Hashing

Databases and keys. A *database* is a collection of *records* with various attributes. It is commonly represented as a *table*, where the *rows* are the records, and the *columns* are the attributes:

Number	Name	Dept	Alias
20090612	오재훈	산디과	Pichu
20100202	강상익	무학	Cleffa
20100311	손호진	무학	Bulbasaur

Often one of the attributes is designated as the *key*. Keys must be unique—there can be only one row with a given key. A table with keys is called a *keyed table*.

A database index allows us to quickly find a record with a given key. In data structure terms, this is just a map from the key type to the record type.

Basic hashing with chaining. Let's implement a database of all the students in the class. We start with a `Student` class: Each record will be an object of this class.

```
class Student():
  def __init__(self, id, name, dept, alias):
```

We will use the `id` field as the key, and want to build an index so that we can quickly find the `Student` object with a given key.

A simple idea is to make an array with 100 slots, called a *hash table*. We use the last two digits of the student id as the index into this array. This is the *hash function* of the student id. Unfortunately, this doesn't quite work, because there are students with identical last two digits, like these:

20100874	정민수	무학	Pikachu
20080174	방태수	산디과	Mew

So what we do instead is to store in each slot of the array a *linked list* of (key, record) - pairs, as in Fig. 1.

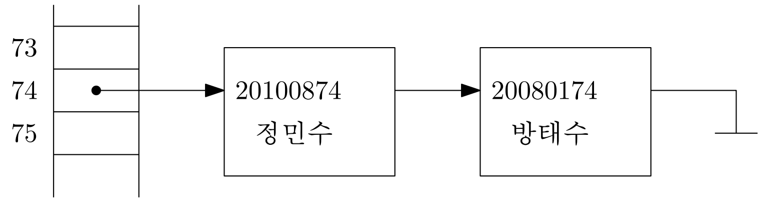

Figure 1: Chaining

To search for a key k in this data structure, we take the hash function of k (that is, the last two digits of k) to obtain the head of a linked list from the array. We then search the entire linked list by comparing k with each key in this linked list.

Implementation. The implementation uses a private `_Node` class, and a function to compute the hash code:

```
class _Node():
  def __init__(self, key, value, next):
    self.key = key
    self.value = value
    self.next = next

def _hash(key):
  return (key) % 100
```

We can then implement a `dict` using a list with 100 slots:

```
class dict():
  def __init__(self):
    self._data = [ None ] * 100
```

We find a record using the linked list:

```
  def _findnode(self, key):
    i = _hash(key)
    p = self._data[i]
    while p is not None:
      if p.key == key:
        return p
      p = p.next
    return None

  def __contains__(self, key):
    return self._findnode(key) is not None

  def __getitem__(self, key):
    p = self._findnode(key)
    if p:
      return p.value
    else:
      raise ValueError(key)
```

To insert a (key, record) pair, we first check if the key already exists. If not, we make a new node and add it at the beginning of the linked list:

```
  def __setitem__(self, key, value):
    p = self._findnode(key)
    if p:
      p.value = value
    else:
      h = _hash(key)
      self._data[h] = _Node(key, value, self._data[h])
```

Analysis of hash tables. How can we analyze this data structure? Clearly, in the worst case all students could have the same last two digits, and then the running time of insert and search operations would be $O(n)$, where n is the number of students.

So to analyze hashing, we need to make the assumption that the hash function (in our case, mapping the keys to their last two digits) is good: it should distribute the keys onto the slots of the array uniformly.

Formally, we will pretend as if the hash function was *random*: We will analyze the *expected running time* of insertions and search operations, where the expectancy is with respect to the random "choice" made by the hash function. In reality, of course, the hash function cannot be random: Given the same key twice, it must return the same hash value each time. Nevertheless, it turns out that with a good hash function, this analysis reflects the behavior of hash tables well.

So this is our setting: Given a key x, the hash value $h(x)$ is a random element of $\{0, 1, \dots, N-1\}$, where N is the size of the hash table. The random choices must be *independent*, so that given two keys x and y, the probability that $h(x) = h(y)$ is exactly $1/N$. (Further down, we will even need independence for the entire set of keys hashed into the table.)

Analysis of chaining. We assume that a set Y of n items has already been hashed into a hash table of size N. We now consider the insertion of a new key x. The running time of this insertion is proportional to the length of the linked list in slot $h(x)$. So what is the expected length of this list, with respect to the random choices made by the hash function?

