**Project One: Pseudocode and Runtime Analysis**

Alex Baires

Southern New Hampshire University

CS-300-H2999 DSA: Analysis and Design

Professor Nathan Lebel

December 3rd, 2023

**Pseudocode**

**For all Data Structures – Pseudocode to Open and Parse through File:**

**//** The below process will be utilized for both file operations and error-checking in all three of the prospective data structures under consideration.

START the program.

INVOKE *checkDataIntegrity()* function with the file path as an argument.

INITIALIZE a vector called *courseParameters*

INVOKE *loadDefaultCourseList*() with the file path as an argument.

INITIALIZE a vector of strings called *defaultCourses.*

INITIALIZE a vector of strings called *temp*.

CREATE ifstream object called *CourseData*.

OPEN the *ABCU\_Advising\_Program\_Input* text file.

IF the file does not open:

DISPLAY an error message: “File not found; please check the directory”

RETURN an empty vector.

WHILE we have not reached the end of the file:

CLEAR the *temp* vector.

RETRIEVE each line from the file and store into a string variable: *word.*

APPEND each word to the *temp* vector.

APPEND the first word of the *temp* vector to the *courseParameters* vector.

// This will maintain a list of courses in the catalog as a base case.

CONTINUE for all lines in the file.

CLOSE the file.

RETURN the *defaultCourses* vector for use in the *checkDataIntegrity* function.

CREATE ifstream object called *CourseData*.

REOPEN the *ABCU\_Advising\_Program\_Input* text file.

IF the file does not open:

DISPLAY an error message: “File not found; please check the directory”

RETURN an empty vector.

WHILE we have not reached the end of the file:

CLEAR the *tokens* vector.

RETRIEVE each line from the file and store into a string variable: *line*.

TOKENIZE each line of the file by pushing each word separated by a comma into the *tokens* vector.

IF the vector size is less than 2:

DISPLAY “Error: line does not have at least 2 parameters”

BREAK.

IF the size of the temporary *tokens* vector is greater than 2:

ITERATE from the third element to the end of the vector and check if they are courses in the *courseParameters* vector.

IF true:

USE *tokens* to create *Course* object.

ELSE:

DISPLAY “Prerequisite course does not exist in catalog.”

BREAK

CONTINUE until all CSV lines are parsed.

CLOSE the file.

**VECTOR Pseudocode – Create course objects and store in the vector.**

CREATE a *Course* STRUCT object. // I prefer a struct for a plain old data object.

INITIALIZE a string called *courseID.*

INITIALIZE a string called *courseName.*

INITIALIZE a vector of strings called *coursePrerequisites.*

INITIALIZE a vector of *course* objects called *Courses.*

REOPEN the *ABCU\_Advising\_Program\_Input* text file.

IF the file is open:

DISPLAY “Course list file successfully opened.”

WHILE the file is good and has not reached the end:

Create an istringstream object to store the split words from the line.

INITIALIZE a string called word to store each word in the line.

INITIALIZE a blank *Course* object.

.// Since each line should be of the same formatting, the below pseudocode applies:

IF there is a first word from the comma-separated line,

ASSIGN to the *courseID* member of the *Course* object.

IF there is a second word from the comma-separated line,

ASSIGN to the *courseName* member of the *Course* object.

FOR each additional word in the line:

Append to the *coursePrerequisites* member vector of the *Course* object.

IF there are no additional words (pre-reqs):

ASSIGN null to the *coursePrerequisites* member of the *Course* object.

PUSH BACK the brand-new *Course* object to the *Courses* vector.

ELSE: // The file failed to open.

DISPLAY “open failed, check if the csv file is in the correct directory.”

CLOSE the file.

**VECTOR Pseudocode – Search the vector for a course and print its relevant information and prerequisites**

INITIALIZE a string called *searchCourseID*.

INVOKE printCourseInformation using the *courses* vector and the *searchCourseID* as arguments.

FOR each course in the *courses* vector:

IF the course *courseID* is equal to the *searchCourseID*:

DISPLAY all course information: courseID and name.

FOR each prerequisite of the course:

DISPLAY all the prerequisite course information.

ELSE:

DISPLAY “Course not found.”

**HASH TABLE Pseudocode – Create course objects and store in the hash table.**

***//*** *The below pseudocode will run after the file has been parsed and its contents are inside the*

*// “courses” vector of vectors.*

CREATE a *Course* STRUCT object. // I prefer a struct for a plain old data object.

