**1.Resubmit pseudocode from previous pseudocode assignments and update as necessary. In the previous assignments, you created pseudocode for each of the three data structures: vector, hash table, and tree. Be sure to resubmit the following pseudocode for each data structure:**

**Design pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for formatting errors.**

**Design pseudocode to show how to create course objects so that one course object holds data from a single line from the input file.**

**Design pseudocode that will print out course information and prerequisites.**

Vector sorting pseudocode:

**Program Name: VectorSorting**

1. Start Program

- welcome message.

2. Define Data Structures

- Define a `Bid` structure with `bidId`, `title`, `fund`, and `amount`.

3. Define Helper Functions

- `displayBid(bid)`: Outputs the details of a bid.

- `getBid()`: Prompts the user for bid details and returns a `Bid`.

4. Load Bids from File

- Open CSV file.

- Parse lines into `Bid` objects.

- Store bids in a list.

5. Sorting Algorithms

- \*\*Selection Sort\*\*: Repeatedly find the smallest element and move it to the front.

- \*\*Quick Sort\*\*: Recursively divide and sort based on a pivot.

6. Main Menu

- Options:

1. Load bids from file.

2. Display all bids.

3. Sort using selection sort.

4. Sort using quick sort.

9. Exit program.

- Perform user’s choice.

7. Exit Program

- Display goodbye message.

- End Program.

**Class HashTable:**

Node Structure:

- bid: Bid

- next: Node pointer

- key: Integer

Variables:

- table: Array of Node pointers

- size: Integer (default = 179)

Constructor:

- Initialize table with size `size` and set each entry to nullptr

Method hash(key):

- Return key % size (hashing the key)

Method Insert(bid):

- Generate key using `hash(bid.bidId)`

- If table[key] is nullptr:

- Insert bid at table[key]

- Else:

- Traverse to find the last node

- Add new node with bid at the end of the list

Method PrintAll():

- For each node in table:

- If node is not nullptr:

- Print bid details

- Traverse the linked list to print all bids in chain

Method Search(bidId):

- Generate key using `hash(bidId)`

- Traverse the list at table[key] to find the bid with matching bidId

- Return the bid if found, else return empty Bid

Method Remove(bidId):

- Generate key using `hash(bidId)`

- If table[key] is the first node and matches bidId:

- Remove the node, update the table entry to nullptr

- Else:

- Traverse the list to find the matching node

- Remove the node and update pointers

Main:

Initialize HashTable

While not exit:

Display menu with options:

1. Insert Bid

2. Print All Bids

3. Search Bid by ID

4. Remove Bid by ID

9. Exit

Get user choice and execute corresponding method

Display "Goodbye!"

**Class BinarySearchTree:**

Node Structure:

- bid: Bid

- left: Node pointer

- right: Node pointer

Variables:

- root: Node pointer

Constructor:

- Set root to nullptr

Method Insert(bid):

- If root is nullptr:

- Set root to a new node with the bid

- Else:

- Call InsertRecursively with root and bid

Method InsertRecursively(node, bid):

- If bid's ID is less than node's ID:

- If node's left is nullptr:

- Set node's left to a new node with the bid

- Else:

- Call InsertRecursively with node's left and bid

- If bid's ID is greater than node's ID:

- If node's right is nullptr:

- Set node's right to a new node with the bid

- Else:

- Call InsertRecursively with node's right and bid

Method Search(bidId):

- Call SearchRecursively with root and bidId

- Return the bid if found, else return empty Bid

Method SearchRecursively(node, bidId):

- If node is nullptr:

- Return empty Bid

- If bidId matches node's ID:

- Return node's bid

- If bidId is less than node's ID:

- Call SearchRecursively with node's left and bidId

- Else:

- Call SearchRecursively with node's right and bidId

Method Remove(bidId):

- Call RemoveRecursively with root and bidId

Method RemoveRecursively(node, bidId):

- If node is nullptr:

- Return nullptr

- If bidId matches node's ID:

- If node has no children:

- Delete node and return nullptr

- If node has one child:

- Replace node with the child

- If node has two children:

- Find the inorder successor and replace node with it

- Call RemoveRecursively to delete the successor

- If bidId is less than node's ID:

- Call RemoveRecursively with node's left and bidId

- Else:

- Call RemoveRecursively with node's right and bidId

Method InOrder():