Every element $y \in Y$ contributes one to the length of this list if $h(y) = h(x)$. Since we assume that hash values are independent, this happens with probability $1/N$. By additivity of expectation, this means that the expected number of items $y \in Y$ with $h(y) = h(x)$ is n/N, and so the expected length of the linked list in slot $h(x)$ is n/N.

Let's define the *load factor* λ of a hash table to be the ratio $\lambda = n/N$. By the above, the expected running time of insertions is $O(\lambda)$, and the same analysis applies to search operations and deletions.

Closed and open addressing. Hashing with chaining works well, and is often implemented in practice. However, the linked lists are not very memory-efficient, as we need space for node objects. We could make the data structure much more compact if we could store *all* the data inside the hash table. *Open addressing* means that we allow an item to be stored at a slot that is different from the "correct" slot indicated by its hash function. On the contrary, *closed addressing* means that items have to be stored at the slot given by their hash function, that is by chaining.

Linear probing. The most commonly used form of open addressing is *linear probing.* It also seems to be the best method to be used in practice, as long as the load factor remains low enough.

In linear probing, the first choice for a key x is the slot $h(x)$. If this slot is already in use, we try the slot $h(x)+1$. If this is also already in use, we continue with $h(x)+2$, and so on. For an insertion, we continue until we find an empty slot, and insert the item there. For a search operation, we continue until we either find the item, or find an empty slot. In the latter case, we know that the key is not in the hash table.

Figure 2 shows the result of a sequence of insertions into a hash table of size $N = 10$. The hash function of key x is its last digit, that is $h(x) = x \bmod 10$. Keys 89 and 18 are inserted in their "correct" slot. Key 49

	insert 89	insert 18	insert 49	insert 58	insert 9	insert 30
0			49	49	49	49
1				58	58	58
2					9	9
3						30
4						
5						
6						
7						
8		18	18	18	18	18
9	89	89	89	89	89	89

Figure 2: Insertions with linear probing.

should go in slot 9, but that is already occupied, so it goes into slot 0. Key 58 needs to try 3 slots before finding the empty slot 1, and the same is true for keys 9 and 30.

Consider the search for key 61: We first try slot $h(61) = 1$, but $58 \neq 61$. We then try slot 2, but $9 \neq 61$. We continue to slot 3, but $30 \neq 61$. We then check slot 4, which is empty, and so we can conclude that key 61 is not in the hash table.

Implementation. The implementation uses a list of 100 slots, each of which can contain an `_Entry` object:

```
class _Entry():
  def __init__(self, key, value):
    self.key = key
    self.value = value
```

Searching for entries uses the private method `_find_key`. It either returns `True` and the index of the slot containing the key, or returns `False`, and the index where the `key` can be added:

```
  def _findkey(self, key):
    i = _hash(key)
    while self._data[i] is not None:
      if self._data[i].key == key:
        return (True, i)
      i = (i + 1) % 100
    return (False, i)
```

The remaining functions are then easy to implement using `_findkey`:

```
  def __contains__(self, key):
    found, i = self._findkey(key)
    return found

  def __getitem__(self, key):
    found, i = self._findkey(key)
    if found:
      return self._data[i].value
    else:
      raise ValueError(key)

  def __setitem__(self, key, value):
    found, i = self._findkey(key)
    if found:
      self._data[i].value = value
    else:
      self._data[i] = _Entry(key, value)
```

Deletions in linear probing. Deleting items from a hash table with linear probing is not as easy as for chaining. Consider the table of Figure 2, and assume we simply delete key 58. The result is shown in Figure 3(a). However, this doesn't work: Image if we now search for key 9. The search would terminate at slot 1 and report that 9 is not in the hash table!

There are two solutions to this problem: The easiest and common solution is shown in Figure 3(b): we mark slot 1 as "available". This means that we consider the slot as empty during insertions (and so a new item may be placed here), but not during search operations (and so we can still correctly find keys 9 and 30).

With a bit more thinking, we can find a solution that doesn't need a special marker. Whenever we delete an item from a slot h, we create a *hole*. We then continue to scan the hash table from slot $h+1$. If we find an item y in slot k, we move it to the hole at h if h lies in between $h(y)$ and k. This move will create a new hole at slot k, and so we continue scanning at slot $k+1$ to fill the hole at slot k. The process continues until we find an empty slot. Figure 3(c) shows the result of deleting 58 from the hash table of Figure 2. Figure 4 shows a more interesting example: Key 89 is deleted and creates a hole in slot 9. This hole cannot be filled by key 30, but is filled with key 68. This creates a new hole in slot 1, which is filled by key 91. This creates a new hole in slot 2. Note that this hole is never filled, as the scan ends in slot 5 without finding an item that can be moved.

