**Main Menu**

**FUNCTION** mainMenu()

**CREATE** integer variable called choice and set to 0

**WHILE** choice does not equal 9

**PRINT** “Main Menu:”

**PRINT** “1. Load Courses”

**PRINT** “2. Display Courses”

**PRINT** “3. Find Course”

**PRINT** “9. Exit”

**COLLECT** user input and set that equal to choice

**SWITCH** based on choice variable

**CASE 1:**

**CREATE** timer variable called ticks and set to clock()

**CALL** openFile with desired file name

**SET** ticks to clock() - ticks to calculate elapsed time

**PRINT** “Time: “ and ticks

**PRINT** “Time: “ and ticks \* 1.0 divided by

CLOCKS\_PER\_SEC to get seconds

**BREAK** from loop

**CASE 2:**

**CALL** displayCourses function

**BREAK** from loop

**CASE 3:**

**CALL** findCourse function with desired course

**BREAK** from loop

**CASE 9:**

**PRINT** “Exiting program”

**BREAK** from loop

**Structure of Course Object**

**STRUCT** Course

**String** courseId

**String** courseTitle

**Integer** elements

**List** prereqs

**Create Course with Inputted Elements**

**FUNCTION** createCourse(Input: string vector elements)

**CREATE** new course object called course

**FOR** i = 0; as long as i is less than elements.length; ++i

**IF** i is 0

**SET** course’s courseId to current element

**IF** i is 1

**SET** course’s courseTitle to current element

**IF** i > 1

**ADD** current element to course’s prereqs list

**RETURN** course

**Vector Data Structure**

**FUNCTION** openFile(input: string fileName)

**CREATE** variable called inputFile

**CREATE** string vector variable called elements

**CREATE** course object vector called courses

**CREATE** string variable called tempInput

**LOAD** file with name equal to fileName into inputFile

**FOR EACH** line in inputFile

**WHILE** getLine from inputFile, storing it in tempInput and using the

delimitation character: ‘,’

**PUSHBACK** tempInput into elements

**SET** tempInput to “”

**IF** elements.length is less than 2

**PRINT** “ERROR: An entry has less than 2 parameters”

**BREAK** out of the for loop

**CALL** createCourse using elements and add the course to courses

**CLEAR** elements for next line with elements.clear()

**FOR EACH** course in courses

**FOR EACH** prereq in course’s prereqs

**CREATE** course object called reqCheck

**CALL** search function with current prereq store in reqCheck

**IF** reqCheck’s courseId does not equal current prereq

**PRINT** “ERROR: A course has a prerequisite that does

not exist in the list of courses.”

**DELETE** reqCheck

**FUNCTION** Search(Input: courses vector and string courseId, Returns a course)

**FOR EACH** course in courses

**IF** current course’s courseId matches courseId

**PRINT** course’s courseId and courseName

**FOR EACH** prereq in course’s prereqs

**PRINT** current prereq

**RETURN** course

**ELSE**

**PRINT** “Desired course not found”  
**FUNCTION** selectionSort(Input: courses& vector)

**CREATE** size\_t object called size and set to courses.size()

**CREATE** integer variable called min

**FOR** size\_t position starting at 0, courses.size() – 1, ++position

**SET** min to current position

**FOR** size\_t j starting at position + 1, size – 1, ++j

**IF** j’s courseId is less than min’s courseId

**SET** min to j

**IF** min does not equal current position

**SWAP** course at position with min’s course

**FUNCTION** printAll(Input: courses vector)

**CALL** selectionSort with courses

**FOR EACH** course in courses

**PRINT** course Id and course name

**FOR EACH** prereq in course’s prereqs

**PRINT** current prereq

**Hash Table Data Structure**

**STRUCT** Node

**Course** course

**Unsigned Int** key

**Node** \*next

**CREATE** vector of nodes called nodes

**CREATE** HashTable called Hcourses

**CONSTRUCTOR** HashTable()

**RESIZE** nodes to desired size(most likely a constant defined in practice)

**CONSTRUCTOR** HashTable(Input: unsigned integer called size)

**SET** this object’s tableSize to size

**RESIZE** nodes to size

**FUNCTION** hash(Input: integer key, courses vector)

**RETURN** the key modulo tableSize

**FUNCTION** openFile(input: string fileName)

**CREATE** variable called inputFile

**CREATE** string variable called tempInput

**CREATE** string vector called elements

**CREATE** course object called tempCourse

**LOAD** file with name equal to fileName into inputFile

**FOR EACH** line in inputFile

**WHILE** getLine from inputFile, storing it in tempInput and using the

delimitation character: ‘,’