- Call InOrderRecursively with root

Method InOrderRecursively(node):

- If node is not nullptr:

- Call InOrderRecursively with node's left

- Print node's bid

- Call InOrderRecursively with node's right

Destructor:

- Call DeleteAll() to clean up all nodes

Method DeleteAll():

- If root is not nullptr:

- Call DeleteAllRecursively with root

Method DeleteAllRecursively(node):

- If node is not nullptr:

- Call DeleteAllRecursively with node's left

- Call DeleteAllRecursively with node's right

- Delete the node

Main:

Initialize BinarySearchTree

While not exit:

Display menu with options:

Insert Bid

Print All Bids

Search Bid by ID

Remove Bid by ID

Exit

Get user choice and execute corresponding method

Display "Goodbye!"

Class Menu:

**Method DisplayMenu():**

- DISPLAY menu options:

1. Load data into data structure

2. Print alphanumerically ordered list of courses

3. Print course title and prerequisites

9. Exit program

Method LoadData():

- Open file

- Read each line

- Parse data

- Create course objects

- Store course objects in data structure

Method PrintCourses():

- Sort courses alphanumerically

- For each course in sorted list:

- PRINT course details

Method PrintCourseInfo():

- Get course number from user

- Search and print course title and prerequisites

Main:

Initialize Menu

While not exit:

Call DisplayMenu()

Get user choice and execute corresponding method:

1. Load data into data structure (Call LoadData())

2. Print alphanumerically ordered list of courses (Call PrintCourses())

3. Print course title and prerequisites (Call PrintCourseInfo())

9. Exit program

Display "Goodbye!"

**3. Design pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order. Continue working with the Pseudocode Document linked in the Supporting Materials section. Note that you will design for the same three data structures that you have been using in your previous pseudocode milestones: vector, hash table, and tree. This time, you will create the final pieces of pseudocode that you will need for ABCU’s advising program. To complete this part of the process, do the following actions:**

* 1. **Sort the course information by alphanumeric course number from lowest to highest.**
  2. **Print the sorted list to a display.**

Node Structure:

- courseNumber: String

- name: String

- prerequisites: List of Strings

- next: Node pointer

Variables:

- courses: Vector of Course objects

Constructor:

- Initialize courses as an empty vector

Method AddCourse(courseNumber, name, prerequisites):

- Create a new Course object with courseNumber, name, and prerequisites

- Add the new Course object to courses

Method compareCourses(course1, course2):

- Return true if course1.courseNumber < course2.courseNumber, else false

Method SortCourses():

- Sort courses using compareCourses function

Method PrintSortedCourses():

- For each course in courses:

- Print course.courseNumber + ", " + course.name

Main:

Initialize courses

AddCourse("CSCI100", "Introduction to Computer Science", [])

AddCourse("CSCI101", "Introduction to Programming in C++", ["CSCI100"])

AddCourse("CSCI200", "Data Structures", ["CSCI101"])

AddCourse("MATH201", "Discrete Mathematics", [])

AddCourse("CSCI300", "Introduction to Algorithms", ["CSCI200", "MATH201"])

AddCourse("CSCI301", "Advanced Programming in C++", ["CSCI101"])

AddCourse("CSCI350", "Operating Systems", ["CSCI300"])

AddCourse("CSCI400", "Large Software Development", ["CSCI301", "CSCI350"])

SortCourses()

PrintSortedCourses()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **Evaluate the run time and memory of data structures that could be used to address the requirements. In previous assignments, you created pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for formatting errors and to show how to create course objects so that one course object holds data from a single line from the input file.**
   1. **Using the pseudocode you wrote for the previous assignments, analyze the worst-case running time of each, reading the file and creating course objects, which will be the Big O value. This analysis should not include the pseudocode written for the menu or the search/print functions Print Course List (Option 2) above. To complete this part of the project, do the following actions:**
   2. **Specify the cost per line of code and the number of times the line will execute. Assume there are n courses stored in the data structure. Assume the cost for a line to execute is 1 unless it is calling a function, in which case the cost will be the running time of that function.**

**Data structure time complexity analysis(vector)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| code | Line cost | #times executes | Total cost | |
| All courses | 1 | n | n | |
| If course == courseNumber | 1 | n | n | |
| Cout the course info | 1 | 1 | 1 | |
| For each course prerequisite | 1 | n | n | |
| Cout course prerequisite info | 1 | n | n | |
| Total cost | | | | 4n+1 |
| Runtime | | | | 0(n) |

