**Runtime Analysis:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Vector** | | | |
| Code | Cost | Times | Total |
| Open file | 1 | 1 | 1 |
| While loop | 1 | n | n |
| Split line | 1 | n | n |
| Validation checks | 1 | n | n |
| Check for duplicates | n | n | n² |
| Add course to vector | 1 | n | n |
|  | | | |
| Total: | | | O(n²) due to duplicate checking loop |
| Memory: | | | O(n) - linear space for n courses |

|  |  |  |  |
| --- | --- | --- | --- |
| **Hash Table** | | | |
| Code | Cost | Times | Total |
| Open file (first pass) | 1 | 1 | 1 |
| While loop (first pass) | 1 | n | n |
| Split and validate (first pass) | 1 | n | 1 |
| Reset file | 1 | 1 | n |
| While loop (second pass) | 1 | n | n |
| Hash function | 1 | n | n |
| Insert with collision resolution | 1 | n | n |
|  | | | |
| Total: | | | O(n) - two passes but still linear |
| Memory: | | | O(n) - space for n courses plus hash table array |

|  |  |  |  |
| --- | --- | --- | --- |
| **Binary Tree** | | | |
| Code | Cost | Times | Total |
| Open file | 1 | 1 | 1 |
| While loop | 1 | n | n |
| Split and validate | 1 | n | n |
| Insert into BST | Log n | n | n Log n |
| Validate prerequisites | Log n | n | n Log n |
|  | | | |
| Total: | | | O(n log n) |
| Memory: | | | O(n²) - space for n nodes |

**Evaluation**

**VECTOR:**

So, looking at the vector implementation, it's basically the simplest way to store our course data - kind of like a big list that can grow as needed. The main issue is that when we're loading courses, we must check each new course against all the ones we already have to make sure there aren't any duplicates. This gives us an O(n²) runtime, which honestly gets slow when you have a lot of courses. While it's nice that everything is stored together in memory (good for speed), searching for a specific course means we must go through the entire list one by one. It's like trying to find a specific textbook by checking every single book in the library - not very efficient! The memory usage is decent since it only stores what we need, but the slow search and loading times make it impractical for what we're trying to do.

**HASH TABLE:**

The hash table is like having a well-organized filing cabinet. It uses a two-step process to load the courses - first checking what's valid, then actually storing everything. What's cool about this is that even though we go through the data twice, it's still way faster than the vector approach, with O(n) runtime. The best part is how fast we can find courses - it's almost instant, like knowing exactly which drawer and folder to check in that filing cabinet. Yeah, we must deal with some technical stuff like collision handling (when two courses try to go in the same spot), but the super-fast lookup speed makes it worth it. Think of it as having an efficient library catalog system where you can find any book instantly instead of searching shelf by shelf.

**BINARY SEARCH TREE:**

The BST is kind of like a family tree for our courses, with each course pointing to two others. It takes O(n log n) time to load everything, which isn't as fast as the hash table but still better than the vector. Finding courses is quick using O(log n) because we can eliminate half the possibilities each time we look, kind of like playing a number guessing game where you can ask higher or lower. The cool thing about BST is that it keeps everything in order automatically, which is nice for printing out sorted course lists. The downside is that if we're not careful about how we add courses, the tree can get lopsided and slow things down.

**RECOMMENDATION:**

After looking at all three options, I'm going with the hash table for this project.

First off, it's FAST, really fast. When you need to look up a course, it takes basically the same time whether you're looking through 10 courses or 1000. That's huge for an advisory system where you'll be looking up courses all the time. While it's a bit more complicated to code than just using a vector, the speed boost is totally worth it.

The loading process might seem weird with its two-pass system, but it's smart, it’s safe, and it makes sure all our prerequisites are valid without slowing things down too much. Plus, course numbers make perfect keys for a hash table (like CS101, MATH201, etc.), so it just makes sense.

Yeah, the BST would work too, but the hash table is going to be faster for what we need. The vector might be simpler to code, but it would be painfully slow with a real, full-size course catalog. Nobody wants to wait around while the program checks every single course one by one, that would take too long.

I know implementing a hash table means dealing with some more complex stuff like collision handling, but when we think about what this system needs to do, mainly quick course lookups, it’s the best choice. It's like choosing between taking the elevator (hash table), taking the stairs (BST), or checking every single room in the building one by one (vector) when you're trying to find someone. The elevator might be more complicated than the stairs, but it's going to get you there faster!

The memory usage is reasonable too; we're not wasting a ton of space, and the performance benefits make any extra memory worth it. For a real-world academic advisory system, the hash table is the way to go based on the data in my opinion.