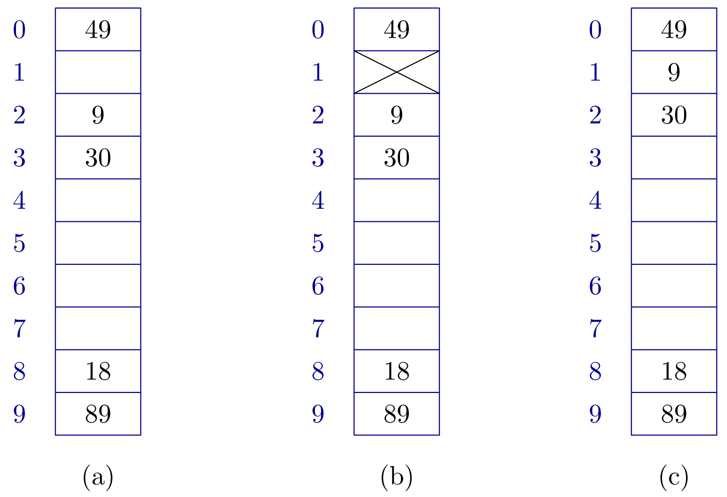

Figure 3: Deletions in linear probing: (a) this doesn't work, as we can no longer find 9 and 30. (b) mark deleted slot as "available". (c) moving items to fill the holes.

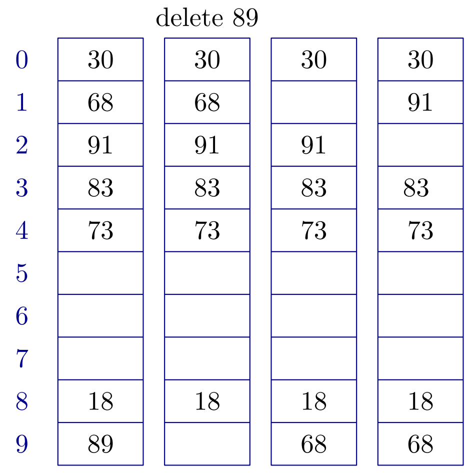

Figure 4: Deletions in linear probing by filling the hole.

Simplified analysis of linear probing. We assume that a set Y of n items has already been hashed into a hash table of size N using linear probing. We now consider the insertion of a new key x. The running time of this insertion is proportional to the number of *probes* we need to make to find an empty slot. Here, a probe means that we check a slot to see if it is empty.

Since there are n items in a hash table of size N, "on average" a slot contains an item with probability $n/N = \lambda$. This means that a slot is free with probability $1 - \lambda$. So we are repeating the following experiment: We flip a coin. With probability λ, it comes up "occupied", with probability $1 - \lambda$, it comes up "free". The expected number of times we have to repeat this until we see the outcome "free" is $1/(1 - \lambda)$.

In other words, when $\lambda = 1/2$, we expect to check only two slots. When $\lambda = 3/4$, we already need to check four slots, and for $\lambda = 0.9$, it would be ten slots. This already shows that linear probing can only work well when the load factor remains significantly smaller than one.

If we use a special marker to implement deletions as in Figure 3(b), then the marked slots do not count as free in the computation of the load factor. It is therefore possible that a hash table is nearly empty, but still has a high load factor! Such a hash table should be cleaned up by *rehashing* all the items.

Real behavior. Unfortunately, the argument above is not correct, because the probabilities for each slot to be filled are not independent. Let's call a *run* a maximal sequence of consecutive filled slots (that is, a sequence of slots $h, h+1, \dots, h+k-1$ (modulo N) that are all filled, while $h-1$ and $h+k$ are empty). Consider a run of length k. For every item that is hashed into the run, the run will grow by one element. This happens with probability $(k+2)/N$, and so the probability grows with the length of the run. This effect causes *clusters* to appear in the hash table.

We can implement an experiment to verify this: for a given table size N and load factor λ, we first fill a table A of size N by independently filling each slot with probability λ. We then fill a second table B of size N by inserting $\lambda * N$ randomly chosen items using linear probing. Finally, we evaluate the expected search cost in both tables as follows: For each slot h, we count the number of probes needed to find an empty slot.