**Data structure time complexity analysis (binary search tree)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Code | Line cost | #times execute | | Total cost |
| Method for adding node | 0 | 0 | | 0 |
| If(root == nullptr) + node | 1 | 1 | | 1 |
| If(node < root) + to left node | 1 | n | | n |
| If(left node doesn’t exist) | 1 | n | | n |
| This node == left node | 1 | n | | n |
| ELSE IF(node > root) + to right | 1 | n | | n |
| If(right node does not exist) | 1 | n | | n |
| This node == right node | 1 | n | | n |
| For(every line in file) | 1 | n | | n |
| Create vector<course\_item> | 1 | n | | n |
| While prerequisites exists) | 1 | n | | n |
| Append the prerequisite | 1 | n | | n |
| Inset function for course item | 1 | n | | n |
| Total cost | | | 11n + 2 | |
| runtime | | | 0(n) | |

**Data structure time complexity analysis (HashTable)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Code | Line cost | #times execute | | Total cost |
| Creating hashtable | 1 | 1 | | 1 |
| Insert method | 0 | 0 | | 0 |
| Creating key for courses | 1 | n | | n |
| If (not found) | 1 | n | | n |
| Assign node (key) | 1 | n | | n |
| Else |  | n | | n |
| Assign key = UNIT\_MAX  Set old node to course, old node next ->nullptr | 3 | n | | 3n |
| Else | 1 | n | | n |
| Locate next new node | 1 | n | | n |
| Adding new node to end | 1 | n | | n |
| For(each line in the file) | 1 | n | | n |
| Create the vector w/ <course item> | 1 | n | | n |
| If (prerequisite exists) | 1 | n | | n |
| append | 1 | n | | n |
| Insert course item | 1 | n | | n |
| Total cost | | | 15n + 1 | |
| runtime | | | 0(n) | |

1. **Based on the advisor’s requirements, analyze each of the vector, hash table, and tree data structures. Explain the advantages and disadvantages of each structure in your evaluation.**

Vectors are great for storing static data and allow fast access, but they become inefficient with frequent insertions or deletions. Hash tables excel in fast lookups and handling unique items, but they don't maintain order and can suffer from performance issues due to collisions. Binary Search Trees (BSTs) provide ordered data and support efficient range queries, but they can become unbalanced and slower if not carefully managed. For static data, vectors work well; for fast lookups with unique entries, hash tables are best; and for maintaining sorted data with range queries, BSTs are the ideal choice.

1. **Now that you have analyzed all three data structures, make a recommendation for which data structure you plan to use in your code. Provide justification for your recommendation based on the Big O analysis results and your analysis of the three data structures.**

Based on the Big O analysis and my evaluation, I recommend using a hash table in my code. Hash tables offer constant-time complexity (O(1)) for lookups, insertions, and deletions on average, which makes them highly efficient for scenarios where quick access to unique elements is crucial. Although they don't maintain order, the speed advantage outweighs this limitation for most use cases, especially when handling large datasets or frequent queries.

**Milestone 1 pseudocode:**

// Define the course structure

STRUCT Course

STRING courseNumber

STRING courseTitle

LIST of STRING prerequisites // List to store prerequisites for each course

END STRUCT

// Create a list to store all the courses

VECTOR<Course> courseList

// Function to read the course data from a file

FUNCTION readDataFromFile(FILE file)

OPEN file to read

// Read each line from the file

WHILE there are more lines in the file

LINE = read next line from file

SPLIT LINE into smaller parts (separate by commas or spaces)

// Check if the line is correct: It must have at least two parts (course number and title)

IF there are less than two parts THEN

PRINT "Error: This line is missing information: " + LINE

SKIP to the next line

// Get the course number and title from the parts

courseNumber = parts[0]

courseTitle = parts[1]

// Create a new course

NEW course = NEW Course

course.courseNumber = courseNumber

course.courseTitle = courseTitle

// If there are prerequisites (other courses to take before this one)

IF there are more parts after the title THEN

FOR EACH prerequisite IN remaining parts

// Make sure the prerequisite course exists in the list

IF prerequisite does not exist in courseList THEN

PRINT "Error: Prerequisite course " + prerequisite + " is missing for " + courseNumber

