## **Pseudocode Summary**

STRUCT Course  
 STRING number, title  
 LIST<STRING> prereqs  
END STRUCT

**Menu**

MAIN()  
 choice ← prompt("1=Vector 2=Hash 3=Tree")  
 loaded ← FALSE  
 WHILE TRUE  
 print("1-Load 2-List 3-Find 9-Exit")  
 opt ← readInt()  
 IF opt=9 THEN BREAK  
 IF opt=1 THEN loaded←Load(choice)  
 ELSE IF NOT loaded THEN print("Load data first")  
 ELSE IF opt=2 THEN PrintAll(choice)  
 ELSE IF opt=3 THEN PrintOne(choice,prompt("Course?"))  
 END WHILE  
END MAIN

**Vector**

Load\_Vector(): open file → read each line → parse → validate → append to VECTOR  
PrintAll\_Vector(): sort VECTOR by number → print  
PrintOne\_Vector(k): linear search → print if found

**Hash Table**

Load\_Hash(): open file → parse → validate → insert Course into HASH[key]  
PrintAll\_Hash(): sort HASH keys → print in order  
PrintOne\_Hash(k): print HASH[k] if exists

**Binary Search Tree**

Load\_Tree(): open file → parse → validate → BSTInsert(root,key,Course)  
BSTInsert(t,k,v): standard recursive insert  
PrintAll\_Tree(): in-order traversal (sorted)  
PrintOne\_Tree(k): BSTFind(root,k) → print if found

**PrintCourse**

print(number + ", " + title)  
IF prereqs empty → print("Prerequisites: None")  
ELSE → print("Prerequisites: " + join(prereqs,", "))

## **Runtime Analysis**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Structure** | **Per Insert** | **Total Time** | **Memory** | **Advantages** | **Disadvantages** |
| **Vector** | O(1) | **O(n)** | Low | Simple; minimal overhead | Lookup O(n); must re-sort |
| **Hash Table** | O(1) avg | **O(n)** avg | Moderate | Fast lookups; easy inserts | No order; collisions; extra memory |
| **BST** | O(log n) avg | **O(n log n)** avg | Moderate | Sorted order; fast lookups | Can degrade to O(n) if unbalanced |

## **Evaluation and Recommendation**

All three structures meet project goals of loading, searching, and listing courses.

* **Vector:** Best for simplicity and small datasets but inefficient for frequent searches.
* **Hash Table:** Excellent average speed O(1) but lacks ordering, requiring an extra sort for listing.
* **Binary Search Tree:** Naturally sorted with O(log n) average lookups and O(n) traversal for listing. Its performance gracefully supports both advisor requests.

**Recommended Structure:** The **Binary Search Tree** provides the best balance—efficient searches, ordered output without extra sorting, and strong scalability—making it the ideal choice for ABCU’s advising system.