**ABCU Department of Computer Science**

Data Structure Evaluation

CS 300 Analysis and Design

James Porter

**Table of Contents:**

Table of Contents 2

Problem and Information 3

Constant Declarations 4

Vector 8

Hash Table 12

Binary Search Tree 16

Solution 19

**Problem:** ABCU is looking to make it easier for their academic advisors to find and print off the required classes and their prerequisites. To accomplish this ABCU’s Computer Science department would like to implement a program that will print a list of all the courses offered and their prerequisites.

**Information:** The following three pages include the pseudocode to run the software given the indicated data structure. The time constraints of reading the data from a file will not be discussed extensively in the discussion, because it is not the focus of this evaluation, and will not contribute to the runtime or memory constraints.

**Declarations that remain consistent independent of the data structure used:**

//Course class variables

String courseAbbreviation

String courseName

Vector<String> prerequisites

//Course printCourse() function

//This function is used by all the data structures listed in the following section

Print out course with the following format “courseAbbreviation, courseName” (Optional) “,prerequisite, prerequisite, …, prerequisite”

Note: Appropriate data structure in the following pseudocode refers to either Vector, Hash Table, or Binary Search Tree. The specific information regarding these data structures will be discussed in greater detail in the next three sections where it is noted in the pseudocode below.

//Main menu

Create appropriate data structure

Switch

//Load data from file and add to data structure

Case 1:

Open file containing course info

If file did not open properly

Throw an invalid location error

Else

Do

Create a temporary course object

Read the line from the file

If the line has no commas (or other delimiter choice)

Throw an invalid line error

Else if the line has only 1 comma

Read data up to the first delimiter

Put this data to the temporary course object’s courseAbbreviation

Read data up to the end of the line

Put this data to the temporary course object’s courseName

Else

Read data up to the first delimiter

Put this data to the temporary course object’s courseAbbreviation

Read data up to the second delimiter

Put this data to the temporary course object’s courseName

Do

Read data up to next delimiter or end of line

Put this data to temporary prerequisite string

Add this data to the temporary course’s prerequisite list

While not at the end of the line

//This next line is discussed in greater detail in the next sections

Add temporary course to data structure

While not at the end of the file

//Print all courses in alphanumeric order

Case 2:

//This next section is discussed in greater detail in the next sections

Print all courses according to the appropriate data structure

//Search for and print specific course

Case 3:

//This next section is discussed in greater detail in the next sectionsd

Use a search function according to the appropriate data structure

Case Q:

Print Exit Message

Quit Program

**Case 1:** Case 1 will have a big O that is more dependent on the data structure implemented. However, besides the line adding the course to the appropriate data structure, in most cases this case will run with a big O of O(N). This is because each line in the file with the course information must be read, and there can be N number of lines in the file.

**Case 2:** Case 2 will always have a big O of O(1), because there is only ever 1 line of code read in this section. However, depending on the data structure implemented, the big O of the function could be different.

**Case 3:** Case 3 will always have a big O of O(1), because there is only ever 1 line of code read in this section. However, depending on the data structure implemented, the big O of the function could be different.

**Case Q:** Case Q will always have a big O of O(2), because there are only ever 2 lines of code read in this section.

As a side note, these cases will all run with similar time constraints no matter the data structure. Only the accompanying functions within these cases will cause different runtimes.

**Vector:**

Note: An additional sorting function would need to be called in case 1 after all the data has been read in from the file.

//Add data to the structure

Get course from file

Push course to back of vector

Note: To sort the data structure as desired (using the quicksort algorithm), there must be two functions. The first function is called partition() and is used to pick a pivot and switch elements. This function takes a vector called courseList and two integers called lowIndex and highIndex as parameters. The second of the two functions are called quicksort() and is the function that actually sorts the data. This function takes a vector called courseList and two integers called lowIndex and highIndex.