The following shows the result for $N = 10000$, taking the average over 1000 experiments to get more accurate results:

λ	Table A	Table B
0.5	2.0	2.5
0.7	3.3	6.0
0.9	10.0	49.5
0.95	20.0	182.1
0.99	100.0	1750.5

It turns out that the simplified argument above is far too optimistic. When the load factor increases, the expected search time grows much faster than $1/(1-\lambda)$.

Complexity of linear probing. It can be shown that the expected number of probes needed for the insertion of a new item (or equivalently, for an unsuccessful search) is:

$$\frac{1}{2}\Big(1 + \frac{1}{(1-\lambda)^2}\Big)$$

So the running time grows with the *square* of $1/(1-\lambda)$, which explains why it is important to keep the load factor small. It should not exceed 0.5 or maybe 0.75.

Analysis of linear probing. We cannot prove the precise formula above here, but we want to at least show that the expected time for insertions and search operations is bounded by a constant independent of n, for the case $\lambda = 1/2$.

So we assume that we have a set Y of n items that have already been inserted into an array of length $N = 2n$ using linear probing. We consider the insertion of a new item with key x.

We start at slot $h = h(x)$, and visit slots $h, h+1, h+2, \dots, h+t$ (modulo N), where slot $h+t$ is the first free slot visited. The item will be inserted in slot $h+t$.

The number of probes needed for this insertion is $t+1$. We observe that the number t depends only on the hash function $h = h(x)$ of the key x, so let's denote it as $t = t(h)$.

We assume that the hash function $h(x)$ is a random element of $\{0, 1, \dots, N-1\}$, so to find the expected number E of probes for the insertion, we need to compute the average

$$E = \frac{1}{N}\sum_{h=0}^{N-1} t(h).$$

We note next that $t(h) = 0$ if slot h is an empty slot. The sum is therefore determined only by the occupied slots. A run of length k in the hash table contributes at most k^2 to the sum. (The precise contribution is $k(k+1)/2$.)

We can therefore bound the sum above as

$$E = \frac{1}{N}\sum_{h=0}^{N-1} t(h) \leqslant \frac{1}{N}\sum_{h=0}^{N-1}\sum_{k=1}^{n} P(h,k)\cdot k^2,$$

where $P(h,k)$ is the probability that the slots $h, h+1, h+2, \dots, h+k-1$ form a run of length k. (This probability is with respect to the insertions of the n items in Y into the hash table.)

When do these k slots form a run? A necessary condition is that exactly k of the n items have a hash index in the range $\{h, h+1, \dots, h+k-1\}$, while the remaining $n-k$ items have a hash index *not* in this range.

For a fixed item y, the probability that it is in the range is k/N, and the probability that it is not in the range is $1 - k/N$. It follows that

$$P(h,k) \leqslant P_k = \binom{n}{k}\big(\frac{k}{2n}\big)^k\big(1-\frac{k}{2n}\big)^{n-k}$$

Since P_k is independent of h, we can bound

$$E \leqslant \frac{1}{2n}\sum_{h=0}^{2n-1}\sum_{k=1}^{n} P_k k^2 = \sum_{k=1}^{n} P_k k^2.$$

We now need two inequalities:[1]

$$\binom{n}{k} \leqslant \frac{n^n}{k^k(n-k)^{n-k}}, \tag{1}$$

$$\big(1+\frac{x}{n}\big)^n \leqslant e^x \qquad \text{for } x \geqslant 0. \tag{2}$$

It follows that

$$\begin{aligned}
P_k &\leqslant \frac{n^n}{k^k(n-k)^{n-k}}\Big(\frac{k}{2n}\Big)^k\Big(1-\frac{k}{2n}\Big)^{n-k} \\
&= \Big(\frac{n}{k}\Big)^k\Big(\frac{k}{2n}\Big)^k\Big(\frac{n}{n-k}\Big)^{n-k}\Big(1-\frac{k}{2n}\Big)^{n-k} \\
&= \frac{1}{2^k}\Big(\frac{n}{n-k}\,\frac{2n-k}{2n}\Big)^{n-k} = \frac{1}{2^k}\Big(\frac{2n-k}{2(n-k)}\Big)^{n-k} = \frac{1}{2^k}\Big(\frac{2(n-k)+k}{2(n-k)}\Big)^{n-k} \\
&= \frac{1}{2^k}\Big(1+\frac{k}{2(n-k)}\Big)^{n-k} = \frac{1}{2^k}\Big(1+\frac{k/2}{n-k}\Big)^{n-k} \leqslant \frac{1}{2^k}e^{k/2} = \Big(\frac{\sqrt{e}}{2}\Big)^k.
\end{aligned}$$

Since $\sqrt{e}/2 < 1$, the infinite series $\sum_{k=1}^{\infty}(\frac{\sqrt{e}}{2})^k k^2$ converges to some constant C, and we have $E \leqslant C$.

