**Joseph Thomas**

**April 19, 2025**

**CS-300: Project One**

**Pseudocode for opening file, reading data from file while parsing each line and checking for file format errors:**

Create file stream and open file

While lines exist

Split by commas into tokens

If tokens < 2

Print an error message and stop program

Else

courseNumber = tokens[0]

courseName = tokens[1]

For every token in tokens[2]:

If courseNumber is invalid

Print Error

Else

Add token to prereq vector

**Pseudocode for creating course objects and storing them in appropriate data structure:**

CREATE class courseBST:

CREATE variables courseNumber, courseName

CREATE vector prereq

CREATE left and right courseBST pointer nodes

DECLARE constructor

courseBST(str courseNumber, str courseName, vector<str> prereq)

courseBST\* left = NULL

courseBST\* right = NULL

this->courseNumber = courseNumber

this->courseName = courseName

this->prereq = prereq

CREATE insert function

insertCourseBST(courseBST\* root, str courseNumber, str courseName, vector<str> prereq)

if (root == null)

root = new courseBST(courseNumber, courseName, prereq)

return

if courseNumber < root->courseNumber

root->left = insert(root->left, courseNumber)

else

root->right = insert(root->right, courseNumber)

return

**Pseudocode for printing course information and prerequisites:**

void searchCourse(Tree<courseBST> courses, str courseNumber)

Node\* current = root

WHILE current is not null

IF current-> courseNumber == courseNumber

RETURN current->courseNumber

IF current->courseNumber < courseNumber

Current=current->left

ELSE

Current=current->right

Return courseNumber;

**Menu option 1:**

Call loadBids and store data in courseBST

**Menu option 2:**

Call validateTree() passing courseBST

**Menu option 3:**

Get user input and store in variable “userInput”

Call printCourseTree() passing userInput

**printCourseTree() pseudocode:**

Create node pointer “root”

Root = null

If root = null, return

Call node’s left pointer recursively

Output courseID, courseName

|  |  |  |
| --- | --- | --- |
| **Hash Table** | Line cost | # times |
| Open file | 1 | 1 |
| Loop to process each line | 1 | n |
| Split the line | 1 | n |
| Create Object | 1 | n |
| Insert course | 1 | n |
| **Binary Search Tree** | Line cost | # times |
| Open file | 1 | 1 |
| Loop to process each line | 1 | n |
| Split the line | 1 | n |
| Create Object | 1 | n |
| Insert course | 1 | nlogn |
| **Vector** | Line cost | # times |
| Open file | 1 | 1 |
| Loop to process each line | 1 | n |
| Split the line | 1 | n |
| Create Object | 1 | n |
| Insert course | 1 | O(n+m) |
|  |  |  |

Worst case runtime for Hash is O(n).

Worst case runtime for BST is O(nlogn).

Worst case runtime for vector is O(n+m).

I would choose the Hash table with worst case runtime of O(n) as this would be the most efficient algorithm compared to O(n+m) and O(nlogn), especially if there is little collision.