1. ***Resubmit pseudocode from previous pseudocode assignments and update as necessary***

Milestone one: vector

Written pseudocode to open, read, parse and validate the file

```
// Pseudocode to open the file, read data, analyze each line, and check any format errors

BEGIN

// Open the file
FILE coursesFile = OPEN(“courses”)

// Create a vector to store course data
    VECTOR<Course> courseVector = INITIALIZE()

    // Read file line through line

    // Read a line from the file
    STRING ling = READ_LINE(coursesFile)

    // Split the line into different tokens (course number, title)
        LIST<STRING> tokens = SPLIT

    // Check if there are at least two courses
    IF LENGTH < 2 THEN
    PRINT “Error: Line does not meet the requirement”
```

```
CONTINUE
        ENDIF

// Extract course number and title

STRING courseNumber = tokens [0]

STRING courseTitle = tokens[1]


// Create a course object and store any of the data

Course = CREATE_COURSE(courseNumber, courseTitle)


// Add course

APPEND(courseAdd)


ENDWHILE
```

Pseudocode to create the course objects and the store in vector

```
// Pseudocode to create the course objects and to be able to store them in a vector


STRUCT Course

STRING courseNumber

STRING courseTitle

LIST<STRING> prerequisites

END


FUNCTION TO CREATE_COURSE(courseNumber, courseTitle, prerequisites)

Course

Course.courseNumber = courseNumber
```

Course.courseTitle = courseTitle

Course.prerequisites = prerequisites

RETURN to course

END

Pseudocode to be able to search for a certain course

// Pseudocode to search the vector for a specific course

FUNCTION PRINT_COURSE_INFO(courseNumber, courseVector)

BOOLEAN courseFound = incorrect

FOR EACH course IN courseVector DO THIS

IF course.courseNumber == courseNumber THEN

PRINT “course number”

PRINT “course title”

PRINT “prerequisites”

ENDIF

courseFound is found

IF courseFound is not found

PRINT “Course,” “courseNumber,” “NOT FOUND.”

END

Milestone two: Hash Table

Pseudocode for Loading the Data into the Hash Table

1. **The first step would be to open and read the file that is presented**

Procedure OpenAndReadFile (provide the filename here):

Open file with the filename

IF file is not opening, PRINT error message and EXIT

INITIALIZE the hash_table as a hash table data structure

FOR each line present in the file:

IF line is not empty :

Parse(has_table)

CLOSE file

2. **Parse each line in the code and validate the code**

PROCEDURE ParseLine(line, hash_table):

SPLIT line into tokens using whitespace

IF the number of tokens are < 2:

PRINT “Error: Invalid data in line”

EXIT

course_number = tokens

course_title = tokens (1 to ...)

Prerequisites = tokens // prerequisites would be the last token

FOR EACH prerequisite:

IF prerequisite is not found in the hash table:

PRINT “ERROR: Prerequisite course can’t be found”

EXIT

3. **Psuedocode to print any Course Information from the Hash Table**

PROCEDURE to print the course information(hash_table):

FOR each entry that is put into the hash_table:

PRINT “Course Number: entry.course.number”

PRINT “Course Title: entry.course.title”

IF entry.prerequisites has data:

PRINT “Prerequisities”

EXIT

Milestone three: Binary Search Tree

// 1. Open and Read File
PROCEDURE OpenAndReadFile(filename):
OPEN file with filename
IF file is not open:

```
PRINT “Error: Unable to open file.”
EXIT

INITIALIZE tree as a binary search tree data structure

// Skip the header row
READ a line from the file

FOR each line in the file:
IF line is not empty:
ParseLine(line,tree)

CLOSE file

// 2. Parse Each Line and Validate
PROCEDURE ParseLine(line,tree):
SPLIT line into tokens using ‘,’ (comma)

IF number of token is less than 4:
PRINT “ERROR: Invalid data in line.”
EXIT

bidID = first token
description = second token

// Create a Bid object
CREATE bid_object with bidId, description, amount, and fund

// Insert the bid into the binary search tree
INSERT bid_object into tree based on bidId

// 3. Print Bid Information from Binary Search Tree
PROCEDURE PrintBidInformation(tree):
PERFORM In-order Traversal of the tree:
FOR each node in the traversal:
PRINT “Bid ID: “ + node.bidId
PRINT “Description: “ + node.description
PRINT “Amount: “ +node.amount
```

```
PRINT “Fund: “ node.fund
PRINT “_________”

EXIT

// Data Structures
struct Bid {
        string bidId;
         string description;
        double amount;
        string fund;
}

struct Node {
        Bid bid;
        Node* left;
        Node* right;
}

// Binary Search Tree Class
class BinarySearchTree {
        private:
        Node* root;

        public:
                BinarySearchTree() { root = null; }
                ~BinarySearchTree() { /* Recursive deletion */ }

        // Insert a bid
        void Insert(Bid bid) {
                 if (root is null) {
                        root = new Node(bid, null, null);
                         } else {
                          addNode(root, bid);
    }
   }

   // Helper for insertion
```

```
  void addNode(Node* node, Bid bid) {
        if (bid.bidId < node.bidId) {
                if (node.left is null) {
                        node.left = new Node(bid, null, null);
                 } else {
                        addNode(node.left, bid);
        }
   } else {
        // ... (similar for right subtree)
        }
  }

// Remove a bid
  void Remove(string bidId) {
        root = removeNode(root, bidId);
  }

  // Helper for removal
  Node* removeNode(Node* node, string bidId) {
    // ... (handle cases for leaf, one child, two children)
  }

  // Search for a bid
  Bid Search(string bidId) {
        // ... (traverse the tree)
  }

  // Display all bids
  void InOrder() {
        inOrderTraversal(root);
  }

  // Recursive in-order traversal
  void inOrderTraversal(Node* node) {
          if (node is not null) {
           inOrderTraversal(node.left);
          // Print bid information (bidId, description, amount, fund)
                inOrderTraversal(node.right);
```

```
        }
    }
}

// Function to load bids from a CSV file
PROCEDURE loadBids(filename):
        // ... (Open file, skip header, parse each line, create Bid objects)

// Main function
int main() {
        BinarySearchTree tree;
         int choice;
        do {

          // Display the menu (1. Load, 2. Display, 3. Find, 4. Remove, 9. Exit)
                // Read user's choice
   switch (choice) {
     case 1:
      // Load bids from the CSV file
      // ... (Read filename, call loadBids, insert bids into tree)
      break;

     case 2:
      // Display all bids
      tree.InOrder();
      break;

     case 3:
      // Find a bid
      // ... (Read bidId, search for bid, print if found)
      break;

     case 4:
      // Remove a bid
      // ... (Read bidId, remove bid from tree)
      break;

     case 9:
```

```
      // Exit
      break;

    default:
      PRINT "Invalid choice.\n";
   }
 } while (choice is not 9);
 return 0;
}
```

