# CS 300 Final Project Pseudocode Document

## Pseudocode to display and run the main menu.

The application can be opened from the command line while reading the CSV file address and additional courseKey argument, so it will be searched later. The main menu will print four menu options and exit the app as the “9” input has been read.

* Main():

Local arguments:

String: csvPath, courseKey

Course: course

Int: choise = 0

Define switch cases for argv[0] and argv[1] command line input.

csvPath = argv[0], courseKey = argv[1].

Define a binary search tree (BST) structure to hold all courses.

Call new BinarySearchTree() and store it as BST.

Repeat while loop until user choice = 9

Print menu options:

1. Load Courses, 2. Display All Courses, 3. Find Course

4. Remove Course, 9. Exit

case 1: Load Courses

while csvPath is not empty, get filename from a user.

Call the loadCourses() method to read the courses from the file and load it

to the BST. Pass csvPath and BST arguments.

In loadCourses():

Try opening the file(csvPath), Print Error message if failed, and

return to the main menu.

If the file exists and is readable:

Keep reading line by line and parsing data from each column.

Validate that the read line has both course number and name

otherwise, skip the line (row.size() < 2).

Validate prerequisites:

Check that every prerequisite (row[≥2]) exists in the Binary Search Tree (BST) data structure as a coursceKey.

If **all** prerequisites for each courseKey exist in the BST:  
 Create a new node(), and Push the course to the BST.

If a course has **no** prerequisites (row[≥2]=null):

Create a new node(), and Push the course to the BST.

If **any** of the course prerequisites do not yet exist in the BST:

// \*Allows the program to handle **unsorted data input**, where prerequisites may appear **after** dependent courses in the read file.

Recursively Call the loadCourses() to load the missing prerequisite and search for the courses in the rest of the file.

If found, insert the prerequisite into the BST and return to the previous course.

If **any** of the course prerequisites can not be found, print an Error prompt and Continue to the next row.

Back in Main(): Display the counter for successfully read courses.

case 2: Display All Courses

Call BST->InOrder() method to Print an alphanumerically ordered list of

all the courses.

case 3: Find Course

If courseKey (argv[0]) is empty, Get the course Number from the user.

Call BST->Search(courseKey) to search for a specific course if such

exists in the BST. Returns course instance if a course is found.

If the requested course is found in the BST, print the course

Information:

Call displayCourse(), pass course instance to Print course number

and course name to the display.

Call displayPrerequisites(), pass course instance to Print course

prerequisites to the display if such exist.

Else (requested course was not found), Print error prompt.

case 4: Remove Course

Get courseKey from the user to be removed from the BST.

Call BST->Remove() and pass courseKey argument.

case 9: Exit

Call the BinarySearchTree\* BST Destructor to clear memory.

Quit.

## Pseudocode to print out the list of all the coursces from the BST data structure.

The read data from the CSV file will reside in a Binary Search Tree (BST) in the form of nodes. Each node will hold an instance of a course (), and two pointers, \*left and \*right, to point to the next tree branch or a leaf. Each course, will have the following arguments: courseId, courseTitle, and the prerequisites vector<string>. When inOrder(root)is called by passing the root argument, it will recursively traverse the BST and will print each course’s number (courseId) and name (courseTitle).

* PrintInOrder():

Call inOrder(root) method and pass the root node to start traversing the BST.

In void inOrder():

if node is not empty (!=nullptr)

Recursively visit the left subtree to go down the branches:

Call InOrder(node->left),

A lowest tree leaf is reached, output course data:

Call displayCourse(course)

Recursively visit the right subtree to go down the branches:

Call InOrder(node->right)

As the data that resides in a Binary Search Tree (BST) is inserted into the tree, and each node is organized by its alphanumeric size, e.g., (A1**<**B2), recursively calling the InOrder()method while passing the addresses of the left and right nodes, will output a printout of a course number (courseId) from lowest to highest, e.g., form leaf to the root.

