Robin Hood Hashing

Robin Hood is a legendary figure in English history. Ballads and stories about him have been written and told and sung since the middle ages. At least 10 movies about him have been produced. Robin Hood was a bandit, skilled in archery and sword fighting. By many accounts, he lived in Sherwood Forest with his band of merry men, and he fought against the Sheriff of Notingham. But he was a *good* guy, robbing the rich in order to give to the poor.

And that is how Robin Hood hashing got its name. We will see how probing in Robin Hood hashing robs some hash table values to give to poorer ones.

Probe sequence length (psl)

a, e, f hash to 0. b, c to 1. d to 2

	0	1	2	3	4	5	...
b	a	b	c	d	e	f	

Although the expected time to search a hash table using linear probing is in O(1), the length of the sequence of probes needed to find a value can vary greatly. Suppose values `a`..`f` hash as described to the right. Then, inserting values in alphabetical order, `a`, `b`, `c`, `d`, `e`, `f`, produces the hash table shown to the right.

The length of the probe sequence to find `a` and `b` is 0[1]; they are found just by looking in the bucket to which they hash. We call that the *psl* of `a` and `b`. The psl of `c` and `d` is 1. For example, to find `c` requires looking in buckets 1 and 2. But poor value `f` —its psl is 5!

	0	1	2	3	4	5	...
b	a	e	f	b	c	d	

The expected time to search may be constant, but the worst-case time is in O(size of the set). We say that the *variance* of the distances of values from their hashed buckets can be large. When inserting values, we want to ensure that the largest distance of values from their hashed buckets is as small as possible. This means putting all the values that hash to 0 first, then those that hash to 1, then those that hash to 2, etc., as shown to the right. Now, `c` and `d` have the largest psl: only 3.

In order to achieve the minimal possible psl when inserting, we have to change the insertion procedure. It will rob the rich —values in the table with low psl— to give to the poor —value with high psl that have to be inserted— by replacing a rich value by a poor one and finding another place for the rich one.

Maintaining psl values

	0	1	2	3	4	5	...
b	a 0	e 1	f 2	b 2	c 3	d 3	

We augment the hash table to include psls as well as values, as shown to the right. We'll use the notation `b[p].v` and `b[p].psl` to refer to the value and psl in element `b[p]`. The table to the right tells us that `f` hashed to 0, and its probe sequence is (0, 1, 2), which has length 2. Also, `d` hashed to 2, and its probe sequence is (2, 3, 4, 5), which has length 3.

The following invariant will be maintained:

> INV: For any `i`, the values that hash to bucket `i` precede the values that hash to bucket `i+1` (this naturally includes wraparound)

The insertion procedure

To the right is the typical method using linear probing to insert a value `v` known not to be in the set, but augmented to maintain the psl of `v`. The loop continues until a null bucket is found and then stores `v` and `vpsl` in that bucket. The invariant of the loop can be stated as follows:

> INV: Starting with the bucket to which `v` hashed, `vpsl` successive buckets are not null, and `b[p]` is the next bucket that might be null. */

```
void insert(v) {
   p= (hash of v) % b.length;
   vpsl= 0;
   while (b[p] != null) {
      // TODO

      p= (p+1) % b.length;
      vpsl= vpsl+1;
   }
   b[p].v= v; b[p].psl= vpsl;
}
```

[1] You might think the probe sequence (0, 1, 2) has length 3, but we say it is 2. Think of this sequence as we do a path (p0, p1, p2) in a graph; the path consists of the two edges from p0 to p1 and from p1 to p2, so the length is 2.

We now want to insert code at the TODO comment so that invariant INV, above, is maintained by the repetend. Before showing the code, we analyze an example, showing what we want to happen.

The hash table to the right, is a copy of the one above. It satisfies invariant INV. Suppose that `g` is to be inserted into the table and that it hashes to 1. Below and to the right, we show `p`, `vpsl`, and `v` at each iteration of the loop.

	0	1	2	3	4	5	...
b	a 0	e 1	f 2	b 2	c 3	d 3	

Consider the case `p` = 1. We have: `vpsl` < `b[p].psl`. This indicates that `v` must appear after `b[p]` in the table because its hash location is after that of `b[p].v`. So, continue looking for an empty bucket. This happens also on the second iteration, when `p` = 2.

Consider case `p` = 3. We have: `vpsl` = `b[p].psl`. This means that `v` and `b[p].v` hash to the same location. So, continue looking for an empty bucket. This happens also when `p` = 4.

p	vpsl	v	vpsl ? b[p].psl
1	0	g	<
2	1	g	<
3	2	g	=
4	3	g	=
5	4	g	>
6	4	d	b[p] is null

When `p` = 5, we have: `vpsl` > `b[p].psl`. This means that `v`'s hash location is before that of `b[p].v`. Therefore, to maintain invariant INV, `v` must be placed in a bucket before that of `b[p].v`. The easiest way to do this is to take rich value `b[p].v` out of the table and put in the poorer `v`. Then, continue to look for a null bucket to place the new value `v`! Here's the code to insert at the TODO point in method insert. That's all there is to it! Wow.

if (`vpsl` > `b[p].psl`) { swap `v` and `b[p].v`; swap `vpsl` and `b[p].psl`; }

Other methods, optimizations

Other methods can be written in a similar fashion. For example, `contains(v)` need only search until one of three things happen: (1) `b[p]` = null, (2) `v` = `b[p].v`, and (3) `vpsl` > `b[p].psl` (in which case v is not in the set). Also many more improvements can be made to Robin Hood hashing.

Benefits of Robin Hood hashing

Robin Hood Hashing was developed in a PhD thesis by Pedro Celis from the University of Waterloo (Canada) in 1986. Celis says in his thesis that Robin Hood hashing, with all the optimizations he has discussed in his thesis (not discussed here!), has these properties:

- Simple to program.
- The expected value of the longest probe sequence length for full tables is O(ln n) even for a full table.
- Unsuccessful search is just as fast as successful search.
- Cache friendly.

A bunch of further optimizations can be done, some taking advantage of the fact that all values that hash to the same bucket at in contiguous buckets.