Linked Lists are best for an application that involves data that is frequently inserted or removed. Parsing CSVs for them however might not be the most optimal option for them if the program needs to frequently access or search for certain objects.

* CSV Parsing and reading the rows in CSV file
  + Time Complexity: O(n), where n is the number of rows (courses) in the CSV file.
  + BigO: O(n) for holding the rows temporarily during parsing.
* Looping through rows:
  + Time Complexity: O(n) due to iteration.
  + BigO: O(n \* m) where m is the prerequisites.
* Creating Course Objects:
  + Time Complexity: O(1) per object creation. So again, it’s O(n)
  + BigO: Each Course object includes a course number, course name, and a vector for prerequisites. Memory for each Course object is O(n \* m) where m is the prerequisites and n is the course.
* Inserting into the Linked List
  + Time Complexity: O(1) for inserting a new node at the beginning of the list. of which there are many so O(n)
  + BigO: each Node stores a reference to a Course and a pointer to the next node leading to O(n) memory for all nodes.
* Validating Prerequisites:
  + BigO: O(m \* n) as each course may have multiple prerequisites, and checking if prerequisite exists involves a linear search through the rows.
* Overall Time Complexity:
  + Each node holds a reference to a course and a pointer to the next node.
  + Memory for Course Objects O(n \*m) for storing all courses and their prerequisites.
  + Memory for Node References: O(n), where n is the number of courses.
  + Total Memory: O(n \* m) for storing courses to make the overall memory complexity.

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **Reading CSV File** | 1 | n | n |
| **Loop through rows** | 1 | n | n |
| **Validate row format** | 1 | n \* m | n \* m |
| **Creating course objects** | 1 | n | n |
| **Check Prerequisites and add** | 1 | n \* m | n \* m |
| **Insert into linked list** | 1 | n | n |
| **Total Cost** | | | O(3n + 2n \* m) |
| **Runtime** | | | O(n \* m) |

**Advantages:**

* Simple Insertion: Linked lists provide a O(1) insertion time, making it efficient for adding new courses without having to worry about resizing the tables.
* Dynamic size: they’re able to do the above benefit because they can either shrink or grow in size
* Simple structure management: linked lists avoid the overhead associated with hash tables such as collisions or maintaining balance trees.

**Disadvantages:**

* Slow Search times: searching for a specific course by course number is O(n) because it involves traversing the list sequentially.
* High Memory Overhead: each node contains an additional pointer, increasing memory usage compared to other methods.

**The Hash table** is a different choice for this use case because it offers efficient storage and quick access for course data through its array and value data structure. With a well-designed hash function, it can minimize collisions and ensure that prerequisites can be checked quickly, aligning well with the requirement to validate course data. These collisions can be mitigated with choosing a large initial table size or using a more complex hash function.

* CSV parsing and readings the rows of the CSV file
  + Time complexity: O(n), where n is the number of rows(courses) in the CSV file. This complexity results from iterating over each row in the file.
  + Memory Usage: O(n) due to storing the rows temporarily during parsing
* Looping Through Rows:
  + Time Complexity: O(n \* m) due to iterating through each row, where n is the course object, and m is the prerequisites.
* Creating Course Objects:
  + Big O: O(1) per course creation. Since there are many, it’s O(n \*m) essentially because of course objects and prerequisites.
* Inserting into the Hash Table (hashTable->Insert(course)):
  + hashing: calculating the hash value involves iterating over characters in the course number.
    - Big O: O(k), where k is the length of the course number.
  + handling collisions: Uses chaining (linked lists) for collision resolution.
    - Big O: O(n) is the worst case scenario if all entries hash to the same key, on average it’ll likely be O(1) because that’s unlikely to happen.
  + Memory Usage: O(n) for storing the hash table itself, with each node potentially holding a linked list.
* Validating Prerequisites(isCourseInFile):
  + For each prerequisite, checking its existence in the file is performed linearly.
    - Big O: O(m \*n) because it’s validating all, it’s going to scan through m prerequisites, and n objects.
* **Overall Complexity:**
  + O(m \* n) due to prerequisite validation.

Advantages:

* Fast Lookup: on average, O(1) time for searching, inserting, and deleting courses, which is ideal for frequent access patterns like checking prerequisites.
* Efficient Use: Addressing of courses by their hashed keys can minimize the time spent in accessing data compared to linear searches.
* Handles Large datasets: suitable for storing a large number of courses due to constant time operations

Disadvantages:

* Collisions: Hash function design can lead to clusters of data in specific buckets thus increasing the access time, so a table should usually start with a large size, or insertion should be done optimally.
* Fixed size: a hash table may require resizing if the number of entries grows significantly, leading to temporary increases in time complexity.

