|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| T1 | |  | T2 | |
| 0 | 88 |  | 0 | 88 |
| 1 | 77 |  | 1 | 77 |
| 2 | 23 |  | 2 | 23 |
| 3 | 14 |  | 3 | 14 |
| 4 | 99 |  | 4 | 99 |
| 5 |  |  | 5 |  |
| 6 |  |  | 6 |  |
| 7 | 95 |  | 7 | 95 |
| 8 | 41 |  | 8 | 41 |
| 9 |  |  | 9 |  |
| 10 |  |  | 10 |  |

1.c – The T1 hash function is a linear probing, so the chance of a clustering happening is very high, meaning the insert of a new value to the list is not very efficient, the T2 function works more efficient but T2 hash function will give the same result as T1 on theirs two first calls, so the insert function T(2) is more efficient but only after two calls of the function (two collisions).

1.d – The load factor is represented by alpha = n/m, where n is the number of actual elements in the list and m is the number representing all the slots in the list.

The load factor of the table is: alpha = 7/11 = 0.636

2.a – In a linear probing the hash function will always increase by 1 when it get a collision, meaning the only possible options for a finding function in a linear hash is iterate through all the list and check slot by slot if the element is there, if the slot given by a hash function is empty (and without a marker of a remove function), it means the element is not in the list.

So the hash function h(x) to find the element x ran 10 times and h(y) to find y ran 4 times. It’s possible that the last 4 times that h(x) ran were on the same slots that’s h(y) ran, and both found an empty slot after it, meaning they’re not on the list.

That for, we have: h0(y) = h6(x), h1(y) = h7(x), h2(y) = h8(x) and h3(y) = h9(x).

2.b – It’s impossible, because in a linear hashing when you get a collision, the next hash function will always return the value of the last hash function plus one, meaning it might run from h0(x) until hn(x), for h0 being the first value of the hash function for an element x and hn being the last value of the hash function in a list of length n.

3 –

For logn we can take n = 2^x

So for the whole function we get (2^x)^3 \* 2^x = (2^3x) \* (2^x) = 2^4x => T(n), is θ(2^4x)

a - It’s necessarily false, since Ω(n^2) = Ω(2^2x) and the average case for the algorithm is θ(2^4x), meaning it’s not possible to get a best-case time complexity of Ω(n^2).

b – For O(n^4) = O(2^4x).

It’s necessarily true that the worst-case time complexity of the algorithm is O(n^4).

c – It may be true that the average-case time complexity of the algorithm is O(n^4).