Main()

Read cmd arguments

For each cmd argument

Store as csv file path

While choice != 9

Print menu

menuChoice = input

dataChoice = input

if userChoice not in 1, 2, 3, 4, 9

throw error

if dataChoice not in 1, 2, 3, 4, 9

throw error

if menuChoice is 1

if dataChoice is binarySearchTree

binarySearchTree = loadBids()

else if dataChoice is vector

vector = loadBids()

else if dataChoice is hashTable

hashTable = loadBids()

else if menuChoice is 2

if dataChoice is binarySearchTree

validateTree(binarySearchTree)

else if dataChoice is vector

validateList(vector)

else if dataChoice is hashTable

validateTable(hashTabel)

else if menuChoice is 3

userSearch = input

if dataChoice is binarySearchTree

printCourseTree(userSearch)

else if dataChoice is vector

printCourseList(userSearch)

else if dataChoice is hashTable

printCourseTable(userSearch)

else if menuChoice is 4

if dataChoice is binarySearchTree

printTree()

else if dataChoice is vector

sortList()

printList()

else if dataChoice is hashTable

sortTable()

printTable()

else if menuChoice is 9

` exit

print “Good bye”

struct Course ()

courseID

courseName

preCount

prelist

Course()

Class BinaryTree()

struct Node

Course

right pointer

left pointer

root

printTree()

BinaryTree()

Class HashTable

struct bucket

Course

Key

Next pointer

hash()

printTable()

List<>hashTable

sortList()

lowEndIndex = partition()

quicksort(vector, lowest index, lowEndIndex)

quicksort(vector, lowEndIndex + 1, highest index)

partition()

pivot = (highest index + lowest index) / 2

while lowest index >= highest index

while pivot < lowest index

lowest index = current position

while pivot > lowest index

highest index = current position

temp = highest index

highest index = lowest index

lowest index = temp

lowest index += 1

highest index -= 1

return highest index

printList()

for each course in courseList

print courseID and courseName

while I < preCount

for each course in prelist

print CourseID

printTree()

root = Node pointer

root = Null

if Node is null

return

print courseID courseName of Node’s left pointer

while I < preCount

for each course in prelist

print courseID

print courseID courseName of Node’s right pointer

printTable()

Node pointer = nodes beginning

for each course

print courseID, courseName

while I < preCount

for each course in prelist

printCourse(preList)

Run Time Analysis

|  |  |  |  |
| --- | --- | --- | --- |
|  | Binary Tree | Vector | Hash Table |
| Loading | O(log N) | O(1) | O(1) – O(N)  (depending on collisions) |
| Searching | O(log N) – O(N)  (depending on balance of tree) | O(n) | O(1) – O(N)  (depending on collisions) |
| Sorting and Printing | O(N)  (in order of traversal) | O(N log N)  (using quick sort) | O(N)  (assuming table is in order) |

All three data structures have their pros and cons. Loading data into an unsorted vector using an append method is extremely fast, but sorting it afterward has the slowest performance. In theory, a hash table could always operate at its average O(1) if it were large enough to prevent all collisions. However, because neither time nor memory are infinite, the table must handle some collisions, which could push the hash table's performance to somewhere between O(1) and O(N).

A binary tree generally operates most consistently at or near O(log N), depending on how the data is input. If the tree becomes heavily unbalanced, such as when sorted data is loaded, its performance slows down to O(N).

Choosing the right data structure depends on how and how frequently the data will be accessed. For instance, if the data only needs to be loaded infrequently, there are no advantages after the initial load. If the data needs to be searched often, a hash table could be more efficient than a binary tree, assuming an efficient and well-designed hash function or an unbalanced tree.

Furthermore, a binary tree doesn't need to be sorted and can be traversed in order, which could save memory if both sorted and unsorted "lists" do not need to be stored. Overall, a binary tree and a hash table will perform better and be preferable to sorting a vector.

The data will most likely be read into memory and fully printed a lot less than it will be searched. This would made the Hash Table the best option. The Hash function and table size would of course need to be created and maintained in a way to prevent collisions, making the code closer to operating at O(1) rather than at O(N).