**Project One Runtime and Summary**

**Vector Runtime**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Open file** | 1 | 1 | 1 |
| **Read the lines** | 1 | n | n |
| **Parse the lines** | 1 | 1 | 1 |
| **Make course object** | 1 | n | n |
| **Add them to the vector** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

**Hash Table Runtime**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Open file** | 1 | 1 | 1 |
| **Read the lines** | 1 | n | n |
| **Parse the lines** | 1 | n | n |
| **Make course object** | 1 | n | n |
| **Add them to the vector** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(1) |

**Binary Search Tree Runtime**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Open file** | 1 | 1 | 1 |
| **Read the lines** | 1 | n | n |
| **Parse the lines** | 1 | 1 | 1 |
| **Make course object** | 1 | n | n |
| **Add them to the vector** | 1 | n | N |
| **Add course to Tree** | Log n | n | n log n |
| **Total Cost** | | | 3n + nLogn+1 |
| **Runtime** | | | O(n log n) |

The first advantage that comes to mind using a vector list, is the simplicity, they are easy to use and very adaptable. Scalability is important when it comes to courses, as new ones come along and old ones get removed, a vector list will scale dynamically. However, while searching a vector, each element is visited and this can slow things down, especially with a huge set of data. On top of that, vectors need a separate sorting step before the courses can be shown alphabetically.

Hash tables have speed on their side, especially for an application like this. Large amounts of data can be handled efficiently using key value pairs to locate specific courses instantly. This also means that no matter how much the course catalog grows, the hash table will grab the information within the same time frame. Given how fast this method is, it is also more difficult to implement. Collisions have to be delt with appropriately, which is tricky, and if anything is amiss it can wreak havoc on performance and stored data.

Binary search trees stand out with their ease of access to data. This comes from the sorted order that this method automatically uses. This, in turn, adds to the efficiency of the system which can quickly display the stored course information in alphanumeric order. Given the way it stores data, searching for courses is fast and when adding or removing courses the tree automatically updates. Given these traits, a BST also has to be well balanced to stay efficient, requiring monitoring and maintenance. Pointers have to be used in each node to link the system together and this requires more memory.

My recommendation is a Binary Search Tree. A BST naturally keeps courses sorted, making it easy to display them in order. They're great when it comes to finding specific courses quickly. They do need a bit more memory for the node pointers and keeping them balanced requires some maintenance. But I think the tradeoff is worth the efficiency. Fast searches, automatic sorting, and the ability to add and remove information easily makes this method stand out. BSTs seem like the logical choice for a dynamic course catalog that can be accessed easily and updated regularly.