A *hash table* is a data structure for efficiently maintaining a set of elements. For example, the Python data structures `set` and `dict` are implemented using a hash table.

A hash table consists of N locations indexed $0, 1, \ldots, N-1$. Each element has a specific location in the hash table based on its hash value.

A *hash function* determines the location of an element in the hash table. The function takes an element x as a parameter and returns an integer $h(x)$ as the *hash value* of the element x. The location of x in the hash table is $h(x) \bmod N$, i.e., the remainder from dividing $h(x)$ by N.

When the element is added to the hash table, it will be inserted at the location $h(x) \bmod N$. Similarly, when checking if the hash table contains the element x, it will be searched at the location $h(x) \bmod N$.

Example

Consider an example, where $N = 10$ and $h(x) = 7x$. We will store the elements of the set $\{2, 4, 5, 9, 18, 30\}$ into the hash table. We obtain the following locations for the elements:

Element	Location
2	$7 \cdot 2 \bmod 10 = 4$
4	$7 \cdot 4 \bmod 10 = 8$
5	$7 \cdot 5 \bmod 10 = 5$
9	$7 \cdot 9 \bmod 10 = 3$
18	$7 \cdot 18 \bmod 10 = 6$
30	$7 \cdot 30 \bmod 10 = 0$

Thus the hash table as follows after the additions:

Location	0	1	2	3	4	5	6	7	8	9
Elements	30	-	-	9	2	5	18	-	4	-

When we want to check if the hash table contains an element x, we search for it at the location $7x \bmod 10$. For example, we will search for the element 4 at the location $7 \cdot 4 \bmod 10 = 8$.

Collisions

A *collision* is a situation, where two elements have the same location in the hash table. An implementation of a hash table has to be prepared for collisions, because the size N of the hash table is typically much smaller than the number of possible element values.

For example, if we add the elements 4 and 14 into our example hash table, both will be assigned to the location 8, because $7 \cdot 4 \bmod 10 = 7 \cdot 14 \bmod 10 = 8$.

There are two commonly used techniques for handling collisions:

- *Chaining*: Each hash table location contains a list that stores all the elements assigned to that location.
- *Open addressing*: Each location contains at most one element. The location of an element x is determined by using a hash function $p(x, i)$ with a second parameter i so that we can try different choices $i = 0, 1, 2, \dots$ until an unoccupied location is found.

Efficiency

A hash table operates efficiently if the elements are fairly evenly distributed over the whole hash table. The distribution is determined by the hash table size and the choice of the hash function.

Chaining is efficient if each location contains only a small number of elements. Then the lists are short and the list operations are fast.

Open addressing is efficient if a small number of tries is sufficient for finding a location of an element, i.e., only a few calls to the function $p(x, i)$ is needed.

However, in the worst case, a hash table can become slow and the operations can take $O(n)$ time. For example, this happens with chaining if every element is assigned into the same location.

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI