//Menu

void printMenu() {

SET choice integer to -1

SET searchedCourse string

WHILE choice isn’t 9

PRINT “Menu:”

PRINT “1. Load courses”

PRINT “2. Print all courses”

PRINT “3. Search and print a course”

PRINT “9. Exit”

PRINT “Enter choice :”

GET choice from user

SET choice to choice integer

SWITCH(choice)

CASE 1:

CALL loadCourses()

CASE 2:

FOR all courses

CALL printCourse()

CASE 3:

GET course to search from user

SET course to search to searchedCourse

CALL searchCourse()

IF return value is empty

PRINT “no such course”

ELSE

CALL printAllCourses()

PRINT “Goodbye.”

}

//Vector

// Written with help of Stack Overflow

// https://stackoverflow.com/questions/4892680/sorting-a-vector-of-structs

// Question by calccrypto: https://stackoverflow.com/users/341683/calccrypto

// Answer by Oliver Charlesworth:

https://stackoverflow.com/users/129570/oliver-charlesworth

bool isLess(const Course& x, const Course& y) {

RETURN x.courseNumber < y.courseNumber

}

CALL sort on courses vector, using isLess as comp parameter

void printAllCourses(Vector<Course> courses) {

IF vector isn’t empty

CALL sort(courses.begin(), courses.end(), isLess)

FOR each course in vector

CALL printCourse(course)

}

//HashTable

// Written with help of Stack Overflow

// https://stackoverflow.com/questions/4892680/sorting-a-vector-of-structs

// Question by calccrypto: https://stackoverflow.com/users/341683/calccrypto

// Answer by Oliver Charlesworth:

https://stackoverflow.com/users/129570/oliver-charlesworth

bool isLess(const Course& x, const Course& y) {

RETURN x.courseNumber < y.courseNumber

}

// Copies hash table to vector and sorts vector

DECLARE empty vector of courses

void HashTable::toVector(Vector<Course> coursesFromHashTable) {

SET course

FOR each node in iteration

GET the course from the node

APPEND course to vector coursesFromHashTable

SET nextNode to next node

WHILE nextNode isn’t null

GET course from nextNode

APPEND course to vector coursesFromHashTable

SET nextNode to next node

CALL sort on vector coursesFromHashTable, using isLess as comp parameter

}

// Prints vector

void HashTable::printAllCourses(Vector<Course> coursesFromHashTable)) {

IF vector isn’t empty

CALL sort(coursesFromHashTable.begin(), coursesFromHashTable.end(), isLess)

FOR each course in vector

CALL printCourse(course)

}

//BinarySearchTree

void BinarySearchTree::preOrder(Node\* node) {

IF the node isn’t null

CALL printCourse(course)

CALL preOrder, passing in the node’s left child

CALL preOrder, passing in the node’s right child

}

// Call preOrder() to print all nodes in alphanumeric order

Void BinarySearchTree:printAllCourses() {

CALL preOrder(root) function

}

Runtime Analysis:

Vector

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Print “Loading file” | 1 | 1 | 1 |
| For each line in file | 1 | n | n |
| Set course | 1 | 1 | 1 |
| Set prerequisite vector | 1 | 1 | 1 |
| If at least 2 parameters | 1 | 1 | 1 |
| Get course number | 1 | 1 | 1 |
| Get course name | 1 | 1 | 1 |
| For all prerequisites in line | 1 | n | n |
| Get prereq course number | 1 | 1 | 1 |
| Call searchCourse with course number | 1 | O(n) | O(n) |
| IF prereq course search is empty | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| SET string with course number | 1 | 1 | 1 |
| APPEND string to course prereq vector | 1 | 1 | 1 |
| SET course with course information | 1 | 1 | 1 |
| APPEND course to courses | 1 | 1 | 1 |
| **Total Cost** | | | N3+7N2+8N+1 |
| **Runtime** | | | O(N3) |

