Temporal Complexity analisis:

Our first analysis will be for the public void put(K key, V value), followed by the next code:

|  |  |  |
| --- | --- | --- |
| # | Line | Times executed |
| 1 | if (key != null) { | 1 |
| 2 | int hash = getHash(key); | 1 |
| 3 | NodeHash<K, V> node = new NodeHash<>(key, value); | 1 |
| 4 | if (tabla[hash] == null) { | 1 |
| 5 | tabla[hash] = node; | 1 |
| 6 | } else {  NodeHash<K, V> current = tabla[hash]; | 1 |
| 7 | while (current.getNext() != null) { | n + 1 |
| 8 | current = (NodeHash<K, V>) current.getNext(); | n |
| 9 | current.setNext(node);  } | 1 |

Now we must do the summatory of each “Times executed” so we can define the Big O notation:

*T(n)= 1+1+1+1+1+1+1+n+(n+1)*

*T(n) = 8 + 2n*

And so, we can say that the algorithm time complexity is: Θ(n)

Second Algorithm:

Our second analysis will be for the removeElement(T element), followed by the next code:

|  |  |  |
| --- | --- | --- |
| # | Line | Times Executed |
| 1 | if (element != null) | 1 |
| 2 | return false;  } | 1 |
| 3 | Heap<T> tempHeap = new Heap<>(); | 1 |
| 4 | while (!isEmpty()) { | n + 1 |
| 5 | T currentItem = dequeue(); | n \*n |
| 6 | if (!currentItem.equals(element)) { | n |
| 7 | tempHeap.insert(currentItem); | n \* n |
| 8 | heap = tempHeap; | 1 |
| 9 | return !isEmpty(); | 1 |

Now we must do the summatory of each “Times executed” so we can define the Big O notation:

*T(n) = 1 + 1 + 1 +(n + 1) + ( n \* n ) + n + ( n \* n ) + 1 + 1*

*T(n ) = 2n2 + 2n + 6*

And so, we can say that the algorithm time complexity is: Θ(n2)