Adam Koenig

8/4/2023

CS 300

Project One

// Define course object and data structures to store them

DEFINE a Course object

String “courseNumber”

String “name”

String Vector “prerequisites”

DEFINE another string vector of all prerequisites

// Quicksort function to sort vectors

DEFINE Quicksort function with string vector argument, beginning index, end index

INIT int mid index = 0

IF beginning index is greater than or equal to end index

RETURN

END IF

SET mid equal to CALL partition on vector, beginning, and end index

RECURSIVELY CALL Quicksort on vector, beginning, and mid index

RECURSIVELY CALL Quicksort on vector, mid index + 1, and end index

DEFINE Partition function taking string vector argument, beginning index, and end index

INIT Low Index equals beginning index

INIT High Index equals end index

INIT Middle equals Low index plus (High Index minus Low Index divided by two)

INIT String Pivot equals vector at middle index

INIT Bool Done equals False

WHILE not Done

WHILE Vector at Low Index is less than the Pivot

INCREMENT Low Index

END WHILE

WHILE Pivot is less than Vector at High Index

DECREMENT High Index

END WHILE

IF Low Index is greater than or equal to High Index

SET Done equal to True

ELSE

SWAP Vector at Low Index with Vector at High Index

INCREMENT Low Index

DECREMENT High Index

END IF  
 END WHILE

RETURN High Index

// Vector Data Structure

DEFINE a vector of course objects called Course list

// Hash Table Data Structure

DEFINE Hash Table Data Structure

- DEFINE Node Object

Course Object “Course”

Integer “key”

Node Pointer “next”

- Unsigned int “tableSize”

- Node Vector “nodes”

-String Vector “keys”

- DEFINE Hash Function with “key” argument

RETURN key mod tablesize

+ DEFINE Insert Function with course argument

SET Index equal to Hash Function(Course.courseNumber)

IF node at Index is empty

SET node at Index’s course object to Course

SET node at Index’s key to Course.courseNumber

ENDIF

ELSE

INIT new node equal to Course

SET last node in linked list at Index to new node

ENDIF

+ DEFINE Print All Function

FOR each element in the nodes vector

IF the node at the index is not empty

OUTPUT the course information stored

IF there is a linked list at the node

WHILE there are more nodes in the linked list

OUTPUT the course information

ITERATE to the next node

ENDWHILE

ENDIF

ENDIF

ENFOR

+DEFINE Print In Order Function

INIT empty Course Object

CALL Quicksort on keys vector

FOR each element in keys vector

SET Course Object equal to CALL Search function on index

OUTPUT Course Object

END FOR

+ Define Search Function with course name argument and course object return type

INIT empty Course Object

SET Index equal to Hash Function(course name)

IF nodes at Index’s key is equal to course name

RETURN nodes at Index’s course object

ELSE IF nodes at Index is empty

RETURN empty Course Object

ELSE

INIT Node Pointer equal to nodes at Index’s next pointer

WHILE Node pointer is not equal to null pointer

IF Node pointer’s key is equal to course name

RETURN Node pointer’s course object

ENDIF

SET Node pointer equal to Node pointer’s next pointer

ENDWHILE

ENDIF

RETURN Course Object if no matching course number was found

// Binary Search Tree Data Structure

DEFINE Binary Search Tree Object

- DEFINE Node Object

Course Object “Course”

Node Pointer “left”

Node Pointer “right”

- Node Pointer root

- DEFINE addNode function with node pointer and course argument

IF node’s course name is greater than course argument’s course name

IF node’s left pointer is a null pointer

INITIALIZE Node pointer newNode with course argument

SET node’s left pointer equal to newNode

ELSE

RECURSIVELY CALL addNode passing node’s left pointer and

and course argument

END IF

ELSE

IF node’s right pointer is a null pointer

INITIALIZE Node pointer newNode with course argument

SET node’s right pointer equal to newNode

ELSE

RECURSIVELY CALL addNode passing node’s right pointer and

Course argument

END IF

END IF

- DEFINE inOrder function with node pointer argument

IF node is not equal to null pointer

RECURSIVELY CALL inOrder passing node’s left pointer

OUTPUT node’s course information

RECURSIVELY CALL inOrder passing node’s right pointer

END IF

+ DEFINE Insert Function with course argument

IF root is equal to null pointer

INITIALIZE Node pointer “newNode”

SET newNode’s course with course argument

SET root equal to new Node

ELSE

CALL addNode passing root and course

END IF

+ DEFINE InOrder function

CALL inOrder function passing root

+ Define Search Function with course name argument and course object return type

INITIALIZE Node pointer currentNode equal to root

WHILE currentNode is not equal to null pointer

IF currentNode’s course name is equal to the course name argument

RETURN currentNode’s course object

ELSE IF currentNode’s course name is greater than the course name

SET currentNode equal to currentNode’s left pointer

ELSE

SET currentNode equal to currentNode’s right pointer

END IF

END WHILE

\\ If no matching course was found, return an empty course object

RETURN course object

\\ Open and read data from the file, checking for errors and storing them in the course object data structures

