6-2 Project One

Parker G McKay

Southern New Hampshire University

CS-300-11851-M01 DSA: Analysis and Design

Nathan Lebel

August 10th, 2025

6-2 Project One

Pseudocode:

Milestone 1:

Load text parsing libraries and headers

Define a struct to hold course data

struct Course {}

courseID

courseName

preCount

preList

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

Main()

Create new List named courseList of the struct-type Course

Get CSV file path from user

If no data passed use default location

Call txtParser() passing CSV file path

Call validateList() passing courseList

Get user value to search for and Store in userSearch

Call printCourse() passing userSearch

End

txtParser(String)

Create a local List named tempList

Open file found at the path in String by invoking parser libraries

Loop row by row until end of file (eof)

If first and second string are present

Add the first String to struct at courseID

Add the second String to Struct at courseName

Loop until file handler has no value in a column (indicates no more prerequisite)

Increment a variable named preCount for each prerequisite found

Concatenate a localString named preNames for each prerequisite

Add preCount to struct at preCount

Add preNames to struct at preList

Return tempList

End

searchList(String)

Create tempCourse of type Course

Loop through list For Each Course

If String is the same as courseID

Set tempCourse to Course

Return tempCourse

End

printCourse(String)

Create tempCourse of type Course

Set tempCourse equal to searchList(String)

Output courseID to console

Output courseName to console

Loop 0 to preCount

For each Course in preList

Call printCourse() passing preList

End

validateList()

Create tempCourse of type Course

Create variable valid and Set to True

For Each Course

If valid is False break

Loop 0 to preCount

Set tempCourse equal to searchList(preList token)

If tempCourse courseID is empty Set valid to False

Return valid

End

Milestone 2:

Load text parsing libraries and headers

Define a struct to hold course data

struct Course {}

courseID

courseName

preCount

prelist

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

Class HashTable{}

-struct bucket

Course

key

next pointer

+hash()

+printAll()

+List<> hashTable

Main()

Create new List named courseList of the struct-type CourseMap

Get CSV file path from user

If no data passed use default location

Call txtParser() passing CSV file path

Call validateList() passing courseList

Get user value to search for and Store in userSearch

Call printCourse() passing userSearch

End

txtParser (String)

Open file found at the path in String by invoking parser libraries

Loop row by row until end of file (eof)

If first and second string are present

Call hash passing the first string

Add to struct at hash position within tempList

Add the first String to struct at courseID

Add the second String to Struct at courseName

Loop until file handler has no value in a column (indicates no more prerequisite)

Increment a variable named preCount for each prerequisite found

Concatenate a localString named preNames for each prerequisite

Add preCount to struct at preCount

Add preNames to struct at preList

Return tempList

End

searchList(String)

Create tempCourse of type bucket

Set tempCourse to the bucket at the hash location of String

Loop through list For Each Course

If String is the same as courseID

Set tempCourse to Course

Return tempCourse

End

printCourse(String)

Create tempCourse of type bucket

Set tempCourse equal to hash (String)

Loop through all chained buckets at tempCourse

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 printCourse() passing preList

End

validateList()

Create tempCourse of type bucket

Create variable valid and Set to True

For Each Course

If valid is False break

While tempCourse next is not null

Loop 0 to preCount

Set tempCourse equal to searchList(preList token)

If tempCourse courseID is empty Set valid to False

Return valid

End

int Hash(key)

Milestone 3:

Load text parsing libraries and headers

Define a struct to hold course data

struct Course {}

*courseID*

*courseName*

*preCount*

*prelist*

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

Class BinaryTree{}

-struct *Node*

*Course*

*right* pointer

*left* pointer

-*root*

*+printCourse()*

+*BinaryTree()*

Main()

Create new BinaryTree named *courseTree* of the struct-type Course

Get CSV file path from user

If no data passed use default location

Call txtParser() passing CSV file path

Call validateList() passing *courseTree*

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

Call printCourse() passing *userSearch*

End

txtParser (String)

