CS 300 Pseudocode Document

### Vector - Milestone 1

Data Structure Definition

First, define a structure to hold the information for one single course.

// Define a structure to represent a course

STRUCT Course

STRING courseNumber

STRING name

VECTOR<STRING> prerequisites

END STRUCT

Main Program Logic

This is the main entry point and control flow of the application. It shows a menu to the user and calls functions based on their specific menu selection.

// Main program execution block

int main()

// A vector to hold all the Course objects

VECTOR<Course> courseCatalog{};

// Variable to hold user's menu choice

STRING menuSelection{};

PRINT "ABCU Course Planner Menu"

// Loop while the user choice is not to exit

WHILE menuSelection != 9

PRINT "Menu:"

PRINT " 1. Load Course Data"

PRINT " 2. Print Course List"

PRINT " 3. Find a Course"

PRINT " 9. Exit"

PROMPT for menuSelection

READ menuSelection

IF menuSelection == 1

// Function to load data from the file

courseCatalog = loadCourses()

ELSE IF menuSelection == 2

// Function to print all the data

printCourseCatalog(courseCatalog)

ELSE IF menuSelection == 3

// Function to call + print a specific course

searchForCourse(courseCatalog)

ELSE IF menuSelection == 9

PRINT "Goodbye."

ELSE

// Simple input validation.

PRINT "Not a menu option. Try again."

END WHILE

END main

Return 0;

Load and Validate Data

This section has the pseudocode for loading data from a file into the vector data structure and doing the necessary validation checks.

// Function loading courses from file->vector of Course objects

FUNCTION loadCourses()

// New temp vector to hold courses during loading

VECTOR<Course> courses

// Assumes configuration of file path has been set up

OPEN file at specified path in config

// Check if file opened successfully

IF file !open

PRINT "Error: File could not be opened."

RETURN an empty Course(); vector

END IF

// Read file line by line

WHILE nextline is not null

READ the next line

// Split line by commas into list containing tokens

VECTOR<STRING> tokens = parseLine(line, ',')

// First file format ValidationCheck == for minimum parameters

IF size of tokens < 2

PRINT "Skipping line:" + line + “invalid token size”

CONTINUE on to the following line

END IF

// Create new Course object

Course newCourse

newCourse.courseNumber = tokens[0]

newCourse.name = tokens[1]

// If there exists prerequisites, add to course prerequisite vector

IF tokens.size() > 2

FOR i{} + 2 < tokens.size() – 1

APPEND tokens[i] to newCourse.prerequisites

END FOR

END IF

// Add fully made Course object to the vector

APPEND newCourse to courses

END WHILE

CLOSE file

// Second file format validation == Check that all prerequisites exist

// Check must be done after all the courses are loaded

FOR EACH course : courses

FOR EACH prereqId : course.prerequisites

// Search for prerequisite in entire main course’s list

BOOL courseFound = false

FOR EACH altCourse : courses

IF altCourse.courseNumber == prereqId

courseFound = true

BREAK from inner loop

END IF

END FOR

// If prereq is not found as course, print error

IF courseFound == false

PRINT "Error: Prerequisite '" + prereqId + "' for course '" + course.courseNumber + "' not found."

END IF

END FOR

END FOR

PRINT Positive, + successful load and validation statement.

RETURN courses

END FUNCTION

Search and Print Course Information

This pseudocode shows the search sequence for finding a specific course and then printing its details plus all of its prerequisites.

// Prompt the user for a course ID and print its information

VOID searchForCourse(VECTOR<Course> courses)

PROMPT user for a courseNumber to search for

// Variable to track if the course was found

BOOLEAN courseFound = false

// Loop through courses vector to find the matching course

FOR EACH course : courses

// If the current course matches the user's input

IF course.courseNumber == courseNumber

// Print the course information

PRINT course.courseNumber + ", " + course.name

// Print all the prerequisites for that course

PRINT "Prerequisites: "

IF course.prerequisites == empty

PRINT " None"

ELSE

// Loop through and print each prerequisite ID

FOR EACH prereqId : course.prerequisites

PRINT " " + prereqId

END FOR

END IF

courseFound = true

BREAK // Exit loop once courseFound

END IF

END FOR

// If the loop finishes && not courseFound

IF courseFound == false

PRINT "Course '" + courseNumber + "' not found."