**PUSHBACK** tempInput into elements

**SET** tempInput to “”

**IF** elements.length is less than 2

**PRINT** “ERROR: An entry has less than 2 parameters”

**BREAK** out of the for loop

**CALL** createCourse using elements and store in tempCourse

**CREATE** unsigned key called currentKey and call hash with

tempCourse’s courseId

**CREATE** node variable called oldNode to retrieve node at currentkey

**IF** oldNode is null

**CREATE** node variable called newNode with tempCourse

**INSERT** newNode into Hcourses hashTable

**ELSE IF** oldNode is unused

**UPDATE** it to equal tempCourse

**ELSE IF** oldNode is used

**WHILE** oldNode’s next is not null

**SET** oldNode to oldNode’s next

**SET** oldNode’s next to new node using tempCourse and

currentKey

**CREATE** string vector called checkReqs

**FOR EACH** prereq in tempCourse’s prereqs

**ADD** current prereq to checkReqs

**DELETE** tempCourse

**FOR EACH** prereq in checkReqs

**CREATE** boolean variable hasMatch and set to false

**CREATE** hash key with hash using current prereq

**CREATE** node variable called sNode and retrieve node at key

**IF** the node is not null, the key is not empty, and the prereq

matches the Id

**SET** hasMatch to true

**BREAK** out of the for loops

**WHILE** sNode is not null

**IF** node’s key is not empty, and the prereq matches the Id

**SET** hasMatch to true

**BREAK** out of the for loops

**SET** sNode to sNode’s next

**IF** hasMatch is false

**PRINT** “ERROR: A course has a prerequisite that does not exist in the list of courses.”

**BREAK** out of loop

**FUNCTION** Search(Input: string courseId)

**CREATE** course object called course

**CREATE** key using hash and courseId

**CREATE** node variable called node and retrieve node at key

**IF** retrieved node is not null pointer, key is not empty, and courseId matches

**RETURN** node’s course

**IF** retrieved node is null pointer, or key is empty

**RETURN** empty course object

**WHILE** node is not null pointer

**IF** node’s key is not empty, and the courseId matches

**RETURN** node’s course

**SET** node to node’s next

**RETURN** empty course object if course is not found

**FUNCTION** printAll()

**CREATE** vector of courses called coursesSort

**FOR** auto i starting at start of hashTable nodes, as long as i is not node’s end

++i

**IF** i’s key is not empty

**APPEND** course associated with i’s key to coursesSort

**CREATE** node object called nextNode and set to i’s next

**WHILE** nextNode is not null pointer

**APPEND** course associated with i’s key to coursesSort

**CALL** selectionSort with coursesSort

**FOR EACH** course in coursesSort

**PRINT** course Id and course name

**FOR EACH** prereq in course’s prereqs

**PRINT** current prereq

**FUNCTION** selectionSort(Input: coursesSort& vector)

**CREATE** size\_t object called size and set to courses.size()

**CREATE** integer variable called min

**FOR** size\_t position starting at 0, courses.size() – 1, ++position

**SET** min to current position

**FOR** size\_t j starting at position + 1, size – 1, ++j

**IF** j’s courseId is less than min’s courseId

**SET** min to j

**IF** min does not equal current position

**SWAP** course at position with min’s course

**Binary Search Tree Data Structure**

**STRUCT** Node

**Course** course

**Node** \*left

**Node** \*right

**CREATE** node\* object called root

**CONSTRUCTOR** BinarySearchTree

**SET** root equal to null pointer

**FUNCTION** openFile(input: string fileName)

**CREATE** variable called inputFile

**CREATE** string variable called tempInput

**CREATE** string vector called elements

**CREATE** course object called tempCourse

**LOAD** file with name equal to fileName into inputFile

**FOR EACH** line in inputFile

**WHILE** getLine from inputFile, storing it in tempInput and using the

delimitation character: ‘,’

**PUSHBACK** tempInput into elements

**SET** tempInput to “”

**IF** elements.length is less than 2

**PRINT** “ERROR: An entry has less than 2 parameters”

**BREAK** out of the for loop

**CALL** createCourse using elements and store in tempCourse

**IF** root equals null pointer

**SET** root to new node created with tempCourse

**ELSE**

**CALL** this BinarySearchTree’s addNode function with

root and tempCourse

**CREATE** string vector called checkReqs

**FOR EACH** prereq in tempCourse’s prereqs

**CREATE** course object called reqChecker

**CALL** search function with current prereq as parameter and set

reqChecker equal to it

**IF** reqChecker’s courseId does not equal current prereq

**PRINT** “ERROR: A course has a prerequisite that does

not exist in the list of courses.”

