**Algorithm to Print All Courses in Alphanumeric Order (Tree Structure)**

1. Traverse the tree structure in-order
2. For each node:

a. Print the course information (course number, title, and prerequisites)

1. Return the result

**Algorithm to Print a Given Course's Title and Prerequisites (Tree Structure)**

1. Search for the node with the given course number in the tree structure
2. If the node is found:

a. Print the course title and prerequisites

1. If the node is not found:

a. Print "Course not found"

1. Return the result

**Algorithm to Print All Courses in Alphanumeric Order (Hash Table)**

1. Sort the keys of the hash table
2. For each key:

a. Access the course object for the current key

b. Print the course information (course number, title, and prerequisites)

1. Return the result

**Algorithm to Print a Given Course's Title and Prerequisites (Hash Table)**

1. Check if the hash table contains the key with the given course number
2. If the key is found:

a. Access the course object for the current key

b. Print the course title and prerequisites

1. If the key is not found:

a. Print "Course not found"

1. Return the result

**Algorithm to Print All Courses in Alphanumeric Order (Vector)**

1. Sort the vector of course objects based on course number
2. For each course object in the vector:

a. Print the course information (course number, title, and prerequisites)

1. Return the result

**Algorithm to Print a Given Course's Title and Prerequisites (Vector)**

1. Use a for loop to iterate through the vector of course objects
2. Within the for loop, use an if statement to check if the course number of the current course object matches the given course number
3. If the course number matches:

a. Print the course title and prerequisites

1. If the course number does not match, move to the next object in the vector
2. Repeat steps 2-4 until the end of the vector is reached
3. If the course is not found:
4. a. Print "Course not found"
5. Return the result

**Vector:**

The vector is an array-based data structure that can be resized dynamically. The time complexity of inserting an element into a vector is O(1) in the average case and O(n) in the worst case, when the vector needs to be resized. The time complexity of searching for an element in a vector is O(n).

**Advantages:** Simple to implement, constant time complexity for inserting elements in average case, efficient memory usage.

**Disadvantages**: Linear time complexity for searching elements, resizing can be slow in the worst case.

**Hash Table:**

The hash table is a data structure that uses a hash function to map keys to indices in an array. The time complexity of inserting an element into a hash table is O(1) on average. The time complexity of searching for an element in a hash table is O(1) on average, but can be O(n) in the worst case if there are collisions.

**Advantages:** Constant time complexity for inserting and searching elements on average, efficient memory usage.

**Disadvantages:** Possible collisions, resizing can be slow in the worst case.

**Tree:**

The tree is a hierarchical data structure where each node has zero or more children. The time complexity of inserting an element into a tree depends on the implementation, but it is generally O(log n) in balanced trees and O(n) in the worst case in unbalanced trees. The time complexity of searching for an element in a tree is O(log n) in balanced trees and O(n) in the worst case in unbalanced trees.

**Advantages:** Logarithmic time complexity for inserting and searching elements in balanced trees, efficient memory usage.

**Disadvantages:** Requires a balanced tree for efficient performance, can be slower than a hash table in some cases.