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **Reading CSV File** | 1 | n | n |
| **Loop through rows** | 1 | n | n |
| **Validate row format** | 1 | n \* m | n \* m |
| **Creating course objects** | 1 | n | n |
| **Check Prerequisites and add** | 1 | n \* m | n \* m |
| **Insert into hash table (no collisions)** | 1 | n | n |
| **Total Cost** | | | O(3n + 2n \* m) |
| **Runtime** | | | O(n \* m) |

The **binary search tree** is good for maintaining sorted access to course data, such as the need to keep data in an alphanumerical order. The binary structure of these trees’ attributes to a logarithmic time complexity thus benefiting it greatly, However, the choice between a hash table and a BST depends on the need for sorted access versus faster average-case lookup. If sorted data is not critical, a hash table may be preferable for its O(1) average insertion and search times. In this case, a system where prerequisites are frequently checked, and sorted access is valuable, a BST is a good choice, however in simpler implementations with a focus on direct look ups. A hash table might be preferred.

* CSV parsing and Reading rows:
  + BigO - O(n) where n is the number of courses in the CSV file.
* Looping through rows.
  + BigO - O(n) due to iterating through each row. O(n \* m) since it also has to account for prerequisites.
* Creating Course Objects:
  + BigO – O(1) for initializing each Course object, but adding the prerequisites so O(m)
* Inserting into the Binary Search Tree (bst->Insert(course)):
  + BigO: O(log n) for a balanced BST since the insertion requires traversing the tree to find the correct location, however worst-case, if unbalanced, it becomes O(n)
* Validating Prerequisites(isCourseInFile);
  + checking its existence in the file involves a linear search.
  + BigO: O(n) per prerequisite. So O(m \* n) again.
* Overall Time Complexity:
  + Dominant Term: O(m \* n) from the prerequisite.
  + assuming that prerequisites don’t go above 2. O(m \* n) again

Advantages:

* Automatic sorting: BSTs are well suited for sorting data. By the very nature of their binary left or right design, they make for retrieving of courses easy through its automatic alphanumeric order.
* Automatic Insertion and Deletion: The structure of BSTs makes for the time complexity of either inserting or deleting a values makes for a O(log n) time complexity, or just in general searching.

Disadvantages:

* Unbalanced Tree: While there are measures meant to prevent this, an unbalanced BST can degrade to an O(n) for the searching/insertion/deletion operations, similar to a linked list.

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **Reading CSV File** | 1 | n | n |
| **Loop through rows** | 1 | n | n |
| **Validate row format** | 1 | n \* m | n \* m |
| **Creating course objects** | 1 | n | n |
| **Check Prerequisites and add** | 1 | n \* m | n \* m |
| **Insert into BST(balanced)** | 1 | Log n | Log n |
| **Total Cost** | | | O(3n + 2n \* m + log n) |
| **Runtime** | | | O(n \* m) |

**Recommendation:**

I’m going to break the fourth wall a bit, and say I think I’ll go with the hash Tables. A **Hash table’s** main concern is to avoid as many collisions as possible, but even if one happens the effects are not as detrimental, and unlikely to reach the worse-case run time that with being that the entire table turns into a linked list because every value surfaces as a collision thus turning the average run time into a O(n) runtime. BSTs’ however were giving me a headache, and I’d have to create circumventions to stop the tree from being unbalanced which seems rather daunting currently and will create a O(n) runtime. Both structures have fast search times should the problems not occur, but even the Hash Tables manage to beat BSTs. The prerequisites are always assigned to the respective bucket, making for only a O(1) search time, whereas a BST would have to continuously traverse the tree with a O(log n) look up time. There’s also just the needs and desires of what’s to come. While a BST would have a faster average time of printing all the courses in alphanumeric order, since they’re already in that order if the tree is balanced. That will likely not be needed as often as looking up specific courses and their prerequisites, validating prerequisites which is what the hash tables does more often, and viewing those prerequisites is what the is most likely going to be needed.

Resources

Gupta Himanshu. (2024). Advantages of BST over Hash Table. <https://www.geeksforgeeks.org/advantages-of-bst-over-hash-table/>

Baeldung. (2024) Hastable vs Balanced binary Tree. Baeldung. <https://www.baeldung.com/cs/hash-table-vs-balanced-binary-tree>

Sambnol, M. (2017, April 6). Big-O Notation in 5 Minutes[Video]. Youtube. <https://youtu.be/__vX2sjlpXU?si=0nH49Eggn1NNAyNe>

Olawanle, J. (2022). Big O Cheat Sheet – Time Complexity Chart. freeCoderCamp. <https://www.freecodecamp.org/news/big-o-cheat-sheet-time-complexity-chart/>