**Main** Function() //Menu Loop

**Read** cmd arguments

**Store** argument as CSV file path

**If** no cmd arguments load default CSV file path

**Loop** while choice is not equal to ‘9’

**Output** menu block

**Get** user input; **Store** in *menuChoice* //what the program is to do

**Get** user input; **Store** in *dataChoice* //what data structure to use

**Validate** user input

**If** choice is not 1-4 or 9 throw an error

**If** choice equals ‘1’

//Call file parser and load data into each data structure

**If** BinarySearchTree

**Call** loadBids and store CSV data in BinarySearchTree *bst*

**Else** **If** vector

**Call** loadBids and store CSV data in vector *courseList*

**Else** **If** HashTable

//loadBids to have a hash function that orders the map in ascending order

**Call** loadBids and store CSV data in HashTable *courseTable*

**Output** number of records in the CSV file

**If** choice equals ‘2’

//Validate the List

**If** BinarySearchTree

**Call** **validateTree**() passing *bst*

**Else** **If** vector

**Call** **validateList**() passing *courseList*

**Else** **If** HashTable

**Call** **validateTable**() passing *courseTable*

**If** choice equals ‘3’

//Search and print course

**Get** user value to search for and **Store** in *userSearch*

**If** BinarySearchTree

**Call** **printCourseTree()** passing *userSearch*

**Else** **If** vector

**Call** **printCourseList()** passing *userSearch*

**Else** **If** HashTable

**Call** **printCourseTable()** passing *userSearch*

**If** choice equals ‘4’

//Print each course in alphabetic order

**If** BinarySearchTree

**Call printTree()**

**Else** **If** vector

**Call** **sortList()**

**Call printList()**

**Else** **If** HashTable

**Call** **sortTable()**

**Call printTable()**

**If** Choice equals ‘9’

**Exit** the application

**Output** ‘Good bye’

**End**

**struct Course {}**

*courseID*

*courseName*

*preCount*

*prelist*

Course() (constructor) {courseID = courseName = ””; preCount = 0; preList = “”}

**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()**

**Get** vector to sort, lowest index of vector and highest index of vector

**If** lowest index if greater than or equal to highest index return nothing

**Call** partition() function

**Set** *lowEndIndex* equal to the value returned by the partition function

Recursively **call** quicksort passing the vector, lowest index, and *lowEndIndex* (from above)

Recursively **call** quicksort passing the vector, *lowEndIndex* (from above) plus one, and highest index

**End**

**partition**()

**Get** the vector to partition, the lowest index and the highest index

Determine the vector element at the midpoint between the lowest and highest index

**Set** pivot equal to this vector element

**Loop** until the lowest index is greater than or equal to the highest index

**Loop** through the vector from lowest index until a vector element larger than the pivot is found

**Overwrite** lowest index with this element’s position

**Loop** through the vector from lowest index until a vector element smaller than the pivot is found

**Overwrite** highest index with this element’s position

**Swap** the vector elements at the new highest and lowest index

**Overwrite** the lowest index by incrementing it one

**Overwrite** the highest index by decrementing it one

**Return** the highest index

**End**

**printList()**

**Loop** through *courseList*

**Output** to console: *courseID, courseName,*

**Loop** 0 to *preCount*

**For each** *Course* in *preList*

**Output** to console: *courseID*

**End**

**printTree**()

**Create** new Node pointer named root

**Set** root to NULL

**Check** if Node is null and if so return

**Call** via recursion Node’s left pointer which will find the left most Node

**Output** to console: *courseID, courseName,*

**Loop** 0 to *preCount*

**For each** *Course* in *preList*

**Output** to console: *courseID*

**Call** via recursion Node’s right pointer which will find the right most Node

**End**

**printTable()**

**Create** a new Node pointer and **Set** to the address of the nodes beginning

**Loop** through the list; starting at the beginning

**Output** *courseID* in Course struct found within *tempCourse* to console

**Output** *courseName* in Course struct found within *tempCourse* to console

**Loop** 0 to *preCount*

**For each** *Course* in *preList*

**Call** p**rintCourse**() passing *prelist*

**End**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Vector** | **Hash Table** | **Binary Tree** |
| **Loading Data** | O(1) | O(1) – O(N)  *\*depends on if there are collisions* | O(log N) |
| **Search** | O(n) | O(1) – O(N)  *\*depends on if there are collisions* | O(log N) – O(N)  *\*depends on balance of the tree* |
| **Sort/Print** | O(N log N) *\*using quick sort* | O(N)  *\*assumes the table is created in order* | O(N)  *\*in order traversal* |

Each of the three data structures has benefits and drawbacks. Sorting the data after it has been loaded has the slowest speed, however loading it into an unsorted vector using an add technique is extremely fast. If a hash table were big enough to avoid all collisions, it could theoretically always function at its average Δ(1). But because memory and time are finite, the hash table must be able to handle certain collisions, which would cause it to move between O(1) and O(N). Depending on how the data is read in, the binary tree will typically perform at or near O(log N). In other words, the Binary Tree slows down to O(N) if the tree becomes significantly imbalanced, such as when sorted data is loaded. The data structure to use relies on how and how often the data will be accessed. There are no benefits after the initial load, for instance, if the data only needs to be loaded occasionally. Assuming a very imbalanced tree or an effective and well-designed hash algorithm, the hash table may perform better than the binary tree if the data needs to be searched frequently. Finally, if both the sorted and unsorted "lists" do not need to be stored, the binary tree can be traversed in order without the need for sorting, which may save some memory. Additionally, sorting the vector will not perform as well as the binary tree and hash table. The Hash Table should be preferred since it is assumed that the data will only need to be searched regularly, rather than frequently read into memory and printed completely. To minimize collisions, the hash function and table size must be optimized, though, such that the algorithm runs more like O(1) than O(N)