**Pseudocode**

FUNCTION menu(coursesFile)

ASSIGN choice as an integer equal to 0

ASSIGN coursesLoaded as Boolean equal to False

WHILE choice does NOT equal 9

OUTPUT “Menu:”

OUTPUT “1. Load courses”

OUTPUT “2. Print alphabetically sorted list of all courses”

OUTPUT “3. Print a specific course title and prerequisites”

OUTPUT “9. Exit”

TRY

READ INPUT as choice

IF choice equals 1 THEN

loadCourses(coursesFile)

coursesLoaded equals True

IF choice equals 2 THEN

IF NOT coursesLoaded THEN

OUTPUT “Please load courses first!”

ELSE

printSortedCourses()

END IF

IF choice equals 3 THEN

IF NOT coursesLoaded THEN

OUTPUT “Please load courses first!”

ELSE

searchCourses(struct, courseNumber) // “struct” varies depending on implementation

END IF

CATCH invalid input // anything except ints 1-3, 9

OUTPUT “Invalid input”

END TRY

END WHILE // Ends when choice equals 9

OUTPUT “Closing program”

END FUNCTION

FUNCTIONS printSortedCourses() // A group of possible funcs, depending on choice

// For sorting a BST, inOrder works because of BST rules

FUNCTION printSortedCourses()

ASSIGN new node to access tree

IF node is NOT null THEN

CALL inOrder(node->left) // recursive to left

OUTPUT course information

CALL inOrder(node->right) // recursive to right

END IF

END FUNCTION

// For sorting a vector, sort() works well

FUNCTION printSortedCourses()

// make sure vector is accessible

CALL sort(coursesVector.begin(), coursesVector,end())

FOR each course IN coursesVector

OUTPUT course information

END FOR

END FUNCTION

// Hash tables are trickier to sort, but it’s simple to use a temp vector with sort()

FUNCTION printSortedCourses()

// make sure hashTable is accessible

// make temporary vector

INITIALIZE tempVector as empty vector of Course objects

FOR each bucket in hashTable

IF bucket is not empty THEN

APPEND bucket’s Course to tempVector

END IF

END FOR

CALL sort(tempVector.begin(), tempVector.end())

FOR each course IN coursesVector

OUTPUT course information

END FOR

END FUNCTION

END FUNCTIONS

**Evaluation**

**Worst Case Runtimes**

Each of the three structures has a similar main() function that handles reading and inserting the courses. They differ in their exact insertion methods, which are considered as separate calculations. Each **file reading runtime is O(n),** and the additional runtimes are considered below.

The runtimes were calculated by giving each “normal” statement a value of 1 and any loops a value of “n”. Because constants can be ignored, as Big O runtimes are approximations, many of the calculations ended up as O(n). The built-in function sort() was researched to determine its runtime.

***Vector***

The worst-case runtime for the vector involves traversing the vector in the searchCourses() function while the sought-after course is either the last element or isn’t found. This would result in the “for” loop iterating over every item in the list, executing the maximum number of times.

Calculated worst case runtimes:

* **SEARCH: O(n)**, linear time, given its linear search function.
* **SORT: O(n log(n))**, quasilinear time, per its standard comparison functionality (Digital Ocean, 2022).
* **PUSH\_BACK:** **O(1)**, constant time, because the end of the vector is always immediately available (CPP Reference, n.d.).

***Hash Table***

The hash table’s worst-case runtime involves every course being mapped to the same bucket, resulting in the maximum number of collisions, and searchCourses() having to search through the entire chained list of the bucket until the course is found at the last spot or isn’t found at all.

If a course is found right away as the first (or only) item in its bucket, then searching for a course would have O(1), or constant, runtime. This would be the best case. As for the worst case, searching a linked list sequentially is a linear search, so it has the same runtime complexity as the vector.

Calculated worst case runtimes:

* **SEARCH: O(n)**, linear time.
* **SORT: O(n log(n))**, quasilinear time, because a temporary vector is used with the sort() function.
* **INSERT: O(n)**, linear time, thanks to the presence of a “while” loop.