Comparison of linear probing and chaining. Linear probing works well when the load factor remains low, say $\lambda \leqslant 0.5$. It is, however, more sensitive to the quality of the hash function—chaining is more robust when the hash function is not so good.

Linear probing works particularly well when all data is allocated inside the array in C or C++. In that case, it requires no dynamic memory allocation at all, and consecutive probes are extremely fast as they exploit caching in the processor (a processor can access consecutive memory locations much faster than data spread out over the memory). Open addressing is therefore the method of choice for hashing in devices like routers etc.

Hash functions. When keys are integers, we only need to somehow map them to the range $\{0, \dots, N-1\}$. In many applications, however, keys are other objects, such as strings. In that case, we first have to map the string to an integer.

In this case, we speak about the *hash code* as a function h_1 from keys to integers, and the *compression function* as a function h_2 from integers to the index range of the hash table, that is $\{0, 1, \dots, N-1\}$. The *hash function* is then the function $x \mapsto h_2(h_1(x))$ from keys to indices.

[1]The first one follows by writing $n^n = (k + (n-k))^n = \sum_{i=0}^{n}\binom{n}{i}k^i(n-k)^{n-i}$, which implies $n^n \geqslant \binom{n}{k}k^k(n-k)^{n-k}$. The second is true because $(1+x/n)^n$ is an increasing series whose limit is e^x, as you should have learnt in Calculus I.

Hash codes and compression functions are a bit of a black art. The ideal hash code function should map keys to to a uniformly distributed random slot in $\{0, \ldots, N-1\}$. This ideal is tricky to obtain. In practice, it's easy to mess up and create far more collisions than necessary.

Rehashing. When the load factor has become too large, or when a hash table contains too many slots marked as "available" due to deletions, we need to *rehash* the table.

This means that we create a new table (typically of about twice the size of the previous one), and change the compression function to map to the index range of the new table. We consider all items in the current table one by one, and insert them into the new table.

You *cannot* just copy the linked lists of a hash table with chaining to the same slots in the new table, because the compression functions of the two tables will certainly be incompatible. You have to rehash each entry individually.

You can also shrink hash tables (for instance when $\lambda < 0.25$) to free memory, if you think the memory will benefit something else. (In practice, it's only sometimes worth the effort.)

Obviously, an operation that causes a hash table to resize itself takes more than constant time; nevertheless, the *average* over the long run is still constant time per operation.

Compression functions. Let's consider compression functions first. Suppose the keys are integers, and each integer's hash code is itself, so $h_1(x) = x$. The obvious compression function is

$$h_2(x) = x \bmod N.$$

But suppose we use this compression function, and $N = 10,000$. Suppose for some reason that our application only ever generates keys that are divisible by 4. A number divisible by 4 mod 10,000 is still a number divisible by 4, so three quarters of the slots are never used! We have at least four times as many collisions as necessary. (This is an important example, because in many programming languages, such as C and C++, memory addresses of objects, when converted to integers, are always divisible by 4 or even by 8.)

The same compression function is much better if N is a prime number. With N prime, even if the hash codes are always divisible by 4, numbers larger than N often hash to slots not divisible by 4, so all slots can be used.

For reasons we won't explain

$$h_2(x) = ((a \cdot x + b) \bmod p) \bmod N$$

is a better compression function. Here, a, b, and p are positive integers, p is a large prime, and $p \gg N$. Now, the number N doesn't need to be prime.

Use a known good compression function like the two above instead of inventing your own. Unfortunately, it's still possible to mess up by inventing a hash code that creates lots of conflicts even before the compression function is used.

Hash codes. In Python, the built-in function `hash` computes a hash code for many kinds of object. In particular, for basic types like `int`, `float`, and `str`, this method returns a useful hash code.

When you create your own objects, you may need to design a hash code specially for this object. Here is an example of a good hash code for strings:

```
def hash_code(s):
  h = 0
  for ch in s:
    h = (127 * h + ord(ch)) % 16908799
  return h
```

By multiplying the hash code by 127 before adding in each new character, we make sure that each character has a different effect on the final result. The `%` operator with a prime number tends to "mix up the bits" of the hash code. The prime number is chosen to be large, but not so large that `127 * hash + ch` will ever exceed the maximum possible value of a 32-bit integer.