END IF

END Method

// Helper function to print all loaded courses alphabetically

VOID printCourseCatalog(VECTOR<Course> courses)

// First sort the vector alphabetically by courseNumber

SORT courses by courseNumber

PRINT "List of available courses:"

FOR EACH course : courses

PRINT course.courseNumber + ", " + course.name

END FOR

END Method

### Hash Table - Milestone 2

Load Courses from File to Hash Table

This pseudocode shows the method for loading courses from a file, the same CSV as before, but into a hash table rather than a vector. It is creating a hash table of Course objects by creating an array of lists, where the lists are pairs.

FUNCTION loadCourses()

// This function needs to create a temp HashTable to hold all the Course objects

HASHTABLE<STRING, Course> courses

// Assumes configuration of file path has been set up

OPEN file at specified string path in config

// Check if file opened successfully

IF file IS NOT open

PRINT "Error: File could not be opened."

RETURN an empty HashTable

END IF

// Read file line by line

WHILE there IS a next line in file

READ the next line

// Split line by commas into list of tokens

VECTOR<STRING> tokens = parseLine(line, ',')

// First file format ValidationCheck == a check for minimum parameters

IF size of tokens < 2

PRINT "Skipping line: " + line + “ due to invalid format”

CONTINUE loop back to next line

END IF

// Create new Course object

Course newCourse

newCourse.courseNumber = tokens[0]

newCourse.name = tokens[1]

// If there exists prerequisites, add them to the course's prerequisite vector

IF size of tokens > 2

FOR i + 2 < tokens.size() - 1

APPEND tokens[i] to newCourse.prerequisites

END FOR

END IF

// Insert the fully made Course object into the hash table using the course number as the key

courses.Insert(newCourse.courseNumber, newCourse)

END WHILE

CLOSE file

// Second file format validation == Check that all the prerequisites exist

// This check will be much quicker due to the change in time complexity than it was previously

FOR EACH course : courses.GetAllItems()

FOR EACH prereqId : course.prerequisites

// Will make use of the method search for prerequisite using hash table’s built in Search method

IF courses.Search(prereqId) == null

PRINT "Error: Prerequisite '" + prereqId + "' for course '" + course.courseNumber + "' not found."

END IF

END FOR

END FOR

PRINT Successful message telling user course data has been loaded +validated

RETURN courses

Helper Function to Load Alphabetically

This simple helper function is relatively self-explanatory. The code will just use a vector sorting method, implementation is ADT for now, to sort the hash table elements alphabetically.

// Helper function to print all loaded courses alphabetically

VOID printCourseCatalog(HASHTABLE<STRING, Course> courses)

/\*\* The goal of this function is solely to display alphabetically. \* This requires getting all of the items and sorting them in a temp \* vector since hash tables aren’t ordered by default and can’t

\*/ be sorted using <include> algorithm, for instance.

VECTOR<Course> sortedCourses = courses.GetAllItems()

// temp vector now sorted alphabetically using it’s courseNumber

SORT sortedCourses by courseNumber

PRINT "List of available courses:"

FOR EACH course : sortedCourses

PRINT course.courseNumber + ", " + course.name

END FOR

END Method

Search Function for Course in Hash Table

The goal of this function is to replace the linear search (complexity O(n)) with a direct hash table search (complexity O(1)).

void searchCourse(HashTable<Course> courses, String courseNumber) {

// This searches the hash table using a given key

Course course = courses.Search(courseNumber)

// Checks if the search returned a valid course

IF course \exists in table

// Display that course’s info

PRINT course.courseNumber + ", " + course.name

// Display all prerequisites of the selected course

PRINT "Prerequisites: "

IF course.prerequisites.isEmpty()

PRINT "No prerequisites for this course."

ELSE

// Loop through + print each individual prerequisite ID

FOR EACH prereqId : course.prerequisites

PRINT " " + prereqId

END FOR

END IF

ELSE

// If code reaches here, then course was not found.

PRINT "Course '" + courseNumber + "' was not found in the hash table."

END IF

}

**Binary Search Tree – Milestone 3**

Load Courses from File to Binary Search Tree

This function will read all the course data from the CSV file and insert it into a BST. The BST will automatically order the Course objects based on their courseNumber.

// This function loads courses from file -> BST of Course objects

FUNCTION loadCourses()

// Create a Binary Search Tree to hold all of the Course objects

BINARY\_SEARCH\_TREE<Course> courses

// Assumes configuration of file path has been set up

OPEN file at specified path in config

// Check if file opened successfully

IF file IS NOT open

PRINT Error: "File couldn’t open the file. Check directory."

