**4-2 Assignment: Hash Tables**

Cristiano Miranda

Southern New Hampshire University

CS-300: Analysis and Design

Professor Muthukudage

28 September 2025

**4-2 Assignment: Hash Tables**

**Code Reflection:**

This program was created to allow a user to input data from a file and store it in a hash table, which assigns the position of the data based on its bidId. In the instance where collisions occur, a singularly linked list is established at said hash table index in which new nodes are added to the singularly linked list to avoid overwriting data. The program also allows the user to search for and remove specific data by removing the nodes from their position in the singularly linked list associated with their key index in the hash table. The purpose of creating this program was to practice implementing a hash table and to understand how effective they are at searching for and potentially removing data. Having the chance to practice using a hash table in this scenario helps highlight that its near-constant time complexity improves the user experience when attempting to search for or remove data. Other data structures would have resulted in longer run times when trying to search for or access any data within the list. In this program, the hash table utilizes chaining as opposed to other methods, such as linear probing, to solve issues with collisions. While this introduces additional complexities as there are multiple singularly linked lists to manage, it results in a faster-running program. Rather than having to search the entire vector or linked list, one element at a time, the hash table utilizes the bidId value from the data to calculate a key that becomes associated with where the data is stored in the hash table. Thus, even though there is the potential of having to search through the singularly linked list to find the desired data, the singularly linked list's length is far smaller and thus quicker to search through. The hash table demonstrates its effectiveness through nearly constant execution time, as it performs searches at almost the same runtime, regardless of the data being searched for, since the calculation for the hash key remains consistent regardless of the data. Still, there are instances where accessing specific data can result in longer execution times, such as if all data shared the same hash key value. The worst-case runtime complexity could be O(N), but this scenario is rare and can be easily solved by resizing the array or modifying the hash key calculation. Having the ability to access data with an average time complexity of O(1) can significantly improve a program where data searching is common, as it allows the search of said data to perform nearly consistently, providing a better user experience since wait time is consistent and not random, regardless of the data the user is searching for.

            Implementing the hash table required a combination of two data structures: a vector and a singularly linked list. To start with, a vector is initialized to the desired size, and a hash key calculation function is created so that all added data can calculate their respective index. In this case, we used the unique ID values of the bids and calculated their remainder after dividing them by the vector's size value, using modulus, to determine their placement in the vector. From here, node objects are initialized with the data and the key value, and if the index is empty, they are placed at said index. However, in the instance that the index is already occupied, a singularly linked list is used by assigning the last node in the list to point towards the new node via its next pointer property. Through this, a vector is used to store multiple singularly linked lists, all of which are separated based on the hash key calculated using the hash key function. Through this, the elements are separate throughout the singularly linked list, and by hashing a bid ID when searching, the program will know exactly which singularly linked list to search through for the data; thus, it avoids spending time searching through all elements and instead focuses its search on the singularly linked list in which it should be stored.

            By far, the biggest challenge I faced while implementing the hash table was the constant confusion and issues involving the various pointers that need to be initialized, re-pointed, and converted to their associated data. For instance, there were many instances in which I struggled to determine whether a certain index was null, if it had been previously populated, or if it was currently populated. While I understood that pointers were the solution to my issue, I had trouble implementing them because I experienced many problems involving type conflicts, as I ended up getting my process mixed up. Instead of assigning pointers to point towards new nodes, I attempted to assign the node's data to the pointer. To finally solve this problem and avoid experiencing it throughout the project, I took the time to reflect on the purpose of the pointers and try to identify instances where I no longer needed them. In doing this, I realized that the main issue I was experiencing was needing to assign the information associated with the pointer, rather than the memory addresses the pointer was referring to. Upon figuring this out, it became easier to research ways to retrieve data from a pointer and assign it to a variable assigned with the associated data type. In the end, while this mistake was simply a misunderstanding and error on my part, it highlighted the importance of fully understanding the issues one faces in order to research and solve them more effectively.

