**Costs That Persist for Each Data Structure**

For each course:

1. **Open File**  
     - Cost: 1 executed once  
     - Overall: O(1)
2. **Read a Line**  
     - Cost: 1 per line  
     - Executed n times -- Total cost: n
3. **Split the Line into Tokens**  
     - Cost: Assuming a constant cost c per line with a fixed, small number of tokens  
     - Executed n times -- Total cost: c\*n = O(n)
4. **Validate Tokens**  
     - Cost: 1 per line  
     - Executed n times -- Total cost: n
5. **Create a Course Object**  
     - Cost: 1 per object  
     - Executed n times - Total cost: n
6. **Insert the Course Object into the Data Structure**  
     - The cost for insertion depends on the data structure.

**1. Vector Data Structure**

* **Insertion Cost:**  
    - Using push\_back on a dynamic array is O(1) time per insertion.  
    - Total insertion cost for n courses: O(n)
* **Overall File-Processing Time:**  
    - Reading splitting validating and inserting: O(n) + O(n) + O(n) + O(n) = O(n)
* **Memory Usage:**  
    - A vector stores the n course objects in a contiguous block of memory.  
    - Memory usage is O(n) per element.
* **Advantages:**  
    - Simple to implement and fast for adding items.  
    - Low per-item overhead and efficient cache utilization during iteration.
* **Disadvantages:**  
    - Does not maintain sorted order naturally. Requires an O(n log n) sort when printing courses in order.  
    - Searching is O(n) runtime.

**2. Hash Table Data Structure**

* **Insertion Cost:**  
    - Average-case insertion is O(1) per course.  
    - Total insertion cost for n courses: O(n) on average  
    - In the worst-case insertion could degrade to O(n) per course.
* **Overall File-Processing Time:**  
    - Similar to the vector: O(n) for reading, splitting, validating, and inserting on average.
* **Memory Usage:**  
    - Hash tables typically require extra space for buckets and may allocate more space than necessary.  
    - Memory usage is O(n).
* **Advantages:**  
    - Fast average-case lookups O(1) for search operations.  
    - Efficient insertion when using a well-designed hash function.
* **Disadvantages:**  
    - Does not maintain an order. Printing the courses in alphanumeric order requires extracting keys and sorting them this is O(n log n) extra time.  
    - Higher memory overhead due to additional data structure components.

**3. Binary Search Tree (BST) Structure**

* **Insertion Cost:**  
    - Each insertion is O(log n) on average.  
    - Total insertion cost for n courses: O(n log n)  
    - Worst-case*:* Insertion can be O(n) per course (total O(n^2)).
* **Overall File-Processing Time:**  
    - Reading, splitting, and validating remain O(n).  
    - Insertion adds an extra O(n log n) time, making the overall process O(n log n) on average.
* **Memory Usage:**  
    - Each course is stored in a node with pointers to child nodes.  
    - Memory usage is O(n) with additional overhead for pointers in each node.
* **Advantages:**  
    - Naturally maintains sorted order; an in-order traversal yields the courses in alphanumeric order with O(n) time.  
    - Efficient search (O(log n)) and insertion if balanced.
* **Disadvantages:**  
    - Insertion is slower compared to vectors or hash tables (O(log n) vs. O(1) average).  
    - Requires additional memory overhead per node due to pointers.  
    - If not balanced, performance can degrade to O(n) per operation.

**Recommendation**

Based on the analysis:

* **Vector:**  
    Pros: Simple, fast O(n) file reading and insertion.  
    Cons: Lacks inherent order requires an extra sort step (O(n log n)) for ordered display. Linear search (O(n)) if not sorted.
* **Hash Table:**  
    Pros: Fast average-case lookups and insertion (O(1)). Efficient for search operations.  
    Cons: Does not preserve order, so printing a sorted list needs additional processing (extraction and sort, O(n log n)). Extra memory overhead for buckets.
* **BST:**  
    Pros: Naturally maintains sorted order via in-order traversal (O(n) for printing) and efficient search (O(log n)).  
    Cons: Insertion is slightly more expensive (O(log n) per course, total O(n log n)). Higher per-node memory overhead. Worst-case performance concerns if not balanced.

Given that the advising program requires printing the courses in alphanumeric order and efficient course lookup for a specific course, the **BST** is recommended. Despite the higher cost during insertion, its ability to provide sorted data without extra sorting steps and maintain efficient search at O(log n) time complexity makes it well-suited for the advisor’s requirements. If the number of courses (n) is moderate and a self-balancing tree is implemented, the O(n log n) cost is ok compared to the added overhead of sorting when using a vector or hash table.

I recommend using a self-balancing BST for this project because it naturally maintains order, allowing an in-order traversal to print courses in alphanumeric order and provides efficient search capabilities, aligning well with the functional and performance requirements of the advising program.