Ashley Santo

**Vector Data Structure Pseudocode**

readFile() {

OPEN file

IF file not found:

THROW exception

ELSE:

WHILE not end of file:

READ each line of file

SPLIT string on commas

COUNT that there are at least two parameters

IF there are less than two parameters:

THROW exception

CREATE a new course object for ID, course name, and prerequisites

ADD course object to vector

}

validateCoursePrerequisites() {

FOR each course in vector

IF a course has a prerequisite:

VERIFY prerequisite exists in one of the course objects within the vector

IF course not found:

OUTPUT no prerequisite course exists

}

outputCourseInformation() {

INPUT course name or ID

FOR all courses in vector:

IF input matches course in vector:

OUTPUT course information for entire course object

FOR prerequisites found in course object:

OUTPUT prerequisite information

ELSE:

OUTPUT course not found

}

**Hash Table Data Structure Pseudocode**

readFile() {

OPEN file

IF file not found:

THROW exception

ELSE:

WHILE not end of file:

READ each line of file

SPLIT string on commas

COUNT that there are at least two parameters

IF there are less than two parameters:

THROW exception

CREATE a new course object for ID, course name, and prerequisites

ADD course object to hash table

}

validateCoursePrerequisites() {

FOR each course in hash table:

IF a course has a prerequisite:

VERIFY prerequisite exists in one of the course objects within the hash table

IF course not found:

OUTPUT no prerequisite course exists

}

outputCourseInformation() {

INPUT course name or ID

FOR all courses in hash table:

IF input matches course in hash table:

OUTPUT course information for entire course object

FOR prerequisites found in course object:

OUTPUT prerequisite information

ELSE:

OUTPUT course not found

}

**Binary Search Tree Data Structure Pseudocode**

readFile() {

OPEN file

IF file not found:

THROW exception

ELSE:

WHILE not end of file:

READ each line of file

SPLIT string on commas

COUNT that there are at least two parameters

IF there are less than two parameters:

THROW exception

CREATE a new course object for ID, course name, and prerequisites

ADD course object to binary search tree

}

validateCoursePrerequisites() {

FOR each course in binary search tree:

IF a course has a prerequisite:

VERIFY prerequisite exists in one of the course objects within the tree

IF course not found:

OUTPUT no prerequisite course exists

}

outputCourseInformation() {

INPUT course name or ID

FOR all courses in binary search tree:

IF input matches course in tree:

OUTPUT course information for entire course object

FOR prerequisites found in course object:

OUTPUT prerequisite information

ELSE:

OUTPUT course not found

}

**Menu Pseudocode**

main() {

SET user input to 0

WHILE user input does not equal exit:

OUTPUT “Menu:”

OUTPUT “1. Load Course Data”

OUTPUT “2. Print Course List”

OUTPUT “3. Print Single Course”

OUTPUT “4. Exit”

INPUT user choice

SWITCH (user input):

CASE 1:

LOAD course information

BREAK

CASE 2:

OUTPUT all courses in order alphanumerically

BREAK

CASE 3:

FIND single course

OUTPUT course information

BREAK

OUTPUT “Goodbye”

**Print Course List Alphanumerically Pseudocode**

printCourseList() {

FOR all courses in data structure:

IF first entry is higher:

REPLACE first entry with next on list

ELSE:

ADD next item to end of list

OUTPUT sorted list in ascending alphanumerical order

**Run-Time Analysis of Vector Data Structure**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open file | 1 | n | n |
| IF file not found: | 1 | n | n |
| THROW exception | 1 | 1 | 1 |
| ELSE: | 1 | n | n |
| WHILE not end of file: | 1 | n | n |
| READ each line of file | 1 | n | n |
| SPLIT string on commas | 1 | n | n |
| COUNT that there are at least two parameters | 1 | n | n |
| IF there are less than two parameters: | 1 | n | n |
| THROW exception | 1 | 1 | 1 |
| CREATE a new course object for ID, course name, and prerequisites | 1 | n | n |
| ADD course object to vector | 1 | n | n |
| **Total Cost** | | | 10n + 2 |
| **Runtime** | | | O(n) |

