# Rich Knoll

# CS 300 Pseudocode Document

//Vector - Milestone 1

Void loadCSV(string csvPath) {

Validate the file is of the csv type

For all records:

Make sure that there are at least two attributes

If there are any pre-requisites, these courses must exist as their own record

}

Struct Course {

String courseNumber

String courseDescription

Vector <string> prerequisites (numElements)

}

While (the next item in the vector is not empty) {

Start at head of vector and create Course Struct with line 1 information. Then, append course to vector. Finally, move to next record in csv file.

}

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

**for all courses**

**if the course is the same as courseNumber**

**print out the course information**

**for each prerequisite of the course**

**print the prerequisite course information**

}

// Function the prints all courses in alphabetical order

void printAllCourses(Vector<Course> courses) {

// Insertion sort

For(i = 1; i < coursesSize; ++i) {

j = i;

while(j > 0 && courses[j] < courses[j-1]){

temp = courses[j];

courses[j] = courses[j-1];

courses[j-1] = temp;

--j;

}

}

For(i = 0; i < coursesSize; ++i) {

Print i;

}

}

//Hash Table - Milestone 2

// course structure to hold course number, course description, and vector of prerequisites

Struct Course {

String courseNumber;

String courseDescription;

Vector <string> prerequisites (numElements);

// Node structure to hold course, key, and pointer to the next node in chain

Struct Node {

Course course;

Unisigned int key;

Node\* next;

}

// Node constructors

//Default constructor

Node(){

Key = UINT\_MAX;

Next = nullptr;

}

// Initialized with a course

Node(Course aCourse): Node() {

Course = aCourse;

}

// Initialized with a course and a key

Node(Course aCourse, unisigned int aKey): Node(aCourse) {

Key = aKey;

}

// vector of nodes

Vector <Node> nodes;

// initialize variable to hold the size of the vector as an int

Unsigned int tableSize = size of nodes vector;

// Hash function to return hash table key using argument of function and previously defined tableSize variable

Unisigned int hash(int key){

Return tableSize % key;

}

// Insert function to insert course into nodes vector using chaining if necessary

Void Insert (Course course) {

Unsigned key = hash(courseNumber);

Node\* oldNode = address of the node at key;

If (no node found at key) {

Create new node with key and course number;

}

Else if (node is not used) {

Assign new key;

Assign new course;

Assign next to nullptr;

}

Else {

While (next node does not equal nullptr){

oldNode = nextNode -> next;

}

oldNode->next = new Node(course, key);

}

}

// loads and parses csv file

Void loadCourses(string csvPath, HashTable\* hashTable ) {

Validate the file is of the csv type

For all records:

Make sure that there are at least two attributes

If there are any pre-requisites, these courses must exist as their own record;

Create course data structure and add to collection of courses;

Insert(course);

}

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

Course course;

// create key for given course

Unisigned key = hash(courseNumber);

Node\* node = address of node at key;

If (entry is found at key) {

Return course;

}

If (no entry found at key){

Return empty course;

}

If (entry found at key but course numbers do not match) {

While (node does not equal nullptr){

If (course numbers match) {

Return course;

}

Else{

Node = node->next;

}

}

}

}

void printAllCourses(hashTable<Course> courses) {

for all items in hashTable{

add each item to courses vector

}

// Insertion sort

For(i = 1; i < coursesSize; ++i) {

j = i;

while(j > 0 && courses[j] < courses[j-1]){

temp = courses[j];

courses[j] = courses[j-1];

courses[j-1] = temp;

--j;

}

}

For(i = 0; i < coursesSize; ++i) {

Print i;

}

//Binary Search Tree – Milestone 3

Void loadCSV(string csvPath) {

Validate the file is of the csv type

For all records:

Make sure that there are at least two attributes

If there are any pre-requisites, these courses must exist as their own record

}

// course structure to hold course number, course description, and vector of prerequisites

Struct Course {

String courseNumber;

String courseDescription;

Vector <string> prerequisites (numElements);

