Kylee Brown

CS 300

10/17/25

Project One

Create pseudocode for a menu

```
WHILE number is not equal to 9
  GET number from user
  DISPLAY a menu of options 1, 2, 3 and 9
  IF number is 1
    LOAD the data into the data structure
  ENDIF
  IF number is 2
    PRINT the list in alphanumeric order
  ENDIF
  IF number is 3
    PRINT the course number and prerequisites for a specific course
  ENDIF
ENDWHILE
OUTPUT the string “Goodbye”
EXIT program
```

Design pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order

Vector | Chosen sorting method: Merge Sort | avg. runtime: $O(Nlog_2N)$

Merge

CREATE an empty array to temporarily hold the merged course objects named tempVector

INITIALIZE position variables leftPos, rightPos and mergedPos

COMPARE the IDs in each courseVector partition and add the smallest to merged vector

WHILE left partition is not empty AND right partition is not empty

IF course ID at leftPos is less than course ID at rightPos

tempVector at mergedPos equals leftPos

INCREMENT leftPos by one

ELSE

tempVector at mergedPos equals rightPos

INCREMENT rightPos by one

ENDIF

INCREMENT mergedPos by one

ENDWHILE

ADD remaining course objects to tempVector

WHILE left partition is not empty

tempVector at mergedPos equals leftPos

INCREMENT leftPos by one

INCREMENT merge by one

ENDWHILE

WHILE right partition is not empty

tempVector at mergedPos equals rightPos

    INCREMENT rightPos by one

    INCREMENT mergedPos by one

  ENDWHILE

  COPY the tempVector back to courseVector

  FOR index in tempVector

    courseVector at index plus mergePos equals tempVector at mergePos

  ENDFOR

MergeSort

INITIALIZE j

IF i is less than k

  Midpoint j is equal to i plus k divided by two

  RECURSIVELY sort left and right partitions

  MERGE left and right partitions in sorted order

ENDIF

Print Alphanumerical list

LOAD vector with course objects

MERGE SORT the courseVector

FOR course object in courseVector

  PRINT the course ID, title

  FOR prerequisite in prerequisites

    PRINT prerequisite

ENDFOR

<u>Hash Table</u> | Chosen sorting method: Merge Sort | avg. runtime: $O(Nlog_2N)$

<u>Print Alphanumerical list</u>

LOAD the course objects into the hashtable using course ID as the key

CREATE an empty vector to hold the hashtable items named sortedVector

FOR bucket in hashTable

  ASSIGN current node to bucket

  WHILE current node does not equal a null pointer

    APPEND each course Object stored in node into sortedVector

    ASSIGN current node to the next node

ENDFOR

MERGE SORT sortedVector

FOR course object in sortedVector

  PRINT the course ID, title

  FOR prerequisite in prerequisites

    PRINT prerequisite

  ENDFOR

ENDFOR

<u>Binary Tree</u> | Chosen sorting method: Inorder Traversal | avg. runtime: $O(N)$

<u>Inorder</u>

IF the node is null

  RETURN

ENDIF

RECURSIVELY traverse the left side of tree and print sub tree nodes

PRINT course ID and title

  IF prerequisite list is not empty

    FOR prerequisite in prerequisites

      PRINT prerequisite

    ENDFOR

  ENDIF

RECURSIVELY traverse the right side of tree and print subtree nodes

BSTPrintInOrder

CALL InOrder on the root BST node

Print the BST alphanumerically

LOAD the course objects into the BST

CALL BSTPrintInOrder

Evaluatiuon

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

Runtime Analysis Chart: Reading & parsing the file			
Code	Line cost	# of times executed	Total Cost
OPEN the file	1	1	1

Runtime Analysis Chart: Reading & parsing the file			
WHILE the file end is not reached	1	n	n
IF the first two items in the file line are course ID and course title	1	n	n
STORE only the course ID into a vector named valid Courses	1	n	n
ELSE THROW ERROR "file is corrupted, not enough data provided"	1	n	n
WHILE the file end is not reached	1	n	n
CREATE a row vector that holds each file row	1	n	n
IF the file line is blank	1	n	n
THROW ERROR "file is corrupted, not enough data provided"	1	1	1
IF the first two items in the file line are course ID and course title	1	n	n
APPEND the course ID to the row vector	1	n	1
APPEND the course title to the row vector	1	n	1

