

IF course->courseID = courseID

DISPLAY course details

void printDetailsHash(hash table, courseID)

FOR ALL courses in hash table

IF courses->courseID = courseID

DISPLAY course details

void printDetailsTree(BST, courseID)

DECLARE pointer currNode to tree root

IF currNode is NULL

DISPLAY “Empty BST”

WHILE currNode is NOT NULL

IF currNode->courseID equals courseID

DISPLAY currNode details

return

ELSE IF courseID is less than currNode->courseID

currNode = currNode->left

ELSE IF courseID is greater than currNode->courseID

currNode = currNode->right

DISPLAY “Course ID not found”

return

**1. File Check Functions (fileCheck, checkFormatting, and validPrereqs)**

**Time Complexity:**

* **checkFormatting(line)**:
  + **Splitting line into parts**: Splitting a string of length m into components has a time complexity of O(m).
  + **Checking each part's conditions**: These checks are constant time operations for each component, so O(1) for each condition.
  + **validPrereqs(prereqs)**: Assuming p prerequisites, it iterates over each and checks if it's alphanumeric, so the complexity is O(p).
* **fileCheck(file)**:
  + **For each line in the file**: This function calls checkFormatting(line) for each line, so if the file has n lines, each of length m, and each with p prerequisites, the total time complexity would be: O(n \* (m + p))

**Space Complexity:**

* The space complexity is O(n \* (m + p)) to store the file's data in memory while performing checks.

**2. Creating Course Objects**

**Vector Data Type (createVectorObjects):**

**Time Complexity**:

* **File check**: Calling fileCheck takes O(n \* (m + p)).
* **Creating and appending course objects**: For each of the n lines, the following steps occur:
  + **Splitting** the line: O(m)
  + **Assigning and appending** to the vector: Appending to a vector takes O(1).

So the overall time complexity for creating the vector is:

O(n \* m)

Because we assume p is small compared to n and mmm.

**Space Complexity**:

* The space complexity is O(n \* (m + p)) where n is the number of course objects and m + p is the data stored per course.

**Hash Table Data Type (createHashObjects):**

**Time Complexity**:

* **File check**: Same as before, O(n \* (m + p))).
* **Creating course objects and inserting into the hash table**:
  + **Splitting the line**: O(m)
  + **Inserting into hash table**: Insertion into a hash table is O(1) on average.

Thus, the overall time complexity for the hash table approach is:

O(n \* m)

**Space Complexity**:

* Same as vector, O(n \* (m + p)), but the hash table has overhead for storing the key-value pairs, making space usage slightly higher.

**Binary Search Tree (BST) (createTreeObjects):**

**Time Complexity**:

* **File check**: O(n \* (m + p))
* **Creating course objects and inserting into the BST**:
  + **Splitting the line**: O(m)
  + **Inserting into BST**: Insertion into a BST takes O(logn) on average for balanced trees, but in the worst case (if the tree is unbalanced), it can take O(n).

Thus, the worst-case time complexity for the BST approach is:

O(n \* (m + p))

In the average case, this is O(n \* (m + logn)).

**Space Complexity**:

* The space complexity is O(n \* (m + p)) similar to the vector, with additional overhead for BST node pointers.

**3. Analysis of Data Structures: Advantages and Disadvantages**

**Vector:**

* **Advantages**:
  + Simple to implement and manage.
  + Efficient for appending new course objects.
  + Sorting is straightforward with algorithms like selection sort.
* **Disadvantages**:
  + Searching and sorting require O(n) and O(n logn), respectively.
  + Memory allocation can lead to increased space usage if the vector resizes.

**Hash Table:**

* **Advantages**:
  + Fast lookups for course details, typically O(1).
  + Simple insertions of new course objects.
* **Disadvantages**:
  + Collisions can reduce efficiency to O(n) in the worst case.
  + Inherently unordered, requiring additional steps to sort the data.

**Binary Search Tree (BST):**

* **Advantages**:
  + Naturally ordered, so searching for alphanumeric course IDs and traversing in sorted order is efficient.
  + Efficient insertion in a balanced tree (average O(logn)).
* **Disadvantages**:
  + If the tree becomes unbalanced, insertion and searching can degrade to O(n).
  + Requires additional overhead to maintain the tree structure.

**4. Recommendation**

Given the analysis, **Hash Table** is the most efficient in terms of lookups and insertions (average O(1)). However, since sorting is a requirement and the hash table does not inherently store items in order, you’d need to extract the courses and sort them separately, which adds overhead.

**Binary Search Tree (BST)** is a strong candidate due to its natural ordering, allowing both insertion and in-order traversal for sorted output to be efficient. If the tree stays balanced, it offers O(logn) operations for insertion and searching, making it a good balance between time and space efficiency.

Thus, **Binary Search Tree (BST)** is recommended, especially if a self-balancing BST (like an AVL or Red-Black Tree) is used to ensure efficiency across all operations.