**Run-Time Analysis of Hash Table Data Structure**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open file | 1 | n | n |
| IF file not found: | 1 | n | n |
| THROW exception | 1 | 1 | 1 |
| ELSE: | 1 | n | n |
| WHILE not end of file: | 1 | n | n |
| READ each line of file | 1 | n | n |
| SPLIT string on commas | 1 | n | n |
| COUNT that there are at least two parameters | 1 | n | n |
| IF there are less than two parameters: | 1 | n | n |
| THROW exception | 1 | 1 | 1 |
| CREATE a new course object for ID, course name, and prerequisites | 1 | n | n |
| ADD course object to hash table | 1 | n | n |
| **Total Cost** | | | 10n + 2 |
| **Runtime** | | | O(n) |

**Run-Time Analysis of Binary Search Tree Data Structure**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open file | 1 | n | n |
| IF file not found: | 1 | n | n |
| THROW exception | 1 | 1 | 1 |
| ELSE: | 1 | n | n |
| WHILE not end of file: | 1 | n | n |
| READ each line of file | 1 | n | n |
| SPLIT string on commas | 1 | n | n |
| COUNT that there are at least two parameters | 1 | n | n |
| IF there are less than two parameters: | 1 | n | n |
| THROW exception | 1 | 1 | 1 |
| CREATE a new course object for ID, course name, and prerequisites | 1 | n | n |
| ADD course object to binary search tree | 1 | n | n |
| **Total Cost** | | | 10n + 2 |
| **Runtime** | | | O(n) |

**Analyzation of Each Data Structure and Recommendation**

The vector data structure is definitely one of the more common methods for holding information. The advantages of this structure are that the size of the vector can constantly be changed even during runtime. On top of this, items can easily be inserted or deleted from a vector at the front or end. So, if the list is constantly changing or being updated, this would be the structure to go with. Also, because the index of a vector is an expression, it makes it extremely easy to look up specific items on a list via integers. That being said, there are some disadvantages to it. One of the biggest ones involving memory management. Insertions or deletions may need additional memory by causing memory to spike when adding or deleting items. This may lead to running out of memory even though only a smaller portion is actually required. Also, items can only be added to the head and tail ends fast, any other insertions could lead to slower processing. Overall, vectors are generally a safe go to for data structures.

With hash tables, the biggest advantage is being able to quickly search or pull up information especially when there are a lot of entries to sort through. Since they work similar to a dictionary, they can quickly find the search object and access the data. The problem with hash tables comes from collisions. This is when an item being inserted into the table maps to the same bucket as another existing item. Too many collisions can cause problems down the road and make the program inefficient. With that in mind, there are corrections that can be made to account for this. Chaining is a technique where buckets have a list of items or there is open addressing where collisions are resolved by looking for an empty bucket somewhere else in the table.

Lastly, there are binary search trees. They are extremely memory efficient and don’t reserve more memory than they need. They are much more useful for storing data if there is a hierarchy. Items in the tree can also easily be ordered such as through in-order transversals or reverse in-order transversals. Binary search trees are also very useful for looking up ranged queries. So, if there are certain keys that need to be found in a large range, it can easily be done as compared to other data structures. If the tree is balanced, it can make inserting, deleting or looking up items simple to do. The disadvantages are this structure is mainly efficient when there is a large number of elements to be sorted through. With less elements, binary search trees can become more work than they are worth. There’s also the fact that they only work on lists that are sorted and kept that way. If items are constantly being added to the list, this would not be the best structure to go with. It’s also not known to best at quickly searching and finding elements in comparison to some other structures such as hash tables.

Based on everything stated above, my recommendation would be going with hash tables. My choice for this is because of the benefits it has compared to the other structures. A hash table excels are searching for items which is one of the main purposes of this program. With a vector, items can only be searched by integers and it’s not as efficient. And since there wouldn’t be many changes to the size of the course catalog, it takes away the benefits of using one. And while a binary search tree would also have been a great option due to the hierarchy sorting, there just aren’t enough elements to justify implementing all that work for so little information. On top of that, it won’t search and find items as quickly as a hash table would. Looking at the disadvantages of a hash table, they can be corrected if necessary but with the little number of courses, collisions shouldn’t be too much of an issue. With needing to look up information on a class quickly, I just think a hash table is the way to go in order to efficiently get that done.