# CS 300 Pseudocode Document

//Vector - Milestone 1

// Function to load courses from a file into a vector

void loadCourses(string filename, Vector &courses){

**while not end of file**

**read line from file**

**split line into tokens**

**if number of tokens < 2**

**print "Error: Invalid file format"**

**continue**

**create a new Course object**

**set courseNumber to first token**

**set courseTitle to second token**

**for each remaining token**

**add token to prerequisites list**

**add Course object to courses vector**

**close file**

}

// Function to validate course prerequisite

void validateCourses(Vector &courses) {

**for each course in courses**

**for each prerequisite in course.prerequisites**

**if prerequisite is not in courseNumbers set**

**print "Error: Prerequisite " + prerequisite + " does not**

**exist"**

}

// Function to search for and print a course's information

void searchCourse(Vector &courses, string courseNumber) {

**for each course in courses**

**if course.courseNumber equals courseNumber**

**print course.courseNumber + " " + course.courseTitle**

**print "Prerequisites:"**

**for each prerequisite in course.prerequisites**

**print prerequisite**

**return**

**print "Course not found"**

}

[\\Menu](file:///\\Menu)

**function menu():**

**repeat:**

**print "1. Load Data"**

**print "2. Print Course List"**

**print "3. Print Course Information"**

**print "9. Exit"**

**userInput = getUserInput()**

**if userInput == 1:**

**filename = getFilenameFromUser()**

**loadData(filename)**

**else if userInput == 2:**

**printAllCourses(dataStructure)**

**else if userInput == 3:**

**courseNumber = getCourseNumberFromUser()**

**searchCourse(dataStructure, courseNumber)**

**else if userInput == 9:**

**print "Exiting program."**

**break**

**else:**

**print "Invalid option. Please try again."**

//Hash Table - Milestone 2

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

**IF courseNumber NOT in courses:**

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

**RETURN**

**SET course = courses[courseNumber]**

**PRINT courseNumber + ": " + course.title**

**IF course.prerequisites is empty:**

**PRINT " Prerequisites: None"**

**ELSE:**

**PRINT " Prerequisites: " + JOIN(course.prerequisites, ", ")**

}

//Loading Data

void loadCourses(String filename, HashTable<Course> &courses) {

**OPEN file with name filename**

**IF file cannot be opened:**

**PRINT "Error: Unable to open file."**

**RETURN**

**FOR each line in file:**

**SPLIT line into tokens by comma delimiter**

**IF number of tokens < 2:**

**PRINT "Error: Invalid format. Each line must have at least a course number and title."**

**CONTINUE**

**SET courseNumber = first token**

**SET courseTitle = second token**

**SET prerequisites = remaining tokens (if any)**

**CREATE new Course object with courseNumber, courseTitle, and prerequisites**

**INSERT Course object into courses using courseNumber as key**

**CLOSE file**

}

//Validating Prerequisites

void validatePrerequisites(HashTable<Course> &courses) {

**FOR each course in courses:**

**FOR each prerequisite in course.prerequisites:**

**IF prerequisite NOT in courses:**

**PRINT "Error: Prerequisite " + prerequisite + " for course " + course.courseNumber + " does not exist."**

}

//Printing Info

void printCourses(HashTable<Course> &courses) {

**SORT courses keys alphabetically**

**FOR each courseNumber in sorted keys:**

**SET course = courses[courseNumber]**

**PRINT courseNumber + ": " + course.title**

**IF course.prerequisites is empty:**

**PRINT " Prerequisites: None"**

**ELSE:**

**PRINT " Prerequisites: " + JOIN(course.prerequisites, ", ")**

}

//Binary Search Tree – Milestone 3

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

//Print all courses

**Function PrintAllCourses(tree)**

**Print "Here is a list of all available courses:"**

**Call InOrderPrint on the root of the tree**

//In order traversal

**Function InOrderPrint(node)**

**If the node is not empty**

**Call InOrderPrint on the left child**

**Print the course number and title from this node**

**Call InOrderPrint on the right child**

//Print Detailed Info

**Function PrintCourseDetails(tree, courseNumber)**

**Search for the course in the tree using FindCourse**

**If no course is found**

**Print "Course not found."**

**Stop**

**Print the course number and title**

**If the course has prerequisites**

**Print "Prerequisites:" followed by the list of them**

**Else**

**Print "Prerequisites: None"**

//Search course

**Function FindCourse(node, courseNumber)**

**If the current node is empty**

**Return nothing**

**If the course number matches the one in this node**

**Return the course stored in this node**

**If the course number comes before this node’s course number**

**Search in the left subtree**

**Else**

**Search in the right subtree**

//Search and announce result

**Function searchCourse(tree, courseNumber)**

**Look for the course using FindCourse**

**If a course is found**

**Print "Course Found:"**

**Print the course number and title**

**If the course has prerequisites**

**Print them as a comma-separated list**

**Else**

**Print "Prerequisites: None"**

**Else**

**Print that the course was not foun****d**

**Runtime Analysis**

**Vector – Search Function**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **for each prerequisite of the course** | 1 | 1 | 1 |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 3n + 1 |
| **Runtime** | | | O(n) |

The vector performs consistently with a linear time complexity of O(n) for this stage. Appending to a vector is a simple and efficient operation. However, vectors do not support efficient searching or sorting by default. To sort the courses later for display, an additional O(n log n) step would be needed. This makes the vector a good choice for basic loading, but not ideal for searching or displaying data in order without more processing.

**Hash Table – Search Function**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **get course from hash table** | 1 | 1 | 1 |
| **if the course exist** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | p | p |
| **Print each prerequisite of the course** | 1 | p | p |
| **Total Cost** | | | 2p + 2 |
| **Runtime** | | | O(p) |

In the average case, the hash table performs extremely well. Insertions are done in constant time, and searches for specific courses are also O(1). However, in the worst case (due to hash collisions), inserting and retrieving can become O(n), turning total loading time into O(n²). Another key issue is that hash tables do not store items in order. This means that while search performance is great, printing courses in order requires additional logic and computation, lowering overall efficiency for this specific program.

**Binary Search Tree – Search Function**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Find course in BST** | Log n | 1 | Log n |
| **If course exist** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | p | p |
| **print the prerequisite course information** | 1 | p | p |
| **Total Cost** | | | Log n + 2p + 1 |
| **Runtime** | | | O(log n + p) |

The binary search tree performs well in both search and ordering. Unlike the vector, it doesn’t need extra sorting after loading. Unlike the hash table, it keeps everything in alphanumeric order as data is inserted. In a balanced state, it achieves very good performance across all required operations. However, the tree must be managed carefully to avoid degenerating into a linear structure (which happens when inserting sorted data into an unbalanced BST), or its performance will drop to O(n²). A self-balancing BST (like AVL or Red-Black Tree) would solve this issue.

**Final Recommendation**

Based on the findings from the runtime tables and how the algorithms behave, the binary search tree is the most efficient and reliable choice for this project. It supports fast searching, naturally maintains course order, and avoids the need for additional sorting logic. While the hash table offers faster average-case lookup, it lacks ordering and has poor worst-case behavior. The vector is simple but lacks performance for searching and sorting. For the advising system's needs—searching individual courses and displaying all courses in order—the BST is clearly the best fit.