**Pseudocode:**

* STRUCT Bid
  + INIT bidId
  + INIT title
  + INIT fund
  + INIT amount
  + CONSTRUCTOR ()
    - SET amount = 0.0
  + ENDCONSTRUCTOR
* ENDSTRUCT
* CLASS HashTable
  + DEFINE PRIVATE STRUCT Node
    - INIT BID
    - INIT KEY // key should never be negative
    - INIT POINTER next
    - CONSTRUCTOR ()
      * START
      * SET key = MAXIMUM INTEGER VALUE
      * POINT next TO null
      * END
    - END CONSTRUCTOR
    - CONSTRUCTOR (aBid)
      * START
      * SET bid = aBid
      * CALL CONSTRUCTOR () // call default constructor
      * END
    - END CONSTRUCTOR
    - CONSTRUCTOR (aBid, aKey)
      * START
      * SET key = aKey
      * CALL CONSTRUCTOR (aBid) // call the other constructor
      * END
    - END CONSTRUCTOR
  + ENDDEFINE STRUCT
  + INIT PRIVATE VECTOR nodes
  + INIT PRIVATE tableSize = DEFAULT\_SIZE // Set to value provided on program start
  + INIT PRIVATE hash(key)
  + PUBLIC CONSTRUCTOR ()
    - START
    - RESIZE nodes VECTOR TO tableSize
    - END
  + ENDCONSTRUCTOR
  + PUBLIC CONSTRUCTOR (size)
    - START
    - SET tableSize = size
    - RESIZE nodes VECTOR TO size
    - END
  + ENDCONSTRUCTOR
  + PUBLIC DECONSTRUCTOR
    - START
    - ERASE all elements FROM nodes
    - END
  + ENDDECONSTRUCTOR
  + PUBLIC METHOD hash (key)
    - START
    - CALCUALTE key MOD tableSize
    - RETURN calculation result
    - END
  + ENDMETHOD
  + PUBLIC METHOD Insert(bid)
    - START
    - INIT nodeKey
    - CALL hash (bid.bidId) AND SET nodeKey = result
    - INIT POINTER currNode AND POINT TO element AT nodes[nodeKey]
    - IF currNode IS POINTING TO null THEN
      * INIT new Node (bid, nodeKey) and POINT currNode to new Node
    - ELSE IF currNode->key IS EQUAL TO MAXIMUM INTEGER VALUE THEN
      * SET currNode->bid = bid
      * SET currNode->key = nodeKey
    - ELSE
      * WHILE currNode->next IS NOT POITING TO null
        + POINT currNode TO currNode->next
      * ENDWHILE
      * INIT new Node (bid, nodeKey) AND POINT currNode->next TO new Node
    - ENDIF
    - END
  + ENDMETHOD
  + PUBLIC METHOD PrintAll ()
    - START
    - FOR index FROM 0 UNTIL GetLength(nodes)
      * INIT POINTER currNodes AND POINT TO element AT nodes[index]
      * IF currNode IS NOT PIONTING TO null AND currNode->key IS NOT EQUAL TO MAXIMUM INTEGER VALUE THEN
        + WHILE currNode IS NOT POINTING TO null

INIT bid = currNode->bid

DISPLAY currNode->key + “ | ” + bid.bidId + “: ” + “ | ” + bid.title + “ | ” + bid.amount + “ | ” bid.fund

POINT currNode TO currNode->next

* + - * + ENDWHILE
      * ENDIF
    - ENDFOR
    - END
  + ENDMETHOD
  + PUBLIC METHOD REMOVE (bidId)
    - START
    - INIT nodeKey
    - CALL hash (bid.bidId) AND SET nodeKey = result
    - INIT POINTER currNode AND POINT TO element AT nodes[nodeKey]
    - IF currNode IS NOT POINTING TO null AND currNode->key IS NOT MAXIMUM INTEGER VALUE THEN
      * IF currNode->next IS POITING TO NULL THEN
        + SET currNode->key TO MAXIMUM INTEGER VALUE
        + RETURN
        + END
      * ELSE
        + WHILE currNode->next IS NOT POINTING TO null

POINT currNode TO currNode->next

IF currNode->bid.bidId IS EQUAL TO bidId THEN

INIT bid

SET bid = currNode->bid

DISPLAY currNode->key + “ | ” + bid.bidId + “: ” + “ | ” + bid.title + “ | ” + bid.amount + “ | ” bid.fund

SET element AT nodes[nodeKey] TO currNode->next node

DISPLAY “Removed Bid”

RETURN

END

ENDIF

* + - * + ENDWHILE
      * ENDIF
    - ENDIF
    - DISPLAY “Associated node not found!”
    - END
  + END METHOD
  + PUBLIC METHOD Search (bidId)
    - START
    - INIT bid = new Bid ()
    - INIT nodeKey
    - CALL hash (bid.bidId) AND SET nodeKey = result
    - INIT POINTER currNode AND POINT TO element AT nodes[nodeKey]
    - IF currNode IS NOT POINTING TO null AND currNode->key IS NOT EQUAL TO MAXIMUM INTEGER VALUE THEN
      * WHILE currNode IS NOT POITING TO null
        + IF currNode->bid.bidId IS EQUAL TO bidId THEN

