Runtime Analysis for Vector:

Code	Line Cost	# Times Executes	Total Cost
courses = empty vector of Course	1	1	1
Open the file with filename	1	1	1
If the file cannot be opened, print an error message and return the empty vector	1	1	1
For each line in the file:	1	n	n
Split the line with commas	1	n	n
Parse the line into courseId, courseName (first two parts of the line)	1	n	n
Put rest of the split parts as prerequisites	1	n	n
If courseId or courseName is empty, print an error message and skip to the next line	1	n	n
Create a Course object with the parsed values and add it to the courses vector	1	n	n
Close the file	1	1	1
Return the courses vector	1	1	1
		Total Cost	6n + 5
		Runtime	O(n)

Runtime Analysis for Hash:

Code	Line Cost	# Times Executes	Total Cost
courses = empty hash table of Course	1	1	1
Open the file with filename	1	1	1
If the file cannot be opened, print an error message and return the empty hash table	1	1	1
For each line in the file:	1	n	n
Split the line with commas	1	n	n
Parse the line into courseId, courseName (first two parts of the line)	1	n	n
Put rest of the split parts as prerequisites	1	n	n
If courseId or courseName is empty, print an error message and skip to the next line	1	n	n
Create a Course object with the parsed values and add it to the courses hash table using courseId as the key	1	n	n

Code	**Line Cost**	**# Times Executes**	**Total Cost**
`Close the file`	1	1	1
`Return the courses hash table`	1	1	1
Total Cost			6n + 5
Runtime			O(n)

Runtime Analysis for Tree:

Code	**Line Cost**	**# Times Executes**	**Total Cost**
`course_tree = empty CourseTree`	1	1	1
`Open the file with filename`	1	1	1
`If the file cannot be opened, print an error message and return the empty tree`	1	1	1
`For each line in the file:`	1	n	n
`    Split the line with commas`	1	n	n
`    Parse the line into courseId, courseName (first two parts of the line)`	1	n	n
`    Put rest of the split parts as prerequisites`	1	n	n
`    If courseId or courseName is empty, print an error message and skip to the next line`	1	n	n
`    Create a Course object with the parsed values`	1	n	n
`    course_tree = Call addCourse(course_tree, course)`	n	n	n2
`Close the file`	1	1	1
`Return the course_tree`	1	1	1
Total Cost			n2 + 6n + 5
Runtime			O(n2)

Advantages and Disadvantages

Vector:

Vectors perform well when reading data from a file and creating a vector with that data. However, when it comes to basic lookups, vectors have a time complexity of O(n), which can lead to poor performance, especially when searching for prerequisites. Vectors are a good choice for small datasets due to their simplicity and ease of use.

Hash Table:

Hash tables also excel in reading data from a file and creating a hash table. They offer fast and easy access to data with an average lookup time complexity of O(1). However, sorting data in a hash table can be challenging since it lacks a natural order. Hash tables are well-suited for scenarios where quick data retrieval is essential, but sorting is not a primary concern.

Tree:

On the other hand, trees performed the least efficiently when reading data from a file. While a balanced tree can achieve an average lookup time complexity of O(log n), our data may not be suitable for a tree structure, resulting in a worse-case O(n) time complexity. This is because we traverse each line in the file, resulting in an overall time complexity of O(n^2). However, trees shine when it comes to sorting data, as they can achieve sorting in O(n), which is faster than vectors and hash tables (O(n*log(n))). Trees are a good choice when sorting is a primary requirement.