//Sort data structure (This function uses the quicksort algorithm)

//partition(courseList, lowIndex, highIndex)

Get the middle element value for the pivot (low + (high – low) / 2)

Set the pivot to the element at the midpoint of the courseList vector

Create Boolean called done and set it to false

While not done

While the value at the low index is less than the pivot

Increment the low index by 1

While the value at the high index is greater than the pivot

Decrement the high index by 1

If the low index is greater than or equal to the high index

Set done to true

Else

Swap the courses at the high index and low index

Increment the low index by 1

Decrement the high index by 1

Return high index

//quicksort(courseList, lowIndex, highIndex)

If the lowIndex is greater than or equal to the highIndex (Base case)

Return

Call the partition function using courseList, lowIndex, and highIndex as parameters

Assign the integer result of the partition function to lowEndIndex

Recursively call quicksort using courseList, lowIndex, and lowEndIndex (sort low partition)

Recursively call quicksort using courseList, lowEndIndex + 1, and highIndex (sort high partition)

//Print in alphanumerical order

For the length of the vector

printCourse() (See above)

Note: The binary search function will be used to search for the appropriate value. This function will take a vector called courseList and a string called key which is the value being searched for.

//Search for a course

Create three integer values mid, low, and high

Set mid and low to be 0

Set the high value to be the length of the vector – 1

While the high value is greater than or equal to the low value

Set the mid to be the value of high + low / 2

If the value of courseList at the mid element is less than the key

Set low to be the mid value + 1

If the value of courseList at the mid element is greater than the key

Set high to be the mid value – 1

Else

Return mid

Print message stating that the course was not found

Return null (Empty course)

**Add Data:** The runtime of adding a course to a vector has a big O of O(2) or even O(1) if implemented properly. Only 1 or 2 lines of code are needed to add a course to a vector.

**Sort Data:** For the sort function, I decided to use the quicksort function. This function is considered a fast sorting algorithm (by definition a fast sorting algorithm must operate with a runtime of O(NlogN) or better) because it has a big O of O(NlogN). This function also does not require extra memory to function unlike other sorting algorithms (for example the merge sort requires additional memory to run).

**Print Data:** The runtime for printing all the data in the vector has a big O of O(N) this is the worse, average, and best case of the print function. To print out every item in the vector, every item must be visited once.

**Search for a Course:** For the search function, I decided to use the binary search algorithm as it is efficient at finding values. The worst case big O of this function is O(logN) with a best case of O(1) which is quite efficient for a search algorithm. This function also doesn’t have additional memory requirements.

**Evaluation:** The vector data structure seems like good candidate for the data structure to use, because all the function run using efficient runtimes (big O). Additionally, the vector data structure has many functions that can be coded with relative ease. Lastly, the vector data structure can increase in size as more courses are added with relative ease. The downside of using the vector data structure for this project is that the code for the search and sort functions are more complex. These functions also increase the likelihood that an error or other bug be introduced into code written by the programmer.

**Hash Table:**

This function takes a key as a value and hashes it to get the bucket that it belongs to

//Hash function (Assuming the key is the alphanumeric course abbreviation)

Calculate the integer value of the string by adding all the single char values

Take that integer value and find the modulo of the hash table size

Return the modulo of the integer value with the hash table size (integer value % table size)

This function requires that the user passes a course and then it is placed in the hash table

//Add data to the structure

Given the course passed, get the course abbreviation

Take the course abbreviation and calculate the hash value (using the hash function)

If the spot is not taken in the table

Add the course to the hash table using the hash value as the element

Else

Move through the array 3 times (Move through three elements)

If an empty spot is found

Add the course to the hash table using the new element as the location

Else

Create a new array double the size of the original

For every element in the old array

Calculate a new hash key

Assign that to the new array

Replace old array with new array

Delete old array

//Print in alphanumerical order

For every element in the hash table

Put the course elements to a vector

Sort that vector using the quicksort (See vector above for details)

For every element in the vector

printCourse() (See above)

This function requires the user passes a string to search for

//Search for a course

Take the string passed and calculate the hash value (using the hash function)

Check the value of the array using the hash value as the element

If the array element matches the string passed

Return that value

Else

While the element in the array has a course

If the element in the array matches the string passed

Return that value

Else

Move to the next value in the array

Print message explaining that the value was not found

Return null (Empty course object)

**Hash Function:** This function is not totally necessary to include but makes implementing the hash table data structure a little easier implement. This function runs with a big O of O(2) and thus doesn’t add much to the runtime.

