# CS 300 Pseudocode Document

**// Vector pseudocode**

int numPrerequisiteCourses(Vector<Course> courses, Course c) {

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| totalPrerequisites = prerequisites of course c | 1 | 1 | 1 |
| for each prerequisite p in totalPrerequisites | 1 | n | n |
| add prerequisites of p to totalPrerequisites | 1 | n | n |
| print number of totalPrerequisites | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 2 |
| **Runtime** | | | O(n) |

}

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

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| for all courses | 1 | n | n |
| if the course is the same as courseNumber | 1 | n | n |
| print out the course information | 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) |

}

void printAllCourses(Vector<Course> courses) {

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Vector<Course> sorted = sortCourses(courses) | 3 | 1 | 3 |
| for all courses | 1 | n | n |
| display course info | 1 | n | n |
| **Total Cost** | | | 2n + 3 |
| **Runtime** | | | O(n) |

}

Vector<Course> readCoursesFile(string file) {

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| If file exists: | 1 | 1 | 1 |
| tempCourses = Vector<Course> | 1 | 1 | 1 |
| Open file for reading | 1 | 1 | 1 |
| courseIDs = Vector<string> | 1 | 1 | 1 |
| For each line of file: | 1 | n | n |
| Get substring of line ending in comma | 1 | n | n |
| Add substring to courseIDs | 1 | n | n |
| For each line of file: | 1 | n | n |
| tempCourse = new Course | 1 | n | n |
| Vector<string> courseInfo = Vector<string> | 1 | n | n |
| While getting each delimited line: | 1 | n | n |
| Add line to courseInfo | 1 | n | n |
| If length of courseInfo at least 2: | 1 | n | n |
| tempCourse->ID = courseInfo.pop(0) | 1 | n | n |
| tempCourse->name = courseInfo.pop(0) | 1 | n | n |
| for each prerequisite p in courseInfo: | 1 | n | n |
| if prerequisite of p is in courseIDs: | 1 | n | n |
| add prerequisite of p to tempCourse->prerequisites | 1 | n | n |
| add tempCourse to tempCourses | 1 | n | n |
| return tempCourses | 1 | 1 | 1 |
| **Total Cost** | | | 15n + 5 |
| **Runtime** | | | O(n) |

}

Class Course {

string ID

string name

Vector<string> prerequisites

}

bool compareCourse(Course a, Course b) {

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Return a->ID < b->ID | 1 | 1 | 1 |
| **Total Cost** | | | 1 |
| **Runtime** | | | O(1) |

}

Vector<Course> sortCourses(Vector<Course> courses) {

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| sort(courses.begin(), courses.end(), compareCourse) | 3 | 1 | 3 |
| **Total Cost** | | | 3 |
| **Runtime** | | | O(1) |

}

**// Hashtable pseudocode**

int numPrerequisiteCourses(HashTable courses, Course c) {

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| totalPrerequisites = prerequisites of course c | 1 | 1 | 1 |
| for each prerequisite p in totalPrerequisites | 1 | n | n |
| add prerequisites of p to totalPrerequisites | 1 | n | n |
| print number of totalPrerequisites | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 2 |
| **Runtime** | | | O(n) |

}

void printCourseInformation(HashTable courses, String courseNumber) {

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| int key = HashTable->Hash(courseNumber) | **2** | **1** | **2** |
| for all courses at courses->courses[key] | 1 | n | n |
| if the course is the same as courseNumber | 1 | n | n |
| print out the course information | 1 | 1 | 1 |
| for each prerequisite of the course | 1 | n | n |
| print the prerequisite course information | 1 | n | n |
| **Total Cost** | | | 4n + 3 |
| **Runtime** | | | O(n) |

}

void printAllCourses(Vector<Course> courses) {

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Vector<Course> sorted = sortCourses(courses) | 1 | n | n |
| for all courses | 1 | n | n |
| display course info | 1 | 1 | 1 |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

}

HashTable readCoursesFile(string file) {

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| If file exists: | 1 | 1 | 1 |
| tempCourses = new HashTable | 1 | 1 | 1 |
| Open file for reading | 1 | 1 | 1 |
| courseIDs = Vector<string> | 1 | 1 | 1 |
| For each line of file: | 1 | n | n |
| Get substring of line ending in comma | 1 | n | n |
| Add substring to courseIDs | 1 | n | n |
| For each line of file: | 1 | n | n |
| tempCourse = new Course | 1 | n | n |
| Vector<string> courseInfo = Vector<string> | 1 | n | n |
| While getting each delimited line: | 1 | n | n |
| Add line to courseInfo | 1 | n | n |
| If length of courseInfo at least 2: | 1 | n | n |
| tempCourse->ID = courseInfo.pop(0) | 1 | n | n |
| tempCourse->name = courseInfo.pop(0) | 1 | n | n |
| for each prerequisite p in courseInfo: | 1 | n | n |
| if prerequisite of p is in courseIDs: | 1 | n | n |
| add prerequisite of p to tempCourse->prerequisites | 1 | n | n |
| tempCourses->Insert(tempCourse) | 6 | n | n |
| return tempCourses | 1 | 1 | 1 |
| **Total Cost** | | | 15n + 5 |
| **Runtime** | | | O(n) |