Hash Table

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Print “Loading file” | 1 | 1 | 1 |
| For each line in file | 1 | n | n |
| Set course | 1 | 1 | 1 |
| Set prerequisite vector | 1 | 1 | 1 |
| If at least 2 parameters | 1 | 1 | 1 |
| Get course number | 1 | 1 | 1 |
| Get course name | 1 | 1 | 1 |
| For all prerequisites in line | 1 | n | n |
| Get prereq course number | 1 | 1 | 1 |
| Call searchCourse with course number | 1 | n | n |
| IF prereq course search is empty | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| SET string with course number | 1 | 1 | 1 |
| APPEND string to course prereq vector | 1 | 1 | 1 |
| SET course with course information | 1 | 1 | 1 |
| APPEND course to courses | 1 | 1 | 1 |
| **Total Cost** | | | N3+7N2+8N+1 |
| **Runtime** | | | O(N3) |

BST

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Print “Loading file” | 1 | 1 | 1 |
| For each line in file | 1 | n | n |
| Set course | 1 | 1 | 1 |
| Set prerequisite vector | 1 | 1 | 1 |
| If at least 2 parameters | 1 | 1 | 1 |
| Get course number | 1 | 1 | 1 |
| Get course name | 1 | 1 | 1 |
| For all prerequisites in line | 1 | n | n |
| Get prereq course number | 1 | 1 | 1 |
| Call searchCourse with course number | 1 | Log2n | Log2n |
| IF prereq course search is empty | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| SET string with course number | 1 | 1 | 1 |
| APPEND string to course prereq vector | 1 | 1 | 1 |
| SET course with course information | 1 | 1 | 1 |
| CALL insert function with course | 1 | Log2n | Log2n |
| **Total Cost** | | | 2N2log2(N) +6N2+8N+2 |
| **Runtime** | | | O(N2) |

Comparisons and Recommendations:

A vector is particularly useful because it has built-in methods, such as sort() and swap(), making vectors easier to use than a hash table or a binary search tree. Vectors can also quickly insert or remove an element at the beginning or the end of the vector, but inserting or removing an element from the middle of the vector takes O(N) time (Miller et al). Searching for an element also takes O(N) time. If a wanted element is at the end of the vector, all elements will need to be visited to find that element.

Inserting into, deleting from, and searching for an element in a hash table can take O(1) time, which is the main advantage to using a hash table (Vahid et al, 2019, Section 5.1). Using a hash table will be faster than a vector, but a hash table doesn’t have the built-in methods that vectors do. From when writing the pseudocode to print the courses in alphanumeric order, I found it difficult to do within the hash table structure. I ended up copying all the courses to a vector and then sorted the vector.

A binary search tree is the most complex data structure when compared to hash tables and vectors, which I would say is its biggest disadvantage. When writing the pseudocode, I had the most trouble following the logic for a binary search tree. Another disadvantage is, if nodes are already in order when inserted, the tree will be the maximum height. Removal insertion algorithms all have a worst-case runtime of O(N), which isn’t ideal, but best-case runtime for each is O(logN) (Vahid et al, 2019, Sections 6.5-6.6). To visit all the nodes in a binary search tree, we can use inorder, preorder, and postorder traversal methods. This means that it’s easy to output all the nodes in alphanumeric order, assuming the nodes were inserted using the course number.

My recommendation is to use a binary search tree, particularly if a major concern is runtime complexity. Hash tables have the quickest best-case runtime complexity, but printing all the elements in alphanumeric order was difficult. Looking at the runtime analysis on the previous pages, the binary search tree also had the fastest runtime complexity of the tree structures when loading the file and creating the data structure. If the biggest concern is ease of use, I would instead recommend using a vector.

References

Miller, B. N., Ranum, D. L., Luther College, Pearce, J. L., & Berea College. (n.d.). *2.6. Analysis of Array and Vector Operators*. Runestone Academy. https://runestone.academy/ns/books/published/cppds/AlgorithmAnalysis/VectorAnalysis.html

Vahid, F., Lysecky S., Wheatland, N., & Siu, R. (2019). *CS 300: Data Structures and Algorithms*. Zyante Inc. https://learn.zybooks.com/zybook/CS-300-18508.202486-1