Ethan Klukkert

Professor Andujar

CS-300

9 October 2022

Project One: Data Structure and Runtime Analysis

This analysis includes sections of pseudocode for opening a file, reading it, loading it into each data structure, and a section for evaluating each data structure. Each data structure has a section where the structure is created, course objects are loaded with the course data, and then printed with the courses in alphanumeric order. Then, there is a Menu Pseudocode section that specifies the menu functionality that would work dynamically for each data structure. At the end of the pseudocode section is pseudocode for sorting vectors, which can be used in vectors and hash tables, but not needed for binary search trees. Lastly, there will be an evaluation section of each data structure’s ability to search for a course and print its information, as well as how fast it can insert items. There is an Advantages and Disadvantages section that describes the data structure in more detail. The following pseudocode is written where each tab is a section (or scope) of code.

**PSEUDOCODE**

**Opening the File:**

FUNCTION VerifyCourses:

Open file in read mode

While the end of file is not reached:

Readline for prerequisites after name comma:

For each prerequisite until newline char:

Store in prereqList

If error in readline:

Throw error

Return to menu

While end of file is not reached and prerequisites exist in prereqList:

Find whether prerequisites exist by searching by courseNumber:

Found = 0

For each prerequisite in prereqList:

If courseNumber == currentCourseNumber:

found++

Go to Next Line

If Found == prereqList length:

Go to Next line starting at step 2

Else:

Return “course prerequisite not found in file for this course”

Go to Next line starting at step 2

Close file

**Vector Pseudocode:**

CLASS Course:

String courseNumber

String name

Array prereqList

FUNCTION createCourseObjects:

Open file in read mode

Initialize Courses List

While end of file is not reached:

Readline

Create course object:

courseNumber = string course number before comma

Name = string after comma

Initialize object -> course(courseNumber, Name)

For each item after second comma until newline char:

Append prerequisite course number to object’s prereqList

Append course object to Courses List SPECIFY THIS

Go to next line

FUNCTION searchAndPrint(givenCourseNumber):

For each course in Courses List:

1a. If courseNumber == givenCourseNumber:

Print Course Number

Print Course Name

For each prerequisite in Course prereqList:

Print Course Number

Return to menu

Print “Course not found.” Return to menu

**Hash Table Pseudocode:**

CLASS Course:

* 1. String courseId
  2. String courseName
  3. Array prereqList = []

CONSTRUCTOR Course(courseId, courseName, prereqList)

* 1. this->courseId = courseId
  2. this->courseName = courseName
  3. this->prereqList = prereqList

CLASS HashTable:

(default constructor)

Vector<Node> nodes // has nodes in vector

Node // nodes have key and courseObject

Key

courseObject = default constructor

METHOD Hash(int key):

Return index using hash function on key

METHOD Insert(int key, course):

hashIndex = Hash(key)

Append course at hashIndex’s vector

FUNCTION createCourseHashTable:

courseHashTable = HashTable() // default constructor

For each line in file:

String courseId = courseIdInFile

String courseName = courseNameInFile

Array prereqList = []

For each prereqCourse after courseName:

Append prereqCourse to prereqList

newCourse = Course(courseId, courseName, prereqList) // create Course object

Insert(courseId, newCourse) ANALYZE THIS

FUNCTION printCourses:

For each node in courseHashTable:

Print Course object Id, name, and prereqList

FUNCTION searchAndPrint(givenKey):

indexToSearch = Hash(givenKey)

go to hash index at indexToSearch:

For each course in vector at indexToSearch:

if givenKey == courseNumber:

print courseName and courseNumber

for each coursePrereq in prereqList:

print coursePrereq

Return

Print “course not found” and Return

**Binary Search Tree Pseudocode:**

CLASS course:

String courseNumber

String name

Array prereqList

FUNCTION createBinaryTree():

for each line in file:

read courseName, courseId

for each prereq after comma:

append prereq to coursePrereqList

create course object from id, name, and list

insert object into binary search tree

FUNCTION printCourses()

traverse binary search tree in order:

print course name, id, and prereq list

FUNCTION searchAndPrint(courseNumber)

Current is root

While current is not null:

if current matches courseNumber:

print courseName and prereqList

return

else if courseNumber is less than current’s courseNum:

current is now current’s left child

else courseNumber is greater:

current is now current’s right child

If current is null:

print “course not found”

Return

**Menu Pseudocode:**

While userAnswer is not 0:

Print “Choose and option:”

Take user input in userAnswer

Switch userAnswer:

