Joseph Dengler CS-300

**Milestone 3 (Vector)**

| **Function** | **Line Cost** | **# Times Executes** | **Total Cost** | **Runtime** |
| --- | --- | --- | --- | --- |
| parseCourseFile(String filename) |  |  |  |  |
| - Initialize courses and lineNumber | 1 | 1 | 1 | O(1) |
| - Open the file | 1 | 1 | 1 | O(1) |
| - Loop through each line in the file | 1 | n | n | O(n) |
| - Process non-empty lines | 1 | n | n | O(n) |
| - Process course data | 1 | n | n | O(n) |
| - Loop through prerequisites (if applicable) | 1 | n | n | O(n) |
| - Create Course object and add to courses | 1 | n | n | O(n) |
| - Close the file | 1 | 1 | 1 | O(1) |
| **Total Cost** |  |  | **5n + 4** | **O(n)** |
| printCourseInformation(vector<Course> courses, String courseNumber) |  |  |  |  |
| - Initialize foundCourse | 1 | 1 | 1 | O(1) |
| - Loop through each course in courses | 1 | m | m | O(m) |
| - Check if the courseNumber matches | 1 | m | m | O(m) |
| - Print course information (if found) | 1 | at most 1 | at most 1 | O(1) |
| - Print prerequisite information (if found) | 1 | at most 1 | at most 1 | O(1) |
| **Total Cost** |  |  | **2m + 2** | **O(m)** |

\*In the table, "n" represents the number of courses in the file, and "m" represents the number of courses in the courses vector passed to the printCourseInformation function. The time complexity for each function is given in the "Runtime" column. For both functions, it is evident that the performance is linear with respect to the number of courses, either in the file or the vector.

**Milestone 4 (Hash Table)**

| **Function** | **Line Cost** | **# Times Executes** | **Total Cost** | **Runtime** |
| --- | --- | --- | --- | --- |
| parseCourseFile(String filename) |  |  |  |  |
| Initialize courses and lineNumber | 1 | 1 | 1 | O(1) |
| Open the file | 1 | 1 | 1 | O(1) |
| Loop through each line in the file | 1 | n | n | O(n) |
| Process non-empty lines | 1 | n | n | O(n) |
| Process course data | 1 | n | n | O(n) |
| Loop through prerequisites (if applicable) | 1 | n | n | O(n) |
| Check prerequisite existence in courses | 1 | n\*avg\_prereqs | n\*avg\_prereqs | O(n\*avg\_prereqs) |
| Create Course object and add to courses | 1 | n | n | O(n) |
| Close the file | 1 | 1 | 1 | O(1) |
| **Total Cost** |  |  | **5n + n \* avg\_prereqs + 4** | O**(n \* avg\_prereqs)** |
| printCourseInformation(vector<Course>courses, String courseNumber) |  |  |  |  |
| Initialize foundCourse | 1 | 1 | 1 | O(1) |
| Loop through each course in courses | 1 | m | m | O(m) |
| Check if the courseNumber matches | 1 | m | m | O(m) |
| Print course information (if found) | 1 | At most 1 | At most 1 | O(1) |
| Print prerequisite information (if found) | 1 | At most avg\_prereqs | At most avg\_prereqs | 0(avg\_prereqs) |
| **Total Cost** |  |  | **2m + avg\_prereqs + 2** | **O(m + avg\_prereqs)** |

\*In the table, "n" represents the number of courses in the file, "m" represents the number of courses in the courses vector passed to the printCourseInformation function, and avg\_prereqs represents the average number of prerequisites per course.

The time complexity for each function is given in the "Runtime" column.

**Milestone 5 (BST)**

| **Function** | **Line Cost** | **# Times Executes** | **Total Cost** | **Runtime** |
| --- | --- | --- | --- | --- |
| readCourseDataFromFile(filePath) |  |  |  |  |
| Initialize courseList | 1 | 1 | 1 | O(1) |
| Open the file | 1 | 1 | 1 | O(1) |
| Loop through each line in the file | 1 | n(number of lines in the file) | n | O(n) |
| Process line data | 1 | n | n | O(n) |
| Check prerequisites existence | 1 | n\* avg\_prereqs | n\* avg\_prereqs | O(n\* avg\_prereqs) |
| Close the file | 1 | 1 | 1 | O(1) |
| **Total Cost** |  |  | **4n+ n\* avg\_prereqs+ 4** | **O(n\* avg\_prereqs)** |
| createCourseObject(courseNumber, courseTitle, prerequisites) |  |  |  |  |
| Create a course object | 1 | 1 | 1 | O(1) |
| **Total Cost** |  |  | **1** | **O(1)** |
| insertIntoBST(root, course) |  |  |  |  |
| If root is null, create a new node | 1 | At most 1 | At most 1 | O(1) |
| Compare courseNumber and traverse the BST | 1 | At most height of the BST | At most log(n) | O(log n) |
| **Total Cost** |  |  | **At most log(n)+1** | **O(log n)** |
| loadDataIntoBST(filePath) |  |  |  |  |
| Initialize courseBST | 1 | 1 | 1 | O(1) |
| Read course data from file | 1 | 1 | 1 | O(1) |
| Create course objects and insert into BST | 1 | n | n\* log(n) | O(n \*log n) |
| **Total Cost** |  |  | **n\* log(n)+ 2** | **O(n\* log n)** |
| printCourse(courseNode) |  |  |  |  |
| If courseNode is null, return | 1 | At most 1 | At most 1 | O(1) |
| Print course information and prerequisites | 1 | At most m(max number of prerequisites) | At most m | O(m) |
| Recurse for the left and right subtrees | 1 | At most n (number of courses) | At most n | O(n) |
| **Total Cost** |  |  | **At most n+ m+ 2** | **O(n+m)** |
| printCoursesInTree(treeRoot) |  |  |  |  |
| Start the recursion from the root of the tree | 1 | 1 | 1 | O(1) |
| Print courses using recursion | 1 | 1 | 1 | O(1) |
| **Total Cost** |  |  | **2** | **O(1)** |

