ANALYSIS

The time complexity of the method **public V get(K key)** of the HashTable class.

Sentence	Cost
int hash = hash(key);	O(1)
while (table.get(hash) != null && !table.get(hash).getKey().equals(key)) {	O(n)
hash = (hash + 1) % size;	O(n)
if (hash == hash(key)) {	O(n)
return null; } }	O(n)
if (table.get(hash) == null) {	O(1)
return null; }	O(1)
else { return table.get(hash).getValue(); }	O(1)

The algorithm's worst-case time complexity is O(n), where n is the total number of keyvalue pairs stored in the hash table. This is due to the while loop's potential need to visit every element of the hash table until it locates the key, which would result in a temporal complexity proportional to the number of key-value pairs contained in the hash table.

Spatial complexity of the method **public V get(K key)** of the HashTable class.

Sentence	Cost
int hash = hash(key);	O(1)
while (table.get(hash) != null && !table.get(hash).getKey().equals(key)) {	O(1)
hash = (hash + 1) % size;	O(1)
if (hash == hash(key)) {	O(1)
return null; } }	O(1)

if (table.get(hash) == null) {	O(1)
return null; }	O(1)
else { return table.get(hash).getValue(); }	O(1)

Since only the already-existing elements of the table are being accessed and no new elements are being added to the table, this method's space complexity is O(1) rather than the typical O(n) for a hash table with n elements.

Time complexity of the method **public T dequeue()** of the Queue class.

Sentence	**Cost**
if (isEmpty()) {	O(1)
throw new IndexOutOfBoundsException("Queue is empty"); }	O(1)
T data = queue[0];	O(1)
for (int i = 0; i < size - 1; i++) {	O(n)
queue[i] = queue[i + 1]; }	O(n)
size--;	O(1)
return data;	O(1)

Dado que sólo la línea del bucle for, que mueve los elementos de la cola hacia la izquierda, tiene un coste temporal que depende del tamaño de la cola, la complejidad temporal global del algoritmo es O(n).

Spatial complexity of the method **public T dequeue()** of the Queue class.

Sentence	**Cost**
if (isEmpty()) {	O(1)

throw new IndexOutOfBoundsException("Queue is empty"); }	O(1)
T data = queue[0];	O(1)
for (int i = 0; i < size - 1; i++) {	O(n)
queue[i] = queue[i + 1]; }	O(n)
size--;	O(1)
return data;	O(1)

The dequeue function algorithm uses an array of size "size" to hold the queue entries, therefore its space complexity is O(n). The value of queue[i + 1] is assigned to queue[i] at each iteration of the for loop, using O(1) space each time. However, because it repeats over elements of size – 1, the cycle's overall level of space complexity is O(n).

In order to temporarily store the element that will be removed from the queue, the operation to remove an element from the queue additionally needs O(1) of extra space. Additionally, the exception that is thrown when the queue is empty calls for a second exception object, raising the O(1) space complexity.

As a result, the algorithm's total space complexity is O(n) + O(1) + O(1) = O(n).