**DELETE** reqChecker

**FUNCTION** addNode(Input: node\* object called node and course object called

course

**IF** the node’s course’s courseId is less than course’s courseId

**IF** node’s left subtree equals null pointer

**CREATE** new node using course to be the left subtree

**ELSE**

**CALL** this BinarySearchTree’s addNode with the left subtree

and course

**ELSE**

**IF** node’s right subtree equals null pointer

**CREATE** new node using course to be the right subtree

**ELSE**

**CALL** this BinarySearchTree’s addNode with the right subtree

and course

**FUNCTION** Search(Input: string variable called courseId, Returns: Course object)

**CREATE** node\* object called current and set to root

**WHILE** current is not null pointer

**IF** courseId matches current’s course’s courseId

**PRINT** course’s courseId and courseName

**FOR EACH** prereq in course’s prereqs

**PRINT** current prereq

**RETURN** current’s course

**IF** courseId is less than current’s course’s courseId

**SET** current to current’s left subtree

**ELSE IF** courseId is greater than current’s course’s courseId

**SET** current to current’s right subtree

**CREATE** Course object called dummyCourse

**PRINT** “No matching course found”

**RETURN** dummyCourse if no match was found during while loop

**FUNCTION** printInOrder(Input: node object called node)

**IF** node is not null pointer

**CALL** printInOrder using node’s left subtree

**PRINT** node’s courseId and courseName

**FOR EACH** prereq in course’s prereqs

**PRINT** current prereq

**CALL** printInOrder with node’s right subtree

**Big- O Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Vectors Big-O Value | Hash Table Big-O Value | Binary Search Tree Big-O Value |
| Opening and loading file | O(1) | O(1) | O(1) |
| Parsing each line in file and creating corresponding course objects | O(n\*(m + k) | O(n\*m + p) | O((n log n + m) + p) |

n = Average number of lines in file

m = Average elements per line

k = Average number of prereqs to validate

p = Average length of prereq list to check

**Evaluation - Vectors­**

**Advantages:**

* **Simple Usage** – Vectors are easy to use for storing collections of elements in a linear order thanks to their straightforward and simple interface. \
* **Contiguous Memory –** The memory allocated for vectors is contiguous, allowing for efficient iteration and element access
* **Dynamic Resizing** – Vectors can dynamically resize themselves, allowing them to be as big or small as needed to be as efficient as possible.

**Disadvantages:**

* **Insertions/Deletions –** When adding or removing elements from a vector, the shifting of elements may be required, slowing down computation.
* **Wasted Memory –** While vectors can dynamically resize themselves, there may also be times when a vector reserves more space than needed for future elements.
* **Search Times –** Searching for a specific element in an unsorted vector could result in long search times as it has to iterate through the entire vector.

**Evaluation – Hash Table­**

**Advantages:**

* **Efficient Searching** – Hash tables have an average runtime complexity of O(1) when performing searches, insertions, and deletions, given an efficient hash function.
* **Dynamic Resizing –** Like with vectors, hash tables can dynamically resize themselves to be efficient with memory allocation.
* **Customizable Hash Functions –** The hash function, which performs mathematical calculations to get a hash key, can be tailored to the requirements of the data and the desired hash table.

**Disadvantages:**

* **Unordered Elements –** Hash tables are intrinsically unordered, meaning there is no way to sort a hash table since the data is stored by the hashcode of keys, not in-order indexes.
* **Hash Collisions** – Tables with large amounts of data could have entries associated with the same hash bucket, which would need to be resolved through a variety of methods.
* **Complexity –** Depending on the hash function and requirements for the data, hash tables can be difficult to implement due to their complexity.

**Evaluation – Binary Search Tree**

**Advantages:**

* **Ordered Structure –** Binary search trees maintain a sorted order, which is important when elements need to be accessed in a specific order.
* **Fast Searching –** BSTs allow for quick and efficient iteration through sorted elements by utilizing in-order traversal.
* **Memory Efficient –** Binary search trees store only elements and do not have a need for additional memory for pointers or other similar structures.

**Disadvantages:**

* **Worst-Case Balancing –** BSTs typically have a time complexity of O(log n), but in situations where a tree becomes unbalanced, the complexity could degrade to O(n).
* **Housekeeping Complexity –** Since improper balance can impact the efficiency of BSTs, additional attention needs to be paid toward maintaining a tree’s balance.
* **Self-Balancing Overhead –** BSTs can implement self-balancing measures, but this results in increased memory usage and operation complexity.

**Conclusion**

After considering the three data structures this project focused on, a binary search tree is what I have decided would be the most appropriate choice. My conclusion stems from two specific characteristics of a BST: the ordered structure of a tree and the speed of its searches. One of the requirements for this project was to have a list sorted and printed alphanumerically, and by utilizing a binary search tree, this sorting can be done automatically. This eliminates the need for a separate sorting function, like with vectors. The project's second requirement, being able to search and print out a specific course, is also handled well by a BST. This is thanks to its speed when it comes to searches and most other operations. The fact that a BST excels in the two main requirements presented by the academic advisors is what makes it stand out in this scenario. While the other two data structures could work for this project, I feel that a binary search tree is the best fit.