DEFINE Load File Function with string file name argument

OPEN the csv file

WHILE there are more lines in the file

INIT a string with the line in the file

SPLIT the string separated by commas

IF the string split into less than two parameters

The file is formatted incorrectly

End reading the file

ENDIF

CREATE a new course object

SET the course number equal to the first parameter

SET the course name equal to the second parameter

FOR the number of remaining parameters

APPEND the parameter to the course object’s prerequisite vector

IF the parameter is not in the vector of all prerequisites

APPEND the parameter to the vector of all prerequisites

ENDIF

ENDFOR

APPEND Course Object to the Course List Vector

CALL Hash Table’s Insert Function using Course Object

CALL Binary Search Tree’s Insert Function using Course Object

ENDWHILE

CLOSE the csv file

FOR every prerequisite in the vector of all prerequisites

IF there is not a matching course number in the vector of course object

The file is formatted incorrectly

ENDIF

ENDFOR

// Menu Pseudocode

DEFINE Int User Choice = 0

WHILE User Choice does not equal 4

OUTPUT 1. Load Data Structure NEWLINE

OUTPUT 2. Print Course List NEWLINE

OUTPUT 3. Print Course NEWLINE

OUTPUT 4. Exit NEWLINE

INPUT User Choice

SWITCH User Choice:

1: CALL Load File function

OUTPUT File successfully loaded

BREAK

2:

\\ For Vector

CALL Quicksort on Course List Vector

FOR the size of Course List

OUTPUT index’s Course Object

ENDFOR

\\ For Hash Table

CALL hash table’s Print In Order function

\\ For Binary Search Tree

CALL BST’s In Order Function

BREAK

3: INPUT String Course Name

// For vector data structure

FOR the size of Course List

IF the Course Name at the index equals Course Name

OUTPUT index’s Course Object

END IF

END FOR

// For hash table data structure

SET Course Object equal to CALL hash table’s Search function

OUTPUT Course Object

// For Binary Search Tree

SET Course Object equal to CALL BST’s Search function

OUTPUT Course Object

BREAK

4 : BREAK

Default:

OUTPUT Invalid Menu Choice

BREAK

END WHILE

|  |  |  |  |
| --- | --- | --- | --- |
| Worst Case Runtime | | | |
|  | Adding Course to Data Structure | Searching for a specific Course | Sorting and Printing Course List |
| Vector | O(1) | O(N) | O(N3) |
| Hash Table | O(N) | O(N) | O(N3) |
| Binary Search Tree | O(N) | O(N) | O(N) |

The Vector data structure excels in adding Course Objects. This operation only requires one line of code and runs in constant time. However, the worst case runtime for the Search operation is linear time. If the Course Object does not exist in the Vector, a loop must iterate through the entire structure. Sorting and printing the vector takes two operations. The Vector uses the Quicksort algorithm, which has a quadratic worst case runtime. The average runtime of Quicksort is much better, being O(N LOG N). In order to print the sorted algorithm, a for loop is used running in linear time.

The Hash Table supports fast inserts and searches. Due to handling collisions using chaining, the worst case runtime for each of these operations is linear. This scenario only happens if each Course Object in the data set corresponds to the same hash value. The average runtime of each of these operations is constant. The main issue with the Hash Table is that there isn’t a convenient way to print the Hash Table in alphabetical order. The solution I came up with was to store each key in a vector as the Course Object is added to the Hash Table. Quicksort is then run on the vector of keys. A for loop is then used to iterate through the sorted vector. The Hash Table’s Seach function is called on the key, and the returned Course Object is printed. All of this means that the runtime of the Hash Table’s Sort and Print operation is just as long as the Vector’s Sort and Print operation.

The Binary Search Tree is great at Searching and Printing the sorted list. The BST uses recursion for inserting Course Objects. In the worst case scenario, objects will be added to the tree in either ascending or descending order. For each item added, the tree would have to call the insert function for the number of course objects in the tree. In this case, the Search function would also have to visit every node in the tree if searching for a Course Object that does not exist. The average runtime for these functions is logarithmic. In an ideal tree, at each level of search would eliminate half of the remaining tree. The main benefit of the Binary Search Tree is that Course Objects are sorted as they are added to the tree. A Print function only needs to visit the nodes in order to print them in alphanumeric order. The Print function runs in constant time.

With all of this in mind, I think the best data structure for this scenario is the Binary Search Tree. The only area that the Binary Search Tree is weaker than the other options is in Inserting Course Objects. It supports faster Search than the Vector, although the average Search time is not as fast as the Hash Table. It really excels in printing the Course List in alphanumeric order, being the only data structure to accomplish this in linear time.