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.

**Cache** **performance**

This discussion introduces a new property concerning speed of execution. Generally, we talk about asymptotic complexity —e.g. one sorting algorithm is in worst-cast time O(n log n) while another is in O(n\*n). Discussing open addressing with probing introduces the notion *cache performance*.

Most computers today have *memory caches*, which contain blocks of memory that were recently used. A cache is close to the core, or processing unit, and is perhaps 25 to 100 times faster to access than memory. When a word is needed from memory, if it is in a cache, great —no need to look in memory. If the word is not in a cache, a block of words in memory that contain the word is copied into a cache and then that word is used.

For more detailed information on caches, read entry *cache* in JavaHyperText.

Important here is that a block of memory is copied, and not just the desired word —at no loss in speed. If other words in that block will be required soon, then time has been saved. For example, in the example of clustering given above, when e5 hashes to bucket 2, and therefore b[2] is retrieved from memory, quite likely, e2, e3, and e4 will be in the block that is copied into a cache, and referencing them will be faster because the cache can be used.

**Quadratic probing**

In preparation for the introduction of different ways of solving collisions, we assume that the value being hashed hashes to h. Linear probing probes the following buckets until null or the desired value is found —*remember*, *all integers below are taken mod the table size*, although we don't show that explicitly:

h, h+1, h+2, h+3, h+4, h+5, ...

Quadratic probing uses a different sequence of probes. It uses a polynomial to determine the sequence. The simplest example uses this sequence:

h, h+1^2, h+2^2, h+3^2, h+4^2, ..., i.e. h+1, h+4, h+9, h+16, ...

Use this sequence of probes instead of linear probing on the example shown above (e1 hashes to 2, e2 and e3 hash to 3, and e4 hashes to 2), and values would be placed as shown to the right. Value e4 is now separated from e1, e2, and e3.

e1 e2 e3 e4

0 1 2 3 4 5 6 7

b

In general, one can choose any polynomial. For example, using pk = h + 2k + 5k^2 , the buckets probed would be p0, p1, p2, p3, ...:

h, h+2+5, h+4+5\*4, h+6+5\*9, ...

Quadratic probing can reduce the number of collisions. But a big problem is to ensure that the probe sequence will cover enough buckets to always find null if the value being probed for is not in the hash table. For example, suppose array b has size 8, suppose *e* hashes to 0, and consider the probe sequence given by pk = h + k^2 — that's the first quadratic probe sequence first shown above. Here's the probe sequence —remember, hash values are taken mode the table size— with some buckets being repeated many times:

0, 1, 4, 1, 0, 1, ...

The period 1966–1975 saw a number of papers on quadratic probing, describing not only what quadratic polynomial to use but also the table sizes to use with that polynomial, and also discussing the problem mentioned in the previous paragraph. However, quadratic probing is not used much these days.

**Double hashing**

Use a second hash function hash(e) to help determine the probe sequence. Suppose e initially hashes to h and H = hash(e). Then use the probe sequence:

h, h+H, h+2H, h+3H, h+4H, h+5H, ...

**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).