Malcolm Williams

CS300

Southern New Hampshire University

December 3, 2023

Analysis and writing

**READING FILE RUNTIME ANALYSIS:**

|  |  |  |  |
| --- | --- | --- | --- |
| **ValidateFile** | **Line Cost** | **# Times Executes** | **Total Cost** |
| validPrereqs[] | 1 | 1 | 1 |
| FOREACH line in file: | 1 | n | n |
| SET parameterCount to count ',' in line + 1 | 1 | n | n |
| IF parameterCount less than 1: | 1 | n | n |
| PRINT "Each line must 2 parameters" | 1 | n | n |
| RETURN false | 1 | 1 | 1 |
|  |  |  |  |
| parameters | 1 | n | n |
| SET parameters to line split by ',' | 1 | n | n |
|  |  |  |  |
| number, prepreqs[] | 1 | n | n |
|  |  |  |  |
| SET number to parameters[0] | 1 | n | n |
| APPEND number to validPrereqs | 1 | n | n |
|  |  |  |  |
| preReqCount | 1 | n | n |
| SET preReqCount to parameterCount - 2 | 1 | n | n |
|  |  |  |  |
| FOR i, 0 to preReqCount: | 1 | n | n |
| SET prepreqs[i] to parameters[i+2] | 1 | n | n |
|  |  |  |  |
| FOREACH prereq in prepreqs: | 1 | n | n |
| IF prereq not in validPrereqs: | 1 | n | n |
| FOREACH line starting from current line: | 1 | n | n |
| SET number to split line by ','[0] | 1 | n | n |
| IF number matches prereq: | 1 | n | n |
| break loop | 1 | n | n |
| ELIF line is last line: | 1 | n | n |
| PRINT "All prerequisites must be a main course" | 1 | 1 | 1 |
| RETURN false | 1 | 1 | 1 |
|  |  |  |  |
| RETURN true | 1 | 1 | 1 |
|  |  |  |  |
|  |  | Total Cost | 20n+5 |
|  |  | Run Time | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **PopulateCourses (Vector)** | **Line Cost** | **# Times Executes** | **Total Cost** |
| FOREACH line in file: | 1 | n | n |
| SET parameters to line split by ',' | 1 | n | n |
|  |  |  |  |
| number | 1 | n | n |
| title | 1 | n | n |
| prepreqs[] | 1 | n | n |
|  |  |  |  |
| FOR i, 0 to parameters size - 1: | 1 | n | n |
| IF i is 0: | 1 | n | n |
| SET number to parameters[0] | 1 | n | n |
| ELIF i is 1: | 1 | n | n |
| SET title to parameters[1] | 1 | n | n |
| ELSE: | 1 | n | n |
| APPEND parameters[i] to prepreqs | 1 | n | n |
|  |  |  |  |
| PUSHBACK new course(number, title, prepreqs) to vector | 1 | n | n |
|  |  |  |  |
|  |  |  |  |
|  |  | Total Cost | 13n |
|  |  | Run Time | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **PopulateCourses (HashTable)** | **Line Cost** | **# Times Executes** | **Total Cost** |
| FOREACH line in file: | 1 | n | n |
| SET parameters to line split by ',' | 1 | n | n |
|  |  |  |  |
| number | 1 | n | n |
| title | 1 | n | n |
| prepreqs[] | 1 | n | n |
|  |  |  |  |
| FOR i, 0 to parameters size - 1: | 1 | n | n |
| IF i is 0: | 1 | n | n |
| SET number to parameters[0] | 1 | 1 | 1 |
| ELIF i is 1: | 1 | n | n |
| SET title to parameters[1] | 1 | 1 | 1 |
| ELSE: | 1 | n | n |
| APPEND parameters[i] to prepreqs | 1 | 1 | 1 |
|  |  |  |  |
| hashtable->Insert(new course(number, title, prepreqs)) | n+10 | n | n^2 + 10n |
|  |  |  |  |
|  |  |  |  |
|  |  | Total Cost | n^2 + 19n + 3 |
|  |  | Run Time | O(n^2) |