***Binary Search Tree***

The BST’s worst-case runtime would involve searching the entire tree in searchCourses() for a course at the deepest depth or a course that doesn’t exist. Further, if the BST is unbalanced, this can make its height (+ 1) equal to ‘n’ should all input favor either the left *or* right side of the tree.

Calculated worst case runtimes:

* **SEARCH: O(n)**, linear time, because an unbalanced BST would turn searchCourses() into a linear search down one long branch, and because the “+ 1” from calculating ‘n’ as its height (which starts at 0) can be ignored (Geeks for Geeks, 2024).
* **SORT: O(n)**, linear time, happens at insert/addNode.
* **INSERT/ADD\_NODE: O(n)**, linear time, for the same reasoning as the search runtime.

**Advantages and Disadvantages**

***Vector***

The vector is an easily implemented structure thanks to its built-in Visual Studio functionality. It’s easy to sort elements with the sort() function without writing extra code. Further, vectors are beginner-friendly, since they’re like arrays or lists, so using them is straightforward.

The vector is less efficient than the hash table or BST because it has a strictly linear structure. Courses are pushed back to it without order, and finding a specific course involves linearly searching the elements using a loop.

***Hash Table***

A hash table offers faster searching than a vector because of its key-sorted structure. When calling searchCourses(), the course can be found right away by using its key. Assuming the table has no collisions, and therefore no chained lists, each course can be found without further traversal by calculating the course’s key (i.e., it’s index).

Despite this advantage, extra measures must be taken to account for collisions, which is where things get slower. The chaining method involves linked lists as well as steps for searching both buckets and their associated linked lists, which turns into linear searching. Additionally, a hash table must be sorted using an additional means, such as a temporary vector or a sorting algorithm.

***Binary Search Tree***

Although a vector has the built-in ease of its sort() function, a BST has the advantage of sorted order from its conception. BSTs have rules for their order, and it’s easy to insert data into one per these rules by using the mathematical operators ‘<’ and ‘>’. Displaying the contents of a BST is, then, just a matter of using a simple inOrder() function. Thanks to the conditional statements in searchCourses() and the BST’s ordered structure, traversing a balanced tree means only traveling down its height, not the entire length of the combined elements (n).

The advantages of using a BST assume that it’s balanced, however. Just like using a hash table can produce many collisions and long linked lists, a BST can become unbalanced and unwieldy by having many values inserted that favor either the left *or* right side more. Taking this possibility into account requires extra forethought and code to mitigate it.

**Recommendation**

Each structure’s main() and searchCourses() functions have worst-case runtimes of O(n), so the other calculated runtimes can be considered in the recommendation decision.

Thanks to its ordered nature from the start (i.e., from loading the courses), **the BST implementation is recommended for use.** The BST does not need to be separately ordered, and a well-balanced BST offers more efficient searching than a vector or collision-heavy hash table, since both use linear search while the BST might only have to search up to its height.

That said, it’s likely that the hash table for *this* implementation would not produce collisions—each course number is unique, whether only the digits substring is used or the entire string is converted to an integer value. This means a hash table would have a faster lookup time than a BST, even with the BST’s height-instead-of-n advantage. The recommendation comes down to the BST’s advantage of a faster sorting method (i.e., sorted during insertion), especially because sorting is a dedicated menu option.

References

CPP Reference. (n.d.). *std::vector<T,Allocator>::push\_back.* Retrieved February 13, 2025, from <https://en.cppreference.com/w/cpp/container/vector/push_back>

Digital Ocean. (2022, August 3). *Using sort() in C++ std library.* <https://www.digitalocean.com/community/tutorials/sort-in-c-plus-plus>

Geeks for Geeks. (2024, September 24). *Insertion in binary search tree (BST).* <https://www.geeksforgeeks.org/insertion-in-binary-search-tree/>