**Vector Data Structure Runtime Analysis**

| **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 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

**Hash Table Structure Runtime Analysis**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Loop through all indexes (buckets) in the hash table:** for (i = 0; i < tableSize; ++i) | 1 | n | n |
| **Accessing the current bucket:** Node\* currentNode = &nodes[i] | 1 | 1 | 1 |
| **Check if the first element in the bucket is not an empty node and print the node’s content:** if (currentNode->key != UINT\_MAX) { cout << "Key " << currentNode->key << ": "} | 1 | n | n |
| **Move the pointer to the next node:** currentNode = currentNode->next; | 1 | 1 | 1 |
| **Print the data for all the rest of the nodes in the list until the pointer reaches nullptr:** while (currentNode != nullptr) {} | 1 | n | n |
| **Total Cost** | | | 3n + 2 |
| **Runtime** | | | O(n) |

**Binary Search Tree Structure Runtime Analysis**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Base case: Check if node exists:**  if (node != nullptr) {} | 1 | n | n |
| **Recursively visit the left subtree:**  inOrder(node->left); | 1 | n | n |
| **Visit node and Output bidID and title:**  cout << node->bid.bidId << ": " | 1 | 1 | 1 |
| **Recursively visit the left subtree:**  inOrder(node->right); | 1 | n | n |
| **Total Cost** | | | 3n + 1 |
| **Runtime** | | | O(n) |

**The advantages and disadvantages of each structure in your evaluation**

Different data structures can behave differently and provide both advantages and disadvantages, considering how they would be applied in the app and how they will be used. The Vector data structure is easy to set up and implement, and it stores the data in the system memory contiguously. This leads to fast data insertion at the end of the vector and direct data retrieval from the Vector if the index is known. It performs well on small datasets and where linear searches are manageable. However, it is inefficient where data should be added in the middle of the Vector or deleted from it as all indexes should be rearranged. On the other hand, the Hash Table allows fast search and data Insertion/deletion at constant time complexity and performs well as data scales. It is Ideal for quick lookups and relatively easy to implement. However, the good performance of the Hash Table algorithm profoundly depends on a clever hash function that must prevent collisions and properly manage chaining and probing. In addition, data in a Hash Table is stored in an unordered manner, which makes ordered traversal more complex. Lastly, we have the Binary Search Tree (BST) data structure, which is considerably more complex to implement and maintain compared to the previously mentioned structures and could suffer from a degraded performance if the inserted data is already sorted and not random. Although the BST provides significant advantages, such as efficient logarithmic search time and efficient insertions and deletions of elements without resizing the structure, and it also excels in returning data in sorted order.

In general, the efficiency and the complexity of the data structures could be summarized and compared in the following table:

| **Operation** | **Vector**  **(Avg/Worst)** | **Hash Table**  **(Avg/Worst)** | **BST**  **(Avg/Worst)** |
| --- | --- | --- | --- |
| **Search Efficiency** | O(n) / O(n) | O(1) / O(n) | O(logn) / O(n) |
| **Insertion Efficiency** | O(n) / O(n) | O(1) / O(n) | O(logn) / O(n) |
| **Deletion Efficiency** | O(n) / O(n) | O(1) / O(n) | O(logn) / O(n) |
| **Time Complexity** | O(n) / O(n) | O(1) / O(n) | O(logn) / O(n) |

**Recommendation and Conclusion**

Based on the advisor’s requirements to read random unsorted data from a CSV file, analyze it before insertion by traversing the data structure through the search methods, and later output it in an ordered and continuous manner, the Binary Search Tree (BST) will provide the best performance considering the fast average and worst case Big O operation time, particularly as the size of the dataset (courses and prerequisites) increases. Despite its more complex implementation in the application, it will allow fast and efficient lookups, insertion, simple sorted list output, and low system resource consumption.

A diagram of a computer tree

Description automatically generated