INITIALIZE a string called *courseNumber.*

INITIALIZE a string called *courseName.*

INITIALIZE a vector of strings called *coursePrerequisites.*

CREATE a default *course* constructor initialized to an empty set.

DEFINE a *HashTable* class object.

Private methods:

DEFINE a NODE struct:

DECLARE a default COURSE object called course.

INITIALIZE a key.

INITIALIZE a NODE pointer called *next* pointed to null.

INITIALIZE a vector of node objects called *nodes.*

CREATE a *hash* member function that utilizes an algorithm to generate a hash key from the table size.

DEFINE a new HashTable called *courseTable.*

REOPEN the *ABCU\_Advising\_Program\_Input* text file.

WHILE we have lines in the file to parse through:

INITIALIZE a *parameters* vector (temporary).

Create an istringstream object to store tokenized words.

INITIALIZE a string called *word* to store each course parameter from the file.

INITIALIZE a blank *Course* object.

WHILE there are characters to read from the stream (line by line), each parameter will be stored into the *word* variable using the comma as a delimiter.

PUSH BACK each word into the *parameters* vector.

// Constructing the *Course* object for each course in the file…

ASSIGN the first element of the *parameters* vector to *Course* object’s *courseID* member.

ASSIGN the second element of the *parameters* vector to the *Course* object’s *courseName* member.

IF the *parameters* vector has greater than 2 elements:

Append to the *coursePrerequisites* member vector of the *Course* object.

INVOKE the *HashTable* *Insert()* function:

GENERATE a key by using the *hash* method with the *Course’s* *courseID* string converted into an integer as an argument.

RETRIEVE a hash table node using the key.

IF there is no entry within the hash table associated with the key:

CREATE a new node using the course and key as arguments and insert into the *courseTable* hash table at the key index.

ELSE:

IF there is an entry for the node found, but it has not been used:

Initialize the course node using the *Course* object information.

POINT the *next* pointer to NULL (because there is no collision).

ELSE:

*// COLLISON!*

Iterate through each node associated with the key by utilizing the next pointer.

Once we find a null pointer, append the new node to the linked list associated with the key.

CONTINUE until all lines of the file have been tokenized and converted into *Course* structs and inserted into the *courseTable*  hash table.

CLOSE the file when finished.

**HASH TABLE Pseudocode – Search the hash table for a course and print its relevant information and prerequisites.**

DEFINE a function that returns void called *printCourseInformation* with a hash table and *courseNumber* string as arguments.

INPUT a course number string.

INVOKE the *printCourseInformation* function with the *courseTable* and *courseNumber* string as arguments.

INVOKE the search function.

CREATE an empty course.

CONVERT the *courseNumber* string into an integer.

RETRIEVE the memory address of the node associated with the key and POINT to it.

IF no node entry is found for the key:

RETURN an empty course.

IF a matching node entry to the *courseNumber* is found as the first element associated with the key:

RETURN the course object associated with the key.

// *otherwise…*

WHILE the node is not null:

IF a matching key is found:

RETURN the course object associated with the key.

CONTINUE to iterate through the linked list of nodes associated with the key until it matches the *courseNumber.*

RETURN a *course*.

*//With the matching course found..*

DISPLAY the course’s course number, and course name.

FOR each prerequisite within the *coursePrerequisite* vector:

DISPLAY the prerequisite course information.

**TREE Pseudocode – Create course objects and store in the tree.**

CREATE a *Course* STRUCT object. // I prefer a struct for a plain old data object.

INITIALIZE a string called *courseNumber.*

INITIALIZE a string called *courseName.*

INITIALIZE a vector of strings called *coursePrerequisites.*

CREATE a default *course* constructor initialized to an empty set.

CREATE a *Node* STRUCT object.

Consisting of:

A *Course* object.

A *Node* pointer to the left child.

A *Node* pointer to the right child.

INITIALIZE both pointers to null.

INITIALIZE the node with a given bid.

DEFINE a *BinarySearchTree* class object.

CREATE a *Node* pointer to the root of the tree.

// Because the courses have been checked for proper formatting, they are ready to load.

REOPEN the *ABCU\_Advising\_Program\_Input* text file.

WHILE we have lines in the file to parse through:

INITIALIZE a *parameters* vector (temporary).

Create an istringstream object to store tokenized words.

