Southern New Hampshire University

CS 300 Project One
GCZ79
02/15/2026

Index

Big O notation measures how performance changes as data grows. For ABCU's ~150 courses, we analyze worst-case time complexity for each operation.

Runtime Analysis Table

Operation	Vector	Hash Table	Binary Search Tree
loadDataStructure	$O(n^2)$	$O(n^2)$	$O(n^2)$
printCourseList	O(n log n)	O(n log n)	O(n)
printCourse (search)	O(n)	O(1) average	O(log n) average
Memory Usage	O(n)	O(n)	O(n)

Notes:

- n = ~150
- Load operations: worst-case complexity
- Search operations: average-case complexity shown because worst-case (O(n)) is identical for all three structures; average case better illustrates practical performance differences

Detailed Analysis

Vector Implementation

■ Advantages:

Simple to implement and understand

Easy to iterate through all elements

■ Disadvantages:

Sorting required before printing ($O(n \log n)$)

Linear search for course lookup ($O(n)$)

Slow for frequent searches with 150+ courses

Must iterate the entire structure for prerequisite validation

Big-O Analysis

loadDataStructure: $O(n^2)$ worst-case

- Two-pass approach required because the original data is unsorted
- First pass: Read n courses → $O(n)$
- Second pass: Validate each course's prerequisites by searching through all n courses → $O(n^2)$
- Nested loops dominate: n courses × average 2 prerequisites × n searches = $O(n^2)$

printCourseList: $O(n \log n)$ worst-case

- Must sort vector using mergesort (guaranteed $O(n \log n)$ worst-case)
- Then iterate to print → $O(n)$
- Sorting dominates the complexity

printCourse: $O(n)$ worst-case

- Linear search through an unsorted vector
- May need to check all n courses if target is last or not found

Hash Table Implementation

■ Advantages:	■ Disadvantages:
Fast average-case lookup (O(1)) for finding specific courses	Potential collisions requiring chaining
No sorting required for insertion	Not naturally sorted (requires collecting all courses into a vector for printing)
Efficient for the "Find Course" requirement	Worst-case lookup degrades to O(n) with a poor hash function or many collisions
	Must still validate prerequisites during load, creating $O(n^2)$ complexity

Big-O Analysis:

loadDataStructure: $O(n^2)$ worst-case

- Same prerequisite validation problem as vector
- Even with O(1) average hash insertion, prerequisite validation requires searching

printCourseList: O(n log n)

- Must collect all n courses from the hash table → O(n)
- Sort collected courses → O(n log n)
- Print sorted list → O(n)
- Sorting dominates

printCourse: O(1) average case, O(n) worst-case

- Average: Direct hash lookup
- Worst: All courses hash to the same bucket (complete collision)

Binary Search Tree Implementation

■ <u>Advantages</u>:

Naturally sorted through in-order traversal (no separate sorting step needed)

Fast search (O(log n) average case) for 150+ courses

Efficient insertion while maintaining order

Directly addresses both main requirements without additional sorting

■ <u>Disadvantages</u>:

Can degrade to O(n) if the tree becomes unbalanced (worst-case with sorted input)

More complex implementation than a vector

Still requires $O(n^2)$ for prerequisite validation during load

Big-O Analysis:

loadDataStructure: $O(n^2)$ worst-case

- Prerequisite validation dominates: $O(n^2)$
- BST insertion for n courses: O(n log n) average, $O(n^2)$ worst-case if unbalanced
- Overall complexity is $O(n^2)$ (prerequisite validation is the dominant term)

printCourseList: O(n)

- Simple in-order traversal of the tree
- No sorting step required (already sorted)
- Most efficient of the three structures for this operation

printCourse: O(log n) average case, O(n) worst-case

- Average: Balanced tree search with ~150 courses = ~7-8 comparisons
- Worst: Degenerate tree becomes a linked list

Recommendations

Recommended Data Structure: Binary Search Tree

Justification

Performance and Ordering Advantages

The Binary Search Tree performs best for the two primary operations required by the system. It maintains courses in alphanumeric order during insertion, eliminating the need for a separate sorting step. Printing the full course list uses an in-order traversal in O(n) time with sorted output, compared to O(n log n) sorting required by vector and hash table. Searching for a specific course runs in O(log n) average time, significantly more efficient than the O(n) linear search required by a vector. Memory usage remains O(n), consistent with the other structures.

Appropriate for the Problem

The BST handles unsorted input naturally by organizing elements during insertion, ensuring that the structure remains ordered without extra processing. It supports efficient full catalog printing and individual course lookup.

Scalability

With approximately 150 courses, BST search (O(log n)) is significantly faster than vector's linear search (O(n)). As the dataset grows, the logarithmic behavior provides increasing performance benefits, making the BST more suitable for larger course catalogs.

Trade-offs

All three structures share $O(n^2)$ load complexity due to prerequisite validation using nested searches. The BST's advantages appear during normal operation, where searching and printing dominate runtime. Although an unbalanced BST can degrade to O(n), this risk is acceptable for the expected dataset size.

Practical Use

Course catalogs naturally benefit from sorted organization and frequent lookups. Advising workflows typically involve repeated course searches combined with periodic full catalog printing, both of which align well with BST behavior.

Conclusion

Because it provides O(n) sorted traversal and O(log n) average search performance while matching the operational needs of the system, the Binary Search Tree is the most appropriate data structure despite sharing the same $O(n^2)$ loading cost as the other options.