}

Class Course {

string ID

string name

Vector<string> prerequisites

}

void HashTable->Insert(Course c) {

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| int key = HashTable->Hash(c->ID) | 2 | 1 | 2 |
| If no entry found for the key: | 1 | 1 | 1 |
| Assign c to HashTable->courses[key] | 1 | 1 | 1 |
| Else an entry is found: | 1 | 1 | 1 |
| Append to list at HashTable->courses[key] | 1 | 1 | 1 |
| **Total Cost** | | | 6 |
| **Runtime** | | | O(1) |

}

int HashTable->Hash(string key) {

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Compute numerical value of key | 1 | 1 | 1 |
| Return numerical value of key modulo HashTable->tableLength | 1 | 1 | 1 |
| **Total Cost** | | | 2 |
| **Runtime** | | | O(1) |

}

bool compareCourse(Course a, Course b) {

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Return a->ID < b->ID | 1 | 1 | 1 |
| **Total Cost** | | | 1 |
| **Runtime** | | | O(1) |

}

Vector<Course> sortCourses(Vector<Course> courses) {

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| sort(courses.begin(), courses.end(), compareCourse) | 3 | 1 | 3 |
| **Total Cost** | | | 3 |
| **Runtime** | | | O(1) |

}

**// Tree pseudocode**

int numPrerequisiteCourses(BST courses, Course c) {

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| totalPrerequisites = prerequisites of course c | 1 | 1 | 1 |
| for each prerequisite p in totalPrerequisites | 1 | n | n |
| add prerequisites of p to totalPrerequisites | 1 | n | n |
| print number of totalPrerequisites | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 2 |
| **Runtime** | | | O(n) |

}

void printCourseInformation(BST courses, string courseNumber) {

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Course current = courses->root | **1** | **1** | **1** |
| While current != null: | 1 | n | n |
| If the current->ID is the same as courseNumber: | 1 | n | n |
| print out the course information | 1 | 1 | 1 |
| for each prerequisite of the course | 1 | n | n |
| print the prerequisite course information | 1 | n | n |
| Else if the current course is less than the courseNumber: | 1 | n | n |
| current = current->leftNode | 1 | 1 | 1 |
| Else: | 1 | n | n |
| current = current->rightNode | 1 | 1 | 1 |
| **Total Cost** | | | 6n + 4 |
| **Runtime** | | | O(n) |

}

void printAllCourses(Course c) {

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| If c is null: | 1 | 1 | 1 |
| Return | 1 | 1 | 1 |
| Else: | 1 | 1 | 1 |
| printAllCourses(c->leftNode) | 1 | 1 | 1 |
| Display course info | 1 | 1 | 1 |
| printAllCourses(c->rightNode) | 1 | 1 | 1 |
| **Total Cost** | | | 6 |
| **Runtime** | | | O(1) |

}

Class Course {

string ID

string name

Vector<string> prerequisites

Course leftNode

Course rightNode

}

BST readCoursesFile(string file) {

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| If file exists: | 1 | 1 | 1 |
| tempCourses = new BST | 1 | 1 | 1 |
| Open file for reading | 1 | 1 | 1 |
| courseIDs = Vector<string> | 1 | 1 | 1 |
| For each line of file: | 1 | n | n |
| Get substring of line ending in comma | 1 | n | n |
| Add substring to courseIDs | 1 | n | n |
| For each line of file: | 1 | n | n |
| tempCourse = new Course | 1 | n | n |
| Vector<string> courseInfo = Vector<string> | 1 | n | n |
| While getting each delimited line: | 1 | n | n |
| Add line to courseInfo | 1 | n | n |
| If length of courseInfo at least 2: | 1 | n | n |
| tempCourse->ID = courseInfo.pop(0) | 1 | n | n |
| tempCourse->name = courseInfo.pop(0) | 1 | n | n |
| for each prerequisite p in courseInfo: | 1 | n | n |
| if prerequisite of p is in courseIDs: | 1 | n | n |
| add prerequisite of p to tempCourse->prerequisites | 1 | n | n |
| tempCourses->Insert(tempCourse) | 3n + 14 | 3n + 14 | 3n + 14 |
| return tempCourses | 1 | 1 | 1 |
| **Total Cost** | | | 18n + 19 |
| **Runtime** | | | O(n)DCx z |

