Runtime Analysis and Recommendation

Shelby St. Clair

From my understanding based on my pseudocode and the previous modules’ coding assignments, loading the file seems to be pretty much the same across the board for each data structure. The course creation aspect is different depending on the structure, but the actual function to load the file won’t change (besides the formatting to call the course creation/insert function - however, it would still equal 1 since the insert function is called once in the loading process). Based on this, the beginning of each chart where I explore the file loading will look quite similar for each structure.

**Vector Chart**

| Code | Line Cost | Number of Times Executed | Total Cost |
| --- | --- | --- | --- |
| Open file | 1 | 1 | 1 |
| If file is not found | 1 | 1 | 1 |
| Print “file not found” | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| Read each line of file | 1 | n | n |
| Call LineValidity to ensure line is formatted properly | 1 | n | n |
| Parse lines - courseDetails = Split(line,”,”) | 1 | n | n |
| courseNumbers = courseDetails[0] | 1 | n | n |
| courseName = courseDetails[1] | 1 | n | n |
| coursePrerequisites = courseDetails[2] | 1 | n | n |
| Append to vector | 1 | n | n |
| Else | 1 | 1 | 1 |
| Print “Invalid formatting within file” | 1 | 1 | 1 |
| lineDetails = Split(line, “,”) | 1 | n | n |
| IF length(lineDetails) < 2 | 1 | n | n |
| Print “Invalid formatting in line - not enough details” | 1 | 1 | 1 |
| If courseDetails[2] does not match any courseDetails[0] | 1 | 1 | 1 |
| Print “Prerequisite course not found as a main course” | 1 | 1 | 1 |
| Return false | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| Return true | 1 | 1 | 1 |
| Create a new node (new Node\* = node) | 1 | 1 | 1 |
| If node\* head = null | 1 | 1 | 1 |
| Node = head | 1 | 1 | 1 |
| Node = tail | 1 | 1 | 1 |
| else | 1 | 1 | 1 |
| Tail points to new node (tail -> node) | 1 | 1 | 1 |
| Tail = node | 1 | 1 | 1 |
| Increase size by 1 | 1 | 1 | 1 |
| Total Cost | | | 20 + 9n |
| Runtime | | | O(n) |

**Hash Table Chart**

| Code | Line Cost | Number of Times Executed | Total Cost |
| --- | --- | --- | --- |
| Open file | 1 | 1 | 1 |
| If file is not found | 1 | 1 | 1 |
| Print “file not found” | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| Read each line of file | 1 | n | n |
| Call LineValidity to ensure line is formatted properly | 1 | n | n |
| Parse lines - courseDetails = Split(line,”,”) | 1 | n | n |
| courseNumbers = courseDetails[0] | 1 | n | n |
| courseName = courseDetails[1] | 1 | n | n |
| coursePrerequisites = courseDetails[2] | 1 | n | n |
| Else | 1 | 1 | 1 |
| Print “Invalid formatting within file” | 1 | 1 | 1 |
| lineDetails = Split(line, “,”) | 1 | n | n |
| IF length(lineDetails) < 2 | 1 | n | n |
| Print “Invalid formatting in line - not enough details” | 1 | 1 | 1 |
| If courseDetails[2] does not match any courseDetails[0] | 1 | 1 | 1 |
| Print “Prerequisite course not found as a main course” | 1 | 1 | 1 |
| Return false | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| Return true | 1 | 1 | 1 |
| Unsigned key = hash(convert to string) | 1 | 1 | 1 |
| Node\* oldNode = &(nodes.at(key)) | 1 | 1 | 1 |
| If oldNode is null | 1 | 1 | 1 |
| Node\* newNode = newNode(course, key) | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| If oldNode -> key = max | 1 | 1 | 1 |
| oldNode-> key = key | 1 | 1 | 1 |
| oldNode->course = course | 1 | 1 | 1 |
| oldNode->next = null | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| While oldNode-> next does NOT equal null | 1 | n | n |
| oldNode = oldNode ->next | 1 | 1 | 1 |
| oldNode->next = new | 1 | 1 | 1 |
| Total Cost | | | 9n + 24 |
| Runtime | | | O(n) |

**Binary Search Tree**

| Code | Line Cost | Number of Times Executed | Total Cost |
| --- | --- | --- | --- |
| Open file | 1 | 1 | 1 |
| If file is not found | 1 | 1 | 1 |
| Print “file not found” | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| Read each line of file | 1 | n | n |
| Call LineValidity to ensure line is formatted properly | 1 | n | n |
| Parse lines - courseDetails = Split(line,”,”) | 1 | n | n |
| courseNumbers = courseDetails[0] | 1 | n | n |
| courseName = courseDetails[1] | 1 | n | n |
| coursePrerequisites = courseDetails[2] | 1 | n | n |
| Else | 1 | 1 | 1 |
| Print “Invalid formatting within file” | 1 | 1 | 1 |
| lineDetails = Split(line, “,”) | 1 | n | n |
| IF length(lineDetails) < 2 | 1 | n | n |
| Print “Invalid formatting in line - not enough details” | 1 | 1 | 1 |
| If courseDetails[2] does not match any courseDetails[0] | 1 | 1 | 1 |
| Print “Prerequisite course not found as a main course” | 1 | 1 | 1 |
| Return false | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| Return true | 1 | 1 | 1 |
| For all courses in Tree<Courses> | 1 | n | n |
| If root is null | 1 | 1 | 1 |
| Current course will become root | 1 | 1 | 1 |
| If current course < root | 1 | 1 | 1 |
| If left node is null | 1 | 1 | 1 |
| Left node = new Course | 1 | 1 | 1 |
| If left node is not null | 1 | 1 | 1 |
| Current course is added to left branch | 1 | 1 | 1 |
| If current course > root | 1 | 1 | 1 |
| If right node is null | 1 | 1 | 1 |
| Right node = new course | 1 | 1 | 1 |
| If right node is not null | 1 | 1 | 1 |
| Current course is added to right branch | 1 | 1 | 1 |
| Total Cost | | | 9n + 23 |
| Runtime | | | O(n) |

**Recommendation:**

From my understanding, the runtime analysis for loading a file and creating a course object is the same in this situation regardless of which data structure you choose. As the files increase in size, the runtime for each function will also increase in size. I believe the worst structure to use in this case would be the hash table, primarily due to the sorting aspect - hash tables aren’t typically in any sorted order, so we would most likely need to create an empty list to store the course details in order for us to properly sort the file alphanumerically. A vector is a good option due to its simplicity and easy ability to sort, but it would need to be resized every time a course is added - we’d also need an additional function if we wanted to add a vector anywhere aside from the end position in the list. Based on the specific requirement of sorting the list in alphanumeric order and searching for a specific course, I believe a Binary Search Tree would be the best option to use for this program. The “inOrder” function can easily be called to get the alphanumeric list of courses, and BST benefits from a natural binary search due to the nature of the structure. I’ll be implementing my program using the BST option for Project 2.