Freddy Case

Mike Susalla

CS 300

2/16/2025

Project 1 Analysis

Vector Big O table (pages 2-3)

Hash Table Big O table (pages 4-5)

BST Big O table (pages 6-7)

Analysis and Recommendation (8-9)

| **Vector Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **struct Course** | 1 | 1 | 1 |
| **validateFile()** |  |  |  |
| **open file** | 1 | 1 | 1 |
| **for each line in file** | 1 | n | n |
| **partition each parameter between commas** | 1 | n | n |
| **if file has less than 2 parameters** | 1 | n | n |
| **print error message** | 1 | n | n |
| **close file** | 1 | 1 | 1 |
| **loadCourses()** |  |  |  |
| **if validateFile(filename) == false** | 1 | 1 | 1 |
| **return (if valid)** | 1 | 1 | 1 |
| **open file** | 1 | 1 | 1 |
| **for each line in validated file** | 1 | n | n |
| **create course newCourse** | 1 | n | n |
| **set courseId to first partitioned parameter** | 1 | n | n |
| **set title to first partitioned parameter** | 1 | n | n |
| **for each prerequisite** | 1 | n\*m | n\*m |
| **add prerequisite** | 1 | n\*m | n\*m |
| **for all additional parameters** | 1 | n\*m | n\*m |
| **add to newcourse.prerequisite** | 1 | n\*m | n\*m |
| **add course to courses** | 1 | n | n |
| **close file** | 1 | 1 | 1 |
| **validateCourses()** |  |  |  |
| **for each course in courses** | 1 | n | n |
| **for each prerequisite in course.prerequisite** | 1 | n\*m | n\*m |
| **if prerequisite is NOT in courses** | 1 | n\*m | n^2 \* m |
| **print “no prerequisite”** | 1 | n\*m | n\*m |
| **searchCourse()** |  |  |  |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **print out course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **printCourses()** |  |  |  |
| **create empty vector courseList** | 1 | 1 | 1 |
| **for each course in list** | 1 | n | n |
| **add course to courseList** | 1 | 1 | 1 |
| **sortCourse(courseList)** | n log n | 1 | n log n |
| **for each course in courseList** | 1 | n | n |
| **print course.Id, course.title, course.prerequisite** | 1 | n | n |
| **sortCourse()** |  |  |  |
| **for pos from 0 to courseList.size() - 1** | 1 | n | n |
| **min = pos** | 1 | n | n |
| **for j from pos to courseList.size() – 1** | 1 | n^2 | n^2 |
| **if courseList[j].courseId is less than courseList[min].courseId then min = j** | 1 | n^2 | n^2 |
| **if min is not equal to pos then** | 1 | n | n |
| **swap courseList[pos] with courseList[minIndex]** | 1 | n | n |
| **Total Cost** | | | O(n^2\*m +n log n) |
| **Runtime** | | | O(n^2) |

