**Vector sorting in alphabetical order:**

FUNCTION partition(courseVector, low, high)

Mid = low + (high – low) / 2

Pivot = courseVector[Mid]

WHILE low <= high

WHILE courseVector[low] < Pivot

low = low + 1

ENDWHILE

WHILE courseVector[high] > Pivot

high = high – 1

ENDWHILE

IF low <= high

TEMP = courseVector[low]

courseVector[low] = courseVector[high]

courseVector[high] = TEMP

low = low + 1

high = high – 1

ENDIF

ENDWHILE

RETURN high

END FUNCTION

FUNCTION quicksort(courseVector, low, high)

IF low < high

Index = partition(courseVector, low, high)

quicksort(courseVector, low, Index)

quicksort(courseVector, Index + 1, high)

ENDIF

END FUNCTION

FUNCTION main

CALL quicksort(courseVector, 0, SIZE – 1)

PRINT “In alphabetical order:”

FOR each course in courseVector

Display course information

ENDFOR

END FUNCTION

**Hash Table sorting in alphabetical order:**

FUNCTION partition(courseHashTable, low, high)

Mid = low + (high – low) / 2

PivotKey = GetKeyAtIndex(courseHashTable, Mid)

PivotValue = GetValue(courseHashTable, PivotKey)

WHILE low ≤ high

WHILE GetValueAtIndex(courseHashTable, low) < PivotValue

low = low + 1

ENDWHILE

WHILE GetValueAtIndex(courseHashTable, high) > PivotValue

high = high – 1

ENDWHILE

IF low ≤ high

SwapValuesAtIndex(courseHashTable, low, high)

low = low + 1

high = high – 1

ENDIF

ENDWHILE

RETURN high

END FUNCTION

FUNCTION quicksort(courseHashTable, low, high)

IF low < high

Index = partition(courseHashTable, low, high)

quicksort(courseHashTable, low, Index)

quicksort(courseHashTable, Index + 1, high)

ENDIF

END FUNCTION

FUNCTION main

CALL quicksort(courseHashTable, 0, SIZE – 1)

PRINT “In alphabetical order:”

FOR each key in courseHashTable

Display course information for courseHashTable[key]

ENDFOR

END FUNCTION

**Binary Tree sorting in alphabetical order:**

FUNCTION inOrderTraversal(node)

IF node is not null

inOrderTraversal(node.left) // Traverse left subtree

Display course information for node

inOrderTraversal(node.right) // Traverse right subtree

ENDIF

END FUNCTION

FUNCTION main

CREATE rootNode as the root node of the binary tree

PRINT "In alphabetical order:"

CALL inOrderTraversal(rootNode)

END FUNCTION

**EVALUATION:**

**Vectors:**

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Cost | Number of times executed | Total Big O value |
| Reading the file | O(1) | O(n) | O(n) |
| creating course objects | O(1) | O(n) | O(n) |
| Sorting with quicksort | O(n log n) | O(n) | O(n log n) |
| Total worst-case running time | | | O(n log n) |

**Hash Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Cost | Number of times executed | Total Big O value |
| Reading the file | O(1) | O(n) | O(n) |
| creating course objects | O(1) | O(n) | O(n) |
| Total worst-case running time | | | O(n) |

**Binary Tree:**

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Cost | Number of times executed | Total Big O value |
| Reading the file | O(n) | O(n) | O(n) |
| creating course objects | O(1) | O(n) | O(n) |
| Sort using in-order traversal | O(n) | O(n) | O(n) |
| Total worst-case running time | | | O(n) |

**ADVANTAGES AND DISADVANTAGES:**

**Vector:**

The advantages of vector sorting include its ease of implementation and the efficiency of the quicksort algorithm, particularly for small datasets. Vectors also facilitate rapid file reading and the addition of new course objects if required. However, for very large datasets, quicksort may become inefficient and time-consuming due to its average-case time complexity of O(n log n) as it sorts through each course item.

**Hash Table:**

Hash tables offer advantages such as rapid lookup and retrieval of elements by utilizing keys to map to items in the search list. They can efficiently resize to accommodate additional elements. However, they demand substantial memory storage and lack inherent ordering, which can result in time-consuming sorting operations if required. Collision issues may also arise, leading to decreased program performance.

**Binary Search Tree:**

Binary search trees maintain order and excel in searching and sorting operations, making it straightforward to find specific items or elements. Additionally, they efficiently handle insertion and deletion tasks. However, modifying a binary search tree may take longer due to its structure, and it can consume significant memory for tree nodes. Furthermore, they are the most complex to implement compared to vectors or hash tables.

**RECOMMENDATION:**

Considering that the dataset comprises courses and their prerequisites, the emphasis on efficient search and sorting operations becomes paramount. This consideration leads me to favor the adoption of a binary search tree for optimal performance. Binary search trees maintain order and offer efficient searching capabilities, aligning seamlessly with the dataset's requirements. Additionally, they excel in handling insertion and deletion tasks, ensuring adaptability to changes in the dataset. Although binary trees may consume considerable memory due to tree nodes, they still maintain a level of memory efficiency, particularly in comparison to structures like hash tables, especially for smaller datasets. The primary drawback lies in the complexity associated with implementing a binary search tree compared to vectors. However, once implemented, binary search trees facilitate efficient search and sorting operations without the need for additional sorting algorithms. Therefore, I believe a binary search tree stands as the most suitable structure for managing the dataset of course information.