RETURN BINARY\_SEARCH\_TREE{};

END IF

// Read file line by line

WHILE getLine /exists in CSV

READ next line

// Split line by commas into list of tokens

VECTOR<STRING> tokens = parseLine(line, ',')

// Validation 1 == Check for parameters < 2

IF size of tokens < 2

PRINT "Skipping line: " + line + “ due to invalid format”

CONTINUE to next getLine

END IF

// Create new Course object

Course newCourse

newCourse.courseNumber = tokens[0]

newCourse.name = tokens[1]

// If there /exists prerequisites add to course prereq vector

IF size of tokens > 2

FOR i + 2 < size of tokens - 1

APPEND tokens[i] to newCourse.prerequisites

END FOR

END IF

// Insert the fully made Course object into the BST

courses.Insert(newCourse)

END WHILE

CLOSE file

// Validation check 2 == ensure that all prerequisites exist

FOR course : courses.GetAllItems() // Assumes method /exists for node traversal

FOR EACH prereqId : course.prerequisites

// Search for the prerequisite using the BST's Search method

IF courses.Search(prereqId) == null

PRINT "Error: Prerequisite '" + prereqId + "' for course '" + course.courseNumber + "' not found."

END IF

END FOR

END FOR

PRINT "Course data has been loaded and validated successfully."

RETURN courses

END FUNCTION

Helper Function to Print Ordered Course Catalog

The goal of this function is to print the full list of courses. Using BST in order traversal the courses can be printed in in order by courseNumber without needing to sort outside of this.

// Helper function to print all loaded courses alphabetically

VOID printCourseCatalog(BINARY\_SEARCH\_TREE<Course> courses)

PRINT "list of available courses in the catalog:"

// Call the BST's in-order traversal method to print all nodes.

// This naturally prints them in sorted order.

courses.PrintInOrder()

END FUNCTION

// Assumed implementation within the BINARY\_SEARCH\_TREE class

PROCEDURE PrintInOrder()

// Call a recursive helper method starting from the root

inOrder(root)

END FUNCTION

PROCEDURE inOrder(Node node)

IF node IS NOT null

inOrder(node.left)

PRINT node.course.courseNumber + ", " + node.course.name

inOrder(node.right)

END IF

END FUNCION

Search Function for Course in BST

The goal of this function is to search for one course within the BST. The search complexity is O(log n) on average, and it is very efficient at searching this data structure.

// Searches for one course in BST and print its data members

void searchCourse(BINARY\_SEARCH\_TREE<Course> courses, String courseNumber) {

// Search the tree for the given courseNumber –- assumes implementation

Course course = courses.Search(courseNumber)

// Check if search returns a non-null course

IF course !null

// Display the course's info

PRINT course.courseNumber + ", " + course.name

// Display all prerequisites of the course

PRINT "Prerequisites: "

IF course.prerequisites.isEmpty()

PRINT "No prerequisites exist for this course."

ELSE

// Loop through and print each individual prerequisite ID

FOR EACH prereqId : course.prerequisites

PRINT " " + prereqId

END FOR

END IF

ELSE

// If the block gets here, then course pointer is null, thus return not found

PRINT "Course '" + courseNumber + "' was not found."

END IF

}

Runtime Analysis

### Runtime and Memory Evaluation

The following analysis evaluates the worse-case running time for the loadCourses() function within each of the required data structures above. In other words, this is the Big O for the Vector, Hash Table, and Binary Search Tree. The analysis assumes that there are `n` courses within the input file.

### Vector Runtime Analysis

This vector implementation is somewhat straightforward, but it has significant performance downsides during specifically the validation step. This is because the validation step requires nested loops where each of the n courses has to be iterated over, and each of the iterations requires all of the other courses to be iterated to find any prerequisites. Then, for each of the n iterations, there must be n more iterations, thus .

|  |  |  |  |
| --- | --- | --- | --- |
| Code (from loadCourses() for Vector) | Line Cost | # Times Executed | Total Cost |
| WHILE nextline is not null | 1 | n | n |
| APPEND newCourse to courses | 1 | n | n |
| Validation Loop: |  |  |  |
| FOR EACH course : courses | 1 | n | n |
| FOR EACH prereqId : course.prerequisites | 1 | n \* preReq | n\*preReq |
| FOR EACH altCourse : courses | 1 | n \* preReq \* n | n²preReq |
| Total Cost (Dominant Term) |  |  | O(n²) |

Therefore, the final time complexity is , since the dominant operation of the nested loops controls the worst-case runtime.