\*In the table, "n" represents the number of lines in the file, "m" represents the maximum number of prerequisites per course, and "log(n)" represents the height of the binary search tree (BST). The time complexity for each function is given in the "Runtime" column.

**Vector: Advantages**

* Simple and easy to use: Vectors are straightforward data structures, and their operations are easy to understand and implement.
* Direct access to elements: Vectors provide constant-time access to elements by their index, making it efficient to access and modify individual courses directly.
* Preserves order: Vectors maintain the order of elements as they are inserted, which can be beneficial when the courses need to be displayed in the order they were read from the file.

**Disadvantages:**

* Linear search complexity: Finding a specific course in a vector requires a linear search, which can be time-consuming when there are many courses.
* Costly insertions and deletions: Insertions and deletions at positions other than the end of the vector can be inefficient, as elements may need to be shifted to accommodate the changes.
* Memory overhead: Vectors typically allocate memory in advance, leading to potential memory waste if the number of courses varies significantly during program execution.

**Hash Table: Advantages:**

* Fast access and insertion: Hash tables provide fast access and insertion operations (on average) with O(1) complexity, making it efficient to find courses and add new courses to the table.
* Constant-time lookup: Once a course is inserted into the hash table, its retrieval time is constant, making it a suitable choice for quick data access.
* Memory efficiency: Hash tables use dynamic memory allocation, consuming memory only as needed, which can be memory-efficient when the number of courses varies during program execution.

**Disadvantages:**

* Collision handling: Hash tables may suffer from collisions when two different courses map to the same hash bucket, potentially impacting the lookup time. Collision resolution techniques (e.g., chaining or probing) may introduce additional complexity.
* Lack of order: Hash tables do not maintain the order of elements, so printing the courses in alphabetical order may require additional sorting steps.
* Hash function complexity: Designing an efficient hash function that minimizes collisions and distributes keys evenly can be challenging.

**Binary Search Tree (BST): Advantages:**

* Ordered structure: BSTs maintain the courses in a sorted order based on the alphanumeric course numbers, enabling efficient retrieval of the courses in alphabetical order.
* Efficient search and insertion: BSTs offer logarithmic search and insertion times (O(log n)), making them efficient for large datasets.
* Dynamic structure: BSTs adjust dynamically to changes in the number of courses, without any pre-allocation overhead, making them memory-efficient for varying course loads.

**Disadvantages:**

* Imbalanced trees: In the worst case, BSTs can become unbalanced, leading to a skewed tree and degrading search performance to O(n) instead of O(log n).
* Expensive operations: Some operations like finding the minimum or maximum course require traversing the entire left or right subtree, respectively, resulting in O(n) time in the worst case.
* Complex implementation: The implementation and management of BSTs can be more complex than other data structures, requiring careful handling of insertion, deletion, and balancing.

**Recommendation**

If the primary concern is fast access and insertion with unordered data, the hash table can be a good choice. If the courses need to be displayed in the order they were read or if the number of courses is small and stable, a vector can be a simple and effective option. If efficient retrieval of courses in alphabetical order is essential, and the number of courses varies dynamically, a balanced BST can provide the desired ordering and efficient operations.

The choice of data structure should depend on the specific needs of the advising program, including the expected size of the course dataset, the frequency of insertions and retrievals, and the importance of maintaining order in the course list. Based on the analysis of the three data structures, I recommend using the Binary Search Tree (BST) as the primary data structure for my code in ABCU's advising program. The BST's logarithmic search time (O(log n)) for large datasets ensures efficient retrieval and display of courses in alphabetical order, meeting the program's requirement. Additionally, its dynamic structure provides memory efficiency for varying course loads, making it a well-balanced choice for implementing the program's functionalities without compromising performance or consuming unnecessary memory.