Consider open addressing with linear probing and an attempt to see whether a value e is in the set. If e hashes to h, then buckets with indexes h % b.length, (h+1) % b.length, (h+2) % b.length, ... are probed until either e is found or a bucket containing null is found.

e1 e2 e3 e4

0 1 2 3 4 5 6 7

b

Linear probing can result in *clustering*: many values occupy successive buckets, as shown to the right, leading to excessive probes to determine whether a value is in the set. Here,

e1 hashed to bucket 2,

then e2 and e3 to hashed to bucket 3,

then e4 hashed to bucket 2.

If e5 now hashes to bucket 2, five probes are necessary to determine that e5 is not in the set.

Several ways of reducing clustering have been proposed over the years. We outline some of them to give you a greater sense of the lengths people go to in attempting to improve data structures.

**Cuckoo hashing**

The simple variant of cuckoo hashing uses two hash functions h1 and h2. To determine whether a value e is in the table, check the two positions b[h1(e)] and b[h2(e)] (taken modulo the table size, of course). If neither contains e, then e is not in the table; there is no need to worry about collisions. This is worst-case time O(1).

Similarly, to remove e from the table, look at those two buckets. If neither is e, then e is not in the set and nothing need be removed. If one of them is e, then set that bucket to null. This is worst-case time O(1).

Both search and remove take constant time in the worst case! How can that be! Because collisions won't occur. All the work to eliminate collisions takes place in the method to insert e into the hash table. It is shown to the right (look at it after reading everything below). *All indexes are taken module the table size*; to save space, we leave that implicit.

1. void insert(e) {

2. if (b[h1(e)] == e || b[h2(e)] == e) return;

3. int p= h1(e);

4. // inv: Trying to insert e at b[p]

5. loop n times: {

6. if b[p] == null { b[p]= e; return; }

7. Swap e and b[p];

8. if (p == h1(e)) p= h2(e);  
9. else p= h1(e);

10. }

11. rehash; insert(e);

12.}

This code is taken from a paper *Cuckoo hashing for undergraduates* by R. Pagh, written in 2006. You can get this paper from the JavaHyperText entry *hashing*. We found at least 15 papers cited on the web concerning cuckoo hashing and improvements to it!

Here's the major change from earlier probing strategies. If necessary, a value will be pushed out of the table (and placed elsewhere) to make room for the new one. That is what gives the method its name. In some species of cuckoo, the cuckoo chick pushes other eggs out of the nest when it hatches.

Method insert works as follows. Suppose e0 is to be inserted. Remember, two hash functions are used to produce two possible buckets for e0. The algorithm uses a value n; we do not discuss it in this high-level description:

If a bucket is null, store e0 there and return; else, kick some value e1 out of a bucket and store e0 there.

If a bucket is null, store e1 there and return; else, kick some value e2 out of a bucket and store e1 there.

...

If a bucket is null, store en-1 there and return; else, kick some value en out of a bucket and store en-1 there.

Create two new hash functions, h1 and h2, and rehash all values using the new functions.

Now, insert en into the table (a recursive call).