Case 1:

load data by calling loadCourses() using the data structure’s insert method

Case 2:

printCourseList() based on data structure’s traversal method

Case 3:

findAndPrintCourse() based on data structure’s search and print method

Case 0:

Exit() set userAnswer to 0 and/or break

default:

Exit() set userAnswer to 0 and/or break

**Pseudocode for Sorting Vectors:**

FUNCTION selectionSort(vector)

For (i = 0; i < vector.size() - 1; i++):

Minimum\_index = i

For (j = Minimum\_index + 1; j < vector.size(); j++)

If jth bid title is less than Mimimum\_index item’s bid title, then:

The new Minimum\_index = j

Swap ith and minimumIndex item to move minimum value to minimumIndex pos

FUNCTION Partition(vector, startIndex, endIndex)

Done = false

lowIndex = startIndex

highIndex = endIndex

Find Midpoint with: midpoint = lowIndex + (highIndex - lowIndex) / 2

Find Pivot bid title with: pivot = vector.at(midpoint).title

While Not Done:

While element at lowIndex is less than the pivot (aka, loop until greater is found)

Check next element: lowIndex += 1

While element at highIndex is greater than pivot (aka, loop until lesser is found)

Check next element: highIndex -= 1

If no more elements left: lowIndex >= highIndex:

Done = true

Else: swap low and high elements so that lesser elements are left of pivot, and greater are on right of pivot.

Check next set of elements: lowIndex += 1

highIndex -= 1

When done, return highIndex int value

QuickSort(vector, startIndex, endIndex)

Base case: if array has 1 or 0 elements, return.

lowEndPartitionIndex is equal to return value from Partition(vector, startIndex, endIndex)

lowEndPartitionIndex = Partition(vector, startIndex, endIndex)

Recursively call Quicksort to sort lower and higher partitions

QuickSort(vector, startIndex, lowEndPartitionIndex)

QuickSort(vector, lowEndPartitionIndex, endIndex)

**EVALUATION**

**Vector Evaluation:**

**Insertion**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Append(course) | **1** | **1** | **1** |
|  |  |  |  |
|  |  |  |  |
| **Worst Case** |  |  | **O(1)** |
| **Worst Case Big O** |  |  | **O(1)** |

**Searching**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| For (int i = 0; i < n; i++) | **2** | **1** | **2** |
| If (vector.at(i) == nameGiven) then return | **1** | **n** | **n** |
|  |  |  |  |
| **Worst Case** |  |  | **O(n + 2)** |
| **Worst Case Big O** |  |  | **O(n)** |

**Advantages and Disadvantages:**

The advantage of the vector is that the insertion speed is constant when appended to the end of the vector. This, however, has a drawback for sorting where if the vector was sorted, it will have to be sorted again. If the vector uses prepending, then the speed and sorting is lost, where the vector must shift “n” elements down the allocated memory location to add an element to the beginning of the vector. The vector does have a strength in saving memory, where the allocated memory is equal to the number of elements “n”.

This data structure can be considered “slow” since the average speed is at O(n).

**Hash Table Evaluation:**

**Insertion**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Index = Hash(givenKey) | **2** | **1** | **2** |
| HashTable.at(Index).Append(course) | **2** | **1** | **2** |
|  |  |  |  |
| **Worst Case** |  |  | **O(4)** |
| **Worst Case Big O** |  |  | **O(1)** |

**Searching**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Index = Hash(givenKey) | **2** | **1** | **2** |
| VectorToSearch = HashTable.at(Index) | **2** | **1** | **2** |
| For (int I = 0; I < n; I++) | **2** | **1** | **2** |
| If (VectorToSearch.at(I) == nameGiven) then return | **1** | **n** | **n** |
|  |  |  |  |
| **Worst Case** |  |  | **O(n + 6)** |
| **Worst Case Big O** |  |  | **O(n)** |

**Hash Tables can have O(n), where n is number of courses stored per bucket. OR, Hash Tables can have only Indexes for courses. Then, the Big O is O(1) (needs more memory).**

**Advantages and Disadvantages:**

Hash tables have a constant insertion speed where the index of the allocation location is found, and the element is simply appended to the "bucket’s” vector. In a different case, *chaining* is used to account for collisions that may occur when the hash function returns the same index across different values. With this solution, if there is only one bucket, then the speed of searching is the same as the vector above, O(n). If several buckets are made, which would make more practical sense, then the search speed is dependent on how many elements are in each bucket, say k elements. Then the speed could be a constant value of 5 for example, and thus the speed would be O(1). However, this can come with an unnecessary cost in memory where several buckets may not have any elements attached to them. Another disadvantage is when the hash table needs to be sorted alphanumerically, each bucket must be sorted then “merged” or printed accordingly, which can be slower than a vector and certainly a binary search tree.

