Southern New Hampshire University

Chris Trimmer

CS-300-T1159 DSA: Analysis and Design

Assignment 4-2: Hash Tables Reflection

22EW1 – 09/23/2022

The purpose of this document is to provide a code reflection of assignment 4-2. Assignment 4-2 is focused on hash tables. Hash tables are highly effective at searching through large amounts of records. It works by using a key value that is hashed and assigned as a "bucket". Each bucket maintains its own list of values that can be searched based on the key. This is the power of a hash table. We essentially gain constant time O(1) insertion, deletion, and search in a table consisting of huge amounts of data.

Code Reflection

Our code for this assignment includes the core operations that are part of a hash table: the hashing algorithm, insert, remove, search, and print. The key is hashed based on the bidId of the bid object. We set the hash table size to be modulus of 179. Thus, the resulting hash values form the keys for the lookup table, and they will be indexes in the vector. When searching for a bidId, we can map the hash value to the correct index of the vector. When the index is found, we simply traverse the list of bids stored at that index to find the respective bidId.

The main thing to contend with is collisions. A collision occurs when bidIds hash to the same value. In this scenario, we are using chaining to deal with collisions. In effect, when a collision occurs, we store the bid in a linked list at that vector index. Overall, this was an enlightening assignment. I didn't encounter any trouble or issues with implementing the hash table. I followed the steps outlined in our assignment and tested each step as I coded. This helped ensure that I found bugs and corrected any problems before moving to each subsequent step.

Southern New Hampshire University

Chris Trimmer

CS-300-T1159 DSA: Analysis and Design

Assignment 4-2: Hash Tables Pseudocode

22EW1 – 09/23/2022

The purpose of this document is to provide pseudocode of the main functions used in assignment 4-2. Assignment 4-2 is focused on hash tables. Hash tables are highly effective at searching through large amounts of records. It works by using a key value that is hashed and assigned as a “bucket”. Each bucket maintains its own list of values that can be searched based on the key. This is the power of a hash table. We essentially gain constant time O(1) insertion, deletion, and search in a table consisting of huge amounts of data.

The following is the pseudocode for the core functions:

<u>Pseudocode</u>

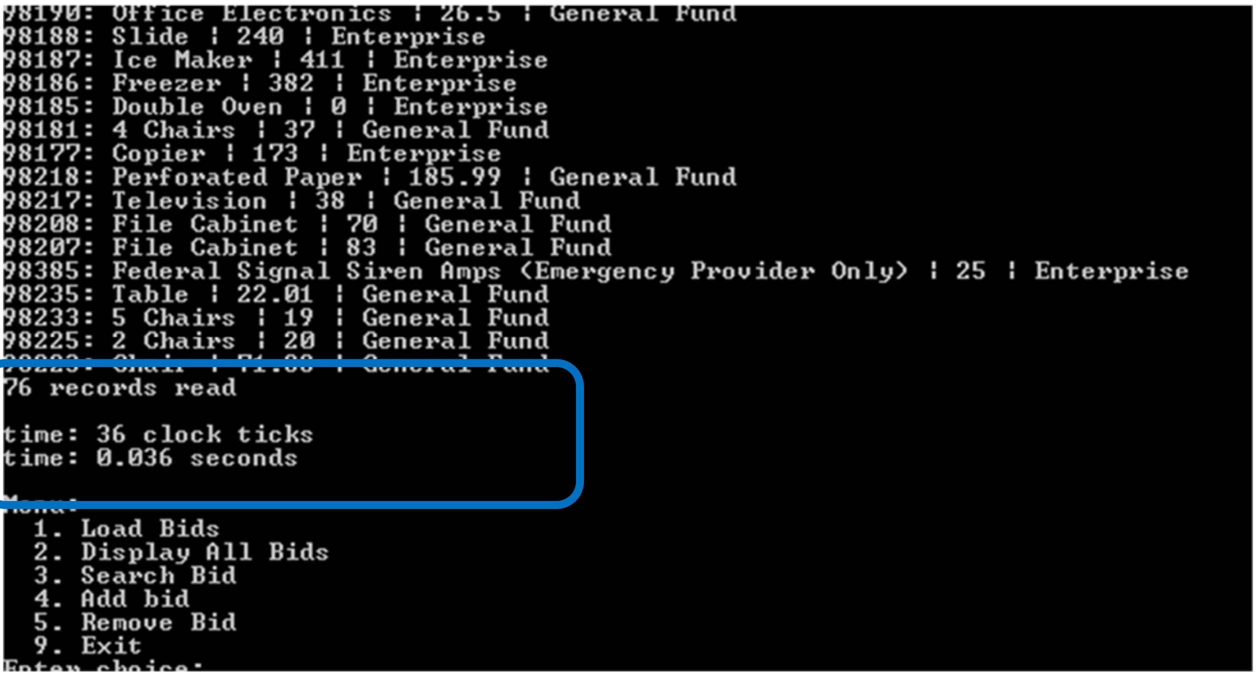
```
// Hash function
Unsigned int HashTable::hash(int key) {
  Return the key modulo tableSize;
}

// Insertion
Void HashTable::Insert(Bid bid) {
  Create key for the node equal to the hash of the bidId
  Retrieve the node using the key

  If the node is null, then assign the bid to this index
    Else, if the node is not used, then set the key for this node, the new bid, and set next to null
      Else, if the node is in use, then walk the list of nodes at this location and add the bid to the
end
}

// Print the table
Void HashTable::PrintAll() {

  For each node in the vector
    If the node at current index is not equal to UINT_MAX
      Then print the bid stored at this node
        Also walk the list of bids stored here if there are more in the list and print them
}

// Remove a node
Void HashTable::Remove(string bidId) {
  Search for the bid using bidId
  If the bid does not exist, then return to caller immediately

  Create a key based on hash value of the bidId
  Create a pointer to the node at the hashed key

  If the next node is null, then we can erase this node and return

  If the node is a head node, then we need to make the next node the head, and delete the old
head in this bucket

  Otherwise, we need to walk the list until we find the bidId that we want to delete
    If we found the node, then reset the pointers and delete the node

}
```

