What to do if the table is full?

We say that a hash table is **full** if the load factor is more than the maximal load factor, that is,

$$\frac{n}{T} > \lambda.$$

Rehashing (idea): If the table becomes full after an insertion, allocate a new table twice the size and `insert` all elements from the old table into it.

Consequences for `insert`:

- the Worst Case time complexity is $O(n)$ (when rehashing) but
- the *amortized* time complexity is $O(1)$!
    - calculation is similar to that for dynamic arrays

(Rehashing can be used for direct chaining, linear probing, or double hashing and always leads to constant amortized time complexities.)

This combines well with our extra assumption that $T = 2^k$ in order to avoid short cycles. If we start from an empty hash table of such size (for example, we initially have $T = 2^3 = 8$), then doubling the size always ensures that $T = 2^k$ for some (natural) number k.

Remark: If we double the size of the hash table, we also need to change the (primary) hash function to make sure that it is *good* again. In practice, `hash(key)` is usually computed as `bigHash(key) mod T` (where `bigHash` computes a "big" hashcode).

Then, after doubling the size of our hash table we only modify `hash(key)` as follows

`bigHash(key) mod 2*T`.

Summary

Hash tables are ADTs with an implementation consisting of an array `arr`, a primary hash function `hash1(key)` (and possibly a secondary hash function `hash2(key)`)

All operations are in $O(1)$ (amortized time) if

1. `hash1` (and `hash2`) computes indexes uniformly,
2. we `rehash` whenever the table becomes full,
3. ($T = 2^k$ for some k, and `hash2` gives odd numbers).

They do not offer an efficient way to obtain entries in key order

Comparison with trees

AVL Trees require keys to be *comparable* and the operations are in $O(\log n)$, best, worst and average case.

Hash tables, on the other hand, require *good hash functions*.
Then, operations are in $O(1)$ *amortized* time complexity.

Final thoughts

- Insert, delete and search in Direct Chaining, Linear Probing or Double Hashing all have $O(1)$ amortised complexity.
- Double hashing has an performance advantage because `allocate_memory` in chaining has a large constant cost and clustering in linear probing is worse.
- In chaining, if the load factor drops below a minimum threshold, we can rehash into a hash table half the size. This is rarely done because it does not speed up performance.
- In open addressing hash tables, We keep track of the number of tombstones in the table. If this exceeds some threshold, we also rehash but without doubling the size. With many tombstones, we might even halve the size of the hash table.
- As a consequence `delete` is also $O(1)$ *amortized* time complexity.