Week 6

Daniel Loranger

**Evaluation**

**Evaluate the run time and memory of data structures that could be used to address the requirements.**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Vector | Hash Table | Binary Tree |
| Load Data from File | O(1) \* Prereqs is variable, but presumably bounded content so not O(N) | O(1) \* Prereqs is variable, but presumably bounded content so not O(N) | O(1) \* Prereqs is variable, but presumably bounded content so not O(N) |
| Search for course | O(N) \* on average, the search will take ½\*O(N) time | O(1) \* presuming the table is created with hashes for each of 3\*26(alpha)\*3\*10 (numeric) values. This will prevent any collisions unless multiple courses share the same course ID but different prereqs/course name that would make them unique. This is assumed to not be true. | O (N Log N) \* search is reentrant while traversing the tree |
| Sorting -> Printing | O(N) \* Sorting will take N cycles to complete, printing will also take N cycles to complete | O(N) \* having no collisions, the hashes will be bucketted in sorted order | O(N) \* in order traversal |
| Memory Consumption | Minimal \* only necessary space allocations will be made | Maximal \* memory will be allocated for all potential courses, many will be unused. | Minimal \* only necessary space allocations will be made |

**Advantages and disadvantages**

The actual time to print the contents will be constant across all implementations as the contents is common to all methods. The loading of the file is likewise constant across all methods, as every entry must be read from file, parsed and validated.

The data handling, searching and sorting is where the differences will appear.

**Data Handling**

vector table – data can either be sorted initially during loading which would be expected to take a long time up-front or sorted only during printing. There are perceived performance tradeoffs here. If sorted during loading, the loading process will be slow, but the printing process would be fast. If sorting is held until printing time, the loading would be fast, but printing quite slow.

Hash table – using the course ID as a hash-key in an appropriately sized table will inherently sort the data by course ID, and would not take any additional time outside of computing the hash key. This will be inherently performant.

The Binary tree method is inherently sorted as the tree is created, and so long as the data is not pre-sorted before input, the tree will have a minimal number of levels making overall traversals very quick.

**Searching**

Vector Table – Searching via vector table will inherently always be slow regardless of the data being pre-sorted or unsorted as the average seek time will be N/2 nodes processing time. This is because no data can be eliminated until the desired node is located.

Hash Table – Searching via hash table will inherently fast as the courseID is used as the hash key, meaning the first node searched will always be the one requested.

Binary Tree – Searching via the binary tree will be slower than the hash table, but will be quite fast overall assuming the data is unsorted prior to entry into the tree structure. Presorted data will have similar performance to the vector table which would eliminate the efficiency gains possible from the Binary tree methods.

**Sorting**

Vector Table – the vector table will inherently always require sorting of some nature on the data either during loading or printing cycles. This will be preferably spent during loading the data as it would be a 1 time penalty vs potentially a repeated penalty (if the sorted data is not used to replace the unsorted data).

Hash Table – By using the course ID as the hash-key, the data is sorted automatically during entry into the table without any penalty in performance. It takes equivalent time to insert into memory A as it does to memory B.

Binary Tree – The Binary tree is inherently sorted during insertion, but does require some re-entrant cycles that could add small additional cycles during entry, but would be better performing than the vector table.

**Recommendations**

Overall, I prefer the use of the hash table as the performance will be high regardless of the tasks being performed ensuring the operator has a smooth predictable experience. While the hash table as defined in the analyses of having 3\*26\*3\*10 key indexes will consume more memory locations than either of the vector table or the binary tree, the overall architecture is cleaner to understand and thus easier to implement and maintain and memory while not unlimited, is not a significant consideration for the potential dataset in this exercise.

**Vector List Pseudo Code**

ParseCSV ()

Read in file contents to memory

For each line

Check string formatting () // String, string, <optional strings in csv format>

For each prerequisite, check for the courseID to exist in the file contents

test each preReq by searching the entire file for a courseID that matches the prereq value

if prereq exists

parse the data into a course(struct) object

add new class to the named list or replace data based on how to handle duplicates

first field goes to course.courseID field

second field goes to course.courseName field

remaining data is the prerequisite list in CSV format

count the number of commas and set class.numPrereq

Add/replace course in vector list

else prereq does not exist

print error and stop

printCourse(courseID){

starting at vector head node

while node != null

if course.CourseID == courseID

//pretty print the information as required by customer specs

return

Else

Node = node.next

printCourses()

starting at vector head node

while node != null

//pretty print the information as required by customer specs

Node = node.next

**HASH Table Pseudo Code**

HashTable::parseCSVData()

Read in file contents to memory

For each line

test each preReq by searching the entire file for a courseID that matches the prereq value

if prereq exists

parse the data into a new course(struct) object

first field goes to course.courseID field

second field goes to course.courseName field

remaining data is the prerequisite list in CSV format

count the number of commas and set class.numPrereq

compute the hash key of the new course based on courseID

Look up the key in the hash table

append new course to the hash table list at the given key

else prereq does not exist

print error and stop

HashTable::printCourse(courseID){

//compute the hash key of the courseID

//use the key to look in the hash table

//search and find the course by ID in the array

//pretty print the information as required by customer specs

HashTable::printCourses(hashtable){

For each hashtable key

Sort vector alphabetically

Print each vector element

**Binary Tree Pseudo Code**

BinaryTree::parseCSV(file)

Check file input path for validity, if blank use default location

Openfile from disk as read only

For each line

test each preReq by searching the entire file for a courseID that matches the prereq value

if prereq exists

parse the data into a new course(struct) object (how to handle duplicates?)

first field goes to course.courseID field

second field goes to course.courseName field

remaining data is the prerequisite list in CSV format

count the number of commas and set class.numPrereq

Insert the course into the binary tree using courseID sorted alphanumerically

else prereq does not exist

print error and stop

BinaryTree::printCourses(root){

While root != null

if (root->left)

printCourses(root->left);

cout << courseID << “, “ << courseName << “, “ << CoursePreReq << endl;

if (root->right)

print(root->right);

}

BinaryTree::printCourse(courseID, & course){

While course != null

// check for a match

If (course->CourseID == courseID)

cout << courseID << “, “ << courseName << “, “ << CoursePreReq << endl;

return

// target is lower

if ( (course->CourseID) < (courseID) )

printCourse(course->left);

//target is higher

else (course->right)

printCourse(course->right);

}

**Menu Pseudo Code (assuming Binary Tree implementation)**

Start program

While true

Cout << “ 1: Load data file” << endl;

Cout <<” 2: Print alphanumerically sorted list” <<endl;

Cout <<” 3: print course title and preReqs” <<endl;

Cout <<” 9: exit” <<endl;

Get user input<>

Switch (<user input>

Case “1” – ParseCSV (command line argument);

Case “2” – BinaryTree::PrintCourses()

Case “3” –

Cout << “enter Course ID : << endl;

Get user input

binaryTree::PrintCourse(<user input>)

Case “9” – exit(0)

default: cout <<”invalid input”<<endl;