RETURN currNode->bid

END

* + - * + ENDIF
        + POINT currNode TO currNode->next
      * ENDWHILE
    - ENDIF
    - RETURN bid
    - END
  + ENDMETHOD
* ENDCLASS
* FUNCTION displayBid (bid)
  + START
  + DISPLAY bid.bidId + “: ” + “ | ” + bid.title + “ | ” + bid.amount + “ | ” bid.fund
  + RETURN
  + END
* ENDFUNCTION
* FUNCTION loadBids (csvPath, hashtable)
  + START
  + DISPLAY “Loading CSV file”
  + INIT FILE
  + CALL csv::Parser (csvPath) AND SET file = result // read data from the csv file
  + INIT vector<string> header
  + SET header = file.getHeader() // get header rows from csv file
  + FOR EACH headerName in header
    - DISPLAY headerName
  + ENDFOR
  + TRY
    - FOR index FROM 0 UNTIL file.rowCount() // iterate through all rows of the file
      * INIT bid
      * SET bid.Id = file[index][1]
      * SET bid.Title = file[index][0]
      * SET bid.fund = file[index][8]
      * SET bid.amount = file[index][4]
      * INSERT bid INTO hashTable
    - ENDFOR
  + CATCH (error)
    - DISPLAY error
  + ENDTRY
  + END
* ENDFUNCTION
* FUNCTION strToDouble (str, ch)
  + START
  + REMOVE ch from str
  + RETURN STR
  + END
* ENDFUNCTION
* FUNCTION main (argc, argv)
  + START
  + INIT csvPath AND bidKey
  + IF argc IS EQUAL TO 2 THEN
    - SET csvPath = element AT argv[1]
    - SET bidKey = “98223”
  + ELSE IF argc IS EQUAL TO 3 THEN
    - SET csvPath = element AT argv[1]
    - SET bidKey = element AT argv[2]
  + ELSE
    - SET csvPath = “eBid\_Monthly\_Sales.csv”
    - SET bidKey = “98223”
  + INIT ticks
  + INIT bidtable
  + INIT bid
  + SET bidTable TO new HashTable
  + INIT choice
  + SET choice = 0
  + WHILE choice IS NOT EQUAL TO 9
    - DISPLAY “MENU:

1. LOADBIDS
2. DISPLAY ALL BIDS
3. FIND BID
4. REMOVE BID
5. EXIT

Enter Choice:”

* + - GET choice
    - IF choice IS EQUAL TO 1 THEN
      * SET ticks = clocks () // gets starting time
      * CALL loadBids (csvPath, bidTable)
      * SET ticks = clock () – ticks //gets execution run time
      * DISPLAY “time: ” + ticks + “ clock ticks”

// converts the clock ticks value to seconds using a predefined constant variable

* + - * DISPLAY “time: ” + ticks \* 1.0 / CLOCKS\_PER\_SEC + “ seconds”
    - ELSE IF choice IS EQUAL TO 2 THEN
      * CALL PrintAll () ON bidTable
    - ELSE IF choice IS EQUAL TO 3 THEN
      * SET ticks = clocks () // gets starting time
      * INIT bid
      * CALL Search (bidKey) ON bidTable AND SET bid = result
      * SET ticks = clock () – ticks //gets execution run time
      * IF bid.bidId IS NOT EMPTY THEN
        + CALL displayBid (bid)
      * ELSE
        + DISPLAY “Bid Id ” + bidKey + “ not found.”
      * ENDIF
      * DISPLAY “time: ” + ticks + “ clock ticks”

// converts the clock ticks value to seconds using a predefined constant variable

* + - * DISPLAY “time: ” + ticks \* 1.0 / CLOCKS\_PER\_SEC + “ seconds”
    - ELSE IF choice IS EQUAL TO 4 THEN
      * CALL Remove (bidKey) ON bidTable
    - ELSE IF choice IS EQUAL TO 9 THEN
      * BREAK
    - ELSE
      * DISPLAY “Invalid Input”
      * SET choice = 0
      * CLEAR user input
    - ENDIF
  + ENDWHILE
  + DISPLAY “Good bye.”
  + END
* ENDFUNCTION