The best way to understand good hash codes is to understand why bad hash codes are bad. Here are some examples of bad hash codes on words.

- Sum up the values of the characters. Unfortunately, for English words the sum will rarely exceed 500 or so, and most of the entries will be bunched up in a few hundred buckets. Moreover, anagrams like "pat," "tap," and "apt" will collide.
- Use the first three letters of a word, in a table with 26^3 buckets. Unfortunately, words beginning with "pre" are much more common than words beginning with "xzq", and the former will be bunched up in one long list. This does not approach our uniformly distributed ideal.
- Consider the "good" `hash_code()` function written out above. Suppose the prime modulus is 127 instead of 16908799. Then the return value is just the last character of the word, because `(127 * hash) % 127 = 0`. That's why 127 and 16908799 were chosen to have no common factors.

Why is the `hash_code()` function presented above good? Because we can find no obvious flaws, and it seems to work well in practice. (A black art indeed.)

Equality and hash codes. Hash tables only work correctly if equal objects have the same hash code. The following example shows what can go wrong. We define a simple `Point` class:

```
class Point():
  def __init__(self, x, y):
    self.x = x
    self.y = y
```

And now we want to store points in a set:

```
>>> s = set()
>>> s.add(Point(3, 5))
>>> s
{Point(3, 5)}
>>> Point(3, 5) in s
False
```

Even though we can see that `s` contains a `Point(3, 5)`, we cannot find it in the set. The reason becomes clear when we try the following:

```
>>> p = Point(3, 5)
>>> q = Point(3, 5)
>>> p == q
False
>>> hash(p)
-9223363271751199253
>>> hash(q)
8765103576559
```

Even though two points have the same coordinates, Python does not consider them equal, and they have different hash codes—so there is no way that the set could find the entry.

Let's try adding the `__eq__` method to the point class:

```
class Point():
  def __init__(self, x, y):
    self.x = x
    self.y = y

  def __eq__(self, rhs):
    return self.x == rhs.x and self.y == rhs.y
```

But now we get a different error:

```
>>> p = Point(3, 5)
>>> q = Point(3, 5)
>>> p == q
True
>>> s = set()
>>> s.add(Point(3, 5))
TypeError: unhashable type: 'Point'
```

Python can now determine that the two points are equal—but it tells us that `Point` objects cannot be used in a hash table. In fact, it's the `hash` function that no longer works:

```
>>> hash(p)
TypeError: unhashable type: 'Point'
```

The Python interpreter will *not* use its default implementation of the `hash` function for objects with an equality operator. Why not? Because the hash code of equal objects needs to be the same, and Python has no way to ensure this.

If we want to be able to put `Point` objects into a set, we also need to define a hash code, for instance like this:

```
class Point():
  def __init__(self, x, y):
    self.x = x
    self.y = y

  def __eq__(self, rhs):
    return self.x == rhs.x and self.y == rhs.y

  def __hash__(self):
    return hash((self.x, self.y))
```

And now the set can handle `Point` objects:

```
>>> s = set()
>>> s.add(Point(3, 5))
>>> s
{Point(3, 5)}
>>> Point(3, 5) in s
True
```

The lesson is: hash tables require that keys satisfy the following "contract":

> If `obj1 == obj2` then `hash(obj1) == hash(obj2)`.

Mutable keys. However, even with our nice `Point` class, things can still go wrong:

```
>>> p = Point(3, 5)
>>> s = set()
>>> s.add(p)
>>> s
{Point(3, 5)}
>>> p.y = 9
>>> s
{Point(3, 9)}
>>> Point(3, 9) in s
False
>>> Point(3, 6) in s
False
```

Even though `s` clearly contains `Point(3,9)`, the set cannot find it. The reason is that `p`'s hash code has changed after it was added to the hash table, so `p` is simply in the wrong slot of the hash table!

The lesson here: *Never modify keys* after they were added to a hash table.

In fact, I would go further and recommend: Never use mutable objects as keys in a hash table. This is yet another example why immutable objects make programming safer and easier.

Python encourages this idea: Python lists and Python sets are themselves not hashable. You cannot put a Python list, or a Python set into a set! What you can do instead is to use a tuple or a `frozenset`. These objects are hashable, and can be used as keys in a map or as elements of a set.

Acknowledgments. Once again I made use of Jonathan Shewchuk's lecture notes. I learnt the analysis of linear probing presented here from David Eppstein, and proofs of Inequality (1) from Antoine Vigneron, Jingil Choi, and Sariel Har-Peled.