**Module 6 – Project One Full Pseudocode and Analysis (UPDATE #3):**

**Vector/Course Creation/Search Pseudocode:**

Define courseName as String;

Define courseNumber as String;

Define prerequisite as String;

Define totalPrerequisites as Int;

Create Vector (Vector<Course> courses);

void printSampleSchedule(Vector<Course> courses) {

Open file 🡪 read data;

}

Define totalCount as Int;

Create new course (Course \*courseName = new Course) {

Search courseNum {

If course number is found

Return course;

List mode set to next listNode;

}

Else If course number isn’t found

Return empty object;

}

Create while loop to iterate though each line

If format = true;

Set counter equal to count;

If count = 1;

Set courseNumber = new courseNumber;

Else if count = 2;

Set courseName = new courseName;

Else if count = 3;

Set prerequisite = new prerequisite;

Add course to vector;

**Print Course Information Pseudocode:**

Print course information (Vector<Course> courses, courseNumber);

For each course in the vector {

If course = courseNum {

Print course information;

For each prerequisite p in totalPrerequisites

Add prerequisites of p to totalPrerequisites

Print number of totalPrerequisites

}

}

Return;

**Menu Pseudocode:**

Open file (Myfile.open);

Define sorting (selectionSort);

selectionSort (courses) {

unsigned int min;

Make list to print alphanumerically (for i<courses.size());

}

Print course titles/prerequisites based on given course information

If sorted course.at(i) == searched course {

Print courseName, totalPrerequisites (in order);

}

Return;

**Analyzing Runtime (Vector/Prerequisites):**

| **Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for each string** | 1 | n | n |
| **courseName -> courseNum = parsed string** | 1 | 1 | 1 |
| **Check for name/num** | 1 | 1 | 1 |
| **For each remaining string** | 1 | n | n |
| **If prerequisite in courseName** | 1 | n | n |
| **prerequisites = parsed string** | 1 | n | n |
| **courseName -> prerequisites = totalPrerequisites** | 1 | n | n |
| **Print totalPrerequisites** | 1 | n | n |
| **Total Cost** | | | 6n + 2 |
| **Runtime** | | | O(n) |

**Hashtable Pseudocode:**

Open file (Myfile.open);

Create Class hashtable;

Struct node (course, key, next);

Node constructors;

Create vector<Node> that look at number of buckets

Create hash (key)

Return key;

Insert (course)

Defines key (key = hash) and determine node position/place;

For each course in hashtable {

Look for each bucket that ISN’T empty;

Print course information;

}

**Menu Pseudocode:**

Open file (Myfile.open);

Define Vector<courses> course = new vector;

For each hash key {

Add node entries (course);

Sort vector by courseName;

}

Make list to print alphanumerically (for i<course.at(i));

Print course titles/prerequisites based on given course information;

Return;

**Analyzing Runtime (Hashtable/Prerequisites):**

| **Hashtable** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create vector<Node>, buckets** | 1 | n | n |
| **Hash (key)** | 1 | n | n |
| **Return hash** | 1 | n | n |
| **Insert vector (course)** | 1 | n | n |
| **Define Key (node pos/place)** | 1 | n | n |
| **Determine correct placements (pos/place)** | 1 | n | n |
| **Print info (courses, prerequisites)** | 1 | n | n |
| **Total Cost** | | | 7n |
| **Runtime** | | | O(n) |
|  | | |  |

**Tree Pseudocode:**

Create line for each course;

getline = new course;

Define courseName as new String;

Define courseNum as new String;

Define numPrerequisiteCourses as new String;

Create struct course {

String courseNum;

String courseName;

Make list of numPrerequisiteCourses;

}

Create struct node {

}

Create Class bst {

Add, remove, search, alphanumerical

}

void printSampleSchedule (node, course) {

for each node in schedule, print course;

}

void printCourseInformation (node, courseName) {

if course found, return course;

}

**Menu Pseudocode:**

alphanumerical (Node\* node) {

call alphanumerical;

bst -> alphanumerical;

Print course info;

}

Search (string courseId) {

Node\* current = root;

Order courses based on course info;

}

Close program;

**Analyzing Runtime (Tree/Prerequisites):**

| **Tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **getline** | 1 | n | n |
| **String courseNum** | 1 | n | n |
| **String courseName** | 1 | n | n |
| **String numPrerequisiteCourses** | 1 | n | n |
| **Class bst (course)** | 1 | n | n |
| **Insert (course)** | 1 | n | n |
| **Call alphanumerical (Node)** | 1 | n | n |
| **Root = new node** | 1 | n | n |
| **Order courses (node, course)** | 1 | n | n |
| **Total Cost** | | | 9n |
| **Runtime** | | | O(n) |
|  | | |  |

**Advantages/Disadvantages of Each Structure:**

Each structure utilized (vector, hashtable, and tree) all have complexities regarding the total line cost of each (n), where ‘n’ equals the number of data values that are currently being processed. When analyzing each data structure, there are advantages and disadvantages to all three. For example, a vector’s ability to sort values is strong, but because information is run through the vector itself, it has worse runtime based on the Big O(n) than a hashtable or tree does, making it ultimately slower as well. Hashtables are good for speeding things up when trying to find particular items within a table but aren’t very applicable for our particular project given their data separation process. We have courses that need to be searched thoroughly by name, number, prerequisites, and sorted alphanumerically.

With a search tree, it’s relatively easy to access values alphanumerically compared to the other two. However, based on our runtime analysis and my basic knowledge, a search tree often sees increases in performance times (O(n)) and was the worst comparatively. Given this information, and in relevance to our project goals, I’m likely to select utilizing the tree structure given its access of valuing components from smallest to largest and alphabetically. This was an important feature asked of our team to create, and while it may take longer to load, it’s an important aspect to tend to. That, and by utilizing a binary search tree, creating custom objects, as well as multiple data fields within, it’s relevance to our project most accurately depicts the direction of our project goals.