INITIALIZE a string called *word* to store each course parameter from the file.

INITIALIZE a blank *Course* object.

WHILE there are characters to read from the stream (line by line), each parameter will be stored into the *word* variable using the comma as a delimiter.

PUSH BACK each word into the *parameters* vector.

// Constructing the *Course* object for each course in the file…

ASSIGN the first element of the *parameters* vector to *Course* object’s *courseID* member.

ASSIGN the second element of the *parameters* vector to the *Course* object’s *courseName* member.

IF the *parameters* vector has greater than 2 elements:

Append to the *coursePrerequisites* member vector of the *Course* object.

INVOKE the *insert* member function of the binary search tree using a course object as the argument.

IF the tree is empty, this course will be the root node.

ELSE the tree already has nodes.

INVOKE the *addNode()* function with the tree root node and new course object as arguments.

IF the course number is less than the root node:

// Traverse the left side

IF there is no left child node, insert the new course object as the left child of the parent node.

ELSE recursively call *addNode()* using the left child node and new course object as arguments.

We continue to call addNode until we find a node that has no left child node. In this case, we satisfy the IF statement and insert the new course object as a left child of the parent node.

IF the course number is greater than the root node:

// Traverse the right side

IF there is no right child node, insert the new course object as the right child of the parent node.

ELSE recursively call *addNode()* using the right child node and new course object as arguments.

We continue to call *addNode* until we find a node without a right child node. In this case, we satisfy the IF statement and insert the new course object as a right child of the parent node.

CONTINUE until all lines of the file have been tokenized and converted into *Course* structs and inserted into the *BinarySearchTree* tree class.

CLOSE the file when finished.

**TREE Pseudocode – Search the tree for a course and print its relevant information and prerequisites.**

DEFINE a function that returns void called *printCourseInformation* with the binary search tree and *courseID* string as arguments.

INPUT a course ID string.

INVOKE the *printCourseInformation* function with the *courseTable* and *courseID* string as arguments.

INVOKE the search function.

CREATE a current node pointer to the root of the BST.

WHILE the current pointer is not null:

IF we find a matching course ID inside of the tree:

RETURN the course object information.

IF the course ID is smaller than the current root node, traverse down the left side of the tree.

ELSE if the course ID is larger than the current root node, traverse down the right side of the tree.

CONTINUE until we find a matching course ID.

IF we do not find a matching course ID:

DISPLAY “Course ID does not exist.”

RETURN an empty course object.

// If we return a matching course object

DISPLAY the course’s course number, and course name.

FOR each prerequisite within the *coursePrerequisite* vector:

DISPLAY the prerequisite course information.

**ALL DATA Structures – Pseudocode to Display the Menu and User Choices**

INITIALIZE an integer variable named *userChoice* to 0.

WHILE the *userChoice* is not equal to 4:

DISPLAY a decorative string of characters “\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*.”

DISPLAY “ABCU Course Catalog” on the next line.

DISPLAY another decorative string of characters “\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*.”

NEWLINE.

DISPLAY “Main Menu.”

DISPLAY “1 – Load Courses into System Data Structure.”

DISPLAY “2 – Print Course List in Alphanumerical Order”

DISPLAY “3 – Print a Specific Course.”

DISPLAY “4 – Exit the Program.”

DISPLAY “Select an Option 1 through 4.”

INPUT a choice.

IF the user selects choice 1:

INVOKE the *loadCourseInformation()* function with the path to the *CourseList* CSV or text file and the specific data structure object (vector, hash table or tree) as arguments.

IF the user selects choice 2:

VERIFY that the data structure has *Course* objects.

IF true:

INVOKE the *printSampleSchedule()* function with the data structure object as an argument.

ELSE:

DISPLAY “No courses found, please load a Course List into the system.”

BREAK.

IF the user selects choice 3:

VERIFY that the data structure has *Course* objects.

IF true:

INVOKE the *printCourseInformation()* function with the data structure object and *courseNumber* as arguments.

ELSE:

DISPLAY “No courses found, please load a Course List into the system.”

BREAK.

IF the user selects choice 4:

DISPLAY “Exiting the Program.”

BREAK out of the loop

EXIT program.

IF the user selects any other number:

DISPLAY “Invalid Input.”

CONTINUE the loop to ask for new user input.

