Hash Tables and Hash Functions

We have seen that with a balanced binary tree (such as a Red-Black Tree) we can guarantee $O(\log n)$ worst-case time for insert, search and delete operations. Our challenge now is to try to improve on that ... if we can.

Suppose we have data objects that consist of pairs - each pair having a key value that is unique to the object, and some "satellite" data that may or may not be be unique. For example, the data might consist of "phone number, name" pairs in which each phone number must be unique (the **key**), but there might be two people named "Cholmondeley Featherstonehaugh Marjoribanks Wriothesley" (duplicate satellite data) – and I would feel sorry for both of them.

Assuming for the moment that the keys are integers, the simplest method to store the data is in an array that has an address for each possible key - this is called **direct addressing**. Then insert, search and delete all take $O(1)$ time – we can't beat that!

The problem of course is that the set of possible keys may be immense - with 10-digit phone numbers there are 10^{10} possible combinations, and most people have at most a couple of hundred phone numbers in their contact list. Even if you have every person in Canada in your contact list, creating an array with 10^{10} elements and only using 35,000,000 of them is not very practical.

However, the idea of using a simple array to store our data is very appealing and as we shall see, with a little care and attention we can get good **average-case** complexity for our three essential operations, even though we may not have optimal **worst-case** complexity.

Since we are introducing average-case complexity we should spend a moment looking at balanced binary trees in this way. In a complete binary tree (ie one in which there are **no** missing children - all the leaves are at the bottom level) - almost exactly half the vertices are at maximum distance from the root. This implies that the average insert/search/delete time is going to be close to $\log n$. We can make the same argument for Red-Black trees.

Since we contemplating using an array that is smaller than the set of all possible keys, we clearly need some way to map key values onto array addresses. For example if our array (which we will call T) has index values 0, 1, ..., 9 and our key value is 34, we need to decide which array address to use to store the data. We call this mapping a **hash function.** We call the array T a **hash table**.

For the rest of this discussion we will use the following notation consistently:

m : the size of T. T has addresses 0, 1, ..., m-1

n : the number of data objects we need to be able to store in T. If this is not known precisely, we should at least be able to put an upper limit on it.

h(k) : a function that takes a key value as its argument and returns a value in the range [0 .. m-1]

We will spend some time later talking about how to choose h(k), but for now we will assume the keys are non-negative integers and we will use

h(k) = k mod m

as our hash function. So continuing the previous example, with m = 10 and k = 34, we get h(k) = 4

The problem is that since the number of possible keys exceeds m, we may get **collisions** - two or more keys in our set that hash to the same address. To deal with collisions we need to define a **collision resolution** method.

Note that due to collisions, we must store the key value as well as its satellite data - otherwise we cannot distinguish between the data associated with different keys that have the same value of h(k)

Collision Resolution Methods

The very bad method: If we are trying to insert a value k and the address $h(k)$ is already occupied, we simply reject the new data. This has the advantage of making insert/search/delete all $O(1)$ worst case - but it has the downside that we are frequently unable to successfully insert new values even though there may be a lot of empty space in T.

Another bad method: If we are trying to insert a value k and the address h(k) is already occupied, we overwrite h(k) with the new data. This has $O(1)$ complexity for all operations, and we are always able to insert a new value. Alas, we are likely to lose a lot of data.

Challenge: come up with a situation where you can argue that one of the above methods is useful.

A good method: **Chaining**

In a chained hash table, T is not used to store the data directly. Each element of T is a pointer to the head of a linked list of objects that have all hashed to the same location in T. If no key values currently in the set have hash value = i, then T[i] is a null pointer. Each data pair is implemented as an object that contains the key value, the satellite data, and a pointer variable that will be used to connect to the next object in the list, if any.

Insert: It is always possible to insert a new key k into the hash table. We add the new object at the head of the list attached to its hash value address in T. This gives insertion O(1) complexity

```
def insert(new_object):
        hash_val = h(new_object.key)
        new_object.next = T[hash_val]
        T[hash_val] = new_object
```

Search is also simple: we go to the hash value address in T and search through the list:

```
    def search(k):
        hash_val = h(k)
        temp = T[hash_val]
        while temp != null and temp.key != k:
                temp = temp.next
        if temp != null:
                return "found it"
        else:
                return "not found"
```

Delete is similar to search - we just need to fix up the pointers in the list

```
def delete(k):
        hash_val = h(k):
        temp = T[hash_val]
        if temp == null:
           return
        elif temp.key == k:
           T[hash_val] = temp.next
        else:
           previous = temp
           current = temp.next
           while current != null and current.key != k:
                   previous = current
                   current = current.next
           if current != null:
                   previous.next = current.next
```

Since the linked lists can be arbitrarily long, there is no upper limit to the number of values that can be stored in the hash table. But the longer the chains, the longer it will take to search and/or delete.

We will make **the uniform hashing assumption**: we assume that our hash function h(k) maps key values uniformly onto addresses - that is, each key is equally likely to be hashed to each address. The validity of this assumption depends on a number of factors, including the distribution of the keys, the size of the table, and the hashing function itself - we will return to this issue later but for now we will just make the assumption. In effect, this means that approximately equal number of keys are mapped onto each address in T.

