parseCourseFile

Code	Line Cost	# Times Executes	Total Cost
If file doesn't exist at "filePath	1	1	1
Output an error message	1	0	0
Return empty vector	1	0	0
If "filePath" string doesn't end with ".txt"	1	1	1
Output an error that the incorrect type of file was input	1	0	0
Return empty vector	1	0	0
Define "courseNames" as vector<String>	1	1	1
Define "courseObjs" as vector<Course>	1	1	1
For every line in the file at "filePath"	1	n	n
Split the line with a comma separator	1	n	n
Store the split in a new "lineSegments" vector<String>	1	n	n
If the size of "lineSegments" isn't at least 2	1	n	n
Ouput that an invalid line was found and skipped	1	0	0
Continue to next iteration	1	0	0
If "courseNames" doesn't have the front item of "lineSegments":	1	n	n
append the front of "lineSegments" to "courseNames"	1	n	n
Define "prerequisites" as empty vector<String>	1	n	n
If the size of "lineSegments" is greater than 2:	1	n	n
Splice the items of "lineSegments" from index 2 to the end	1	n	n

Set “prerequisites” to the splice	1	n	n
If any item in “prequisites” is not in “courseNames”:	1	n	n
Output that in invalid line with missing prerequisite was found and skipped	1	0	0
Continue to next iteration	1	0	0
Set “ID” variable to the front item of “lineSegments”	1	n	n
Set “name” variable to item at index 1 in “lineSegments”	1	n	n
Construct a new Course from “ID”, “name”, and “prequisites”	3	n	3n
Add new course to “courseObjs”	1	n	n
Return “courseObjs”	1	1	1
		Total Cost	17n + 5
		Runtime	O(n)

Creating a Course object

Code	Line Cost	# Times Executes	Total Cost
Set “ID” to first String	1	1	1
Set “name” to second String	1	1	1
Set “prerequisites” to vector<String>	1	1	1
		Total Cost	3
		Runtime	O(1)

Inserting Courses into Data Structure – Worst Case

Vector - Assignment

Code	Line Cost	# Times Executes	Total Cost
Define “courses” as empty vector<Course>	1	1	1
Set “courses” to return of parseCourseFile(“file”)	17n + 5	1	17n + 5
		Total Cost	17n + 6
		Runtime	O(n)

A vector would allow the most seamless transition from the return of parseCourseFile for assignment, but insertion is the least consistent. The best-case insertion for a vector is O(1) when adding an element to the end because it doesn’t involve “shifting” other elements around. However, inserting a course in the middle or, in the worst case, the beginning, of a vector increases the runtime based on the number of shifts. This inconsistency can make large vectors less reliable to add or remove from. A positive, in C++, is that their random access has a constant time complexity of O(1). They’re also the easiest to implement.

Hash Table

Code	Line Cost	# Times Executes	Total Cost
Define “courseHash” as an empty hash table	1	1	1
Define “courses” as the return of parseCourseFile(“courseFile”)	17n + 6	1	17n + 6
For course in “courses”	1	n	n
Call “courseHash”’s Insert method to add the course	2k + 10	n	n(2k + 9)
		Total Cost	n(2k + 9) + 18n + 7
		Runtime	O(n)

Hash Table Insert

Code	Line Cost	# Times Executes	Total Cost
Define “key” as the return of hashing the “courseId” of “course”	2	1	2
Define “currentNode” as a pointer to the node at index “key” in the node vector	2	1	2
Create a Node using “course” and “key”	2	1	2
Define “newNode” as a pointer to the new Node	1	1	1
If “currentNode” is empty (doesn’t hold a course)	1	1	1
Set the Node at index “key” equal to “newNode”	1	1 (if bucket empty)	1
Return	0	1	0
While “currentNode” has a Node after it	1	k (bucket size)	k
Set “currentNode” to the Node after “currentNode”	1	k	k
Set “currentNode”’s next Node to “newNode”	1	1	1
		Total Cost	2k + 10
		Runtime	O(1)

Hash tables' insertion has a constant time complexity which can make them more efficient for larger amounts of data. Their method of hashing items also ensures that buckets have relatively similar sizes, so, in the event of a collision when inserting or searching, it doesn't affect the runtime significantly. They are, however, more complicated to implement than, say, a vector—demanding more effort in designing them to maximize their efficiency. A poorly selected key could negate the constant insert runtime complexity by increasing the average bucket size.

BST

Code	Line Cost	# Times Executes	Total Cost
Define "courseBST" as empty BST	1	1	1
Define "courses" as the return of parseCourseFile("courseFile")	17n + 6	1	17n + 6
For course in "courses"	1	n	n
Call "courseBST". Insert("course")	18log(n) + 1	n	n(18log(n) + 1)
		Total Cost	18nlog(n)+19n+7
		Runtime	O(n^2)

BST Insert

Code	Line Cost	# Times Executes	Total Cost
If “root” node is null	1	1	1
Define “node” as new Node constructed from “course”	3	1	0
Set “root” to “node”	1	1	0
Else	0	1	0
Call addNode(“root”, “course”)	9	log(n)	9log(n)
		Total Cost	9log(n) + 1
		Runtime	O(n)

BST addNode – One call

Code	Line Cost	# Times Executes	Total Cost
Define “newNode” as a new, empty Node	3	1	3
Define “checkingNode” as a new, empty Node	3	1	3
If “course”’s ID is less than “node”’s course’s ID	1	1	1
“checkingNode” equals “node”’s left child	1	1/2	1/2
Else	0	1	0
“checkingNode” equals “node”’s right child	1	1/2	1/2
If “checkingNode” is null:	1	1	1
Set “checkingNode” to “newNode”	1	0	0
Else	0	1	0

Call addNode(“checkingNode”, “course”)	1	1	1
		Total Cost	9
		Runtime	O(1)

While BSTs are the most complicated of the data structures mentioned in this paper, they have consistent runtimes across insertions and searches. They support in-order traversal like printing Course information in alphanumerical order because of their less-than-greater-than structure without needing additional preparation like a vector or hash table. One weakness is that the efficiency of a BST is influenced by the order that items are inserted. For example, if you were to insert a presorted list of Courses, it would result in something called a degenerate BST where each Node only has one child—negating all the benefits of binary search. BSTs also have the most complicated deletion algorithm, making them even more difficult to implement.