**Pseudocode – Print All Courses in Alphanumeric Order**

**VECTOR: // I utilized some of my pseudocode from the 2-3 Assignment.**

// Before printing, we will sort the vector using the Quicksort algorithm. Our first step is to create the partition algorithm that it will recursively call.

DEFINE a *partition* function with the *courses* vector and two integer variables, *begin* and *end (*aka. the indices of the beginning and end of the sub vector to be sorted) as arguments that return the highest index of the low partition.

CALCULATE the middle index as the pivot by adding the *begin* index to the result of subtracting the *begin* from the *end* index and then dividing the answer by 2.

// Since we are sorting by the *courseNumber*

SET the pivot to the string *courseNumber* of the *Course* object at the midpoint of the *courses* vector.

WHILE true. // Intentional infinite loop that exits with a break statement triggered by a conditional

WHILE the *Course* element at the left index is less than the pivot:

INCREMENT the left index.

WHILE the *Course* element at the right index is less than the pivot:

DECREMENT the right index.

IF the left index reaches or passes the right index:

BREAK out of the loop.

ELSE:

Both while loops for the left and right indices stop because the conditional is no longer met.

SWAP the *Course* object at the left index and the *Course* object at the right index.

MOVE both low and high indices closer together by incrementing and decrementing them respectively.

CONTINUE WHILE TRUE until we meet the BREAK statement.

RETURN the highest index of the left partition.

// Quicksort

DEFINE the *quicksort* function with the *courses* vector and two integer variables called *begin* and *end* which represent the two extremes of the vector to be sorted.

IF the subarray has 0 or 1 elements: // Base case!

RETURN // Quicksort completed.

DECLARE the mid (the midpoint variable) and SET as the RESULT of the *partition() function* (again, using the *courses* vector and its two extreme indices as arguments).

INVOKE the *quicksort()* function with the *courses* vector, the first element and the midpoint as arguments, to recursively sort the low partition less than the midpoint.

INVOKE the *quicksort()* function with the *courses* vector, the midpoint incremented by 1, and the last element as arguments, to recursively sort the high partition greater than the midpoint.

// Once the bids are sorted..

INVOKE the *printSortedCourseList()* function with the *courses* vector as an argument:

FOR each *course* object in the *courses* vector:

DISPLAY the *courseID*.

DISPLAY the *courseName.*

FOR each element in the *Course* object’s member vector, *coursePrerequisites*:

DISPLAY each *coursePrerequisite.*

**HASH TABLE:**

// Though not necessarily sorted, we can ensure that the courses are inserted in alphanumerical order by utilizing a good hash function to minimize collisions.

INVOKE *HashMultiplicative*() function with the *courseID* string as an argument.

SET the initial value to 5381. // Bernstein’s hash function from Zybooks 5.7

FOR each character in the *courseID*:

CALCULATE the string hash by multiplying 5381 by the hash multiplier 33 and adding it to the ASCII value of the character.

RETURN the remainder of the string hash divided by the hash table size.

// Printing the Courses…

INVOKE *printSortedCourseList()* member function of the hash table.

FOR each node in the *nodes* vector:

IF the current node’s hash key is used:

DISPLAY the *courseID, courseName* and iterate through the *coursePrerequisites.*

WHILE the current node’s next pointer is not equal to null (checking the linked list if there are any collisions and outputting those courses).

DISPLAY the *courseID, courseName* and iterate through the *coursePrerequisites.*

**TREE:**

// Binary search trees are constructed so that a root node is created at the top. Each of the descendants to the root’s left are less than the root, and those to the right are greater than the root. With this, we can use in-order traversal to call each node in sorted order.

INVOKE the *printCoursesInOrder()* member function of the binary search tree.

IF the node is null:

RETURN.

Recursively CALL the *printCoursesInOrder()* for the left child node.

The function is called until the base case of a node that has a null left child is satisfied.

DISPLAY the *courseID, courseName* and iterate through the *coursePrerequisites.*

CALL the *printCoursesInOrder()* for the right child node.

IF the right child node does not exist, go up one node level in the call stack and DISPLAY that node’s course information before calling the *printCoursesInOrder()* for the latest node’s right child.

Continue moving up node levels in the tree and recursively calling the *printCoursesInOrder()* function until we DISPLAY the root node and then traverse the right side of the tree.