### Hash Table Runtime Analysis

This hash table implementation offers a substantial performance improvement, specifically for the validation step. Instead of a nested loop linear search, there is simply a direct lookup capability built into the hash table. You can simply directly lookup the exact item, which is, on average, a constant time operation.

|  |  |  |  |
| --- | --- | --- | --- |
| Code (from loadCourses() for Hash Table) | Line Cost | # Times Executed | Total Cost |
| WHILE there IS a next line in file | 1 | n | n |
| courses.Insert(key, newCourse) | 1 | n | n |
| Validation Loop: |  |  |  |
| FOR EACH course : courses.GetAllItems() | 1 | n | n |
| FOR EACH prereqId : course.prerequisites | 1 | n \* p | n\*p |
| courses.Search(prereqId) | 1 | n \* p | n\*p |
| Total Cost (Dominant Term) |  |  | O(n) |

Within the hash table, the search and insertion operations are O(1) on average. This makes both the initial data loading and the validation loop after O(n). Therefore, the total and absolute worse case run time complexity is O(n).

### Binary Search Tree (BST) Runtime Analysis

The performance of a BST is heavily dependent on whether the tree stays balanced or not. In the worst-case scenario, like inserting already sorted data, the tree will become unbalanced and will behave more like a linked list. This would mean O(n) for insertions and searches, unfortunately.

|  |  |  |  |
| --- | --- | --- | --- |
| Code (from loadCourses() for BST) | Line Cost | # Times Executed | Total Cost |
| WHILE getLine /exists in CSV | 1 | n | n |
| courses.Insert(newCourse) | n | n | n² |
| Validation Loop: |  |  |  |
| FOR course : courses.GetAllItems() | 1 | n | n |
| FOR EACH prereqId : course.prerequisites | 1 | n \* p | n\*p |
| courses.Search(prereqId) | n | n \* p | n²p |
| Total Cost (Dominant Term) |  |  | O(n²) |

Since the worst-case for both he Insert and Search operations in the BST is O(n), the overall complexity becomes a quite poor This is somewhat misleading, as the average case is much better at O(n log n), but the evaluation requires always using the worst case as a point of consideration.

### Analysis of Data Structures

Each of the data structures has their advantages and disadvantages, of course, but for this specific project’s requirements we can take a nuanced perspective using some analysis.

### Vector

* Advantages:
  + Simple to implement and straightforward to understand. Standard in most programming languages.
  + Memory usage is very direct and simple as well. It’s efficient and has little overhead.
  + It has good performance iterating through all the classes.
* Disadvantages:
  + Immensely slow search time of O(n) for a simple lookup. This is very slow for a simple lookup of a course, and could result in substantial loading times.
  + The loadCourses() function validation requires an even more inefficient runtime, which would be unacceptable for large databases of courses.

### Hash Table

* Advantages:
  + Very, very fast functions across the board. Applies to all, like search, insertion, deletion, and has an average time complexity of O(1).
  + Leads to the absolutely most efficient implementation of loadCourses at O(n).
  + This is ideal for the implementation of lookup by ID for each course.
* Disadvantages:
  + Elements aren’t stored in a sorted order. This means that when printing the catalog in alphabetical order would be a slow operation requiring a lot of space as it will need to be extracted into a temporary vector.
  + Can use significantly more memory itself than a vector due to the structure of the hash table itself.

### Binary Search Tree

* Advantages:
  + This implementation keeps the elements sorted automatically. This means that for one of our requirements of sorting the course catalog in alphabetical order for printing is highly efficient, requiring only an in-order traversal of O(n).
  + Good on average for all functions. Much better than a vector in most applications for insertion and search.
* Disadvantages:
  + Worse case performance degrades significantly to O(n) for searches and insertions if the tree becomes unbalanced. This can easily happen unfortunately, and can happen if the input data is already sorted.
  + Substantially more complex to implement.

### Recommendation and Justification

Based on the project requirements, runtime analysis, and evaluation, I would personally recommend the Hash Table data structure for the implementation of this course catalog project.

I believe that since the main requirements for the ABCU advertising software are to load a potentially extremely large data file and speed is needed to give users a fast experience accessing course information. The efficiency in loading and validation is fantastic and unmatched at worst case. The hash table runtime analysis clearly provides evidence for this, compared to the worse cases of the BST and vector, the hash table completely sweeps the floor. The most common use of the program will be looking up courses. The hash table is unmatched in its average of O(1) for search time, as it is the fastest possible. While the hash table’s main disadvantage is it’s inherently out of order, the cost of printing in alphabetical order of O(n log n) is a perfectly acceptable tradeoff. This function will likely be rarely used, unlike the fast loading, validation, and searching, which will likely be daily operations, and possibly multiple times daily.