Operations on data structures based on hashing are *usually* efficient, but it is possible for the operations to be slow due to collisions. We will next see how the Python dictionary becomes slow, when the keys stored there are chosen in a specific way.

In order to choose the keys, we need to know the exact implementation of the dictionary. The relevant aspect of the implementation are the hash function, the hash table size and the collision handling.

We can find these details in the Pythonin source code.

Hash function

The Python hash function `hash` is implemented differently for different data types. For small enough integers, the hash value is simply the integer itself:

```
> hash(42)
42
> hash(123)
123
> hash(1337)
1337
```

We will use small integers as keys, so that we can assume the above hash function.

Hash table size

The hash table size N of the Python dictionary is initially 8. The hash table is implemented so that at most $2/3$ of the locations are occupied. If this limit is exceeded, the size of the hash table is doubled.

The following code illustrates this:

```
import sys

n = 1000

numbers = {}
old_size = 0

for i in range(n):
    new_size = sys.getsizeof(numbers)

    if new_size != old_size:
        print(len(numbers), sys.getsizeof(numbers))
        old_size = new_size

    numbers[i] = True
```

The code creates a dictionary `numbers` and adds elements to it one at a time. When the memory usage of the hash table changes, the code prints out the number of elements and the memory usage of the hash table. The output of the code is as follows:

```
0 64
1 232
6 360
11 640
22 1176
43 2272
86 4696
171 9312
342 18520
683 36960
```

For example, the size of the hash table increases when the element count reaches 6, and then the hash table size increases from 8 to 16. The size increases, because the ratio $6/8$ exceeds the limit $2/3$. The size increases again when the element count reaches 11, because $11/16$ exceeds $2/3$, and then the new size is 32.

Handling collisions

The following code shows how the Python dictionary chooses the location in the hash table for the pair `(key, value)`:

```
index = hash(key) % N
perturb = hash(key)
while True:
    if not table[index]:
        table[index] = (key, value)
        break
    perturb = perturb >> 5
    index = (5 * index + 1 + perturb) % N
```

Notice that this is not the actual code implementing the dictionary (Python is implemented in the C language), but this code shows the idea.

The variable `index` is the location in the hash table and its initial value is computed with the formula `hash(key) % N`. If the location `index` is occupied, the next location to try is computed with the formula `(5 * index + 1 + perturb) % N`. This continues until an unoccupied location is found.

The purpose of the variable `perturb` is to make collisions less likely. It affects the location formula for the first few tries. The initial value of `perturb` is the hash value of the key, and in each round its new value is computed as `perturb >> 5`. This bit shift operation corresponds to dividing by 32 and rounding down.

Constructing the input

Now we know enough about the Python dictionary implementation, and we can design an input that makes the dictionary operations slow. The idea is to use keys that collide with each other and cause the hash table operations slow down. The following code constructs such an input:

```
def find(table, key):
    N = len(table)
    index = hash(key) % N
    perturb = hash(key)
    count = 0
    while True:
        count += 1
        if not table[index]:
            return index, count
        perturb = perturb >> 5
        index = (5 * index + 1 + perturb) % N
```

```
n = 100000
chain_len = 50000
threshold = 5000

N = 2**18
table = [None] * N
keys = []

key = 1
for i in range(chain_len):
    keys.append(key)
    key = (5 * key + 1) % N

key = chain_len
while len(keys) < n:
    index, count = find(table, key)
    if count > threshold:
        table[index] = True
        keys.append(key)
    key += 1
```

The code creates an input of `n` keys. The hash table size `N` is chosen accordingly so that the limit of $2/3$ is not exceeded. The list `keys` stores the chosen keys.

First, the code adds keys that form a chain of length `chain_len`. The first key in the chain is `1` and the other keys in the chain are computed iteratively with the formula `key = (5 * key + 1) % N`. The chain is designed so that once the sequence of locations tried for a given key gets locked on the chain, it will stay locked on the chain until the end of the chain. Then the code adds the remaining keys so that each key added causes at least `threshold` steps along the chain before an unoccupied location is found. This makes their addition to the dictionary slow.

Testing efficiency

The following code measures how much time it takes to add the keys in the list `keys` to the dictionary:

```
import time

...
```

```
start_time = time.time()
numbers = {}
for key in keys:
    numbers[key] = True
end_time = time.time()

print(round(end_time - start_time, 2), "s")
```

Let us first do a test where the list `keys` contains 10^5 random numbers in the range $1 \ldots 10^8$. The list can be formed as follows:

```
import random

keys = random.sample(range(1, 10**8 + 1), 10**5)
```

This test took 0.02 seconds on the test computer. This is what one would expect when the dictionary operates efficiently.

Let us then do the test where the list `keys` is created with the code described above designed to cause a lot of collisions. Now the test time is 10.57 seconds, which is indeed very slow in comparison due to the collisions.

Can this happen in practice?

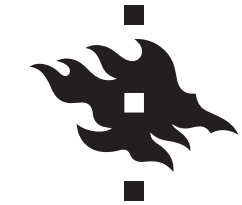
Although it is possible to construct an input that makes the dictionary slow, in practice such inputs occur very rarely. In essence, this happens only when the input was specifically designed to do so.

However, the existence of such designed inputs can be a problem in web programming, because a malicious attacker could try feeding such data to the site, causing the site to slow down. In python this possibility has been addressed by changing the operation of the function `hash` for strings each time Python is started:

```
$ python3
> hash("apina")
-8847049641918498300
> exit()
$ python3
> hash("apina")
5108947336973792736
```

```
> exit()
$ python3
> hash("apina")
-7637574785741815573
> exit()
```

This makes it impossible to construct a list of string keys that causes the dictionary to slow down.



HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI