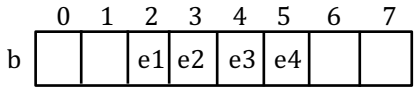
Open Addressing: Dealing with collisions

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.

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



`e1` hashed to bucket 2,
then `e2` and `e3` 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 `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

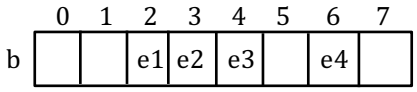
Suppose a value hashes to `h`. Linear probing probes the following buckets until null or the desired value is found —*remember*, *all integers below are taken mod the table size*, although we don't show that explicitly:

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

Quadratic probing uses a polynomial to determine the probe 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 probe sequence instead 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. `e4` is now separated from `e1`, `e2`, and `e3`.



One can choose any polynomial. For example, using the probe sequence $p_0, p_1, p_2, ...$ given by $p_k = h + 2k + 5k^2$, the buckets probed would be

`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 $p_k = h + k^2$ — that's the first quadratic probe sequence first shown above. Here's the probe sequence with some buckets being repeated many times —remember, hash values are taken mod the table size:

`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

Double hashing uses 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, ...`

Here's a major point about double hashing. First, in linear probing, the interval between probes is always 1. Second, in quadratic probing, the interval is the difference between two successive squares, but it's the same sequence of intervals no matter what `e` is. But in double hashing, the sequences of intervals for two different values are completely different, since they depend on `e`. This would seem to reduce collisions, and if you want as few collisions as possible, double hashing seems the way to go.

There are other issues with double hashing. The second hash function has to be non-zero and must be relatively prime to the table length. This means that when the table is resized, a different second hash function may have to be used. There's also the question of defining the second hash function for user-defined types.

Comparison of linear probing, quadratic probing, and double hashing

How does the internet compare these three methods of probing? Here's a quote from Wikipedia[1].

> The main trade-offs between these methods are that
>
> * linear probing has the best cache performance but is most sensitive to clustering,
> * double hashing has poor cache performance but exhibits virtually no clustering;
>   It also can require more computation than other forms of probing,
> * quadratic probing falls in-between in both areas.

That's a rather wishy-washy statement, giving no indication of which probing strategy is best. Some research *does* try to determine whether linear probing or double hashing is better in the face of good cash performance, but there is no theory of cash performance to help here, and one must rely on experiments. Complicating this issue is that there are many different computers with different architectures and caches of different sizes. How do you perform experiments on all of them? Further, the goodness of the hash function —how close to being a uniform distribution is it— influences greatly how many probes are needed.

Looking at many earlier papers, one could conclude that linear probing is a better choice than double hashing do to linear probing's better use of cache memory. Yes, probing sequences may be longer, but the use of the cache rather than memory to process many probes should give linear probing an advantage.

However, one paper that caught our eye is Heileman and Lou's paper *How Caching Affects Hashing*[2]. They report on extensive experiments in different situations. Their advice: Everyone is well-advised to choose double hashing over linear probing.

This is a long paper, not an easy read. But it will give the interested reader a glimpse into the world of research on hashing. Moreover, it has 22 references to other papers on the topic. You can obtain it in the JavaHyperText entry on *hashing*.

Other open-addressing schemes

There are other schemes to deal with probing are the Cuckoo hashing and Robinhood hashing. We'll cover them in another pdf.

---

[1] https://en.wikipedia.org/wiki/Open_addressing

[2] G.L. Heileman and W. Lou. *How Caching Affects Hashing.* Proc of workshop ALENEX/ANALCO, 2005