# CS 300 Pseudocode Document

**PSEUDOCODE**

**Reading File:**

OPEN file

void loadCOurses(string csvPath, dataStructure)

CALL to open file

IF return -1,

THEN file is not found

ELSE file is found

While NOT EOF (End of File)

READ each line

IF values < 2 per line

THEN return ERROR

ELSE read parameters

IF there is a third or more parameter

IF third or more parameter is in the first elsewhere, continue

ELSE return Error

Close file

**Hold Course Information:**

class Course {

String courseNumber

String courseTitle

Vector<String> prerequisites

}

//Vector - Milestone 1

**Vector<Course> loadCourses(String csvPath ) {**

For (int i = 0; i < file.rowCount(); i++) {

Course course;

course.courseId = file[i][1];

course.name = file[i][0];

WHILE not end of line

course.prereq = file[i][8]

course.push\_back(course);}

//Hash Table - Milestone 2

**void loadCourses(string csvPath, HashTable\* hashtable)**

USE loop to read the row of the csv file

For (unsigned int i = 0; i < file. rowCount(); i++) {

Course course;

course.courseId = file[i][1];

course.name = file.[i][0];

while not end of line

course.prereq = file[i][8];

HashTable->Insert(course);

}

//Binary Search Tree – Milestone 3

**Define a binary search tree to hold all courses**

BinarySearchTree\* bst;

bst = new BinarySearchTree();

Course course;

**Create add node method void BinarySearchTree::addNode(Node\* node, Course** course

IF root is NULL

THEN add root

IF node is less than root

THEN add to left

IF no left node

THEN Current node becomes right

IF node is greater than root add right

IF no right node

this node becomes right

**void loadCourses(string csvPath, BinarySearchTree\* bst)**

USE loop to read rows of a CSV file

for (unsigned int i = 0; i < file.rowCount(); i++) {

Course course;

course.couseId = file[i][1];

course.name = file[i][0];

While not end of line

course.prereq = file[i][8];

bst->Instert(course);

**Print Course Info and Prerequisites:**

// Vector

void printCourseInforamtion(**Vector<Course> courseId)**

**GET input for courseId**

ASSIGN key = courseId

ASSIGN node to the node.at(key)

IF current node matches key

RETURN course

displayCourse(nodes[key].course)

IF node points to null, return null

ELSE

WHILE the node is not null, check against the key

IF the key matches the courseId

RETURN course

displayCourse(nodes[key].course)

POINT to next node

**// TREE**

voidPrintCourseInformation(Tree<Course> courses, String couseId)

GET input for couseIdAssing current node to root

While current node is not NULL

IF course.courseId matches current

RETURN current

OUTPUT course.couseId << output course.name

While (prereq = true)

OUTPUT course.prerequisites

IF courseId is less than root

Set current to left

ELSE set current to right

**MENU:**

SET choice to 0;

WHILE choice is not equal to 4

OUTPUT menu choices

1. Load Course File

2. Print Course List

3. Print Indiviudal Course

4. EXIT

CREATE switch(choice)

Case 1: loadCourses(courseFIle, dataStructure) FIXME: use structure of data structure chosen

Case 2: printSorted(courses) call function to print sorted class list

Case 3: printCourseInforamtion(courseId)

Case 4: Terminate Program

Prin Sorted Lists:

**// Vector**

printSOrted(courses)

partition(vecto<Course>course, int begin, int end)

SET lowIndex to first element, set highIndex to last element

SET midpint to lowIndex + (highIndex - lowIndex) / 2

SET pivot to midpoint

Decrement highIndex while pivot is less than highIndex

SWAP lower values to left of pivot, higher values to right of pivot

Set temp value to low index

Set low index to high index

Set high index to temp

Create quicksort method void quickSort(vector<Course>&course, int begin, int end)

SET mid to 0, lowIndex to begin, highIndex to end

IF begin greater than or equal to end,

RETURN lowEndIndex to partition(courses, lowIndex, highIndex)

Make recursive call to quicksort

quickSort(course, lowIndex, lowEndIndex);

quickSort(coursse, lowEndIndex + 1, highIndex)

void displayCourse(Course course) {

cout << course.courseId << ":" << course.name << "|" << course.prereq << endl;

for (int i = 0; < courses.size(); ++i)

displayCourse(courses[i]) /Tree

void BinarySearchTree::inOrder(Node\* node)

IF node does not equal NULL

Check most let side first

inOrder(node->left)

cout << course.courseId << ":" << course.name << "|" << course.prereq << endl;

check next right leaf

inOrder(node->right)

cout << course.courseId << ":" << course.name << "|" << course.prereq << endl

| **Vector** | **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 |
| **Total Cost** | | | 4n + 1 |

| **Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create Vector** | 1 | 1 | 1 |
| WHILE prerequisite exists | 1 | n | n |
| **APPEND prerequisite** | 1 | n | n |
| **PUSH BACK course item** | 1 | n | n |
|  |  | **Total Cost** | 6n + 1 |
|  |  | Runtime | O(n) |
|  | | |  |

| **HashTable** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create HashTable** | 1 | 1 | 1 |
| Insert method | 0 | 0 | 0 |
| **CREATE course key** | 1 | n | n |
| **IF no key entry** | 1 | n | n |
| **ASSIGN node to key** | 1 | **n** | n |
| **ELSE** | 1 | **n** | n |
| **ASSIGN old node key to UNIT\_MAX, SET old node to course and old node to next null pointer** | 4 | **n** | 4(n) |
| **ELSE** | 1 | **n** | n |
| **Find the next open node** | 1 | **n** | n |
| **ADD new node to end** | 1 | **n** | n |
| **For each line of File** | 1 | **n** | n |
| **CREATE Vector course item** | 1 | **n** | n |
| **WHILE prerequisite exists** | 1 | **n** | n |
| **Insert course item** | 1 | **n** | n |
|  |  | **Total Cost** | 16n + 1 |
|  |  | Runtime | O(n) |
|  | | |  |

| **Binary Tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create TREE** | 1 | 1 | 1 |
| ADD node method | 0 | 0 | 0 |
| **IF root NULL, Then add root** | 1 | n | n |
| **IF node < root, then add to left** | 1 | n | n |
| **IF no left node** | 1 | **n** | n |
| **This node added to left** | 1 | **n** | n |
| **IF node > root, then add to right** | 1 | **n** | n |
| **IF no right node** | 1 | **n** | n |
| **This node added to right** | 1 | **n** | n |
| **For each line in file** | 1 | **n** | n |
| **CREATE Vector course item** | 1 | **n** | n |
| **WHILE prerequisite exists** | 1 | **n** | n |
| **APPEND prerequisite** | 1 | **n** | n |
| **Insert course item** | 1 | **n** | n |
|  |  | **Total Cost** | 12(n) + 1 |
|  |  | Runtime | O(n) |
|  | | |  |

**ANALYSIS**

Each data structure has advantages and disadvantages based on the requirements.

Among the three methods, the vector method has the shortest runtime, at 6n+1, with each item simply appended to the end of a vector. It is a very straight forward method that parses the file and adds the course objects. All three methods had the same O(n) notation, but the runtime was shorter at 6n+1. The vector is simple and easy to implement plus efficient for small datasets.

There are some disadvantages to vectors. The program must search through the vectors until a

match is found. It also is inefficient for insertions and deletions and sorting can be expensive.

The Hash Table, however, is efficient at handling large datasets and fast for lookups and

insertions. By creating a key, we will know where the course is located and can easily search and print it. The initial list creation is slow since a key must be created for each item and a place for each course must be found. Furthermore, hash tables are not suited to sorting. It is not possible to sort the table itself. It is necessary to extract, sort, and print each value individually to generate an alphanumeric list of all courses. Furthermore, handling collisions with hash tables can be complex and unstable. As a result, this data structure is not the most suitable choice for this application.

Finally, when compared to vector and the binary tree, binary tree is more efficient at searching when it is in a balanced state and maintains order. It is extremely easy to run down the tree until the value is located if you know the course being searched. A disadvantage is that insertion and deletion operations can lead to an unbalanced tree and are more complex than a vector or a hash table. Thus, this approach is not as straightforward as a hash table, but it is more efficient than a vector. As a worst-case scenario, if only leaves remain, the tree would have to search for each element. This would result in the search time being O(h) where h correspondsto the tree height.

**RECOMMENDATION**

When considering the worst-case running time, memory complexity, and the specific

requirements of the, the Vector is a suitable choice. It provides efficient average-case

performance for both insertion and search operations, making it suitable for the specified tasks.

The advantages of quick look-up and handling large datasets outweigh the potential

disadvantages of handling collisions. The client would benefit from being able to sort and print

the entire catalog quickly. A further benefit of the search utility is that it loses less time in

searching than the sort utility. The vector is the most appropriate option