Exercise Sheet, Week 10

Linear probing is a tucked-in method for dealing with collisions. Each position in the table is marked as either *empty*, *tombstone* **#** or *occupied*. The basic operations then work as follows:

- `lookup(key)` : Starting from the primary position `hash(key)` , we search for the `key` to the right (and loop around if necessary). We skip skip over all tombstones **#** and positions occupied by different keys. We stop if we get to a position marked as *empty*.
- `insert(key):` First, we check if the `key` is stored in the hash table (as above). If it isn't, we store it on the first tombstone position we came across while searching. If there was no tombstone we store the `key` on the *empty* position where we stopped searching.
- `delete(key):` We check if `key` is stored in the hash table. If it is, we replace it by a tombstone.

Double hashing works similarly to linear probing. The only difference is how we calculate the fallback positions. Instead of searching on positions

`hash(key)` , `hash(key) + 1 mod T` , `hash(key) + 2 mod T` , ...

double hashing is searching on positions

`hash1(key)` , `hash1(key) + 1*hash2(key) mod T` , `hash1(key) + 2*hash2(key) mod T` , ...

where `hash1` and `hash2` are two different hash functions.

In the following exercises we will always consider the hash functions `hash` (resp. `hash1`) and `hash2` as given by the following table:

`key`	A	B	C	D	E	F	G
`hash` / `hash1`	3	0	1	1	4	7	2
`hash2`	3	6	3	1	6	1	4

Also, we will start with the hash table that looks as follows:

0	1	2	3	4	5	6	7
	D	G		**#**		**#**	F

Question 1. Use hash tables with (1) *probing sequences*, resp. (2) *double hashing*, and execute the following sequence of commands:

1. `insert(C)`
2. `insert(A)`
3. `insert(E)`
4. `delete(D)`
5. `delete(C)`
6. `insert(C)`

Question 2. Write full pseudocode for `lookup(key)` for hash tables with (1) *probing sequences*, resp. (2) *double hashing*. To mark one of the three states empty/tombstone/occupied we store values 0/1/2, respectively, in an extra array called `state` .

For example, the hash table showed above can be represented as follows

`i`	0	1	2	3	4	5	6	7
`htable[i]`	A	D	G	B	C	D	D	F
`state[i]`	0	2	2	0	1	0	1	2