With this assumption, the expected number of data objects in each chain is $\frac{n}{m}$ (recall that n is the number of values stored in the table, and m is the size of the table). From this it is possible to show that the **expected number** of steps in a search (either successful or unsuccessful) is in $O\left(1+\frac{n}{m}\right)$ - full details of this proof are in the textbook. Since a delete operation requires only O(1) operations after the object has been found, the expected time for both search and delete is in $O\left(1+\frac{n}{m}\right)$.

Writing $O\left(1+\frac{n}{m}\right)$ instead of just $O\left(\frac{n}{m}\right)$ may seem a little odd, but one way to see why it is useful is this: if we treat both n and m as variables, then $\frac{n}{m}$ can be arbitrarily close to 0. However every search operation will take at least a constant amount of time because we have to compute the hash value of the key. Including the 1 in the order reflects the fact that the complexity cannot be arbitrarily small.

Nonetheless, I'm going to be lazy and write the complexity as $O\left(\frac{n}{m}\right)$… for most combinations of n and m it makes no difference.

Is it reasonable to think of n and m as variables? When we are creating our data structure we can ask this question: when the expected number of data items to be stored is n, how big should m be to make our operations efficient? Or looking at it another way: for given values of n and m, what is the expected search time? In these questions both n and m are definitely variables. When building a hash table in a real-world situation, n is most likely going to be dictated by the application, and m would be computed from that … but knowing how to choose m wisely comes from considering the relationship between n, m and efficiency (which is what we are doing now).

The ratio $\frac{n}{m}$ is called the **load factor** of the hash table, and we often see the symbol α used for this. If α is high then collisions are more likely.

The downside of chaining is that indirect addressing (the pointers we use to link together the chains) is physically slower than direct addressing. The most popular alternative is to resolve a collision by finding an empty address in the table and storing the new data object there. This is called **open addressing**.

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 to try to store key k, when i locations have previously been tried for this key

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

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

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

$$f(k,0) = 0$$

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

Open addressing looks like this: Given a key value k, we first compute $h(k,0)$. If there is a collision at that address, we compute $h(k,1)$. If there is a collision there, we try $h(k,2)$… and so on.

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, each probe sequence should contain each 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 fact 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 $f(k, i) = i$

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

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 just use it over and over.

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.

I think these algorithms are marginally different than the way I wrote them on the board in class. Functionally they are the same.

```
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"
```

Here’s something we DIDN’T talk about in class, because it slipped my mind. 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 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"
```

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 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

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

(Side note: We will show in class that k^2 is not a good hash function. Can you see why already?)

If our first key for insertion is k_1 = 13, the probe sequence is 4, 5, …, 9, 0, 1, 2, 3 and we place the data in T[4]

If our second key for insertion is k_2 = 10, the probe sequence is again 4, 5, … 1, 2, 3 and we place the data in T[5].

Now suppose we delete k_1 and place the "empty" flag in T[4].

Now, finally, suppose we search for k_2. We look in T[4], we see "empty", and we stop … even though k_2 is in the table.

We solve this by choosing another flag value to signify "deleted". Again, if the valid keys are all positive integers, we could use -1 as the "deleted" flag. This changes our insert algorithm a bit: **insert** can insert 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. 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 $O(n)$ time for each of the essential operations.

Quadratic Probing

Quadratic probing is similar to linear probing except that instead of $f(i) = i$, we use $f(i) = c_1 * i + c_2 * i^2$, 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. The algorithms we developed for linear probing (using "empty" and "deleted" flag values) need only to have the new $f(i)$ 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
          elif (T[a] == k):
               report "Found it"
          else:
               i ++
        if (i == m):
               report "Search value not found"
```

```
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, consider a simple example: let $c_1 = c_2 = 1$, and let m = 11. Let k_1 and k_2 be two keys. Suppose $h'(k_1) = 0$. Then k_1's probe sequence is

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

(Check to make sure you understand how the values in this probe sequence are computed!)

Now suppose $h'(k_2) = 2$. Then k_2's probe sequence is

i	$h(k_2, i)$
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 the value 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 what is called **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 a possible m! sequences in which we could conceivably search the table. Fortunately, secondary clustering is much less of a problem than primary clustering.

However, 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. For 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 etc. ... 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

$$\begin{aligned} \mathrm{h}(\mathrm{k}_1, i) &= (h'(k_1) + i + i^2) \mod 12 \\ &= (0 + i * (i + 1)) \mod 12 \\ &= (i * (i + 1)) \mod 12 \end{aligned}$$

and since $i * (i + 1)$ is always even, $(i * (i + 1)) \mod 12$ 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 $h'(k) \mod 12$ for this value of i. This means that $h'(k) \mod 12$ will show up quite frequently in the probe sequence for k. At the very least we are frequently 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 even more 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 CISC-235 … but I encourage you to do some independent reading on this topic. The number theory you studied in CISC-203 will help you.