Emily Vasquez

CS 300

Analysis of the run-time and memory for the data structures.

After analyzing the three data structures, each has strengths and weaknesses depending on the task. Vectors are the most straightforward to implement and work well for smaller datasets. However, they are inefficient for searching and sorting because they require scanning through each item linearly and sorting the entire list every time the data needs to be displayed in order. Hash tables offer very fast search and insert times on average, making them ideal for quickly accessing course information. The downside is that they do not maintain any inherent order, so printing the course list alphabetically requires extra steps, such as copying the data to another structure and sorting it manually.

Binary Search Trees provide the most balanced solution. When the tree is kept reasonably balanced, they allow efficient searches and insertions in logarithmic time and naturally keep the data in sorted order. This means that printing the course list in alphanumeric order can be done quickly through an in-order traversal, without needing to sort the data again. Although BSTs are more complex to implement compared to vectors or hash tables, they provide strong performance across all requirements, making them the most effective choice for this advising system.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Data Structure** | **Load Time** | **Search Time** | **Sorted Print Time** | **Advantages** | **Disadvantages** |
| **Vector** | O(n) | O(n) | O(n log n) (need to sort before printing) | - Very easy to implement and debug  - Simple to understand and use  - Insertion at end is efficient  - Ideal for small datasets | - Search is linear — not good for large data  - Must sort the list every time before printing in order  - No efficient way to keep order |
| **Hash Table** | O(n) | O(1) average  O(n) worst (with collisions) | O(n log n) (requires copying and sorting) | - Insertion and search are very fast in most cases  - Efficient for large datasets  - Search time doesn't grow much with more data (avg case) | - Doesn’t store elements in order  - Needs an extra data structure (e.g., vector) and sorting step to print in order  - Can use more memory |
| **Binary Search Tree (BST)** | O(n log n) (if balanced)  O(n²) (if unbalanced) | O(log n) (balanced)  O(n) (worst case) | O(n) (uses in-order traversal to print in order) | - Keeps data sorted automatically  - Fast search, insert, and print if tree is balanced  - No extra sorting step required for printing in order | - Implementation is more complex  - Requires tree balancing (AVL/Red-Black Tree) for optimal performance  - Risk of poor performance if unbalanced |

|  |  |  |  |
| --- | --- | --- | --- |
| **Requirement** | **Vector** | **Hash Table** | **Binary Search Tree (BST)** |
| **1. Load File & Create Course Objects** | - Append each course to end of vector  - Very simple code  - Great for small files | - Hash each course’s ID and insert into hash table  - Need to handle collisions | - Insert each course by comparing course numbers  - More logic needed, especially for balancing |
| **2. Search Course by ID** | - Have to loop through every item  - Gets slow as list grows | - Fast (O(1)) if few collisions  - Performance is consistent for large data (avg case) | - Very efficient (O(log n)) if tree is balanced  - Degrades to O(n) if tree becomes skewed |
| **3. Print All Courses in Alphanumeric Order** | - Need to sort the entire vector every time (O(n log n)) | - Need to copy all items into a vector or list, then sort (extra step and memory) | - Just do in-order traversal of tree  - Naturally prints everything sorted (O(n)) |
| **4. Display Course Info with Prerequisites** | - Requires finding course by ID (O(n)), then printing info | - Find course fast (O(1)), print info | - Search for course (O(log n)), print info |
| **5. Memory Efficiency** | - Moderate memory usage  - No extra pointers | - More memory needed (hash buckets + possible chains or probing structures) | - Moderate memory  - Every node needs two pointers (left/right), which adds up for large trees |
| **6. Implementation Difficulty** | - Easiest to implement  - Minimal bugs likely | - Medium difficulty  - Need to create hash function and collision handling | - Most complex  - Need recursive logic and possibly self-balancing logic |