**Add Data:** The add data function is a relatively simple function to code but does have some downsides. The runtime of this function in a perfect hash table has a big O of O(1), however for this to work the hash table has to map each element to its own bucket. Additionally, all the possible keys must be known to create a function that is perfect. In a more realistic world, this hash table would have a worst-case big O of O(N). However, this function is still quite efficient and gets the job done. One more issue with this function is that if the table size needs to be increased, there needs to be more memory set aside for this function.

**Print Data:** Because this map is not organized, there are extra steps that need to be taken for me to comfortably print all the courses in alphanumeric order. In this case, the big O is O(N2logN). This is because the elements of the table must be incremented through to get them out of the table and then one more time to sort the vector. This function is the Achilles heel of the hash table.

**Search for a Course:** The search function is quite efficient and has a potential big O as efficient as O(1). This is because we are passed the string and can search the table directly and use the hash function to get right to the element that the value is in. Even if the element is not in the first element found, you don’t have to search very far in the list to get to the value or get to an empty space.

**Evaluation:** I do not recommend using the hash table to at ABCU for the computer science courses. This data structure has its pros, but they are very limited. This function can be very efficient, but a lot of variables must be perfect for this to happen, and we don’t know those variables. Additionally, the print function is not very efficient at all, even if these variables are known. It is possible that the print function could be made more efficiently, but I don’t believe that enough of the information is known to use this data structure.

**Binary Search Tree:**

//Node Structure

Course course

Node\* leftLeaf

Node\* rightLeaf

//Add data to the structure

If the root points to nothing

Add the course as the root

Else

Set the current node to be the root

While the current node is not null

If the node to adds key is less than the current key

If the current left leaf is null

Set the current nodes left leaf to be the node to insert

Set current node to null

Else

Set the current node to be the next left leaf

Else

If current right node is null

Set the current nodes right leaf to be the node to insert

Set the current node to null

Else

Set the current node to be the next right leaf

This function takes a node as a parameter

//Print in alphanumerical order (In order traversal)

If the node passed is null

Return

Recursively call this print function on the left leaf node

Print the node

Recursively call this print function on the right leaf node

This function takes a key that is a string to compare to the nodes in the tree to try and find a value

//Search for a course

Set the current node to be the root

While the current node is not null (Not empty)

If the current nodes key matches the key passed

Return the current node

Else if the current nodes key is greater than the key passed

Set the current node to be the next left leaf

Else

Set the current node to be the next right leaf

Print out message stating that the key value wasn’t found

**Node Structure:** The node structure for a binary search tree in necessary to ensure that the structure runs properly. This addition does not take much code and does not add drastically to the runtime or memory.

**Add Data:** The add function has a worst-case scenario of O(N) and a best-case scenario of O(logN). This function is quite efficient and at the worst the code just must go through all the values once. This function also allows the code to be sorted as it is inserted without being inserted in a specific order.

**Print Data:** Info here

**Search for a Course:** Like the add data function, the runtime of the search function is very efficient. The runtime of this function has a best case of O(LogN) and a worse case of O(N). This meant that in the worst-case scenario you only have to check each value once. This function also does not have any substantial additional memory needs.

**Evaluation:** The binary search tree function is quite efficient and is one of the best cases for implementing in a solution for this problem. Along with being very efficient this data structure is very simple to code and implement. To make the code easier to read, some of these functions could also be implemented using recursion. Lastly there are no substantial additional memory allocations necessary to implement using the binary search tree data structure.

**Solution:** This document has served to compare three options for data structure to ensure that the courses offered can be printed efficiently and effectively. Between the data structures we do not recommend that you implement this software using a hash table as the functions are not very efficient and there is a lot of extra code that would need to be implemented to make this program run as intended. Along with that some of the functions that would need to be performed by the hash table would require the allocation of extra memory. Either a vector structure or a binary search tree would be the recommended data structure for this project. Both structures run very efficiently with very little extra memory needed and thus are top candidates for this project. However, the binary search tree would be the candidate that I would most recommend for this project as the code would be easier to implement and write when in comparison to the vector data structure. Given this and everything else mentioned above, it is recommended that ABCU uses a binary search tree to implement their software.