**Justin Schumann**

**8/10/2024**

**Project One**

1. **Vector**
2. **Reading File:**
3. Create method void loadCourse() to open file.
4. For if the file cannot be found, make a “if” that returns -1. Else proceed with file.
5. Read every line, return “error” if the number of values are 1 or 0. If two or above, continue reading. Make sure that the number of parameters number 3 or more, otherwise return “ERROR”.
6. Close file.
7. **Course Information:**
8. For the identifiers courseName, oursed, and prerequisite make a struct title “Course”.
9. Create a vector method, create String csvPath within.
10. Create a “For” “Loop” that should go over all the file’s rows.
11. To make a struct Node, Code a struct as “Course course”, make a unsigned int title “key” and a “Node\*next”.
12. Add a “data structure” to the collection of Course
13. **Menu:**
14. Create variable title Choice and set to 0.
15. Make a “While” “Loop”, make sure the number in Choice don’t go above 4.
16. Code “Casases” for each choice. Case 1 for loaCourse, Case 2 for printSorted, Case 3 for printCourseInformation, and Case 4 for termination.
17. **Print Sorted List:**
18. Create method printSorted(courses) for sorted print.
19. Create method partition(). Define ints “begin” and “end” within the method.
20. Set your indexes and midpoints.
21. **Hash Table**
22. **Reading File.**
23. In order for file to open, use fstream.
24. In case of file not being found, return -1.
25. The number of values in a line should be 2 or more, otherwise return “ERROR”.
26. The number of parameters should be 3 or more to continue, ELSE return “ERROR”.
27. Close file.
28. **Course Objects:**
29. Set Course Vector vector<Node>nodes.
30. Set the classes.
31. Create loop.
32. Hold values within the item temp.
33. **Search and Print.**
34. Set key, and assign input value to it.
35. Create a “If” statement.
36. If Key is found, print information, if not close file.
37. **Search Tree**
38. **Open File:**
39. Use fstream. Give call for file opening.
40. If no file to be found return “ERROR”
41. Else read parameters.
42. If the first parameter have 3 or more, continue.
43. If less than 3 return “ERROR”.
44. Close file.
45. **Structure Objects:**
46. Give initialize Structure Struct Course.
47. Create loop to read each line in file.
48. For the first two, add course ID and Course Name.
49. For third, give “newline found”.
50. **Tree and node building:**
51. Create Binary Tree Classes and root.
52. Set root towards null.
53. Build an Insert Method.
54. If Root = null, current course = Root.
55. If not, add left or right depending on the course number. Less than leaf for left, more than leaf or root add right.
56. If right = null. Repeat.
57. **Searching and printing:**
58. Give call for input.
59. Build a Print Method.
60. If root != null, move left if no input, move right if input found.

**Menu**

1. Make while statement.
2. Print course list and courses.
3. Create cases.
4. Case one: load files, then break.
5. Case two: print course list, then break.
6. Case three: print courses, then break.
7. Case four: make exit option, then break.

| **Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Creating vector** | 1 | 1 | 1 |
| **files** | 1 | n | n |
| **Id, name, preq** | 3 | n | n |
|  |  |  | n |
|  |  |  | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

| **Hush Table** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Hush Table method** | 1 | 1 | 1 |
| **Key** | 1 | 1 | 1 |
| **File lines** | 1 | 1 | 1 |
|  |  |  |  |
| **With no entry in node** | 1 | n | n |
| **Adding node to key** | n | n | n |
| **If not.** | 1 | n | n |
| **Setting old node** | 2 | n | n |
| **Existing/prque Course to null.** | n | n | n |
| **Search for open node** | n | n | n |
| **Input into open node** | 1 | n | n |
| **Attach new node to end.** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

| **Binary Tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Tree method** | 1 | 1 | 1 |
| **If root = null prt** | 1 | 1 | 1 |
| **Add node** | 1 | 1 | 1 |
| **Go left if node is less than root.** | 1 | n | n |
| **If no left node** | 1 | n | n |
| **If node already left.** | 1 | n | n |
| **Go right if node is more than root.** | 1 | n | n |
| **If no right node** | 1 | n | n |
| **If node already right.** | 1 | n | n |
| **Every line of file** | 1 | n | n |
|  |  |  |  |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

Evaluation

All types of data structure have their pros and cons depending the task assign. Vectors are quick to reference, and are easily to modified. With line-base elements such as text, numbers and geometrical images they perfect to use. When it comes to large-scale data and more complex images, they aren’t recommended for use. Also deletion is more complicated when it comes to vectors.

Hash tables are fast when it comes to looking up and sorting data. Unlike Vectors they are more useful when it comes to large scale data. They are also better at adding and deleting elements. The downsides are that Hash Tables take more time to design and take more to implement.

Binary Trees are better at sorting and storing data. Have more greater memory and less search time. But they require more memory and harder on maintenance.

**Recommendation**

As stated, all data structures have their pros and cons, what makes one more recommended than others will depend on the tasks. Analyzing the strengths and weakness of all the following structures I would recommend the Binary Search Tree, for its easily to edit, quick and good at sorting.