|  |  |  |  |
| --- | --- | --- | --- |
| **PopulateCourses (BST)** | **Line Cost** | **# Times Executes** | **Total Cost** |
| FOREACH line in file: | 1 | n | n |
| SET parameters to line split by ',' | 1 | n | n |
|  |  |  |  |
| number | 1 | n | n |
| title | 1 | n | n |
| prepreqs[] | 1 | n | n |
|  |  |  |  |
| FOR i, 0 to parameters size - 1: | 1 | n | n |
| IF i is 0: | 1 | n | n |
| SET number to parameters[0] | 1 | n | n |
| ELIF i is 1: | 1 | n | n |
| SET title to parameters[1] | 1 | n | n |
| ELSE: | 1 | n | n |
| APPEND parameters[i] to prepreqs | 1 | n | n |
|  |  |  |  |
| tree->Insert(new course(number, title, prepreqs)) | 2n+16 | n | (2n^2)+16n |
|  |  |  |  |
|  |  |  |  |
|  |  | Total Cost | (2n^2)+28n |
|  |  | Run Time | O(n^2) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Insert (HashTable)** | **Line Cost** | **# Times Executes** | **Total Cost** |
| SET key to hash(course number) | 2 | 1 | 2 |
| SET bucketList to buckets[key] | 1 | 1 | 1 |
| IF bucketList->Search(course number) is null | n | 1 | n |
| bucketList->Append(course) | 7 | 1 | 7 |
|  |  |  |  |
|  |  | Total Cost | n+10 |
|  |  | Run Time | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Insert (BST)** | **Line Cost** | **# Times Executes** | **Total Cost** |
| IF root is null: | 1 | 1 | 1 |
| SET root to Node(course) | 1 | 1 | 1 |
| ELSE: | 1 | 1 | 1 |
| SET newNode to Node(course) | 1 | 1 | 1 |
| SET currentNode to root | 1 | 1 | 1 |
|  |  |  |  |
| WHILE currentNode is not null : | 1 | n | n |
| IF course number less than currentNode course number: | 1 | n | n |
| IF left child is null: | 1 | 1 | 1 |
| SET left child to newNode | 1 | 1 | 1 |
| SET currentNode to null | 1 | 1 | 1 |
| ELSE: | 1 | 1 | 1 |
| SET currentNode to left child | 1 | 1 | 1 |
| ELSE: | 1 | 1 | 1 |
| IF right child is null: | 1 | 1 | 1 |
| SET right child to newNode | 1 | 1 | 1 |
| SET currentNode to null | 1 | 1 | 1 |
| ELSE: | 1 | 1 | 1 |
| SET currentNode to right child | 1 | 1 | 1 |
|  |  |  |  |
|  |  | Total Cost | 2n+16 |
|  |  | Run Time | O(n) |

**ADVANTAGES & DISADVANTAGES:**

Vector:

Using a vector for storing course data allows for indexed elements and dynamic sizing. Indexed element access provides constant time complexity which is ideal for efficient sorting. Vectors can dynamically grow and allocate additional space as required. Additionally, appending a value to the end of a vector has constant time complexity. However, inserting a value at the front or middle of a vector requires shifting subsequent items.

Hash Table:

The primary strength of a hash table is its fast insertion and efficient search operations. Employing a hashing algorithm with minimal collisions enables rapid identification of the storage location through an ideally unique value association. This avoids sequentially scanning through the structure's items to search for a specific value. A drawback of hash tables is their lack of order. For instance, when aiming to arrange the courses in alphanumeric order, the data must be copied into an alternate structure for sorting purposes. This additional data copying leads to increased space and time consumption within an algorithm.

Binary Search Tree:

The binary search tree data structure offers the advantage of inherent order which eliminates the need to sort data. In this case, the structure's value for comparison is the course number which aligns with the desired order for data printing. This removes the need for pre-sorting the structure for list printing, given that each node's left child is inherently less than its parent. Additionally, a binary search tree is dynamic and does not require resizing. However, a drawback of the binary search tree occurs during the insertion or removal of a value. These operations often require restructuring the tree to maintain its order.

**RECOMMENDATION:**

I recommend utilizing a binary search tree to contain the course data. The inherent ordering property of the tree ensures that the data is sorted, removing the necessity for additional sorting algorithms. Comparatively, a hash table, while efficient in loading data, would result in inefficiency when generating a list of courses due to the increased runtime and space complexity when transferring data to a vector or array for sorting. Additionally, the pre-sorted nature of the BST distinguishes it from a vector, which would still require sorting after data loading. Given these considerations, the binary search tree is a practical and efficient data structure for storing and listing the course data in an orderly fashion.