**Runtime Analysis Chart**

| **Operation** | **Vector** | **Hash Table** | **Tree** |
| --- | --- | --- | --- |
| **Reading and Parsing File** |  |  |  |
| Read line | 1 | 1 | 1 |
| Validate Line | 1 | 1 | 1 |
| Parse Line to Object | 1 | 1 | 1 |
| Add to Data Structure | O(1) amortized, O(n) worst-case for resizing | O(1) average, O(n) worst-case for rehashing | O(log n) |
| Total Cost for n Lines | O(n) + O(n) for potential resizing | O(n) + overhead for hashing | O(n log n) |
| Runtime | O(n) to O(n^2) | O(n) | O(n log n) |

**Memory Considerations**

- **Vector**: Efficient for contiguous allocations, resizing overhead.

- **Hash Table**: Higher due to keys, values, and collision management structures.

- **Tree**: Additional memory for node links but efficiently organized.

**Summary and Recommendation**

- **Vector**: Simple and efficient for sequential operations but has potential resizing costs.

- **Hash Table**: Fastest for direct access but lacks inherent ordering, requiring extra steps for sorting.

- **Tree**: Balances between insertion/search efficiency and maintains an ordered state, ideal for sorted displays and direct access.

Given the balance between efficiency for data insertion, searching, and the requirement for maintaining data in a sorted order, a balanced tree structure is recommended. It optimally supports loading and organizing course information, facilitating efficient access and display of courses in alphanumeric order, which aligns with the project requirements.