| **Hash Table Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **struct Course** | 1 | 1 | 1 |
| **struct HashTable** |  |  |  |
| **array of LinkedList<course> buckets** | 1 | 1 | 1 |
| **insert()** |  |  |  |
| **index = hash(course.courseId)** | 1 | 1 | 1 |
| **insert course into buckets[index]** | 1 | 1 | 1 |
| **search()** |  |  |  |
| **index = hash(courseId)** | 1 | 1 | 1 |
| **for each course in buckets[index]** | 1 | n | n |
| **if course.courseId is courseId** | 1 | n | n |
| **return course** | 1 | 1 | 1 |
| **return null** | 1 | 1 | 1 |
| **validateFile()** |  |  |  |
| **open file** | 1 | 1 | 1 |
| **for each line in file** | 1 | n | n |
| **partition each parameter between commas** | 1 | n | n |
| **if file has less than 2 parameters** | 1 | n | n |
| **print “filename does not have enough parameters”** | 1 | n | n |
| **close file** | 1 | 1 | 1 |
| **loadCourses()** |  |  |  |
| **if validateFile(filename) == false** | 1 | 1 | 1 |
| **return** | 1 | 1 | 1 |
| **open file** | 1 | 1 | 1 |
| **for each line in validated file** | 1 | n | n |
| **create course newCourse** | 1 | n | n |
| **set courseId to first partitioned parameter** | 1 | n | n |
| **set title to second partitioned parameter** | 1 | n | n |
| **for all additional parameters** | 1 | n\*m | n\*m |
| **add to newcourse.prerequisite** | 1 | n\*m | n\*m |
| **add course to courses** | 1 | n | n^2 |
| **close file** | 1 | 1 | 1 |
| **validateCourses()** |  |  |  |
| **for each course in courses.buckets** | 1 | n | n |
| **for each prerequisite in course.prerequisite** | 1 | n\*m | n\*m |
| **if courses.search(prerequisite) is null** | 1 | n\*m | n^2 \* m |
| **print “no prerequisite”** | 1 | n\*m | n\*m |
| **searchCourse()** |  |  |  |
| **Course course is equal to courses.search(courseNumber)** | 1 | n | n |
| **if the course is not null** | 1 | n | n |
| **print course information** | 1 | 1 | 1 |
| **for each prerequisite in course.prerequisite** | 1 | n | n |
| **print the prerequisite information** | 1 | n | n |
| **else print course not found** | 1 | 1 | 1 |
| **printCourses()** |  |  |  |
| **create empty vector courseList** | 1 | 1 | 1 |
| **for each bucket in table.buckets** | 1 | 1 | 1 |
| **for each course in bucket** | 1 | N | N |
| **add course to courseList** | 1 | N | N |
| **sortCourse(courseList)** | n log n | 1 | n log n |
| **for each course in courseList** | 1 | n | n |
| **print course.courseId, course.title, course.prerequisite** | 1 | n | n |
| **sortCourse()** |  |  |  |
| **for pos from 0 to courseList.size() - 1** | 1 | n | n |
| **min = pos** | 1 | n | n |
| **for j from pos to courseList.size() – 1** | 1 | n^2 | n^2 |
| **if courseList[j].courseId is less than courseList[min].courseId then min = j** | 1 | n^2 | n^2 |
| **if min is not equal to pos then** | 1 | n | n |
| **swap courseList[pos] with courseList[minIndex]** | 1 | n | n |
| **Total Cost** | | | n^2\*m + n log n |
| **Runtime** | | | O(n^2) |

