Brooklen Ashleigh

CS300

Professor Jamaliannasrabadi

June 10th, 2023

Project One

PSEUDOCODE

Vector Data Pseudocode

**Opening and Reading Files:**

FOR EACH File

FOR EACH row in File

SPLIT row by comma

IF length of split row >= to 2

CREATE new course object

SET course object’s parameters to course info

APPEND course object to Vector<Courses>

IF length of split row > 2

FOR every string in split row from index 2 to end

IF row[course number] is in <Courses>

CREATE new course object

SET course object’s parameters to row’s course info

APPEND course object to Vector<Courses>

ELSE

PRINT “File contains Invalid input”

**Print:**

FOR EACH Course in Vector<Courses>

IF Course ID = Sought Class ID

PRINT Course information

IF Course has prerequisites

FOR every Prerequisite

PRINT Course information

Hash Table Pseudocode

**Opening and Reading Files:**

FOR EACH File

FOR EACH row in File

Split row by comma

IF length of split row is >= 2

INIT key based on course number

INIT new course object

SET course object’s key to hashed key

SET course object’s parameters to course info

APPEND course object to bucket vector

IF length of split row is > 2

FOR every string in split row from index 2 to end

INIT key based on course number

SET course object’s key to hashed key

SET course object’s parameters to course info

IF course with number exists

CREATE new course object

SET course object’s parameters to row’s course info

APPEND course object to bucket vector

ELSE

OUTPUT “File contains Invalid input”

CONTINUE

**Print:**

FOR EACH Course in Nodes Vector

INIT key based on sought class ID

INIT node with nodes[key]

WHILE node is not null

IF node’s ID == sought ID

OUTPUT Course information

IF Course has prerequisites

FOR every Prerequisite

OUTPUT Course information

INCREMENT node

Binary Search Tree Pseudocode

**Opening and Reading Files:**

FOR EACH File

FOR EACH row in File

Split row by comma delimiter

IF length of split row is >= 2

INIT ID based on course number

INIT new course object node

SET course object’s ID

SET course object’s parameters to course info

IF Root = null

Root = course object node

ELSE

IF current node < course ID

IF right node = null

ADD node to right

ELSE

Recurse down the right node

ELSE

IF left note = null

ADD node to left

ELSE

Recurse down left node

IF length of split row is > 2

FOR every string in split row from index 2 to end

INIT course ID

IF course with number exists

CREATE new course object

SET course object’s parameters to row’s course info

IF course ID < current node

ADD node to left

ELSE

ADD node to right

ELSE

OUTPUT “File contains Invalid input”

CONTINUE

**Print:**

FOR EACH Course in Nodes Vector

INIT key based on sought class ID

INIT node with nodes[key]

WHILE root is not null

FOR i < level

OUTPUT Course information

IF Course has prerequisites

FOR every Prerequisite

OUTPUT Course information

RECURSIVELY CALL print function with root-> left node and level + 1

RECURSIVELY CALL print function with root-> right node and level + 1

MENU PSEUDOCODE

OUTPUT list of menu options

INPUT user selection

IF user selection is valid

IF user selection = 1

CALL insert function

IF user selection = 2

CALL print all courses function

IF user selection = 3

INIT copy of data structure

SORT data structure by course ID

CALL print all courses function

IF user selection = 4

Exit

PRINT LIST OF COURSES IN ALPHANUMERIC ORDER

**Vector Sort and Print**

Void quicksort(vector<Courses>& courses, int begin, int end) {

SET mid to 0

SET low to begin

SET high to end

IF low >= high

RETURN

SET mid to partition of courses, low and high

CALL quickSort with courses, low, and mid

CALL quickSort with courses, mid + 1, and high

}

Void printVector(vector<Courses>& courses) {

FOR course in Courses

OUTPUT course ID

FOR course in Prerequisites

OUT course ID

}

HashTable Sort and Print

HashTables are not sorted, and rely on ordered insertion.

PRINT:

Void printHash() {

FOR course in Courses

IF key != Max

OUTPUT course

Course = Next Course

WHILE course prerequisite is not NULL

OUTPUT prerequisite

Prerequisite = next Prerequisite

}

BINARY SEARCH TREE SORT AND PRINT

BSTs also rely on in-order insertion for alphanumeric order.

