Hash Tables - Part 10

Robin Dawes

February 19, 2021

Begins our examination of the Open Addressing approach to collision resolution.

.

Open Addressing

We will look at three forms of open addressing: linear probing, quadratic probing, and double hashing.

For open addressing we change the notation for our hash function to include a second parameter:

$h(k,i)$ = the address in which we try to store key k, when i locations have previously been tried for this key. So $h(k,0)$ is the first address we we try, $h(k,1)$ is the address we try if $h(k,0)$ is already full, etc.

We break $h(k,i)$ into two independent functions like this:

$$h(k,i) = (h'(k) + f(k,i)) \bmod m$$

where $h'(k)$ is a hash function as we used the term before - any function that produces a value in the range [0...m-1]

$f(k,i)$ is any function that produces integer values and satisfies the requirement that $f(k,0) = 0 \quad \forall k$

Note that $h(k,i)$ includes a final " mod m" to ensure that the value of $h(k,i)$ is always in the range [0...m-1] even if $h'(k) + f(k,i)$ produces a large value.

The search algorithm for a hash table using open addressing examines exactly the same sequence of addresses as the insert algorithm. The search is successful if the key is found, and unsuccessful if either an empty location is found, or all locations are examined without finding the desired key.

The sequence of addresses examined during any of the three essential operations is called the PROBE (OR PROBING) SEQUENCE for that key. It should be clear that the probe sequence is completely determined by the key value. Ideally, the probe sequence for each key should contain every address in the hash table exactly once. These probe sequences would be permutations of the set $\{0, ..., m-1\}$ ... so there are $m!$ possible probe sequences. We will use this to compare the three forms of open addressing.

Linear Probing

THE IDEA OF linear probing is to resolve collisions by looking at the addresses sequentially following the first address tried. To achieve this we simply let $f(k, i) = i$.

When computing $h(k, 0), h(k, 1), h(k, 2)$ etc, the $h'(k)$ part never changes, so in implementation we compute this once and then use it as needed.

Note that we need some way to establish that an address is empty - this is typically done by storing an illegal key value in each address where no key has yet been stored. For example if the legal keys are all positive integers, we can use 0 to signify "empty". Since this depends on the actual set of possible keys I will just use "empty" in the pseudo-code versions of the algorithms.

```
Linear_Probing_Insert(k):
    i = 0
    v = h'(k)
    while (i < m):
        a = (v+i) % m
        if (T[a] is "empty"):
            T[a] = k
            break
        else:
            i ++

    if i == m :
        report "table full, insert failed"
```

In this pseudo-code I am using the command "report" to represent whatever action is appropriate to signal failure to complete the

task. Appropriate actions might be returning a FALSE Boolean value, setting a flag, raising an error condition, etc.

Here's an important point to consider when writing production-level code for hash tables. Remember that the keys are supposed to be unique. Well, are we really going to trust some dumb user to never try to add two data objects with the same key (or even just add the same data object twice)? (Rule 1: never underestimate the user's ability to mess up your program.)

So our insert method should look something like this:

```
Linear_Probing_Insert(k):
    i = 0
    v = h'(k)
    while (i < m):
        a = (v+i) % m
        if (T[a] == k):
            report "Attempt to insert duplicate key"
            break
        elif (T[a] is "empty"):
            T[a] = k
            break
        else:
            i ++

    if (i == m) :
        report "Table full, insert failed"
```

Now for searching - pretty similar to inserting.:

```
Linear_Probing_Search(k):
    i = 0
    v = h'(k)
    while (i < m):
        a = (v + i) % m
        if (T[a] is "empty"):
            report "Search value not found"
            break
        elif (T[a] == k):
            report "Found it"
        else:
            i ++
    if (i == m):
```

```
        report "Search value not found"
```

As mentioned previously, we probably want to do something with the data item once we find it ... so it makes more sense to return the location. Something like this:

```
Linear_Probing_Search(k):
    i = 0
    v = h'(k)
    while (i < m):
        a = (v + i) % m
        if (T[a] is "empty"):
            report "Search value not found"
            return -1
        elif (T[a] == k):
            return a
        else:
            i ++
    if (i == m):
        report "Search value not found"
        return -1
```

Easy peasy, but what about deletion? The problem is that if we delete a key value by replacing it by our "empty" flag value, then a subsequent search for some other key might give an incorrect result due to hitting this empty spot and stopping, when it should have continued and found the key value.