// Node structure to hold course, pointer to left node, and pointer to right node

Struct Node {

Course course;

Node\* left;

Node\* right;

}

//Default constructor

Node() {

Left = nullptr;

Right = nullptr;

}

// Initialize with a bid

Node(Bid aBid): Node(){

Bid = aBid;

}

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

Node\* current = root;

While(current != nullptr){

If (current courseNumber == courseNumber) {

Return current courseNumber

}

If (courseNumber < current courseNumber) {

Current = current->left

}

Else {

Current = current->right

}

}

Course course;

Return course;

}

// Already prints in order

Void printAll(Node\* node) {

If(node != nullptr){

printAll(node->left);

cout << All necessary info << endl;

printAll(node->right);

}

}

// Main Menu

Switch (userInput) {

Case 1:

Load file data into data structure;

Break;

Case 2:

Print alphanumeric list of courses;

Break;

Case 3:

Print course title and pre-requisites for any individual course;

Break;

Exit;

}

## Runtime Analysis

**Vector**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Validate the file is of the csv type** | 1 | 1 | 1 |
| **For all records** | 1 | n | n |
| **Make sure that there are at least two attributes** | 1 | n | n |
| **If there are any pre-requisites** | 1 | n | n |
| **Check to make sure these courses exist as their own record** | 1 | n | n |
| **Create course struct** | 1 | N | N |
| **Append struct to vector** | 1 | N | N |
|  |  |  |  |
|  |  |  |  |
| **Total Cost** | | | 6n + 1 |
| **Runtime** | | | O(n) |

**Hash Table**

**Insert Course Function**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Unsigned key = hash(courseNumber)** | 1 | 1 | 1 |
| **Node\* oldNode = address of the node at key** | 1 | 1 | 1 |
| **If (no node found at key)** | 1 | 1 | 1 |
| **Create new node with key and course number** | 1 | 1 | 1 |
| **Else if (node is not used)** | 1 | 1 | 1 |
| **Assign new key** | 1 | 1 | 1 |
| **Assign new course** | 1 | 1 | 1 |
| **Assign next to nullptr** | 1 | 1 | 1 |
| **Else** | 1 | 1 | 1 |
| **While (next node does not equal nullptr)** | 1 | N | N |
| **oldNode = nextNode -> next** | 1 | 1 | 1 |
| **oldNode -> next = new Node(course, key)** | 1 | 1 | 1 |
|  |  |  |  |
| **Total Cost** | | | n + 11 |
| **Runtime** | | | O(n) |

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Validate the file is of the csv type** | 1 | 1 | 1 |
| **For all records** | 1 | n | n |
| **Make sure that there are at least two attributes** | 1 | n | n |
| **If there are any pre-requisites** | 1 | n | n |
| **Check to make sure these courses exist as their own record** | 1 | n | n |
| **Insert(course)** | 1 | N | N |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| **Total Cost** | | | 5n + 1 |
| **Runtime** | | | O(n) |

**Binary Search Tree**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Validate the file is of the csv type** | 1 | 1 | 1 |
| **For all records** | 1 | n | n |
| **Make sure that there are at least two attributes** | 1 | n | n |
| **If there are any pre-requisites** | 1 | n | n |
| **Check to make sure these courses exist as their own record** | 1 | n | n |
| **Insert(course)** | 1 | N | N |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| **Total Cost** | | | 5n + 1 |
| **Runtime** | | | O(n) |

**Evaluation**

Each data structure – vector, hash table, and binary search tree – has their own advantages and disadvantages. The vector is the most straightforward and easiest to code. However, when it comes to searching and sorting courses, it is the least efficient. The binary search tree is the most complex to code, but makes up for it in its efficiency for sorting as this is inherent in its structure. The hash table falls in the middle. Relatively easy to code, and cuts down on time when searching due to its structure. Although, not as effective as the BST when it comes to sorting. Because of these trade-offs, we recommend moving forward with the hash table as the data structure for this application.