**Pseudocodes and Runtime Analyses for Accessing Course Information**  
by  
John Clark  
for  
ABC University (ABCU)

The following pseudocodes represent algorithms that may be used to upload data from a file into data structures in memory so that data may be accessed, sorted, searched through, and printed. The runtime analyses show the Big O values of each algorithm so the worst-case runtime may be used to consider which data structure is best.

**Pseudocodes**

**Display menu for user interface** While user selection does not equal “Exit”  
 Print the following menu options to the console  
 1. Load Data Structure ()  
 2. Print Course List () in alphanumeric order  
 3. Print Specific Course (Course Number)  
 4. Exit (Leave while loop and free memory used by data structure)  
 Get user choice  
 If user choice equals a menu option  
 Perform the chosen function  
 Else  
 Print error message stating that choice is not a valid option

**Open a File (File Path and Name)**  
 Use an “open” command with the file’s path and name (e.g., open(“path/file.csv”))

**Create List of Every Course Number (Opened File)** Initialize an AvailableCourse list to save every available Course NumberWhile there are more lines in the file  
 Save the first value of each line (i.e., Course Number) to the AvailableCourse list  
 If AvailableCourse is empty  
 Print a message stating the file is empty and   
 Close the file (e.g., file.close())  
 Exit this function and return to the menu

**Read Data from a File (Opened File)**  
 Set a temporary variable to the first line of the file  
 While there are more lines in the file  
 Parse() the line into separate elements  
 Assign the temporary variable to the next line for the next iteration  
 After all lines are read and parsed, close the file (e.g., file.close())

**Parse Each Line (Line)**  
 Split the passed in line using commas at the delimiter for the provided CSV file  
 Store each element as a list to be used to create a Course Object  
 (1st: Course Number, 2nd: Name, 3rd: Prerequisite 1, 4th: Prerequisite 2, … , Nth: Prerequisite N)  
 FoundError = Check for File Format Errors(List)  
 If FoundError is 0  
 Store the list with the associated Create New Course Object(List) function  
 Else   
 Print the first FoundError for the line in the file so it may be corrected  
 Free the memory that has been used, as it is not complete  
 Close the file (e.g., file.close())  
 Exit this function and return to the menu

**Check for File Format Errors (List)** If the list has less than 2 elements  
 Return an error indicating that there are not enough elements  
 For each element in the list  
 If the element is not a string  
 Return an error indicating an invalid element type  
 If the element index is greater than the 2nd element  
 CheckElement = current element being checked  
 PrerequisiteCheck = False  
 For each element in the AvailableCourse list  
 If the CheckElement matches the AvailableCourse element  
 PrerequisiteCheck = True  
 Exit this for loop  
 If PrerequisiteCheck = False  
 Return an error indicating the prerequisite course cannot be found in the AvailableCourse list  
 Return 0

**Define a class structure to create Course Objects**  
 Private declarations  
 String courseNumber  
 String name  
 Vector<string> prerequisites

Public declarations  
 Course Object Constructor (courseNumber, name, prerequisites)  
 Getter functions  
 getCourseNumber  
 return courseNumber  
 getName  
 return name  
 getPrerequisites  
 return prerequisites

------------------------------------------------------------------------------------------------------------------------------------------  
**Vector Data Structure**

**Load Data Structure ()** Define a vector named “CourseList” (e.g., vector<Course> CourseList)  
 Open a File (File Path and Name)  
 Create List of Every Course Number (Opened File)  
 Read Data from a File (Opened File)

**Print Course List () in alphanumeric order** Sort the CourseList vector in alphanumeric order (e.g., the standard sort function in C++)  
 For each Course Object in the CourseList vector  
 Print each element of the Course Object (i.e., Course Number, Name, and any Prerequisites)

**Print Specific Course (Course Number)**  
 For each Course Object in the CourseList vector  
 If the passed in Course Number matches the Course Number of the Course Object Print each element of the Course Object (i.e., Course Number, Name, and any Prerequisites)  
 Exit the for loop  
 If the no match was found signified by the for loop completing  
 Print a message indicating that the course was not found