EXAMPLE: Suppose the keys are integers in the range [1 ... 20], m = 10, and we decide to use

$$h'(k) = \left\lfloor \frac{k^2}{7} \right\rfloor$$

This $h'(k)$ is *not* a good hash function. Can you see why already?

If our first item for insertion has $k = 13$, the probe sequence is $4, 5, ..., 9, 0, 1, 2, 3$ and we place the data in T[4]

If our second item for insertion has $k = 10$, the probe sequence is again $4, 5, ...1, 2, 3$ and after finding that T[4] is full we place the data in T[5].

Now suppose we delete the first item ($k = 13$) and place the "empty" flag in T[4].

Now, finally, suppose we search for the second item ($k = 10$) . We look in T[4], we see "empty", and we stop ... even though the desired item is in the table.

We solve this by choosing another flag value to signify "deleted". For example if the valid keys are all positive integers, we could use -1 as the "deleted" flag. This changes our insert algorithm a bit: we can insert a new value into any address that is either "empty" or "deleted".

```
Linear_Probing_Insert(k):
    i = 0
    v = h'(k)
    while (i < m):
        a = (v+i) % m
        if (T[a] == k):
            report "Attempt to insert duplicate key"
            break
        elif (T[a] is "empty") OR (T[a] is "deleted"):
           T[a] = k
           break
        else:
           i ++

if (i == m) :
    report "Table full, insert failed"
```

The search algorithm does not change at all! Now we can write the delete algorithm:

```
Linear_Probing_Delete(k):
    i = 0
    v = h'(k)
    while (i < m):
        a = (v+i) % m
    if (T[a] == k):
        T[a] = "deleted"
        break
    elif (T[a] is "empty"):
        report "delete failed - value not found"
        break
    else:
        i ++
```

```
    if (i == m) :
        report "delete failed - value not found"
```

Linear probing is quick and easy and it is guaranteed to find an empty address if there is one. Unfortunately it is subject to a phenomenon called primary clustering which can negatively affect the expected times for insertion, search and deletion. The problem is that if (for example) 4 consecutive addresses are filled and the next address is empty, the probability that the next address will be filled on the next insert is higher than it should be: any key that hashes to any of the 4 filled addresses will end up in the next one. Thus blocks of consecutive filled addresses tend to get larger and larger, and the number of probes needed to complete any of the three essential operations gets larger too. In the worst case we can end up with $O(m)$ time for each of the essential operations.

Many Intertubes articles purporting to be tutorials on hashing go no further than linear probing ... to the potential detriment of any readers.

Quadratic Probing

Quadratic probing is similar to linear probing except that instead of $f(k,i) = i$, we use $f(k,i) = c_1 * i + c_2 * i^2$, where c_1 and c_2 are constants (usually but not always positive integers).

Fortunately we don't need to come up with new algorithms for the three essential operations! The algorithms we developed for linear probing (using "empty" and "deleted" flag values) need only to have the new $f(k,i)$ function replace the one we used for linear probing.

```
Quadratic_Probing_Insert(k):
    i = 0
    v = h'(k)
    while (i < m):
        a = (v + c1*i + c2*i^2) % m
        # note: c1 and c2 would be defined externally and
        # shared by all three methods: insert, search, delete
        if (T[a] == k):
            report "Attempt to insert duplicate key"
            break
        elif (T[a] is "empty") or (T[a] is "deleted"):
            T[a] = k
            break
        else:
```

```
            i ++

    if (i == m) :
        report "Table full, insert failed"
        # but this may be a lie - the table may not be full!


Quadratic_Probing_Search(k):
    i = 0
    v = h'(k)
    while (i < m):
        a = (v + c1*i + c2*i^2) % m
        if (T[a] is "empty"):
            report "Search value not found"
            break
            # or
            return -1
        elif (T[a] == k):
            report "Found it"
            # and/or
            return a
        else:
            i ++

   if (i == m):
      report "Search value not found"
      # and/or
      return -1


Quadratic_Probing_Delete(k):
    i = 0
    v = h'(k)
    while (i < m):
    a = (v + c1*i + c2*i^2) % m
    if (T[a] == k):
        T[a] = "deleted"
        break
    elif (T[a] is "empty"):
        report "delete failed - value not found"
        break
    else:
        i ++
```

```
if (i == m) :
    report "delete failed - value not found"
```

Quadratic probing greatly reduces the effect of primary clustering. To illustrate this effect, consider a simple example: let $c_1 = c_2 = 1$, and let $m = 11$. Let k_1 and k_2 be two keys such that $h'(k_1) = 0$ and $h'(k_2) = 2$. Then k_1's probe sequence is

i	$h(k_1, i)$
0	0
1	2
2	6
3	1
4	9
...	...