**Evaluation:**

|  |  |  |  |
| --- | --- | --- | --- |
| **For all Data Structures - Opening and Parsing through the File Using a Vector (ERROR CHECKING/DATA CLEANING)** | | | |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Invoke *loadDefaultCourseList* and return a vector of default courses. | 9 | 1 | 9 |
| Initialize a *courseParameters* vector and assign as vector of defaults. | 1 | 1 | 1 |
| Initialize a temporary *tokens* vector | 1 | n | n |
| Open the csv or text file. |  |  |  |
| If the file does not open: | 1 | 1 | 1 |
| Display an Error Message. | - | - | - |
| While not the end of the file: | 1 | n | n |
| Clear the *tokens* vector. | 1 | n | n |
| Getline from each line in file and store into *line variable* | 1 | n | n |
| Initialize a temporary *tokens* vector | 1 | n | n |
| Tokenize each word in the line separated by a comma and push into *tokens* | 1 | n | n |
| If the *tokens* vector size is < 2: | 1 | 1 | 1 |
| Display an Error Message. | 1 | - | - |
| ELSE | 1 | n | n |
| If the *tokens* vector size is > 3 | 1 | n | n |
| FOR every element after the 2nd element in the vector: | 1 | m | m |
| If the element is in the *courseParameters* vector: | 1 | k | k |
| break | 1 | - | - |
| Else, display an error message. | 1 | - | - |
| Push back the tokens into the *courseParameters* vector. | 1 | n | n |
| Clear the *tokens* vector to prepare for the next line in the file | 1 | n | n |
| Close the file. | 1 | 1 | 1 |
| **Total Cost:** |  |  | 12 + 9n \* m \* k + 1 |
| **Runtime:** |  |  | O(n\*m\*k) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Vector - Create Course Objects and Store in a Vector** | | | |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Open the csv or text file. | 1 | 1 | 1 |
| Initialize a vector of course objects called *courses* | 1 | 1 | 1 |
| If the file opens: | 1 | 1 | 1 |
| While not the end of the file: | 1 | n | n |
| Getline from each line in file: | 1 | n | n |
| Initialize new blank *Course* object. | 1 | n | n |
| If there is a first word in the line: | 1 | n | n |
| Assign to course.courseId. | 1 | n | n |
| If there is a second word in the line: | 1 | n | n |
| Assign to course.courseName. | 1 | n | n |
| For each additional word in the line: | 1 | m | m |
| Push back the word into course.coursePrerequisites vector | 1 | m | m |
| If there are no additional words (pre-reqs): | 1 | 1 | 1 |
| Assign null to course.coursePrerequisites vector | 1 | - | - |
| Push back the *Course* object into the *Courses* vector. | 1 | n | n |
| Close file. | 1 | 1 | 1 |
| ELSE if the file does not open: | 1 | 1 | 1 |
| Display error. | - | - | - |
| **Total Cost:** |  |  | 3 + 8n \* 2m + 3 |
| **Runtime:** |  |  | O(n\*m) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Hash Table - Create Course Objects and Store in a Hash Table** | | | |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Define a new HashTable called *courseTable*. | 1 | 1 | 1 |
| Open the csv or text file. | 1 | 1 | 1 |
| While not the end of the file: | 1 | n | n |
| Create a new blank *Course* object. | 1 | n | n |
| Open an input string stream from the file. | 1 | 1 | 1 |
| While there are characters to read from the stream: | 1 | m | m |
| Split each line using the comma as a delimiter. | 1 | p | p |
| Push back each word into *parameters* vector. | 1 | p | p |
| Assign 1st element to course.CourseID. | 1 | n | n |
| Assign 2nd element to course.CourseName. | 1 | n | n |
| If the *parameters* vector > 2: | 1 | p | p |
| Append the additional paremters to course.CoursePrerequisites vector. | 1 | p | p |
| Invoke the *Insert()* function: | 4 | n | 4 n |
| Generate a key using Hash function: | *1* | 1 | 1 |
| For each character in the *courseID*: | *1* | 1 | 1 |
| Calculate the string hash by multiplying 5381 by 33 and adding to characters ASCII | *1* | 1 | 1 |
| Return hash % table size. | *1* | 1 | 1 |
| Retrieve hash table node using key generated. | 1 | n | n |
| If the bucket is empty: | 1 | n | n |
| Create a new node. | 1 | n | n |
| Insert into hash table key index. | 1 | n | n |
| Else: | 1 | 1 | 1 |
| IF an entry for the node found but not being utilized: | 1 | 1 | 1 |
| Initialize course node using *Course* object information. | 1 | 1 | 1 |
| Else: |  |  |  |
| Iterate through each node attached to key. | 1 | 1 | 1 |
| Add the new node with course info to the end. | 1 | 1 | 1 |
| Close file. |  |  |  |
| **Total Cost:** |  |  | 2 + 12n + m + 4p + 10 |
| **Runtime:** |  |  | O(n \*(m + p)) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Tree - Create Course Objects and Store in a Tree** | | | |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Define a new *BinarySearchTree* object. | 1 | 1 | 1 |
| Open the csv or text file. | 1 | 1 | 1 |
| While not the end of the file: | 1 | n | n |
| Create a new blank *Course* object. | 1 | n | n |
| Open an input string stream from the file. | 1 | 1 | 1 |
| While there are characters to read from the stream: | 1 | m | m |
| Split each line using the comma as a delimiter. | 1 | p | p |
| Push back each word into *parameters* vector. | 1 | p | p |
| Assign 1st element to course.CourseID. | 1 | n | n |
| Assign 2nd element to course.CourseName. | 1 | n | n |
| If the parameters vector > 2: | 1 | p | p |
| Append the additional paremters to course.CoursePrerequisites vector. | 1 | p | p |
| Invoke the Insert() function: | 13 | n | n |
| If the tree is empty: | 1 | 1 | 1 |
| This course will be the root node. | 1 | 1 | 1 |
| Else: | 1 | n | n |
| Invoke *addNode()* function | 10 | n | n |
| If course number is less than root node: | 1 | n | n |
| If no left child node: | 1 | n | n |
| Insert new *Course* object as the left child node. | 1 | n | n |
| Else: | 1 | n | n |
| Call *addNode()* recursively. | 1 | n | n |
| If course number is greater than root node: | 1 | n | n |
| If no right child node: | 1 | n | n |
| Insert new *Course* object as the right child node. | 1 | n | n |
| Else: | 1 | n | n |
| Call *addNode()* recursively. | 1 | n | n |
| Close file. | 1 | 1 | 1 |
| **Total Cost:** |  |  | 5 + 17n \* ( m + 4p) |
| **Runtime:** |  |  | O(n\*(m+p)) |