Void inOrder(Node\* node) {

IF node is not null

CALL InOrder function with node->left

OUTPUT course information

CALL InOrder function with node->right

}

EVALUATION

Vector

| ***Code*** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| FOR EACH row in File | 1 | n | n |
| SPLIT row by comma | 1 | n | n |
| IF length of split row > 2 | 1 | 1 | 1 |
| FOR every string in split row from index 2 to end | 1 | n | n |
| IF row[course number] is in <Courses> | 1 | 1 | 1 |
| CREATE new course object | 1 | 1 | 1 |
| APPEND course object to Vector<Courses> | 1 | 1 | 1 |
| ***Total Cost*** | | | 3n + 4 |
| ***Runtime*** | | | O(n) |

Hash Table

| ***Code*** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| FOR EACH row in File | 1 | n | n |
| Split row by comma | 1 | n | n |
| IF length of split row is > 2 | 1 | 1 | 1 |
| FOR every string in split row from index 2 to end | 1 | n | n |
| INIT key based on course number | 1 | 1 | 1 |
| SET course object’s key to hashed key | 1 | 1 | 1 |
| SET course object’s parameters to course info | 1 | 1 | 1 |
| IF course with number exists | 1 | 1 | 1 |
| CREATE new course object | 1 | 1 | 1 |
| SET course object’s parameters to row’s course info | 1 | 1 | 1 |
| APPEND course object to bucket vector | 1 | 1 | 1 |
| ***Total Cost*** | | | 3n + 8 |
| ***Runtime*** | | | O(n) |

Binary Tree

| ***Code*** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| FOR EACH row in File | 1 | n | n |
| Split row by comma delimiter | 1 | n | n |
| IF length of split row is > 2 | 1 | 1 | 1 |
| FOR every string in split row from index 2 to end | 1 | n | n |
| INIT course ID | 1 | 1 | 1 |
| IF course with number exists | 1 | 1 | 1 |
| CREATE new course object | 1 | 1 | 1 |
| SET course object’s parameters to row’s course info | 1 | 1 | 1 |
| IF course ID < current node | 1 | 1 | 1 |
| ADD node to left | 1 | n | n |
| ADD node to right | 1 | n | n |
| ***Total Cost*** | | | 5n + 6 |
| ***Runtime*** | | | O(n) |

ADVANTAGES AND DISADVANTAGES

Vector

The advantages of using a vector as a data structure is its fast and simple access, with a time complexity of O(1), aided by the fact that they are stored contiguously in memory. Its average and worst-case complexities aside from access are all O(n), which is good but not the best possible. Appending to an array is also fast. The disadvantages come with an application where many insertions are required, as all elements of an array must be shifted to account for insertions. However, in this application, insertions are not likely. As items are not inserted into an array in any particular order, this can make searching slower, and the vector must be sorted. Another disadvantage to vectors is if they need to be resized often, which can be costly.

Hash Table

Hash tables have an optimal average time complexity of O(1) for Searching, Insertion, and Deletion, with a worst-case time complexity of O(n). As items are grouped in a hash table as per their ID, they are easy to insert and search for. The disadvantages to hash tables include if the data used causes a large amount of collisions. Collisions may be a significant issue in the adviser’s application, as course IDs have a pattern of typical structures, and modulo hashing may cause significant collisions. Also, hash tables are not sortable without a map structure, and therefore are limited if many different types of sorting are needed in the end application, which the adviser may want to implement in the future.

Tree

A Binary Search Tree has a good average time complexity of O(log(n)) for accessing, searching, inserting, and deletion. Its worst-case scenario rating is O(n), which is still good. The disadvantages to binary search trees are that they can become imbalanced, which reduces performance, and that they generally take more memory than other data structures.

RECOMMENDATION

The advising application requires an efficient data structure. Overall, the most optimal data structure when speed is considered would be a hash table, which has an average time complexity of O(1) for Search, Insertion, and Deletion. The worst-case scenario Time Complexity for these operations is O(n). Of all data structures, Hash Tables offer the fastest searching, which is a key function of the application. However, there are a few aspects that make hash tables non-optimal in this scenario. Hash Tables are unsorted, and must be inserted in order to be able to be accessed in order. Because printing in alphanumeric order is a main function of this application, hash tables are not optimal. A map would have to be used. Additionally, course IDs are often repeated and have similar digits, and if a key was hashed with a modulo operation, there would likely be many collisions.

Because of these aspects, I recommend using a Vector as a data structure. Insertions are non-optimal, but based on the way the information is presented in near perfect order, this will likely not be an issue. Vectors have very fast access speeds. Vectors have good average and worst-case time complexities, at O(n). Any of the fastest sort algorithms can be used with them, which will make sorting by alphanumeric course ID simple. This would also mean that if the client wished to use other types of sorting in the future, a vector structure would be easy to build off of. Additionally, the number of total objects in this system will not likely exceed several hundred, so a vector would have good performance and easy lookup.

A binary search tree would give a faster search, insertion, and deletion, however in this case I believe because the data is only initially inserted, with mostly searches being done after that operation, the Binary Search Tree doesn’t offer significant advantages, and would also have the same sorting limitations that a Hash Table has.