Maxwell Sheehan
02/13/25
CS 300

All 3 data structures will have a time complexity of 0(1) when parsing a file line by line, or creating a course object. Reading the file line by line will take 0(n) for all data structures so the use of files will not influence our evaluation of data structures. A vector would have a time complexity of 0(1) if it is just appending, but to maintain order of previous entries it would take 0(n). Sorting option 2 would take 0(n log n) because it is a comparison based sorting. Option 3 would take 0(n) if it is a linear search for a course number. A hash table would take similarly 0(1) for insertion, but could take 0(n) because of collisions within a worst case scenario. Option 2 would typically take a hash table 0(n log n) time as you have to extract and sort keys continuously. Similar to insertion, option 3 would take 0(1) on average, and O(n) in worst cases. So far the Hash table appears faster then vector based data structures but we still must examine binary trees.

BST has a time complexity of 0(logn) if balanced, 0(n) if unbalanced, option 2 would have 0(n) as it's an in-order traversal, option 3 would be O(log n) if balance, 0(n) if unbalanced. I would recommend a BST for these reasons, the sorting is almost automatic with in-order traversal, it is very efficient with 0(logn) which is faster then 0(n log n) for our other two methods. Hash tables memory maintenance of collisions will impede performance also, overall leading to multiple reasons to choose BTS.

Below is the visualization of what is described. Although the binary search tree might be lower for insertion compared to the vector and hash table, it is much faster for sorting the list of classes. Realistically in our use case the insertion will not consume that much time even as a function of log n, while BTS has the clear disadvantage within this method, it has the advantage in the most time consuming method which is why it's my recommendation. Even when searching for a course, Hash table is the fastest of 0(1), in the best to average case. The BTS is still very quick and consistent at 0(log n) . In the worst case we have the same time complexity, and in the average case we are faster than vector sorting. BTS has a very solid overall time analysis, which is a huge advantage. It has the fastest time in our most complicated operation, while good times in our other two, making it again my clear recommendation.

Task	Vector	Hash Table	Binary Search Tree
Insertion	0(1)(append) / 0(n) if sorted	0(1) avg, 0(n) with collisions	0(log n) while balanced, 0(n) worst case
Sorting for list	0(n log n)	0(n log n)	0(n) (in order traversal)
Searching for course	0(n)(linear)	0(1)avg, 0(n) with collisions	0(log n)(balance), 0(n) worst

```
Define course Struct
	courseNumber: Int
	courseTitle: string
	Prerequisites: list of String

Define binarySearchTree class
	Root: node

	Function insert(course: course)
	If root is null then
		Root <- new node(course)
```

```
        Else
                Call insertRecursive(root,course)

Function insertRecursive(node, course)
        If course.courseNumber < node.course.courseNumber then
                If node.left is null then
                        Node.left <- new Node(course)
        Else
                Call insertRecursive(node.left, course)
        Else
                If node.right is null then
                Node.right <- new Node(course)
        Else
                Call insertRecursive(node.right,course)

Function inOrderTraversal(node)
        If node is not null then
                Call inOrderTraversal(node.left)
                Print node.course.courseNumber, node.course.courseTitle,
node.course.prerequisites
                Call inOrderTraversal(node.right)



Function search(node, courseNumber)
        If node is null then
                Return null
        Else if  courseNumber = node.course.courseNumber then
                Return code.course
        Else if courseNumber < node.course.courseNumber then
                Return search(node.left,courseNumber)
        Else
                Return search(node.right, courseNumber)


Function loadCoursesFromFile(filename: String)
        Declare courseTree as BinarySearchTree
        Declare courseMap as Map (string-> course)
```

```
        Open file filename for reading

        While not EOF(file)
                Read line from file
                Split line into tokens by ‘,’

        If length(tokens) < 2 then
                Print “error”
                Continue


        Declare course as Course
        course.courseNumber <- tokens[0]
        course.courseTitle <- tokens[1]

        For i from 2 to length(tokens) -1
                Append tokens[i] to course.prerequisites

        Insert course into courseTree
        Insert (course.courseNumber -> course) into courseMap

Close file



For each course in courseMap
        For each prereq in course.prerequisites
                If prereq not in courseMap then
                Print “error prereq not found”
Return courseTree

Function displayMenu()
        Print “1. Load course data from file”
        Print “2. Print all courses in alphanumeric order”
        Print “3. Print course information”
        Print “9. Exit”

Function main()
        Declare courseTree as binarySearchTree
        Declare choice as Integer
```

```
While True
    Call displayMenu()
    Input choice

    If choice = 1 then
        courseTree <- call loadCoursesFromFile("courses.txt")
    Else if choice = 2 then
        Call courseTree.inOrderTraversal(courseTree.root)
    Else if choice = 3 then
        Print "enter course number:"
        Input courseNumber
        Declare course as Course
    Course <- call courseTree.search(courseTree.root, courseNumber)
    If course is not null then
        Print course.courseNumber, course.courseTitle, course.prerequisites
    Else
        Print ("courses not found")
    Else if choice = 9 then
    Print "exiting program."
    Break
Else print "invalid choice, try again'
```