| **BST Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **struct Course** | 1 | 1 | 1 |
| **struct Node()** | 1 | 1 | 1 |
| **class BST()** | 1 | 1 | 1 |
| **insert()** |  |  |  |
| **if root is null** | 1 | 1 | 1 |
| **set root to newNode(course)** | 1 | 1 | 1 |
| **else insertRecursive(root, course)** | 1 | 1 | log n |
| **insertRecursive()** |  |  |  |
| **if course.courseId < node.course.courseId** | 1 | log n | log n |
| **if node.left is null** | 1 | 1 | 1 |
| **set node.left to new Node(course)** | 1 | 1 | 1 |
| **else insertRecursive(node.left,course)** | 1 | log n | log n |
| **else** | 1 | log n | log n |
| **if node.right is null** | 1 | 1 | 1 |
| **set node.right to new Node(course)** | 1 | 1 | 1 |
| **else insertRecursive(Node.right, course)** | 1 | log n | log n |
| **search()** |  |  |  |
| **return searchRecursive(root, courseId)** | 1 | 1 | log n |
| **searchRecursive()** |  |  |  |
| **if node is nullptr** | 1 | log n | log n |
| **return nullptr** | 1 | 1 | 1 |
| **if node.course.courseId is courseId** | 1 | log n | log n |
| **return node** | 1 | 1 | 1 |
| **if courseId < node.course.courseId** | 1 | log n | log n |
| **return searchRecursive(node.left, courseId)** | 1 | log n | log n |
| **else return searchRecursive(node.right, courseId)** | 1 | log n | log n |
| **validateFile()** |  |  |  |
| **open file** | 1 | 1 | 1 |
| **for each line in file** | 1 | n | n |
| **partition each parameter between commas** | 1 | n | n |
| **if file has less than 2 parameters** | 1 | n | n |
| **print “filename does not have enough parameters”** | 1 | n | n |
| **close file** | 1 | 1 | 1 |
| **loadCourses()** |  |  |  |
| **if validateFile(filename) == false** | 1 | 1 | 1 |
| **return** | 1 | 1 | 1 |
| **open file** | 1 | 1 | 1 |
| **for each line in validated file** | 1 | n | n |
| **create course newCourse** | 1 | n | n |
| **set courseId to first partitioned parameter** | 1 | n | n |
| **set title to second partitioned parameter** | 1 | n | n |
| **for all additional parameters** | 1 | n\*m | n\*m |
| **add to newcourse.prerequisite** | 1 | n\*m | n\*m |
| **courses.insert(newCourse)** | 1 | n | n log n |
| **close file** | 1 | 1 | 1 |
| **validateCourses()** |  |  |  |
| **traverseTree (course.root)** | 1 | 1 | n^2 |
| **traverseTree()** |  |  |  |
| **if node is nullptr** | 1 | n | n |
| **return** | 1 |  |  |
| **for each prerequisite in node.course.prerequisite** | 1 | n\*m | n\*m |
| **if courses.search(prerequisite) is null** | 1 | n\*m | n\*m\*log n |
| **print “course” prerequisite “has missing prerequisite.”** | 1 | n\*m | n\*m |
| **traverseTree(node.left)** | 1 | n | n |
| **traverseTree(node.right)** | 1 | n | n |
| **searchCourse()** |  |  |  |
| **Node\* courseNode is equal to courses.search(courseNumber)** | 1 | 1 | log n |
| **if the courseNode is not null** | 1 | n | n |
| **print course information** | 1 | 1 | 1 |
| **for each prerequisite in courseNode.course.prerequisite** | 1 | n | n |
| **print the prerequisite information** | 1 | n | n |
| **else print course not found** | 1 | 1 | 1 |
| **printCourses()** |  |  |  |
| **inOrderTraversal(tree.root)** | 1 | 1 | n |
| **inOrderTraversal()** |  |  |  |
| **if node is null** | 1 | n | n |
| **return** | 1 | 1 | n |
| **inOrderTraversal(node.left)** | 1 | n | n |
| **print node.course.courseId, node.course.title, node.course.prerequisite** | 1 | n | n |
| **inOrderTraversal(node.right)** | 1 | n | n |
| **print course details** | 1 | n | n |
| **Total Cost** | | | O(n\*m +n log n) |
| **Runtime** | | | O(n log N) |

To recommend a data structure to use with managing courses here at ABCU, we first must analyze each data structure. First, we have a vector, which is easy to implement as it needs the least number of lines of code to implement. However, if we need to look for a specific course, the vector will be inefficient since you might have to look at every item in the list. This causes runtime to be slower as the list gets longer, which can be seen with the O(n) worst case runtime in the search method in the vector table. This is seen in the search function, where the code loops through the vector to look for all courses. The table shows the line was executed N times, giving the big O notation O(N) worst-case runtime.

Next, we have a Hash Table, which is more efficient for searching because a Hash Table will use a key in its search function. In the case of ABCU’s courses, we are using the courseId as the key. Using a key in a search function for a Hash table allows for faster access to a course more efficiently than the vector. This is because the run time will be in 0(1) time when compared to the O(N) runtime with the vector. This is seen in the hash table loop over “index = hash(courseId) which costs 1 and then iterates “for each course in buckets[index]” which also costs 1. However, in the worst-case scenario, the Hash table will have a near O(N) runtime if many courses end up in the same bucket. The “key” is to design a hash function that evenly distributes keys across buckets.

Lastly, a BST can keep courses in order automatically by organizing data in a tree. This means each node has a left or right node, where the left child node has lower keys and the right child node has higher keys. If a tree stays balanced, meaning there is an evenly distributed tree where every node has a left and right node, it will have very fast search and insertion runtimes around O(log N). This can be seen in the BST table in the recursive search logic. However, if the tree becomes unbalanced the operations can downgrade to O(n), which is more likely since keeping the tree balanced requires complex code.

After looking at all three data structures, we can see that all three have a similar worse case runtime of O(N^2). A BST offers the best-case scenario runtime with O(log N), with the Hash Table being the second most efficient with a runtime of O(1), and the vector with the worst-case runtime out of the three data structures with a runtime of O(n). Therefore, my recommendation is to use a BST for managing courses at ABCU. A BST is the best option because it offers the best runtime out of the three data structures