**Advantages and Disadvantages of Different Data Structures:**

**Vector:**

The vector data structure has several advantages. Firstly, the computer can set aside a contiguous set of memory to store it. Any additions can be added to the memory allocation because this is a type of dynamic array that can allocate additional memory addresses should the number of elements grow. It also has quick read speeds, provided one knows the element number of the object being searched. Additionally, vectors are one of the most straightforward data structures to implement as they have simple syntax and are built into many languages (like C++). A computer can jump to any memory address of a vector element in a single step, making read speeds the vector’s most significant advantage. However, there are a few disadvantages to utilizing a vector. Searching for a specific object of data can take one step (if found inside of the first vector element) or can require the traversal of the entire vector in the worst-case scenario (searching for data that is not found until the last element) because the computer does not know the values stored inside each element until it is read. Another disadvantage of utilizing a vector is inserting new data. Prepending an additional element to the beginning of the vector is a worst-case scenario because it requires shifting every element to the right to accommodate new data. In contrast, appending an additional element to the end is a best-case scenario because it requires expanding the vector’s memory allocation and pushing the element to the back of the vector.

For the ABCU course catalog implementation, only eight courses will be stored in memory. Sorting them in alphanumeric order will require using a quicksort algorithm to repeatedly halve the data until it is sorted. This would be the most efficient algorithm if we went with a vector solution. One of the requirements is to search for a specific course and to print out its prerequisites. A vector solution is not best suited for this scenario because if an end-user were to search for “CSCI400,” the program would have to search through each element of the vector until it was found because we would not have the memory address to access the course within the data structure directly. The access time will increase if ABCU were to add additional courses. This leads us to another disadvantage: insertion and deletion would be cumbersome because all elements would have to be shifted and resorted whenever a new course is added to the catalog.

