**RUNTIME EVALUATION (Worst Case Big-O for Loading Only)**

(Opening file, parsing each line, creating Course objects, validating prerequisites) Assume: n courses, p total prerequisite references across the file.

Vector (with courseIndex map used in your pseudocode)

* Build phase: append Course -> O(1) each -> O(n)
* Build index (map put) -> average O(1) each -> O(n)
* Validate prerequisites (map lookup per prereq) → O(1) avg → O(p)
* Worst-case (pathological map performance): O(n + p) typical; O(n^2) if map degrades
* Memory: contiguous array + map of n keys -> low/moderate overhead

Hash Table

* Insert Course -> average O(1) each -> O(n); worst O(n^2) if many collisions
* Validate prerequisites via HT\_Contains -> average O(1) each -> O(p); worst O(np)
* Memory: table capacity + buckets + stored objects -> higher overhead but scalable

Binary Search Tree (unbalanced)

* Insert Course -> average O(log n) each -> O(n log n); worst O(n) each (sorted insert) -> O(n^2)
* Validate prerequisites via BST\_Search -> average O(log n) each -> O(p log n); worst O(p·n)
* Memory: one node per Course (+ pointers) -> moderate overhead

Summary Table

Structure Build (avg) Build (worst) Validate (avg) Validate (worst) Notes

Vector+Map O(n) O(n^2)\* O(p) O(p·n)\* \*If map degrades

Hash Table O(n) O(n^2) O(p) O(p·n) Collisions hurt worst-case

BST (plain) O(n log n) O(n^2) O(p log n) O(p·n) Balance matters

**ADVANTAGES / DISADVANTAGES**

* Vector (+ map index)
* Simple to implement and reason about
* Fast linear build; easy memory model
* Great cache locality when scanning/printing

– Requires an explicit sort O(n log n) to print ordered list

– Needs an auxiliary map for fast prerequisite validation

Hash Table

* Fast average insert/lookup for large n
* Easy prerequisite validation (contains/get)

– To print ordered list, must extract keys and sort (extra O(n log n))

– Worst-case time can degrade with poor hashing or high collisions

– More memory overhead (table + buckets)

Binary Search Tree (unbalanced)

* In-order traversal prints in sorted order without sorting (O(n))
* Reasonable average lookup/insert O(log n)

– Worst-case degenerates to a linked list (O(n^2) build/validate) if data arrives sorted

– Requires careful implementation; ideally self-balancing (AVL/Red-Black) for guarantees

**RECOMMENDATION**

Plan to implement the Binary Search Tree for Project Two so that:

* Option 2 (course list) is naturally O(n) via in-order traversal (no extra sort), and
* Option 3 (single course) lookups are efficient on average.

To address worst-case risks, either:

* Insert in randomized order (shuffle lines before insert), or
* Use/build a self-balancing BST (AVL/Red Black) if permitted.

If a balanced BST is not allowed, the Hash Table is a strong alternative for loading and lookups; you’ll just sort keys at print time (O(n log n)). The Vector approach remains the simplest baseline and is fine for small datasets, but it requires sorting and an auxiliary map for fast validation.