Make sure you understand how the values in these probe sequences are computed.

Here is k_2's probe sequence

i	$h(k_1, i)$
0	2
1	4
2	8
3	3
4	0
...	...

Even though the probe sequences both contain 2, they go off in different directions after that. Note that they also both contain location 0 - in k_1's probe sequence it is followed by 2. What is it followed by in k_2's probe sequence?

When we use quadratic probing, two probe sequences may hit the same address at any point but then hit different addresses after that. This greatly reduces the problem of primary clustering - compare this to linear probing, in which two probe sequences are locked together as soon as they share a common value.

Note that with quadratic probing there is still a problem with SECONDARY CLUSTERING: if $h'(k_1) = h'(k_2)$, the probe sequences for k_1 and k_2 will be identical. Thus there are only m different probe sequences, out of the possible $m!$ sequences in which we could conceivably probe the table. Fortunately secondary clustering is much less of a problem than primary clustering.

But quadratic probing has a potentially much bigger problem: unless m, c_1 and c_2 are carefully chosen, a probe sequence may only include a subset of the possible addresses.

EXAMPLE: Let $m = 12$, $c_1 = 1$ and $c_2 = 1$. Suppose $h'(k_1) = 0$. The probe sequence for k_1 is $0, 2, 6, 0, 8, 6, 6, 8, 0, 6 \ldots$ We seem to be trapped in repeated visits to a very small set of addresses. In fact it is easy to see that this probe sequence will never contain any odd addresses: we have

$$\begin{aligned} h(k_1, i) &= (h'(k_1) + i + i^2) \bmod 12 \\ &= (0 + i * (i + 1)) \bmod 12 \\ &= (i * (i + 1)) \bmod 12 \end{aligned}$$

and since $i * (i + 1)$ is always even, $i * (i + 1) \bmod 12$ will also always be even - so this probe sequence will never contain any odd addresses.

Aren't you glad you did all that modular arithmetic in CISC-203? It is a bit more challenging to determine whether or not 4 and/or 10 ever occurs in the probe sequence we have started to write out in this example - I leave that to you as an exercise for a rainy day with nothing good on Disney+.

Why is this important? Suppose we are attempting to insert k_1 into the hash table and all the even addresses are full but all the odd addresses are empty. Our insert attempt will fail because k_1's probe sequence never looks at the odd addresses - so we can't insert the new data even though the table is half empty. This is not good!

You may have noticed a difference between the two examples we have done. In the first one we used $m = 11$ and things worked out ok. In the second example we used $m = 12$ and things went sideways on us. The difference of course is that 11 is a prime number and 12 is not.

As a simple illustration of why this is relevant, when we are computing the expression $c_1 * i + c_2 * i^2$ ... which we can write as $(c_1 + c_2 * i) * i$ ... there are lots of ways this can turn out to be a multiple of 12. For example, the first term can be a multiple of 3 and the second term can be a multiple of 4 (or vice versa), or the first term can be a multiple of 2 and the second term can be a multiple of 6 (or vice versa), or either term can be a multiple of 12. And if this expression *is* a multiple of 12, then $h(k, i)$ becomes just $h'(k) \bmod 12$ for this value of i. This means that $h'(k) \bmod 12$ will show up quite frequently in the probe sequence for k. At the very least we are revisiting an address that we have already looked at (which is a waste of time), and at worst there is a big risk that the probe sequence will contain restrictive patterns such as the one we saw above.

By contrast, there are relatively few ways that $c_1 * i + c_2 * i^2$ (that is, $(c_1 + c_2 * i) * i$) can turn out to be a multiple of 11: it only happens when one or both of the terms are themselves multiples of 11. Thus with a table size of 11 we are less likely to see probe sequences that return to their starting points over and over such as we saw for a table of size 12.

This is just a tiny step towards a proper discussion of the best way to choose the size of your hash table, but it suggests a solid fundamental idea: probe sequences will be less likely to fall into patterns if we let m be a prime number.

A full discussion of the best way to choose m, c_1 and c_2 for quadratic probing is beyond the scope of these notes ... but I encourage you to do some independent reading on this topic. The number theory you studied in CISC-203 will help you.