**Create New Course Object (List)** courseNumber = 1st element of passed in list  
 name = 2nd element of passed in list  
 If the passed in list has more than 2 elements  
 For each element in the list after the second  
 Add the element to a prerequisites vector or list  
 Else  
 prerequisites = NonenewCourseObject = Course Object Constructor (courseNumber, name, prerequisites)  
 Add newCourseObject to the CourseList vector

------------------------------------------------------------------------------------------------------------------------------------------  
**Hash Table Data Structure**

**Load Data Structure ()**  
 Define a hash table named “CourseList”   
 HashTable\* CourseList;  
 Course Object course;  
 CourseList = new HashTable();  
 Open a File (File Path and Name)  
 Create List of Every Course Number (Opened File)  
 Read Data from a File (Opened File)

**Print Course List () in alphanumeric order**  
 Define a list named SortCourseList  
 For each bucket in the hash table  
 For each Course Object in the bucket  
 Add the Course Object to SortCourseList  
 Sort the SortCourseList list in alphanumeric order (e.g., the standard sort function in C++)  
 For each Course Object in SortCourseList  
 Print each element of the Course Object (i.e., Course Number, Name, and any Prerequisites)

**Print Specific Course (Course Number)**  
 Calculate a Hash Value (Course Number)  
 Search for the Course Object in the hash table bucket using the calculated hash value  
 For each Course Object in the bucket the hash value key leads to  
 If Course Object is found  
 Print each element of the Course Object (i.e., Course Number, Name, and any Prerequisites)  
 Exit the for loop  
 If the no match was found signified by the for loop completing  
 Print a message indicating that the course was not found

**Calculate a Hash Value (Course Number)**  
 Define a hash function that takes the Course Number as input and returns a hash value  
 Use a suitable hash algorithm to compute the hash value  
 Return the hash value

**Create New Course Object (List)** courseNumber = 1st element of passed in list  
 name = 2nd element of passed in list  
 If the passed in list has more than 2 elements  
 For each element in the list after the second  
 Add the element to a prerequisites vector  
 Else  
 prerequisites = None

newCourseObject = Course Object Constructor (courseNumber, name, prerequisites)  
 Calculate a Hash Value (Course Number)  
 Insert newCourseObject into CourseList using the calculated hash value

------------------------------------------------------------------------------------------------------------------------------------------  
**Binary Search Tree Data Structure**

**Load Data Structure ()**  
 Define a binary search tree named “CourseList”  
 BinarySearchTree\* CourseList;  
 CourseList = new BinarySearchTree();  
 Course Object course;  
 Open a File (File Path and Name)  
 Create List of Every Course Number (Opened File)  
 Read Data from a File (Opened File)

**Print Course List ()**  
 In Order Traversal (CourseList)

**Print Specific Course (root, Course Number)**  
 If root is null  
 Return null because the course was not found  
 Else if the Course Object Course Number is equal to the root’s Course Number  
 Print each element of the Course Object (i.e., Course Number, Name, and any Prerequisites) at the current root  
 Exit the function  
 Else if the Course Object Course Number is less than the root’s Course Number  
 Recursively search the left subtree with Print Specific Course (root.left, Course Number)   
 Else  
 Recursively search the right subtree with Print Specific Course (root.left, Course Number)

**Create New Course Object (List)** courseNumber = 1st element of passed in list  
 name = 2nd element of passed in list  
 If the passed in list has more than 2 elements  
 For each element in the list after the second  
 Add the element to a prerequisites vector  
 Else  
 prerequisites = None

**In Order Traversal (Node, CourseList))**  
 Set current node to the binary search tree’s root  
 If the current node is not null  
 In Order Traversal (Node->left) // Recursive traverse  
 Print each element of the Course Object (i.e., Course Number, Name, and any Prerequisites) at the current node  
 In Order Traversal (Node->right) // Recursive traverse

**Insert (Node, Course Object)**  
 If root is null  
 Create a new tree node with the course data  
 Else if the Course Object Course Number is less than the root’s Course Number  
 Recursively insert into the left subtree with Insert (root.left, Course Object)  
 Else if the Course Object Course Number is greater than the root’s Course Number  
 Recursively insert into the right subtree with Insert (root.right, Course Object)  
 Return root to keep track of changes to the Binary Search tree