}

void BST->Insert(Course c) {

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| if BST->root is null: | 1 | 1 | 1 |
| Assign c to BST->root | 1 | 1 | 1 |
| Else: | 1 | 1 | 1 |
| Course current = BST->root | 1 | 1 | 1 |
| While current isn’t null: | n | n | n |
| If c->ID is less than current->ID: | n | n | n |
| If current->leftNode is null: | 1 | 1 | 1 |
| Assign c to current->leftNode | 1 | 1 | 1 |
| Assign null to current | 1 | 1 | 1 |
| Else: | 1 | 1 | 1 |
| Assign current->leftNode to current | 1 | 1 | 1 |
| Else: | n | n | n |
| If current->rightNode is null: | 1 | 1 | 1 |
| Assign c to current->rightNode | 1 | 1 | 1 |
| Assign null to current | 1 | 1 | 1 |
| Else: | 1 | 1 | 1 |
| Assign current->rightNode to current | 1 | 1 | 1 |
| **Total Cost** | | | 3n + 14 |
| **Runtime** | | | O(n) |

}

**// Menu pseudocode**

FUNCTION menu(char args[]) {

BST courses

Bid bid

int choice = 0

While choice != 0:

Display menu options

Assign user input to choice

Switch choice:

Case 1:

courses = readCoursesFile(args[0])

Break

Case 2:

printAllCourses(courses)

Break

Case 3:

printCourseInformation(courses, args[1])

Break

Display “Good bye.”

}

FUNCTION menu(char args[]) {

Vector<Course> courses

Bid bid

int choice = 0

While choice != 0:

Display menu options

Assign user input to choice

Switch choice:

Case 1:

courses = readCoursesFile(args[0])

Break

Case 2:

printAllCourses(courses)

Break

Case 3:

printCourseInformation(courses, args[1])

Break

Display “Good bye.”

}

FUNCTION menu(char args[]) {

HashTable courses

Bid bid

int choice = 0

While choice != 0:

Display menu options

Assign user input to choice

Switch choice:

Case 1:

courses = readCoursesFile(args[0])

Break

Case 2:

printAllCourses(courses->root)

Break

Case 3:

printCourseInformation(courses->courses, args[1])

Break

Display “Good bye.”

}

## Evaluation

There are three data structures that we can use: vectors, hash tables and binary search trees. Vectors are a useful data structure for linear data storage and traversal. They are easy to append items to and remove items from the end of. This makes them ideal for abstract data types such as stacks, a FIFO (first-in, first-out) data structure. They are also useful for binary searches, but have the drawback of being less efficient for searching via other methods or for removing items towards the middle or start of the vector, due to the need to reassign all later elements to lower indices.

Hash tables can be useful for inserting and removing data, because there is little need to search for data in a properly-utilized hash table. An effective hashing method and table of requisite length makes for efficient data insertion or removal because it is a direct-access data type, that can quickly compute the location of data in a hash table, which is typically built as a vector or a linked list. Unlike basic vectors, vectors in hash tables do not need to be reorganized when an element is removed, because of the direct-access nature of hashing. The downside is that hash tables are, by nature, unorganized sets of data and thus cannot, and should not, be sorted.

Finally, binary search trees, when used with a properly-organized data set, can efficiently organize data. The binary tree structure allows for effective traversal compared to vectors and hash tables, though the binary nature also makes insertion and removal somewhat less efficient than vectors and hash tables. Binary search trees also suffer when created with sorted data, since the insertion algorithm will effectively turn the sorted data into an either ascending or descending linked list. Sufficient randomization of data is thus necessary before the creation of a binary search table to take optimal advantage of the organized structure.

In conclusion, based on the runtime and Big O analysis of each data type, and the consideration of the ABCU advisor’s requirements, vectors seems optimal. Looking at the requirements, it’s clear that not only do courses need to be accessed for their relation to one another, but also need to be sortable for printing alphabetically. The former requirement would make a hash table ideal, since using the course ID as a key would make lookup, insertion and removal incredibly quick. However, the latter requirement disqualifies hash tables, since sorting them would require circumventing the very data structure.

I initially considered binary search trees due to their efficient insertion, removal and traversal. However, sorting them alphabetically requires recursive method calls. And since insertion and removal may be used infrequently, it is not a necessary component to consider broadly. It may be better to use an algorithm that is more efficiently for frequently-used methods than methods that are infrequently-used.

Vectors would work very well for the sorting requirement, as a single operation could be performed to sort the data set and the nature of university course means that they are very rarely inserted or removed. Thus, the main drawback of the courses is largely avoided. This is why considering the intended usage of a program and the context behind it is important. While a vector without hashing may theoretically be less efficient for searching with a linear search algorithm, a binary search algorithm could provide similar results, reducing the worst-case runtime from n to where m is some value such that .

Thus, I will be using vectors as the data structure. And though this rationale may be short, it is also informative and succinct, both of which are ideal qualities for a developer.