// **Search**

```
Bid HashTable::Search(string bidId) {

  Create a key based on the hash value of the bidId
  If there is an entry, then return the node to the caller

  If there is no entry for the key, and the key is UINT_MAX, then return a dummy bid

  If this not a special case, then walk the sub-list of objects at this index, and search for the bidId
    When found, return the bid object to the caller

}
```

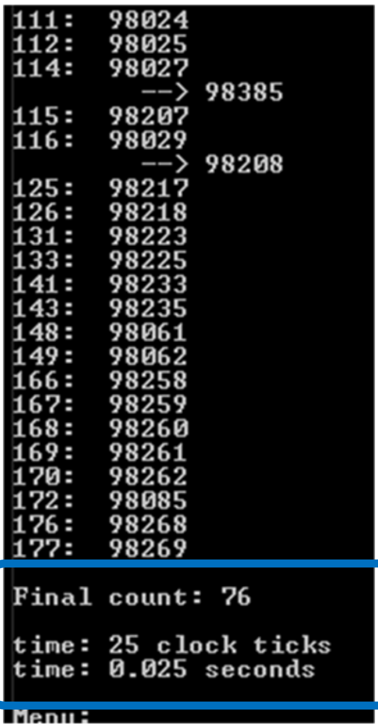
Screenshots

Speed of Load function



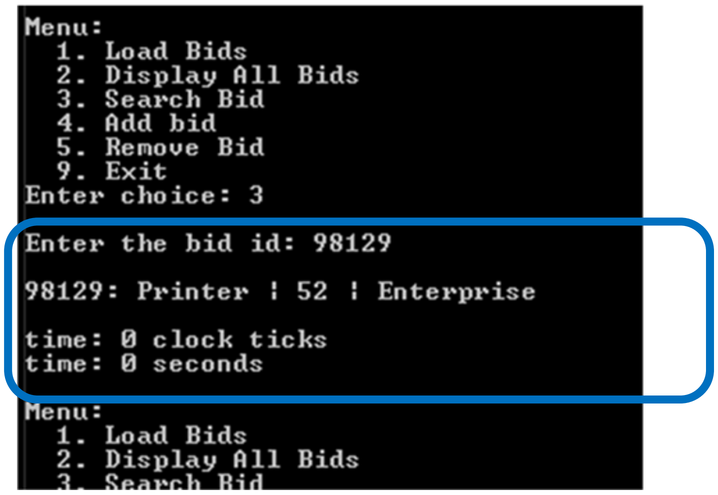
```
98190: Office Electronics ¦ 26.5 ¦ General Fund
98188: Slide ¦ 240 ¦ Enterprise
98187: Ice Maker ¦ 411 ¦ Enterprise
98186: Freezer ¦ 382 ¦ Enterprise
98185: Double Oven ¦ 0 ¦ Enterprise
98181: 4 Chairs ¦ 37 ¦ General Fund
98177: Copier ¦ 173 ¦ Enterprise
98218: Perforated Paper ¦ 185.99 ¦ General Fund
98217: Television ¦ 38 ¦ General Fund
98208: File Cabinet ¦ 70 ¦ General Fund
98207: File Cabinet ¦ 83 ¦ General Fund
98385: Federal Signal Siren Amps (Emergency Provider Only) ¦ 25 ¦ Enterprise
98235: Table ¦ 22.01 ¦ General Fund
98233: 5 Chairs ¦ 19 ¦ General Fund
98225: 2 Chairs ¦ 20 ¦ General Fund
98223: Chair ¦ 71.88 ¦ General Fund
76 records read

time: 36 clock ticks
time: 0.036 seconds

Menu:
  1. Load Bids
  2. Display All Bids
  3. Search Bid
  4. Add bid
  5. Remove Bid
  9. Exit
Enter choice:
```

Speed of Display function



```
111:  98024
112:  98025
114:  98027
          --> 98385
115:  98207
116:  98029
          --> 98208
125:  98217
126:  98218
131:  98223
133:  98225
141:  98233
143:  98235
148:  98061
149:  98062
166:  98258
167:  98259
168:  98260
169:  98261
170:  98262
172:  98085
176:  98268
177:  98269

Final count: 76

time: 25 clock ticks
time: 0.025 seconds

Menu:
```

Speed of Search function



```
Menu:
   1. Load Bids
   2. Display All Bids
   3. Search Bid
   4. Add bid
   5. Remove Bid
   9. Exit
Enter choice: 3

Enter the bid id: 98129

98129: Printer | 52 | Enterprise

time: 0 clock ticks
time: 0 seconds

Menu:
   1. Load Bids
   2. Display All Bids
   3. Search Bid
```