CS 3310: Data and File Structures

HW3: Report

Instructor: Ajay Gupta

TA: Rajani Pingili

Altan Rawal

Problem Statement:

1. Read an input text file and stored into an array(*itemsArray*)
2. Create a pseudo-random list that can decides which element to use to fill hash table first from the *itemsArray*
3. Fill *n* bags with items from *itemsArray* by assigning a random strength
4. Create a hashtable with hashfunction1 and open hashing:
    a. Search for a random item from *itemsArray* in the set of bags(*inventory*)
    b. Record the time for a single search
    c. Record the average search time
5. Repeat for the *inventory* using hashfunction1 and closed hasing(linear, pseudo random and double hashing)
6. Repeat step 3 and 4 with hasingfunction2 and hasingfunction3.

Algorithms Descriptions:

Hashing to table:

Takes input on which hash function and hash table to use and execute appropriate one as per the request and returns the array of hash tables.

Hash function 1:

Uses the inbuilt hash function to hash the string key (combines *rarity* then *itemname*) then takes the modulus by dividing by the table size

Hash function 2:

Uses the inbuilt hash function to hash the string key then square the value and then take the mid 3 values. If the 3 digits are larger than table size then replace the 100^{th} place by a 1. Eg. 922 becomes 122

Hash function 3:

Divides the string into equal size of 4 , adding 1 if necessary. Multiplying the ascii of 4 pairs together after adding them then taking the modulus of the number.

Chain hashing:

Takes input for which hash function to use. Hash the given key for a slot number. If the slot is empty just append the data if not call the rehash function that walks down the DLL and append it to the end.

Double hashing:

Takes input for which hash function to use. Hash the given key for a slot number. If the slot is not empty then use the prime number to rehash for a new slot number if that is also not empty then add the last slot number with the index from prime number hash and take the modulus of that to fit within the table range.

Linear Probe hashing:

Takes input for which hash function to use. Hash the given key for a slot number. If the given slot is not empty, then add 1 to the given slot and rehash it using the same function.

Pseudo Random Hash Table:

Uses the inbuilt library to create a list that is permutation of slots between 1-199 with 0 always being first element. Takes input for which hash function to use. Hash the given key for a slot number. If the slot is not available, then look up at the table add the value from the permutation list which always starts from the initially hashed slot. E.g. If the slot hashed was 16 then the it will look up for 16+12, if not then 16+22, and so on

.

Pseudo-code:

- HashingToTable(GameItem[][] inv, INT count, STRING table_type,STRING hash_type):
    - TABLETYPE tBag = new array[]
    - LIST psudo_list_bag_fill = PERMUTIZE(0, LEN(inv))
    - INT seq_counter
    - FOR (i=0; i< count; i++):
        - IF table_type == ‘open’:
            - Hash_table = OpenHashTable(199, hash_type)
        - ELSEIF table_type == ‘linear’:
            - Hash_table = LinearHashTable(199, hash_type)
        - ELSEIF table_type == ‘double’:
            - Hash_table = DoubleHashTable(199, hash_type)
        - ELSEIF table_type == ‘pseudo’:
            - Hash_table = PseudoHashTable(199, hash_type)
        - FOR(j=0; j<125;j++):
            - IF (seq_counter>= LEN(psudo_list_bag_fill)):
                - seq_counter = 0
            - k = psudo_list_bag_fill[seq_counter]
            - rcvdObj = inv[k]
            - modyObj = NEWCARDS(inv.itemName, inv.rand(strength), inv.rarity)
            - key_for_hash = modyObj.rarity + modyObj.itemName
            - hash_table.set(key_for_hash, modyObj)
            - seq_counter += 1
            - if(i==0):
                - tbag = append(hash_table)
            - tbag = STACK(hash_table)
            -
    - return tbag

Theoretical Complexities of Algorithms:

