**Hash Tables: Code Reflection and Pseudocode**

Justin Starr

Department of STEM

CS 300 – DSA: Analysis and Design

Professor Ricardo Scarello

June 2, 2024

**Hash Tables: Code Reflection and Pseudocode**

**Code Reflection**

The purpose of this code was to be able to load a list of bids from a CSV file and insert the bids into a HashTable used to store bids (each node in the hash table) and their data. The insert function hashes the passed in bidId and creates a new node in the hashTable with the given bid. The PrintAll function iterates over the nodes data structure and prints all of the bids. The Remove function hashes the passed in bidId and assigns it to a key. It then removes the node from the node’s data structure at the key position. The search function again hashes the passed in bidId and assigns it to a key. It then tries to retrieve the node using the key. If the node is found, it returns that node’s bid. Otherwise, it returns an empty bid.

Implementing the various functions demonstrates an understanding of how insertion, deletion, and searching algorithms/operations work when using a hash table as a data structure. This assignment also demonstrates our understanding of chaining (a collision resolution technique) for when it might be necessary to store two items in the same location because the same key is associated with more than one value, or because two keys hashed to the same location in the hash table, and how we can use the method of probing to find the values associated with a specific key. It is also demonstrates what the computational time is for these functions, and how fast they occur, making hash tables a suitable structure for storing data, especially when working with large data sets, and when the size of this data can change dynamically.

For this assignment, I did not encounter too many issues. I was able to follow along with the resource that was provided this week. The only problem I had was when I was implementing the PrintAll function, I was using tableSize at first to iterate in my for loop instead of nodes.size(). This caused my application to exit unexpectedly. It took me a little while to realize what I was doing wrong, however, after spending the necessary time to debug, I was able to correct the issue.

**Pseudocode**

The following is the pseudocode for the main function and all of the functions that required fixing:

HashTable::HashTable(), HashTable::HashTable(unsigned int size), HashTable::~HashTable(), unsigned int HashTable::hash(int key), void HashTable::Insert(Bid bid), void HashTable::PrintAll(), void HashTable::Remove(string bidId), Bid HashTable::search(string bidId)

**int main(int argc, char\* argv[])** function **{**

DEFINE csvPath (type string) and searchValue (type string)

SWITCH (argc) {

case 2:

STORE csvPath = argv[1]

STORE searchValue = “98223”

BREAK

END case 2

case 3:

STORE csvpath = argv[1];

STORE searchValue = argv[2]

BREAK

END case 3

default:

STORE csvPath with the name of the csv file

STORE searchValue = “98109”

END DEFAULT

DEFINE ticks (type clock\_t)

DEFINE bidTable (type HashTable) and INITIALIZE the new hash table

DEFINE bid (type Bid)

DEFINE choice (type int) and INITIALIZE to zero

WHILE LOOP choice not equal to 9 {

DISPLAY “MENU:” and ENDLINE

DISPLAY “1. Load Bids” and ENDLINE

DISPLAY “2. Display all Bids” and ENDLINE

DISPLAY “4. Find Bid” and ENDLINE

DISPLAY “5. Remove Bid” and ENDLINE

DISPLAY “ Enter Choice: “ and ENDLINE

INPUT choice

SWITCH (choice) {

case 1:

STORE ticks = CALL clock() // current clock ticks

CALL loadBids(csvPath, bidTable)

STORE ticks = CALL clock() – ticks // current clock ticks

minus starting clock ticks

DISPLAY “time:”, ticks, “ clock ticks”

DISPLAY “time:”, ticks \* 1.0 / CLOCKS\_PER\_SEC, “ seconds”

BREAK

END case 1

case 2:

CALL PrintAll for bidTable

BREAK

END case 2

case 3:

STORE ticks = CALL clock() // current clock ticks

STORE bid = CALL Search(searchValue) in bidTable

STORE ticks = CALL clock() – ticks // current clock ticks minus

starting clock ticks

IF the returned bid is not empty {

CALL displayBid(bid)

} END IF

ELSE {

DISPLAY “Bid Id”, searchValue, “not found”

} END ELSE

DISPLAY “time:”, ticks, “ clock ticks”

DISPLAY “time:”, ticks \* 1.0 / CLOCKS\_PER\_SEC, “seconds”

BREAK

END case 3

case 4:

CALL Remove(searchValue) on bidTable

BREAK

END case 4

} END SWITCH

} END WHILE LOOP

DISPLAY “Good bye.”

RETURN 0

} END main function

**HashTable::HashTable()** function {

RESIZE node structure with tableSize

} END function

**HashTable::HashTable(unsigned int size)** function {

INVOKE local tableSize to size

RESIZE node structure with tableSize

} END function

**HashTable::~HashTable()** function {

ERASE nodes beginning

} END function

**Unsigned int HashTable::hash(int key)** function {

// calculate hash value and return it

RETURN key modulo tableSize

} END function

**void HashTable::Insert(Bid bid)** function {

ASSIGN key (type unsigned) by CALLING hash and passing the bidId for the given bid

ASSIGN oldNode with the node at the key position

IF oldNode is nullptr {

CREATE newNode with the given bid and created key

INSERT newNode to the nodes structure at the key position

} END IF

ELSE if the node is not used {

IF oldNode’s key is UINT\_MAX {

ASSIGN oldNode’s key to the created key

ASSIGN oldNodes’s bid to the given bid

POINT oldNode’s next node to nullptr

} END IF

ELSE find the next open node {

LOOP WHILE oldNode’s next does not point to nullptr {

ASSIGN oldNode to oldNode’s next node

} END WHILE LOOP

CREATE new node with the given bid and created key and ASSIGN it to

oldNode’s next

} END ELSE

} END ELSE

} END function

**Void HashTable::PrintAll()** function {

LOOP FOR all nodes in the nodes data structure {

ASSIGN searchNode (type node) with the node at the current position

IF searchNode’s key is not equal to UINT\_MAX {

DISPLAY searchNode’s bidId, title, amount, and fund

ASSIGN searchNode to searchNode’s next node

WHILE LOOP searchNode is not nullptr {

DISPLAY searchNode’s bidId, title, amount, and fund

ASSIGN searchNode to searchNode’s next node

} END WHILE LOOP

} END IF

} END FOR LOOP

} END function

**Void HashTable::Remove(string bidId)** function {

DEFINE tempKey (type unsigned) and ASSIGN it by CALLING hash and passing the bidId for the given bid

ERASE node begin and tempKey

} END function

**Bid HashTable::Search(string bidId)** function {

DEFINE tempKey (type unsigned) and ASSIGN it by CALLING hash and passing the bidId for the given bid

ASSIGN node with the node at the tempKey position in the nodes data structure

IF no entry is found for the tempKey {

RETURN bid

} END IF

IF an entry is found for tempKey {

RETURN node’s bid

} END IF

WHILE LOOP node is not nullptr {

IF node’s bidId matches the given bidId {

RETURN node’s bid

} END IF

ASSIGN node to node’s next node

} END WHILE LOOP

RETURN bid

} END function