Open file found at the path in *String* by invoking parser libraries

Loop row by row until end of file (eof)

If first and second string are present

Add the first String to struct at *courseID*

Add the second String to Struct at *courseName*

Loop until file handler has no value in a column (indicates no more prerequisite)

Increment a variable named *preCount* for each prerequisite found

Concatenate a localString named *preNames* for each prerequisite

Add *preCount* to struct at *preCount*

Add *preNames* to struct at *preList*

Return *tempList*

End

searchList(String)

Create *tempCourse* of type Node

Set *tempCourse* to the bucket at the hash location of *String*

Loop through list For Each Course

If *String* is the same as *courseID*

Set *tempCourse* to Course

Return *tempCourse*

End

printCourse(String)

Create *tempCourse* of type bucket

Set *tempCourse* equal to root

Loop until *tempCourse* is Null

If the Node at tempCourse contains a *bidId* equal than to *String*

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 printCourse() passing *preList*

If the Node at *tempCourse* contains a *courseID* less than to *String*

Set *tempCourse* equal to the left Node

If the Node at *tempCourse* contains a *courseID* greater than to *String*

Set *tempCourse* equal to the right Node

End

validateList()

Create *tempCourse* of type Node

Create variable *valid* and Set to True

For Each Course

If *valid* is False break

While *tempCourse* next is not Null

Loop 0 to *preCount*

Set *tempCourse* equal to searchList(*preList* token)

If *tempCourse* courseID is empty Set *valid* to False

Return valid

End

Menu and Course List:

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 printCourse() passing *prelist*

End

Evaluation:

Runtime Analysis:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Loading Data | Search | Sort and Print |
| Vector | **O(n)** – sequential insertions at end (amortized O(1) each) | **O(n)** – linear search | **O(n log n)** – sorting required before printing in order |
| Hash Table | **O(n)** average – O(1) insertions per item (O(n²) worst-case w/ collisions) | **O(1)** average – direct lookup (O(n) worst-case) | **O(n log n)** – must extract keys and sort for ordered output |
| Binary Search Tree | **O(n log n)** – O(log n) insertions per course | **O(log n)** average – tree traversal to find course | **O(n)** – in-order traversal naturally prints sorted |

Advantages and Disadvantages:

**Vector**

* **Advantages**: Very memory-efficient; contiguous storage is cache-friendly; simple to implement; excellent for fast sequential reads.
* **Disadvantages**: Insertions/removals in middle are expensive (**O(n)**); requires resizing when capacity exceeded; search is linear unless data is sorted; sorting adds extra cost.

**Hash Table**

* **Advantages**: Fast average-case insertions and searches (**O(1)**); no need for data to be stored in order; ideal for frequent lookups.
* **Disadvantages**: Worst-case insert/search degrade to **O(n)** with poor hashing or many collisions; unordered by nature, so sorting must be done separately; slightly higher memory usage due to bucket and pointer overhead.

**Binary Search Tree (Balanced)**

* **Advantages**: Maintains sorted order automatically; reasonable average-case insert/search (**O(log n)**); in-order traversal produces sorted output without extra sorting step.
* **Disadvantages**: Higher memory usage from storing multiple pointers per node; more complex to implement than a vector or hash table; performance suffers if tree becomes unbalanced (**O(n)** operations).

Recommendation:

Based on the runtime and memory analysis, I recommend using a **hash table** for storing course data. The advisor’s requirements emphasize efficiency in loading and validating course lists, and a hash table provides **O(1)** average-time insertion and search, making it ideal for quickly processing the data and retrieving course details. While a hash table does not store data in a sorted order, this is not a significant drawback for the main program’s needs because sorting can be done only when necessary for output. The trade-off in slightly higher memory usage is outweighed by the performance gains in search and load operations compared to vectors and trees, especially when working with large datasets. For situations where ordered data is essential on every access, a balanced binary search tree would be the better alternative, but for this project’s priorities, a hash table offers the most practical and efficient solution.