# CS 300 Pseudocode Document: Dylan Stirling

Pseudocode Vector, Hash, Tree:

START

// Vector Pseudocode

// Define a vector to store Course objects

Vector<Course> courseVector;

// Function to read and parse data from the file

Function ReadFileAndCreateObjects(filename):

Open file with filename

if file is open:

for each line in file:

Parse the line to extract course information

Create a Course object with the parsed information

Append the Course object to courseVector

Close the file

// Function to print course information and prerequisites

Function PrintCourseInfo(course):

Print course.title

if course has prerequisites:

Print "Prerequisites: "

for each prerequisite in course.prerequisites:

Print prerequisite

// Function to print alphanumerically ordered list of courses

Function PrintAlphanumericList():

Sort courseVector by course.number in ascending order

for each course in courseVector:

Print course.number

// Hash Table Pseudocode

// Define a hash table to store Course objects

HashTable<Course> courseHashTable;

// Function to read and parse data from the file

Function ReadFileAndCreateObjects(filename):

Open file with filename

if file is open:

for each line in file:

Parse the line to extract course information

Create a Course object with the parsed information

Insert the Course object into courseHashTable

Close the file

// Function to print course information and prerequisites

Function PrintCourseInfo(courseNumber):

Retrieve course from courseHashTable using courseNumber

Print course.title

if course has prerequisites:

Print "Prerequisites: "

for each prerequisite in course.prerequisites:

Print prerequisite

// Function to print alphanumerically ordered list of courses

Function PrintAlphanumericList():

Get all keys from courseHashTable and store in a list

Sort the list in ascending order

for each key in the sorted list:

Retrieve course from courseHashTable using key

Print course.number

// Tree Pseudocode

// Define a binary search tree to store Course objects

BinarySearchTree<Course> courseTree;

// Function to read and parse data from the file

Function ReadFileAndCreateObjects(filename):

Open file with filename

if file is open:

for each line in file:

Parse the line to extract course information

Create a Course object with the parsed information

Insert the Course object into courseTree

Close the file

// Function to print course information and prerequisites

Function PrintCourseInfo(courseNumber):

Retrieve course from courseTree using courseNumber

Print course.title

if course has prerequisites:

Print "Prerequisites: "

for each prerequisite in course.prerequisites:

Print prerequisite

// Function to print alphanumerically ordered list of courses

Function PrintAlphanumericList():

InOrderTraversal(courseTree)

END

Menu Pseudocode:

START

// Menu Pseudocode

// Function to display the main menu

Function DisplayMainMenu():

Display "Menu:"

Display "1. Load Data Structure"

Display "2. Print Course List"

Display "3. Print Course"

Display "4. Exit"

// Function to handle user input for the menu

Function HandleMenuSelection():

userInput = GetUserInput()

// Check user's choice

Switch userInput:

Case 1:

Call LoadDataStructure()

Case 2:

Call PrintAlphanumericList()

Case 3:

Call PrintCourse()

Case 4:

Display "Exiting the program. Goodbye!"

ExitProgram()

Default:

Display "Invalid choice. Please enter a valid option."

// Function to load data into the chosen data structure

Function LoadDataStructure():

Display "Enter the filename to load data from: "

filename = GetUserInput()

// Check if the chosen data structure is a Vector

if dataStructureType == "Vector":

Call ReadFileAndCreateObjectsForVector(filename)

// Check if the chosen data structure is a Hash Table

else if dataStructureType == "HashTable":

Call ReadFileAndCreateObjectsForHashTable(filename)

// Check if the chosen data structure is a Tree

else if dataStructureType == "Tree":

Call ReadFileAndCreateObjectsForTree(filename)

Display "Data loaded successfully."

// Function to print an alphanumerically ordered list of all courses

Function PrintAlphanumericList():

// Check if the chosen data structure is a Vector

if dataStructureType == "Vector":

Call PrintAlphanumericListForVector()

// Check if the chosen data structure is a Hash Table

else if dataStructureType == "HashTable":

Call PrintAlphanumericListForHashTable()

// Check if the chosen data structure is a Tree

else if dataStructureType == "Tree":

Call PrintAlphanumericListForTree()

// Function to print the course title and prerequisites for a specific course

Function PrintCourse():

Display "Enter the course number: "

courseNumber = GetUserInput()

// Check if the chosen data structure is a Vector

if dataStructureType == "Vector":

Call PrintCourseInfoForVector(courseNumber)

// Check if the chosen data structure is a Hash Table

else if dataStructureType == "HashTable":

Call PrintCourseInfoForHashTable(courseNumber)

// Check if the chosen data structure is a Tree

else if dataStructureType == "Tree":