2. Create pseudocode for a menu

```
# Initialize the data structure to store course information
course_data = []

# Display the menu to the user
while True:
        print("Menu:")
        print("1. Load Course Data")
        print("2. Print Sorted Course List")
        print("3. Print Course Details")
        print("9. Exit")

        # Get the user's choice
        choice = input("Enter your choice: ")

        # Process the user's choice
        if choice == "1":
        # Load the course data from the file into the data structure
        load_course_data(course_data) # Replace with your actual loading data
        print("Course data has loaded")
        elif choice == "2":
        # Print the sorted list of courses
                print_sorted_course_list(course_data) # Replace with your actual printing
        logic
        elif choice == "3":
                # Get the course code from the user
```

```
            course_code = input("Enter the course code: ")
            # Print the details of the course
            print_course_details(course_data, course_code)
        # Replace with your actual printing logic
        elif choice == "9":
        # Exit the program
        print("Exiting the program now")
        break
        else:
        # Invalid choice
        print("Invalid, try again")
```

3. ***Design pseudocode that will print out the list of the sources in the Computer Science program in alphanumeric order.***

```
// Function to print the sorted list of courses for each data structure

// Vector
function printSortedCourseListVector(courseData):
// Sort the vector in ascending order by course number
sort(courseData, compareCourseNumbers) // Assuming compareCourseNumbers is a function that compares course numbers

// Print the sorted list
for each course in courseData:
print(course.courseNumber + " " + course.courseName)


// Hash Table
function printSortedCourseListHashTable(courseData):
// Get the sorted keys (course numbers) from the hash table
sortedKeys = getSortedKeys(courseData) // Assuming getSortedKeys is a function that returns a sorted list of keys

// Print the courses based on the sorted keys for each key in sortedKeys:
course = courseData[key] // Assuming courseData is a hash table
print(key + " " + course.courseName)
```

```
// Tree (Assuming a binary search tree)
function printSortedCourseListTree(courseData):
// Perform an in-order traversal of the tree, which naturally prints the elements in sorted
order inOrderTraversal(courseData.root)

//Function for in-order traversal
function inOrderTraversal(node):
if node is not null:
inOrderTraversal(node.left)
print(node.courseNumber + “” + node.courseName)
inOrderTraversal(node.right)

EXIT
```

4. ***Evaluate the run time and memory of data structures that could be used to address the requirements.***

There are three different data structures that are most used and were used throughout this course; vectors, hash tables and binary search tree. Vectors and hash tables are technically seen as the most efficient, when it comes to worse-case time complexities of O(n) (O(n) is a way to describe how the time and memory requirements that revolves around an algorithm or a data structure can grow, n is the input size and the O(n) is the linear growth) for reading and the parsing systems, and creating the objects. Which signifies the performance level of the systems in a linear fashion with the number of courses that are within the database. But binary search trees on the other hand, they are efficient when it comes to specific searching operations, they have a slower reaction for object creation, since the courses would be input in a sorted order, which will make the tree more unbalanced. For memory usage on the other hand, all the three data structures are O(n), which indicates that the memory that is required grows proportionally to the number of courses that are present. Hence, the decision between the data structures would then depend on the different requirements for each project. For example, vectors and hash tables would be recommended when it comes to input efficiency, while binary search trees would be most helpful in efficient searching.

5. ***Vector, hash table and binary search tree advantages and disadvantages.***

Vector: The disadvantages of vectors include it’s not very efficient when it comes to searching, searching for a specific bid in a vector data structure is not as encouraged compared to another structure. However, the advantages: the process of inputting data is

simple and efficient; hence, it is easy to add bids that must be read. And vectors maintain a specific order of elements so that any element can be accessed in a sequence, which means that if iterating is needed, it will be useful since the bids are in a certain order.

Hash Table: With a hash table, unlike a vector, it does not maintain a specific order for the elements, which means that if iteration is needed in a specific order, an additional sorting system will have to be implemented. However, they can change the table's size to accommodate to any size or mass of data, which is most useful for any growing datasets.

Binary Search Tree: This type of data structure has a change of becoming unbalanced because if bids are input into a specific manner, there can be the chance of the structure becoming unbalanced and no longer efficient, however, binary search trees are very efficient when it comes to searching for a specific bid where the tree is balanced, which means that this structure is faster than any linear search.
In conclusion, I think it depends on the task's requirements for a specific data structure to be used. All three data structures have very different advantages and disadvantages, making them efficient in certain scenarios and lesser in others. Hence, understanding and considering the requirements of a specific task, then analyzing the abilities of each data structure in accordance with those requirements will be a more efficient form of recommendation.