Rosalinda Castaneda

CS 300 – Analysis and Design

August 10, 2025

Module 6 – Project One

Function Main()

If command-line argument exists

CSVFilePath ← argument

Else

CSVFilePath ← default path

While menuChoice ≠ '9'

Display Menu:

1. Load Data

2. Validate Data

3. Search Course

4. Print All Courses

9. Exit

menuChoice ← user input

If menuChoice NOT IN [1, 2, 3, 4, 9]

Output "Invalid choice. Please try again."

Continue loop

If menuChoice ≠ '9'

Prompt "Select data structure: BinarySearchTree, Vector, HashTable"

dataChoice ← user input

Switch menuChoice

Case '1': // Load Data

If dataChoice = "BinarySearchTree"

bst ← loadBids(CSVFilePath) // returns BinarySearchTree

Output number of records loaded

Else If dataChoice = "Vector"

courseList ← loadBids(CSVFilePath) // returns vector<Course>

Output number of records loaded

Else If dataChoice = "HashTable"

courseTable ← loadBids(CSVFilePath) // returns HashTable

Output number of records loaded

Case '2': // Validate Data

If dataChoice = "BinarySearchTree"

validateTree(bst)

Else If dataChoice = "Vector"

validateList(courseList)

Else If dataChoice = "HashTable"

validateTable(courseTable)

Case '3': // Search Course

Prompt "Enter course ID to search:"

userSearch ← user input

If dataChoice = "BinarySearchTree"

printCourseTree(bst, userSearch)

Else If dataChoice = "Vector"

printCourseList(courseList, userSearch)

Else If dataChoice = "HashTable"

printCourseTable(courseTable, userSearch)

Case '4': // Print All Courses

If dataChoice = "BinarySearchTree"

printTree(bst.root)

Else If dataChoice = "Vector"

sortList(courseList)

printList(courseList)

Else If dataChoice = "HashTable"

sortTable(courseTable)

printTable(courseTable)

Output "Good bye"

End Function  
  
Struct Course

courseID: string

courseName: string

preCount: int

preList: list<string>

Constructor Course()

courseID ← ""

courseName ← ""

preCount ← 0

preList ← empty list

End Struct  
  
Class BinaryTree

Struct Node

course: Course

left: Node pointer

right: Node pointer

End Struct

root: Node pointer

Constructor BinaryTree()

root ← null

Function printTree(node)

If node = null

return

printTree(node.left)

Output node.course.courseID, node.course.courseName

For i from 0 to node.course.preCount - 1

Output node.course.preList[i]

printTree(node.right)

End Class  
Class HashTable

Struct Bucket

course: Course

key: int

next: Bucket pointer

End Struct

hashTable: List<Bucket>

Function hash(key: string) → int // Implement hash function here

Function printTable()

For each bucket in hashTable

temp ← bucket

While temp ≠ null

Output temp.course.courseID, temp.course.courseName

For i from 0 to temp.course.preCount - 1

Output temp.course.preList[i]

temp ← temp.next

End Class  
Function sortList(courseList: vector<Course>)

quicksort(courseList, 0, courseList.size - 1)

End Function

Function quicksort(list: vector<Course>, low: int, high: int)

If low ≥ high

return

pivotIndex ← partition(list, low, high)

quicksort(list, low, pivotIndex)

quicksort(list, pivotIndex + 1, high)

End Function

Function partition(list: vector<Course>, low: int, high: int) → int

pivot ← list[(low + high) / 2]

While low ≤ high

While list[low] < pivot

low ← low + 1

While list[high] > pivot

high ← high - 1

If low ≤ high

Swap list[low], list[high]

low ← low + 1

high ← high - 1

Return high

End Function  
  
Function printList(courseList: vector<Course>)

For each course in courseList

Output course.courseID, course.courseName

For i from 0 to course.preCount - 1

Output course.preList[i]

End Function

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Binary Search Tree (BST) | Vector (Array/List) | Hash Table |
| Load Data | O(n log n) average (insertion) O(n²) worst (unbalanced) | O(n) load O(n log n) sort | O(n) average (hash insertion) O(n²) worst (collisions) |
| Validate Data | O(n) (in-order traversal) | O(n) (linear traversal) | O(n) (traverse buckets) |
| Search Course | O(log n) average O(n) worst | O(n) linear search O(log n) if sorted | O(1) average O(n) worst |
| Print All Courses | O(n) in-order traversal | O(n) after sorting | O(n) bucket traversal |

|  |  |  |
| --- | --- | --- |
| Data Structure | Advantages | Disadvantages |
| Binary Search Tree (BST) | - Efficient average case insert/search (O(log n)) - Naturally maintains sorted order - Easy to print sorted data | - Can become unbalanced, degrading to O(n) - More complex to implement than vector or hash table |
| Vector (Array/List) | - Simple to implement and use - Fast data loading - Sorting enables binary search (O(log n)) | - Searching unsorted vector is O(n) - Sorting adds overhead (O(n log n)) - Maintaining sorted order costly for inserts/deletes |
| Hash Table | - Very fast average lookup and insertion (O(1)) - Scales well with large data - Efficient direct access by key | - No inherent order (sorting needed for ordered output) - Performance can degrade with many collisions - Slightly more complex collision handling |