This data structure can be considered “fast” since the average speed can be lower than O(n).

**Binary Search Tree Evaluation:**

**Insertion**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Current = root | **1** | **1** | **1** |
| If current < course AND current.leftChild() == null | **2** | **1** | **2** |
| Current.leftChild() = course | **1** | **1** | **1** |
| Else if current < course AND current.leftChild() != null | **2** | **1** | **2** |
| Current = current.leftChild() (continue recursion at top) | **2** |  | **2()** |
| (This recursion can happen for right side, so other operations are skipped depending on the condition in the second line) |  |  |  |
|  |  |  |  |
| **Worst Case** |  |  | **O(2 + 6)** |
| **Worst Case Big O** |  |  | **O()** |

**Searching**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Current = root | **1** | **1** | **1** |
| While current != null | **1** |  |  |
| If current == nameGiven: then return current | **1** |  |  |
| Else if current < nameGiven | **1** |  |  |
| Current = current.rightChild() | **1** |  |  |
| Else current = current.leftChild() | **1** |  |  |
|  |  |  |  |
| **Worst Case** |  |  | **O(5 + 1)** |
| **Worst Case Big O** |  |  | **O()** |

**Advantages and Disadvantages:**

The time complexity of above actually specifies how many comparisons are needed for searching/inserting an element in a perfect binary search tree. “n” is the number of elements in the tree, and the result would be the *maximum* number of comparisons needed to do the operations if the tree is perfectly balanced. For example, for 500 elements, the maximum number of operations needed to find/insert an item is , which is about 9 times. This is much faster than the O(n) time, where 500 comparisons are possibly needed. A small advantage of the binary search tree is that the memory allocation is reasonable, and only a pair of null values are needed for the leaves of the tree. However, this can become unreasonable compared to a hash table, where the leaves are allocating more wasted memory compared to the hash table’s wasted memory when using chaining. The best advantage the binary search tree has is that the sorting is built into the insertion operations. A program only has to traverse the tree “in order” for it to immediately start printing sorted values.

The disadvantage of the binary search tree is that it is very unlikely for the tree to be balanced for the runtimes to be as fast as mentioned above. The memory allocation and speed can be compromised by the order of which the elements are loaded into the structure. The speed can end up being just as fast as a vector, but also have unused memory in the form of null values reserved by elements in the unbalanced tree. So, this data structure can be considered “fast”, but only under ideal conditions, which are not guaranteed.

**RECOMMENDATION**

For creating the structure, all the data structures have a loading O(n) result, due to the nature of loading the data into them. The data must be loaded piece by piece going through each line in the file, which depends on the number of courses, in this case, n. However, searching and printing a specific course is a different scenario, where some of these structures will have a different advantage and disadvantage.

I recommend using a hash table for the requirements of printing in alphanumeric order and searching for a course object. If a vector is used, the comparisons can be as bad as the number of courses, n. The memory of a vector could be better, but a hash table’s search speed can make up for the sacrifice in memory. If a binary search tree is used, the tree may not be ideal for it to accommodate an O() speed. Even if it was ideal, the speed of searching a hash table would only depend on the hash function, which can be constant speed (like a modulus operation), and the elements allowed per bucket in the hash table can be smaller than if enabled. The memory allocation wasted by a binary search tree could be as great as the last leaves of the tree, times two (each leaf has two null pointers). The memory allocation wasted by a hash table can be equivalent to the gaps in the key values, where it affects the placement location received by the hash function. For example, if key’s range from 1-3 and 7-9, then a modulus 10 function would have unassigned locations 4-6, leaving them wasted. However, key values can be assigned to elements as they are loaded into the data structure, and so the gaps can be mitigated. Also, the memory allocation can be resolved by manually assigning data elements a key value with their id’s.

Lastly, sorting in a hash table can be difficult, but manageable. With the given advantages above, the hash table makes up for the loss in sorting ability. A vector can be sorted with various sorting algorithms, and that speed is dependent on the algorithm. The hash table contains vectors, but several, which may require using “merge sort” or other algorithms to sort and print the values in the data structure within a reasonable amount of time. The binary search tree may be pre-sorted, but loss in memory and speed can occur if the conditions of the loaded data are not ideal (when elements are already sorted, the tree is as bad as a vector, but has worst memory allocation). In conclusion, the hash table provides the best speed and memory allocation and is consistent across most scenarios.