Call PrintCourseInfoForTree(courseNumber)

// Function to get user input

Function GetUserInput():

Display "Enter your choice: "

userInput = ReadInputFromUser()

Return userInput

// Function to exit the program

Function ExitProgram():

Exit

END

Print List pseudocode:

START

// Pseudocode to print courses in alphanumeric order

// Vector Pseudocode

Function PrintAlphanumericListForVector():

// Sort the courseVector by course.number in ascending order

Sort courseVector by course.number in ascending order

// Print the sorted list

for each course in courseVector:

Print course.number

// Hash Table Pseudocode

Function PrintAlphanumericListForHashTable():

// Get all keys from courseHashTable and store in a list

keysList = GetAllKeysFromHashTable(courseHashTable)

// Sort the list in ascending order

Sort keysList in ascending order

// Print the sorted list

for each key in keysList:

// Retrieve course from courseHashTable using key

course = RetrieveCourseFromHashTable(courseHashTable, key)

Print course.number

// Tree Pseudocode

Function PrintAlphanumericListForTree():

// Perform In-Order Traversal to print courses in ascending order

InOrderTraversal(courseTree)

// Function to perform In-Order Traversal on a binary search tree

Function InOrderTraversal(treeNode):

if treeNode is not null:

InOrderTraversal(treeNode.left)

Print treeNode.data.number

InOrderTraversal(treeNode.right)

END

Runtime Analysis Chart:

Vector:

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| Create vector | 1 | 1 | 1 |
| for each line in file | 1 | n | n |
| Create vector course item | 1 | n | n |
| While prereq exists | 1 | m (average number of prereqs) | m \* n |
| Append prereq | 1 | m (average number of prereqs) | m \* n |
| Pushback course item | 1 | n | n |
| **Total Cost** | | | 2n + 2m |
| **Runtime** | | | O(n + m) |

Hash:

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| Create hash table | 1 | 1 | 1 |
| Insert method | 1 | n | n |
| Create key for course | 1 | n | n |
| If no entry found for key | 1 | n | n |
| Assign node to key | 1 | n | n |
| If node is less than root | 1 | n | n |
| If no left node | 1 | n | n |
| This node becomes left | 1 | n | n |
| If node is greater than root | 1 | n | n |
| If no right node | 1 | n | n |
| This node becomes right | 1 | n | n |
| Find the next open node | 1 | n | n |
| Add new newNode to end | 1 | n | n |
| For each line in file | 1 | n | n |
| Create vector course item | 1 | n | n |
| While prereq exists | 1 | m (average number of prereqs) | m \* n |
| Append prereq | 1 | m (average number of prereqs) | m \* n |
| Insert course item | 1 | n | n |
| **Total Cost** | | | 9n + 2m |
| **Runtime** | | | O(n + m) |

Tree:

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Create tree | 1 | 1 | 1 |
| Add node method | 1 | n | n |
| If root is null, add root | 1 | n | n |
| If node is less than root | 1 | n | n |
| If no left node | 1 | n | n |
| This node becomes left | 1 | n | n |
| If node is greater than root | 1 | n | n |
| If no right node | 1 | n | n |
| This node becomes right | 1 | n | n |
| For each line in file | 1 | n | n |
| Create vector course item | 1 | n | n |
| While prereq exists | 1 | m (average number of prereqs) | m \* n |
| Append prereq | 1 | m (average number of prereqs) | m \* n |
| Insert course item | 1 | n | n |
| **Total Cost** | | | 7n + 2m |
| **Runtime** | | | O(n + m) |

When evaluating the three data structures for the advising program, namely vector, hash table, and tree, each structure comes with its own set of advantages and disadvantages. The vector is easy to implement but is not suitable for dynamic data as it struggles with efficient insertion, deletion, and sorting operations. The hash table, while enabling quick retrievals, may suffer from collisions and is less effective when a sorted order is crucial. On the other hand, the tree, especially a balanced binary search tree like an AVL tree, offers efficient searching, insertion, and deletion operations. It maintains a sorted order, which aligns well with the program's requirement to print courses in alphanumeric order.

Considering the specific needs of the advising program, it is recommended to employ the Tree Data Structure (Binary Search Tree). Although slightly more complex to implement than vectors or hash tables, the tree structure's advantages of efficient operations and inherent sorting make it a well-suited choice. The ability to maintain courses in alphanumeric order is especially valuable, and the use of a balanced tree helps mitigate potential performance issues, ensuring consistent and reliable behavior in various scenarios. Overall, the tree data structure stands out as a robust solution that balances efficiency, ordered retrieval, and adaptability to the dynamic nature of course data.