1. use double linked-list as bucket to store different key index(after hash), each bucket node head is a double linked-list node, if different key generates same hash value, add new node to the next of original one. Each node has an element(contains key and value), next and prev node pointer to traverse on the same bucket index, after and before pointer to traverse on the bucket list. If duplicate keys, update former value with new value; If size reaches hashSize, the eviction strategy is evict the oldest key(the one least used).

using the deletion function many times in a row, completely emptying the structure
if try to delete non-existing key, print error given key is not existing;
if structure is already empty and keep deleting, similarly, since given key won't be matched, print error given key is not existing.

for the ascending order tree map implementation, it's recursive in nature, each node(treemap) contains a key, value and left & right pointer, the reason I use tree map data structure to implement ascending order is that I can't find a way to implement ascending order using linked-hashmap and treemap is commonly used to achieve ascending order. add node will add to suitable position based on ascending keys(if less put on left or put on right side) and delete will maintain the ascending order as well.

2/3. the first version is keeping the insertion order one.
For the linkedhashmap:

operate on the double linkedlist, therefore O(1)

```
// remove from bucket O(1)
void bucketRemove(dlist_t *bucket, node_t *cur)
```

operate on the double linkedlist, therefore O(1)

```
// remove from hashmap O(1)
void hashRemove(dLinkedHashmap_t *linked_hashmap, node_t *cur)
```

create & initiate the node

```
// create a node O(1)
node_t *createNode(int key, float value)
```

one pass to traverse the bucked linked list

```
// return bucket head with given key O(n), n is the number of dll within bucket
node_t *getNode(dlist_t *bucket, int key)
```

add node on ddl

```
// update bucket with new node when keeping insertion order, O(1)
void updateBucketNode(dLinkedHashmap_t *linked_hashmap, dlist_t *bucket, node_t
*cur)
```

need to traverse on the dll

```
// update bucket with new node for ascend order, O(n) n is the number of dll within bucket
void updateBucketNodeAsc(dLinkedHashmap_t *linked_hashmap, dlist_t *bucket, node_t
*cur)
```

update node on dll

```
// update header to track the newest added / modified node, O(1)
void updateHeader(dLinkedHashmap_t *linked_hashmap, node_t *cur)
```

remove node on dll

```
// remove a bucket with given key, O(n)
void removeKey(dLinkedHashmap_t *linked_hashmap, int key)
```

update node on dll

```
// add a node with given key and value; if no key exist before, create a new bucket index; if
// duplicate key, overwrite with new value; if reach hashSize, evict oldest key bucket. O(1)
void put(dLinkedHashmap_t *linked_hashmap, int key, float value)
```

add node on ddl

```
// add a node with given key and value for ascend order; if no key exist before, create a new
// bucket index; if duplicate key, overwrite with new value; if reach hashSize, evict oldest
key
// bucket. O(1)
void putAseOrder(dLinkedHashmap_t *linked_hashmap, int key, float value)
```

find node on dll

```
// return the value with given key, O(n)
float getValue(dLinkedHashmap_t *dLinkedHashmap, int key)
```

find node on dll

```
// Returns a boolean value if a key has been put into the hashmap, O(n)
//    - Returns a '1' if a key exists already
//    - Returns a '0' if the key does not exist
//    - Returns a '-9999' for invalid hashmaps
int hashmap_hasKey(dLinkedHashmap_t *linked_hashmap, int key)
```

```
// Return the first key-value pair in the ordering, O(1)
//    - Returns NULL if there is no pair
element_t *hashmap_getFirst(dLinkedHashmap_t *linked_hashmap)
```

```
// Return the last key-value pair in the ordering, O(1)
//    - Returns NULL if there is no pair
element_t *hashmap_getLast(dLinkedHashmap_t *linked_hashmap)
```

```
// Frees a hashmap, O(n) n is the number of node
void freeHashmap(dLinkedHashmap_t *linked_hashmap)
```

```
// Prints all of the keys and corresponding values of hashmap in order, O(n) n is the number
of
// nodes
void printHashmap(dLinkedHashmap_t *linked_hashmap)
```

```
// Prints all of the keys and corresponding values of hashmap in order, O(n) n is the number
of
// nodes
void hashmap_printKeys(dLinkedHashmap_t *linked_hashmap)
```

4. Different order treemap:

treemap is more suitable for keeping ascend order, I write another treemap data structure to support this operation,

worst case is skewed tree O(n); average case is height of tree O(logn)

```
// add a node into treemap, O(logn), n is the number of nodes within the treemap
treemap_t *add(treemap_t *treemap, int key, float value)
```

traverse all nodes

```
// Prints all of the keys and corresponding values of hashmap in ascending order, O(n) n is
the number of nodes
void printTreemap(treemap_t *treemap)
```

traverse all node and free each node recursively, O(n)

```
// free the treemap
void deleteTreemap(treemap_t *treemap)
```

```
// get left most node, worst case O(n) in skewed tree, average O(logn)
treemap_t *getLeftmost(treemap_t *treemap)
```

```
// delete node with given key, worst case O(n) in skewed tree, average O(logn)
treemap_t *deleteByKey(treemap_t *treemap, int key)
```

Comparision:
the new complexity is presented in above treemap part,

function	double linkedhashmap	tree map
add	O(n)	worst O(n), average O(logn)
deletebykey	O(n)	worst O(n), average O(logn)
print	O(n)	O(n)
free	O(n)	O(n)

add and deletebykey from treemap is faster than double linkedhashmap in average case, same in worst case since treemap data structure can divide data into two parts during traversal thus faster to get certain node in average case O(logn).

5. Optional: thread-safe linked-hashmap, when need to make changes on existing structure / creating new structures / update values, use a mutex to ensure mutual exclusive access to make chages, thus ensure thread-safety.