Runtime Analysis Chart: Reading & parsing the file			
ELSE THROW ERROR "file is corrupted, not enough data provided"	1	n	n
CREATE an empty list to hold prerequisites	1	n	n
FOR prerequisite in prerequisite list	1	1	1
FOR prerequisite in prerequisite list	1	n	n
IF the prerequisite ID exists in valid Courses	n - *requires vector search* $O(N)$	n	n^2
APPEND the prerequisite to the prerequisite list	1	n	n
APPEND the prerequisite list to the row vector	1	n	n
APPEND the row vector to the file contents	1	n	n
		Total cost:	$n^2 + 14n + 6$
		Runtime:	$O(n^2)$

Runtime Analysis Chart: Creating course objects			
Code	Line cost	# of times executed	Total Cost
PARSE the file, retrieving the file contents vector	n^2	1	n^2
FOR row in file contents	1	n	n
CREATE an empty course object	1	n	n
STORE the first column item in the course object as the course ID	1	n	n
STORE the first column item in the course object as course title	1	n	n
STORE the third column item in the course object as prerequisites	1	n	n
		Total Cost:	n^2+6n
		Runtime:	$O(n^2)$

Explain the advantages and disadvantages of each structure in your evaluation

There are many advantages and disadvantages to certain data structures. A vector can store an ordered list of data and allows for out-of-range checking when accessing items at specific indices. Vectors are the better alternative to arrays because of their range checking, and they also allow for simple indexing in constant time; however, searching, inserting, and removing from the vector generally requires a traversal over N items.

A hash table uses buckets to store unordered data by hashing the item to a location in an array. Hash tables have two implementation methods: open-addressing and chaining. With open-addressing, the tables' buckets never exceed 1, and probing can be done to manage collisions upon insertion. With chaining, each bucket contains a pointer, and to manage collisions, the item is appended to the bucket's linked list. When using open addressing, the amount of space in memory taken up by the operations is much lower because a pointer does not take up unnecessary space. However, chaining allows the table to innately hold more before triggering a resize, which is ideal for big loads of data. When it comes to inserting and removing, the runtimes aren't much better than vectors; however, searching for an item in a hashtable has a runtime $f(N) \approx O(1)$ that is extremely efficient and is the main reason programmers choose hash tables.

Lastly, a binary search tree is used to store hierarchical data such that the left side is ≤ the root node and the right side is ≥ the root node, and each node has up to two children. In a balanced binary tree, insertion, removal, and search operations are performed at an average runtime of $O(log_2 N)$. If a programmer needs to hold data that will need to be displayed alphanumerically, a simple inorder traversal can be done, saving runtime and space that would otherwise be used on a sort function.

make a recommendation for which data structure you plan to use in your code

After evaluating each data structure, I have concluded that a binary search tree would be the best data structure for ABCU. A binary search tree is the most ideal for this project because I am required to print the list alphabetically. A vector would require a sorting method that would be costly in space and runtime complexity, and hash tables are inherently meant for fast insertions/lookups, which is why, to print the hash table in order, I would need to copy the contents to a vector, which would again require a costly sorting method.

With a binary search tree, I can print the tree alphabetically by traversing the tree in order, which overall uses less memory space since I do not have to call a function to sort or copy any items into another data structure. Lastly, I can perform searches with an average run time $log_2 N$, which is faster than a vector ($O(N)$) but slower than a hash table ($O(1)$). Loading course objects into a BST would require a runtime of $O(log_2 N)$. If this project were only concerned with the ability to load, search, and print data, a hash table would be the ideal choice because the runtimes are close to constant. However, because there is a focus on the ability for the list to be displayed in order as well as searching and displaying an item, a BST is the best choice.