SNHU: CS-300 DSA: Analysis and Design

Module 6 Evaluation – Project One

Matthew Bandyk

Below is the Big O analysis of each of the three data structures based on the Pseudocode.

# Parser

|  |  |
| --- | --- |
| **Parse Method** | |
| **<<CODE>>** | **Cost** |
| Create input stream courseFile and open file at filePath | 1 |
| If file is not open | 1 |
| Print “File failed to open” << end line | 1 |
| Define string line | 1 |
| While file open is good | n |
| Getline of courseFile and put in line | 1 |
| If line is not empty | 1 |
| Push back line into originalFile | 1 |
| Close file | 1 |
| Initiate parseRow | 1 + (See below) |
| ***Big O analysis pre initiating parseRow = O(8 + n)*** | **O(n)** |
| ***Big O analysis initiating parseRow (vector) = O(8 + n) + n*** | **O(n)** |
| ***Big O analysis initiating parseRow (Hash Table) = O(8 + n) + n*** | **O(n)** |
| ***Big O analysis initiating parseRow (BST) = O(8 + n) + n*** | **O(n)** |

|  |  |
| --- | --- |
| **ParseRow Method** | |
| **<<CODE>>** | **Cost** |
| Declare vector string | 1 |
| Declare iterator cur | 1 |
| For while cur is not originalFile end, iterate cur | n |
| Define integer countStart to 0 | 1 |
| Define unsigned integer i to 0 | 1 |
| If cur at i is equal to “,” | 1 |
| Push row from cur substring of countStart to i – countStart | 1 |
| Set countStart to i + 1 | 1 |
| If row is less then 2 | 1 |
| Print “Error in data in file” and end line | 1 |
| Push back string to courseData | 1 |
| ***Big O analysis pre loadCourses = O(2 + n + 8)*** | **O(n)** |
|  |  |
| Initiate loadCourses (see Vector) | n |
| /or Initiate loadCourses (see Hash Table) | n |
| /or Initiate loadCourses (see Binary Search Tree) | n |
| ***Big O analysis for Vector = O(2 + n + 8 + n)*** | **O(n)** |
| ***Big O analysis for Hash Table = O(2 + n + 8 + n)*** | **O(n)** |
| ***Big O analysis for BST = O(2 + n + 8 + n)*** | **O(n)** |

# Vector

|  |  |
| --- | --- |
| **LoadCourses Method** | |
| **<<CODE>>** | **Cost** |
| Define vector | 1 |
| Define vecSize as size of courseData | 1 |
| Define string tempString | 1 |
| For unsigned integer i equals 0, while i is less than vecSize, iterate i | n |
| Assign courseData at [i][0] to course.courseNum | 1 |
| Assign courseData at [i][1] to course.name | 1 |
| If courseData[i] is greater than 2 |  |
| For unsigned integer j equals 0, while j is not courseData[i]’s end, iterate j | n |
| Assign tempString to equal courseData[i][j+2] | 1 |
| If (initiate preReqValidation(stream courseFile and tempString) is true) | n |
| push back tempString to course.preReq | 1 |
| Push back Course course to courses | 1 |
| ***Big O analysis = O(3 + ((n + 2) + 1 + (n + 1)) + (n + 1) + 1)*** | **O(n)** |

# Hash Table

|  |  |
| --- | --- |
| **Hash Method** | |
| **<<CODE>>** | **Cost** |
| Declare integer i to equal key length | 1 |
| Declare integer stringHash to equal 6101 | 1 |
| Declare integer hashMult to equal 3 | 1 |
| For each ch in key, while ch is less than i | n |
| Assign stringHash to equal (stringHash \* hashMult) + int(ch) | 1 |
| Return stringHash mod tableSize | 1 |
| ***Big O analysis = O(5 + n)*** | **O(n)** |

|  |  |
| --- | --- |
| **LoadCourses Method** | |
| **<<CODE>>** | **Cost** |
| Define vector data structure vector<Course> courses | 1 |
| Define vecSize as size of courseData | 1 |
| Define string tempString | 1 |
| For unsigned integer i equals 0, while i is less than vecSize, iterate i | n |
| Assign courseData at [i][0] to course.courseNum | 1 |
| Assign courseData at [i][1] to course.name | 1 |
| If courseData[i] is greater than 2 | 1 |
| For unsigned integer j equals 0, while j is not courseData[i]’s end, iterate j | n |
| Assign tempString to equal courseData[i][j+2] | 1 |
| If (initiate preReqValidation(stream courseFile and tempString) is true) | n |
| push back tempString to course.preReq | 1 |
| Push back course to courses | 1 |
| Initiate Insert(Course course) | n |
| ***Big O analysis = O(3 + ((n + 2) + 1 + (n + 1)) + (n + 1) + 1 + n)*** | **O(n)** |

|  |  |
| --- | --- |
| **Insert Method** | |
| **<<CODE>>** | **Cost** |
| Declare key to equal the initiation of hash passing string course courseNum | n |
| If keyNode is equal to nullptr | 1 |
| Assign a new node equal to Node(course and key) | 1 |
| Insert the new node at key location | 1 |
| Else if keyNodes key is equal to UINT\_MAX | 1 |
| Assign keyNodes key to equal key | 1 |
| Assign keyNodes course to equal course | 1 |
| Assign keyNodes next to equal nullptr | 1 |
| Else | 1 |
| while keyNodes next is not equal to nullptr | n |
| Assign keyNode to equal keyNode next | 1 |
| Assign keyNode to equal new Node(course and key) | 1 |
| ***Big O analysis = O(n + 8 + n + 2)*** | **O(n)** |