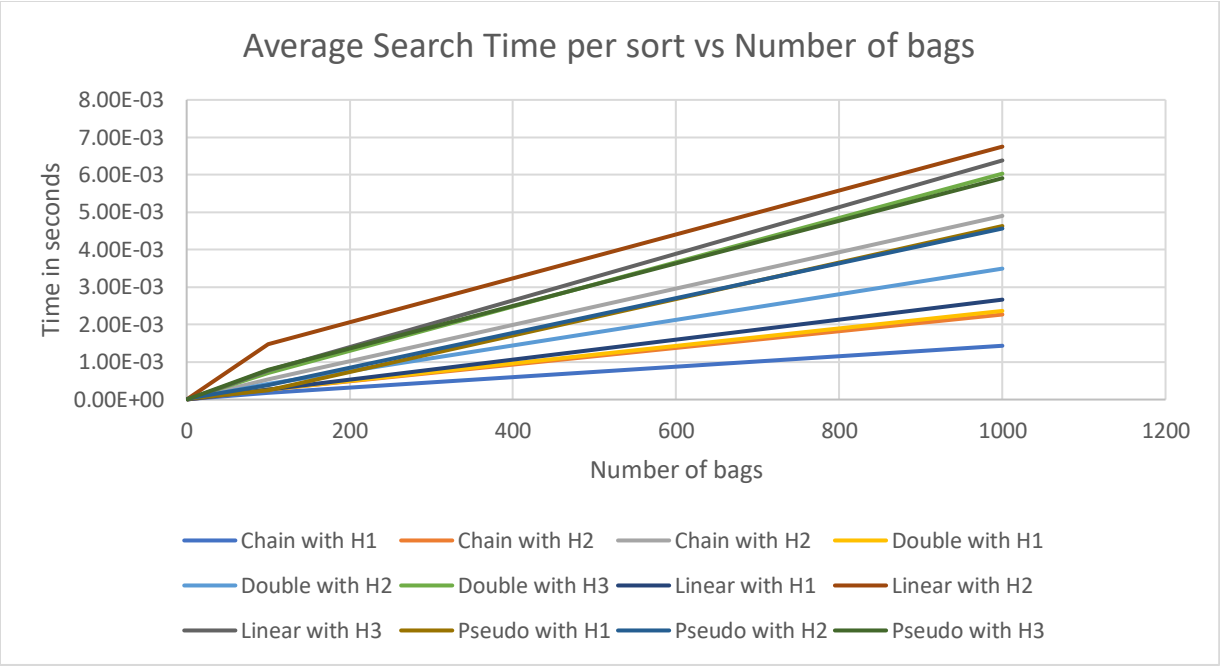
Algorithms	Time Complexity
Hashing to table	O(n)
Hash function 1	O(1)
Hash function 2	O(1)
Hash function 3	O(1)
Chain hashing	O(1)
Double hashing	O(1)
Linear Probe hashing	O(1)
Pseudo random hashing	O(1)

The implementation of Hash table is worked out in such a manner that all the tables should be in Order 1 should the data be present within the table. However, if the data is not in the table then the Hash table will take a linear time for finding the data. As it will look over all the elements within the table.

The Pseudo Random Hash Table implements works in the principle of forming a random 200 permutation value for the table and then randomly selecting on random list form within that list. Since it always generates 200 and selects 1 it should also take a constant time.

Tables of Observed Time Complexities:

Type of hash	Average Time(seconds)			
	1	10	100	1000
Chain with H1	3.87E-06	2.40E-05	0.000172	0.001436
Chain with H2	5.99E-06	3.07E-05	0.000258	0.00227
Chain with H2	9.00E-06	5.95E-05	0.000546	0.004903
Double with H1	6.94E-06	4.01E-05	0.000269	0.002367
Double with H2	8.46E-06	4.55E-05	0.000413	0.003495
Double with H3	9.35E-06	7.62E-05	0.000717	0.00603
Linear with H1	5.47E-06	2.98E-05	0.000258	0.002667
Linear with H2	1.55E-05	0.000147	0.001481	0.006752
Linear with H3	1.12E-05	6.79E-05	0.000784	0.006383
Pseudo with H1	7.32E-06	3.61E-05	0.000251	0.004632
Pseudo with H2	1.12E-05	5.21E-05	0.000382	0.004566
Pseudo with H3	1.01E-05	7.11E-05	0.000792	0.005909



Here we can observe that there is a linear trend for the searching the hashed item as we can predict that the item, we were looking for should not be on the list. This could have generated due to the randomly inserting and looking for an item from a large set of 750 data being mapped on a small 125 scale. Which also help us predict that the table should have a frequent value with same key.

Output:

Please look at the *preserved_ouput* folder.

The linux command '>' was used to create the preserved_output's .txt files