SKIP to next line

END IF

// Add the prerequisite to this course

course.prerequisites.append(prerequisite)

END FOR

END IF

// Add the course to the list

courseList.append(course)

END WHILE

CLOSE file

END FUNCTION

// Function to search for a course by its number

FUNCTION findCourse(courseNumber)

FOR EACH course IN courseList

IF course.courseNumber == courseNumber THEN

PRINT "Course Number: " + course.courseNumber

PRINT "Course Title: " + course.courseTitle

// Show prerequisites if there are any

IF course.prerequisites is not empty THEN

PRINT "Prerequisites: " + JOIN course.prerequisites with ", "

ELSE

PRINT "No prerequisites"

END IF

RETURN

END IF

END FOR

PRINT "Course not found: " + courseNumber

END FUNCTION

// Main program to run everything

FILE file = OPEN "CourseData.txt"

// Read the course data from the file

readDataFromFile(file)

// Example: Search for a specific course

findCourse("CS101")

END

**Milestone 2 pseudocode**

Class Course:

Variables:

- courseNumber: String

- courseTitle: String

- prerequisites: List of Strings

Constructor:

- Initialize courseNumber, courseTitle, and prerequisites

Class HashTable:

Variables:

- table: Dictionary (courseNumber → Course object)

Method Insert(course):

- Insert course into table using courseNumber as key

Method Search(courseNumber):

- If courseNumber exists in table:

- Return corresponding Course object

- Else:

- Return None

Method PrintAll():

- For each Course in table:

- Print course details

Function ReadFile(filename):

- Open file filename

- If file cannot be opened:

- Print error message and exit

- Initialize empty HashTable

- While not end of file:

- Read line from file

- Split line by commas into tokens

- If tokens has less than 2 elements:

- Print error message and continue

- Set courseNumber = tokens[0]

- Set courseTitle = tokens[1]

- Initialize empty prerequisites list

- For each token from index 2 onwards:

- Add token to prerequisites list

- Create Course object with courseNumber, courseTitle, prerequisites

- Insert Course into HashTable

- Close file

- For each Course in HashTable:

- For each prerequisite in Course.prerequisites:

- If prerequisite is not in HashTable:

- Print error message

- Return HashTable

Function PrintCourseInfo(courseNumber, hashTable):

- If courseNumber not in hashTable:

- Print "Course not found." and return

- Retrieve Course from hashTable

- Print courseNumber, courseTitle, and prerequisites

Main:

- Call ReadFile("CourseData.txt") to populate hashTable

- Call PrintAll() to display all courses

**Milestone 3 Pseudocode**

BEGIN

// Define the course structure

STRUCT Course

STRING courseNumber

STRING courseTitle

LIST of STRING prerequisites

END STRUCT

// Create a list to store courses

VECTOR<Course> courseList

// Function to read course data from a file

FUNCTION readDataFromFile(FILE file)

OPEN file to read

// Read each line from the file

WHILE there are more lines

LINE = read next line

SPLIT LINE into parts

// Check if the line has course number and title

IF there are less than two parts THEN

PRINT "Error: Missing information"

SKIP to next line

// Get course number and title

courseNumber = parts[0]

courseTitle = parts[1]

// Create a new course

NEW course = NEW Course

course.courseNumber = courseNumber

course.courseTitle = courseTitle

// If there are prerequisites

IF there are more parts THEN

FOR EACH prerequisite IN remaining parts

IF prerequisite not in courseList THEN

PRINT "Error: Missing prerequisite"

SKIP to next line

END IF

course.prerequisites.append(prerequisite)

END FOR

END IF

// Add course to list

courseList.append(course)

END WHILE

CLOSE file

END FUNCTION

// Function to find a course by number

FUNCTION findCourse(courseNumber)

FOR EACH course IN courseList

IF course.courseNumber == courseNumber THEN

PRINT "Course Number: " + course.courseNumber

PRINT "Course Title: " + course.courseTitle

IF course.prerequisites is not empty THEN

PRINT "Prerequisites: " + JOIN course.prerequisites with ", "

ELSE

PRINT "No prerequisites"

END IF

RETURN

END IF

END FOR

PRINT "Course not found"

END FUNCTION

// Main program

FILE file = OPEN "CourseData.txt"

// Read data from file

readDataFromFile(file)

// Example: Search for a course

findCourse("CS101")

END