**Hash Table:**

The most significant advantage of a hash table is how quickly one can search for a specific value. Each value is paired with a key. This key is created by a hash function that must be consistent: the string must equal the same number each time the hash function is utilized. With the string value’s key, the program knows precisely which memory location the value is stored in. This allows a constant lookup speed for searching, insertion, and deletion. When deleting or inserting an element, only one index within the hash table’s array must be updated, eliminating the need for shifting elements to accommodate the operation. Sometimes, the hash function maps the same key for multiple strings. This is called a collision, and it is handled through subarrays or linked lists. This leads to the disadvantages of a hash table. A hash table is most efficient when the number of collisions is low. Handling a collision involves traversing an array or linked list to find a specific value, which adds more steps to any data operation. One way to mitigate this is through managing the hash table’s load factor. According to Jay Wengrow, a load factor of 7 elements to 10 cells is the optimal solution to minimizing collisions while lessening the amount of memory a hash table occupies. This, too, is a disadvantage because it adds to the complexity of implementing a hash table: a decent hash table should automatically resize based on the amount of data it stores.

For ABCU’s implementation, a big disadvantage to using a hash table is that it does not maintain the order of its values, especially when collisions occur and need to be handled. This would make accessing and printing the courses in alphanumerical order challenging because the hash function would generate key values within the hash table array without considering each string’s alphanumerical position concerning the other values of the input file. While converting to each string’s ASCII value within the function may be a workaround, the most difficult challenge would be ensuring the data of a specific key would be sorted when a collision occurs. However, the benefit of this solution is that searching, deleting, and inserting a course into the hash table would be quicker. The end-user would only need to input the course string ID, and the hash function would convert it into a key to search for the course information. Barring any collisions, this would be the fastest way to access information.

**Tree:**

A tree’s main advantage is its ability to inherently sort data as it is inserted. In a binary search tree, the first node, also known as the “root,” has at most two child nodes. The child node to the left has a lesser value than the root, while the right has a greater value. This pattern repeats for each child node in the tree until a leaf node (a node with no children) is reached. Another advantage of utilizing a tree is how quickly it can search for a specific value. The program would not have to visit every single node inside the tree. A value is either less than, equal to, or greater than any node in the tree. Wengrow succinctly summarized that a search in a binary tree only takes as many steps as there are levels that constitute it, thus eliminating half of the nodes in the tree right off the bat. Inserting a new node is also an advantage because it piggybacks off of the search algorithm in that it takes the number of steps to search plus one (inserting the new node). According to Wengrow, a binary search tree is optimal when one foresees many changes in the data used. However, a binary search tree produces suboptimal results when fed sorted data. In fact, this would generate a linear tree that is only as fast as a linear search of an array which would be a major disadvantage. Deletion is more complex to implement, especially when deleting a node with two children, but is still faster than an array or vector solution.

For ABCU’s program, a tree provides the advantage of having every course presorted. Printing each course involves traversing the tree using an in-order method. If we were to insert a new course, we would compare it to the root and again traverse the array, comparing the new course to each node before we find a suitable spot to plug the node in, so to speak. The disadvantage would be the data provided in the text file. If it is presorted from least to greatest, the tree would be linear, negating its advantages. A workaround would be to take each course within the file and randomly sort it, allowing the root node to have children that are less than and greater than its value, enabling a balanced tree with all its advantages.

**Final Recommendation:**

Considering the advantages and disadvantages of each data structure for ABCU’s implementation, I believe that the tree data structure is the best solution for the use case. It enables the data to be accessed quickly and allows future proofing for the program in the event the university wants to add more courses. Furthermore, when creating the tree, the data will be inserted in alphanumerical order making an in-order traversal to print the courses simple to implement. According to the Big-O analysis, the tree implementation is similar in algorithmic complexity to the hash table, and even the vector. In the worst-case, particularly with an unbalanced search tree, the time complexity is close to that of a linear search in a vector of O(N). The differences in the pseudocode come down to the way in which we parse the data to populate the course objects. For the tree and hash table implementations, *n* represents the number of lines in the file the program parses through, *m* represents the number of characters inside the stream of each line, and *p* represents the number of times we iterate through the prerequisites if they are found. This is why the worst-case big O runtime analysis of this specific tree implementation is *O(n\*(m+p))* instead of *O(n).*

References

GeeksforGeeks. (2023a, March 28). *Applications, advantages and disadvantages of hash data structure*. GeeksforGeeks. https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-hash-data-structure/#

GeeksforGeeks. (2023b, May 21). *Applications, advantages and disadvantages of binary search tree*. GeeksforGeeks. https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-binary-search-tree/

Wengrow, J. (2020). *A common-sense guide to data structures and algorithms level up your core programming skills*. The Pragmatic Bookshelf.