# Binary Search Tree

|  |  |
| --- | --- |
| **AddNode Method** | |
| **<<CODE>>** | **Cost** |
| If nodes courseNum is greater than course courseNum | 1 |
| If nodes left equals nullptr | 1 |
| Assign nodes left to equal new Node passing course | 1 |
| else | 1 |
| Initiate this class addNode method passing nodes left and course | n |
| Else | 1 |
| if nodes right equals nullptr | 1 |
| Assign nodes right to equal new Node passing course | 1 |
| else | 1 |
| Initiate this class addNode method passing nodes right and course | n |
| ***Big O analysis = O(4 + n + 4 + n)*** | **O(n)** |

|  |  |
| --- | --- |
| **LoadCourses Method** | |
| **<<CODE>>** | **Cost** |
| Define vector data structure vector<Course> courses | 1 |
| Define vecSize as size of courseData | 1 |
| Define string tempString | 1 |
| For unsigned integer i equals 0, while i is less than vecSize, iterate i | n |
| Assign courseData at [i][0] to course.courseNum | 1 |
| Assign courseData at [i][1] to course.name | 1 |
| If courseData[i] is greater than 2 | 1 |
| For unsigned integer j equals 0, while j is not courseData[i]’s end, iterate j | n |
| Assign tempString to equal courseData[i][j+2] | 1 |
| If (initiate preReqValidation(stream courseFile and tempString) is true) | n |
| push back tempString to course.preReq | 1 |
| Push back course to courses | 1 |
| Initiate Insert(Course course) | n |
| ***Big O analysis = O(3 + ((n + 2) + 1 + (n + 1)) + (n + 1) + 1 + n)*** | **O(n)** |

|  |  |
| --- | --- |
| **Insert Method** | |
| **<<CODE>>** | **Cost** |
| If root equals nullptr | 1 |
| Assign root to equal new Node passing course | 1 |
| else | 1 |
| Initiate this class addNode method passing root and course | n |
| ***Big O analysis = O()*** | **O(n)** |

|  |  |
| --- | --- |
| **AddNode Method** | |
| **<<CODE>>** | **Cost** |
| If nodes courseNum is greater than course courseNum | 1 |
| If nodes left equals nullptr | 1 |
| Assign nodes left to equal new Node passing course | 1 |
| else | 1 |
| Initiate this class addNode method passing nodes left and course | n |
| else | 1 |
| if nodes right equals nullptr | 1 |
| Assign nodes right to equal new Node passing course | 1 |
| else | 1 |
| Initiate this class addNode method passing nodes right and course | n |
| ***Big O analysis = O(4 + n + 4 + n)*** | **O(n)** |

Based on the Big O analysis of the various algorithms for the data structures, we can see that each has a Big O value of O(n) which is the worst-case running time. After completing the Big O analysis, each would produce the same level of run time at their worst cases, but the hash table and the binary search tree data structures do have additional methods within them to complete the loading functions.

There are advantages and disadvantages to the three data structures. I will be covering each below at a high level. It should be noted that there are ways to overcome most obstacles, but this evaluation is based on the structures individually without the additional “modifications” or work arounds.

**Vector**

*Advantages:*

* Vectors are dynamic in nature, meaning they can scale based on the data size and utilize pointers to point to the next item in the data structure
* Efficient when needing to sort or add items to the structure
* Many libraries supported methods and functions that can be utilized to work with them

*Disadvantages:*

* Ineffective when searching for an item in the data set as it must iterate through the entire list until a match is found, or through the entire list if there is no matching item
* Vectors can be difficult to manipulate individual elements as the location is unknown and must first be found before updating
* While it is dynamic, in order to increase in size, the vector must be copied from the old vector to a new vector, which can affect performance

**Hash Table**

*Advantages:*

* Extremely fast structure for insertion, search and remove as each item is assigned a key
* The hash table is very space efficient
* Hash tables are flexible in their data storage, meaning they can house any data type

*Disadvantages:*

* Hash tables sizes must be declared, and must be maintained if the data set increases to avoid to many collisions occurring
* Hash tables are unordered and there is no effective way to sort the data within them
* They can be complex to develop and implement effectively

**Binary Search Tree**

*Advantages:*

* When the tree is balanced, meaning that each the left and right are similar in height, a BST’s insertion and search capabilities are fast
* A BST is naturally sorted when data is placed into it
* The data structure is flexible when scaling, allowing easy insertion of additional data

*Disadvantages:*

* While an advantage is the speed of the BST, if the tree is not balanced, this advantage is removed, meaning all data could be housed on just one side of the tree making it more linear in nature
* Removing nodes from the tree can be very complex and difficult
* While there are ways around it, duplicate keys are not allowed in BST’s

After reviewing the Big O analysis and the advantages and disadvantages of each of the tree data structures, my recommendation would be to utilize a binary search tree data structure to support the ABCU requirements. The reasoning behind this is that this data structure offers many advantages in this situation. BST boasts strong run time efficiencies' when looking at the functionality required by ABCU, to include searching, printing and inserting. The BST is naturally sorted as well, making it easy to display all the courses in order. This is all based on the assumption that the data being loaded to the structure is not already an ordered list though.

If the list is ordered, or even closely ordered, the benefits of the BST are quickly removed as it would become overloaded on one side of the tree. In that instance, I would advocate the use of a vector data structure instead. The vector is not as effective with its searching and data manipulation but would work better with an ordered list. The vector data structure also has some built in library support methods/functions that would make it easy to sort the data if it is an unsorted list. It would also be the simplest to implement of the three. As the data in question is unknown at this time, I would advocate for the vector to ensure that the end program would run efficiently no matter the case.