------------------------------------------------------------------------------------------------------------------------------------------

**Runtime Analyses**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Code | Line Cost | # Times Executes | Total Cost |
| All Data Structures | Display menu for user interface | 6 | User Defined (n) | n |
| Open a File (File Path and Name) | 1 | 1 | 1 |
| Create List of Every Course Number (Opened File) | 7 | # Lines (n) | n |
| Read Data from a File (Opened File) | 5 | # Lines (n) | n |
| Parse Each Line (Line) | 9 | # Elements (n) | n |
| Check for File Format Errors (List) | 10 | # Elements (n) | n |
| Define a class structure to create Course Objects | 1 | 1 | 1 |
| Total | | | | 5n + 2 |
| O(n) |
| Vector | Load Data Structure () | 4 | 1 | 1 |
| Print Course List () | 3 | Sorting (n log n) | n log n |
| Print Specific Course (Course Number) | 4 | # Courses (n) | n |
| Create New Course Object (List) | 7 | # Elements (n) | n |
| Vector Total | | | | 7n + 3 + (n log n) |
| O(n log n) |
| Hash Table | Load Data Structure () | 6 | # Lines (n) | n |
| Print Course List () | 7 | Sorting (n log n) | n log n |
| Print Specific Course (Course Number) | 6 | # Objects in Bucket (n) | n |
| Calculate a Hash Value (Course Number) | 3 | 1 | 1 |
| Create New Course Object (List) | 5 | # Prerequisites (n) | n |
| Hash Table Total | | | | 8n + 3 + (n log n) |
| O(n log n) |
| Binary Search Tree | Load Data Structure () | 6 | # Lines (n) | n |
| Print Course List () | 1 | # Objects in Tree (n) | n |
| Print Specific Course (root, Course Number) | 5 | # Objects in Tree (n) | n |
| Create New Course Object (List) | 5 | # Prerequisites (n) | n |
| In Order Traversal (Node, CourseList)) | 5 | # Objects in Tree (n) | n |
| Insert (Node, Course Object) | 5 | # Objects in Tree (n) | n |
| Binary Search Tree Total | | | | 11n + 2 |
| O(n) |

**Conclusion**

A vector data structure can be sorted alphanumerically without altering its structure (e.g., O(n log n) worst-case time complexity with quicksort or mergesort) before each Course Object is printed (e.g., print time complexity O(n)). A vector is simple to implement, requires relatively less memory than the other data structures, and can access elements in constant time due to their continuous memory location. Insertion and deletion of elements in a vector may require lots of shifting and may require new vectors to be created as they have a fixed size.

A hash table uses calculated keys, which do not inherently store the Course Objects by order of their Course Number. Printing all Course Objects in alphanumeric order from an unsorted hash table might involve several comparisons and increased time complexity (e.g., O(n2)), as each Course Object may need to be compared against every other Course Object in the hash table prior to printing them in alphanumeric order. In the above runtime analysis for the hash table, I used the storing of each Course Object from the hash table into a new list so it could be sorted (e.g., O(n log n) worst-case time complexity with quicksort or mergesort) before being printed. However, creating a secondary copy of a chosen data structure, such as from a hash table to a list, and sorting that data would require extra memory which may not be available given the amount of data. While hash tables can easily be scaled and keys/buckets can be accessed fast, they require extra memory and collision resolutions if the same hash value is returned.

Using specific recursive calls, a binary search tree can print Course Objects in alphanumeric order without the need for sorting using an in-order traversal (e.g., print time worst-case complexity O(n)), as when objects are stored in a binary search tree, they are placed in relative order to each other. However, if the largest value is the root and each inserted value after is less than the root, searching through a binary search tree would be equivalent to a linear search with no advantages. Additionally, since a binary search tree has nodes that keep track of its left and right children, it requires more memory than a vector.

I would recommend choosing a binary search tree for its efficient searching and printing time complexities for use in ABC University’s course information program. The binary search tree has the best worst-case time complexity of O(n) as seen in the runtime analyses above. If it is possible to balance the tree, possibly by defining the root as the middle element in the series of provided data, the average time complexity would be O(log n), which is much faster than the sorting functions the other data structures must use.