**Interface KWHashMap**

Table

Description automatically generated

**Class Entry**

Graphical user interface, text, application, email

Description automatically generated

There is no setter for the key since key is the unique feature for each key-value pair. When you change the key, you are changing the key-value pair completely. So you have to create a new Entry.

REMEMBER, in hash table each element is key-value pair <K, V>.

Text, letter

Description automatically generated

**Class HashTableOpen**

Table

Description automatically generated

LOAD\_THRESHOLD is better to be final.

START\_CAPACITY (capacity of the array) is better to be prime.

We need numDeletes because it contributes on load factor. To be able to check whether load factor is above the threshold or not, we have to consider number of keys and also number of deleted keys together.

You have to probe through the deleted items as the regular occupied slots. So running time diminishes if number of deleted items increases.

If load factor is above the threshold, we allocate more space and rehash everything to that larger table.

REMEMBER, load factor = (numKeys + numDeletes) / tableSize

Deleted items are not rehashed and table size increased. So load factor is increased by allocating more space.

Text

Description automatically generated

If quadratic probing is used, using 0.75 as LOAD\_THRESHOLD is not possible. Maximum LOAD\_THRESHOLD should be 0.5 for the quadratic probing. We are implementing linear probing.

“null” item in the table is empty slot.   
Entry with key-value pair “null-null” is deleted slot.

Graphical user interface, text, application

Description automatically generated

These 2 methods are private helper methods for KWHashMap interface methods (get, put, remove).

Text

Description automatically generated

Text

Description automatically generatedhashCode method of Object is calculated based on the address of that object not the value of the object. That’s why each class should override hashCode method.

We use “table[index].key” instead of “table[index].getKey()” in while condition because Entry is inner class.

A picture containing text, clock

Description automatically generatedThis is number of probes (number of times while loop executed). This is average case formula.

In the worst case, we have to probe n (numKeys + numDeletes) times. 🡪

In the best case, key is found in the first probe 🡪

In the average case, we use above formula.

For 0.75 load factor, number of probes is 2.5 on average. So average case is 🡪

So while loop is executed 2.5 times on average.

A screenshot of a computer

Description automatically generated with medium confidence

Text, letter

Description automatically generated



Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated



Running time is dependent on find method (linear for worst case, constant for best and average cases).

Graphical user interface, text, application, email

Description automatically generated



Load factor isn’t changed because we increment numDeletes and decrement numKeys. So we don’t need to rehash.

Running time is dependent on find method.

Text, letter

Description automatically generatedA screenshot of a computer

Description automatically generated



Capacity for the new table may not be prime but at least it is odd.

If all the elements are hashed to the same index, running time is quadratic.

If we implement the Map interface, iterator is also must be implemented.

In Java, we have HashMap class, using hashing for implementing the Map interface. Of course there is an iterator.

There is no specific order in the hash table. So you can go through all the elements one by one through end of the table by iterator. next or hasNext methods may not be constant time operations. Since there could be many empty slots that you have to pass before finding the next element in the hash table.

**Class HashTableChain**

Table

Description automatically generated

We don’t need numDeletes bc deleted items is deleted from the linked list completely.

Text, letter

Description automatically generated

Text

Description automatically generated

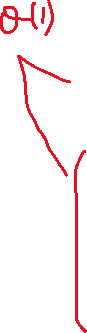
Tb(n) = {either table[index] is null or we found in the first element of the linked list}  
Tw(n) = {one index contains a linked list that contains all the elements inserted in table, other   
 indexes are empty}  
Tav(n) = {on average 2.5 elements are checked to find an element, for loop is performed < 2.5   
 times}

Text, letter

Description automatically generatedc = 1 + L/2 🡪 Average number of items in a list in chaining. If L is 3, c = 2.5. L must be less than 3 (3 is max load factor) in our case so c < 2.5.

Text

Description automatically generated



REMEMBER, key must be unique. So we can’t just directly add new entry to table. We should check whether there is an entry with the same key.

Even rehashing is linear time on average, how come we say lines 6-14 are constant?

Text, letter

Description automatically generatedIt is amortized constant time.

Text

Description automatically generated



Removing an element is linear time for the worst case. If we perform the search by using an iterator and iterator’s remove operation can be used in line 7 to make remove operation constant time.

**Testing the Hash Table Implementation**

Write a method to:

* create a file of key-value pairs
* read each key-value pair and insert it in the hash table
* you can perform some delete operations
* observe how the hash table is filled

Implementation:

* Write a toString method that captures the index of each non-null table element and the contents of the table element
* For open addressing, the contents consists of the string representation of the key-value pair
* For chaining, a list iterator can traverse at the table element and append each key-value pair to the resulting string

Cases to examine:

* Does the array index wrap around as it should?
* Are collisions resolved correctly?
* Are duplicate keys handled appropriately? Is the new value retrieved instead of the original value?
* Are deleted keys retained in the table but no longer accessible via a get?
* Does rehashing occur when the load factor reaches 0.75 (3.0 for chaining)?

Step through the get and put methods to:

* observe how the table is probed
* examine the search chain followed to access or retrieve a key

For performance test, insert randomly generated integers in the hash table to create a large table with O(n) effort:

Text

Description automatically generated

Insertion of randomly generated integers into a table allows testing of tables of very large sizes, but is less helpful for testing for collisions

You can add code to count the number of items probed each time an insertion is made (open addressing) -- these can be totaled and divided by the number of insertions to determine the average search chain length (chaining)

After all items are inserted, you can calculate the average length of each linked list and compare with the number predicted by the formula ( c = 1 + L/2 )

Table

Description automatically generated

You can use this table to check whether number of probes values are obtained or not.