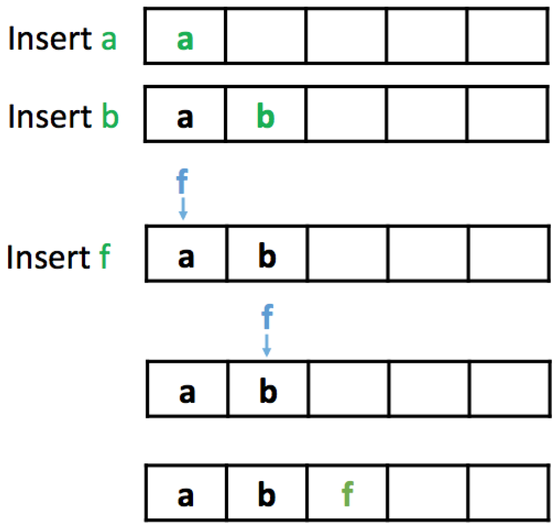
Collision Resolution: Open Addressing

Step 1

As we mentioned repeatedly thus far, we have acknowledged that collisions can happen, and even further, we have proven that they are statistically impossible to fully avoid, but we simply glossed over what to do should we actually encounter one. In this section, we will discuss the first of our **collision resolution strategies** (i.e., what should be done to successfully perform an insert/find/remove operation if we were to run into a collision): **Linear Probing**. The idea behind this strategy is extremely simple: if an object *key* maps to an index that is already occupied, simply shift over and try the next available index.

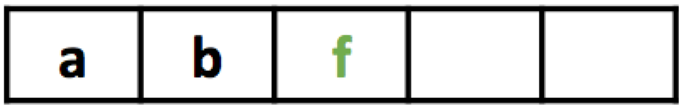
Here is an intuitive example in which the **hash function** is defined as $H(k) = (k+3) \% m$, where k is the ASCII value of *key* and m is the size of the backing array (we add 3 to k to have the letter 'a' hash to index 0 for simplicity):



STOP and Think: Suppose you are inserting a new key into a **Hash Table** that is already full. What do you expect the algorithm to do?

Step 2

EXERCISE BREAK: What is the probability of a collision for inserting a **new** arbitrary key in the **Hash Table** below (the same **Hash Table** *after* we inserted the key 'f' in the previous step)? Input your answer in decimal form and round to the nearest thousandth.



To solve this problem please visit https://stepik.org/lesson/31223/step/2

Step 3

How do we go about actually implementing this simple collision resolution strategy? Here is the pseudocode to implement **Linear Probing** when inserting a key k into a **Hash Table** of capacity M with a backing array called *arr*, using an indexing **hash function** $H(k)$:

```
insert_LinearProbe(k): // Insert k into the Hash Table using Linear Probing
    index = H(k)

    Loop infinitely:
        if arr[index] == k:         // check for duplicate insertions
            return

        else if arr[index] == NULL: // insert if slot is empty
            arr[index] = k
            return

        else:                       // there is a collision, so recalculate index
            index = (index + 1) % M

        if index == H(k):           // we went full circle (no empty slots)
            enlarge arr and rehash all existing elements
            index = H(k)            // H(k) will index differently now that arr is a different
size
```

Notice that the core of the **Linear Probing** algorithm is defined by this equation:

```
index = (index + 1) % M
```

By obeying the equation above, we are ensuring that our key is inserted strictly within an index (or more formally, an address) *inside* our **Hash Table**. Consequently, we call this a **closed hashing** collision resolution strategy (i.e., we will insert the actual key only in an address bounded by the realms of our **Hash Table**). Moreover, since we are inserting the keys themselves into the calculated addresses, we like to say that **Linear Probing** is an **open addressing** collision resolution strategy (i.e., the keys are open to move to an address other than the address to which they initially hashed.

Note: You will most likely encounter people using the terms **closed hashing** and **open addressing** interchangeably since they arguably describe the same method of collision resolution.

STOP and Think: How would we go about *finding* a key using **Linear Probing**?

Step 4

EXERCISE BREAK: Suppose we have an empty **Hash Table**, where $H(k) = k \% M$ and $M = 7$. After inserting the keys 31, 77, and 708 into our **Hash Table** (in that order), which index will the key 49 end up hashing to using the collision resolution strategy of **Linear Probing**?

To solve this problem please visit https://stepik.org/lesson/31223/step/4

Step 5

EXERCISE BREAK: Using the collision resolution strategy of **Linear Probing**, fill in the missing elements in the **Hash Table**, where $H(k) = k \% M$ and $M = 5$, after executing all the operations shown below.

- Insert: 5
- Insert: 10
- Insert: 11
- Insert: 12
- Insert: 13

0	1	2	3	4
?	?	?	?	?

To solve this problem please visit https://stepik.org/lesson/31223/step/5

Step 6

EXERCISE BREAK: Sort the keys below in the order that they must have been inserted into the **Hash Table** below based on the following facts:

- **Linear Probing** was used as the collision resolution strategy.
- *H*(*k*) = *k* % *M*, where *k* is the ASCII value of the key and *M* is the size of the backing array

0	1	2	3	4
R		H	C	D

Feel free to use the ASCII chart located here.

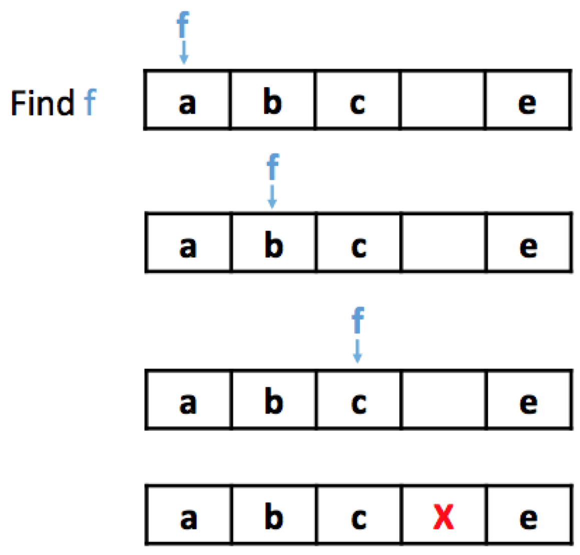
To solve this problem please visit https://stepik.org/lesson/31223/step/6

Step 7

We've discussed how to insert elements using **Linear Probing**, but how do we go about deleting elements?

In order to answer this question, we first need to take a look at how the *find* algorithm works in a **Hash Table** using **Linear Probing.** In general, in order to not have to search the *entire* **Hash Table** for an element, the find algorithm will search for an element until it finds an empty slot in **Hash Table**, at which it will terminate its search.

For example, using the same **hash function** as we did in the previous steps, *H*(*k*) = (*k*+3) % *m*, we would find an element like so:

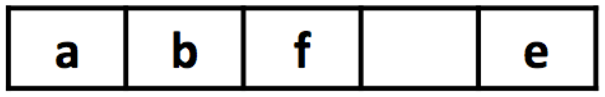


STOP and Think: Why does this always work?

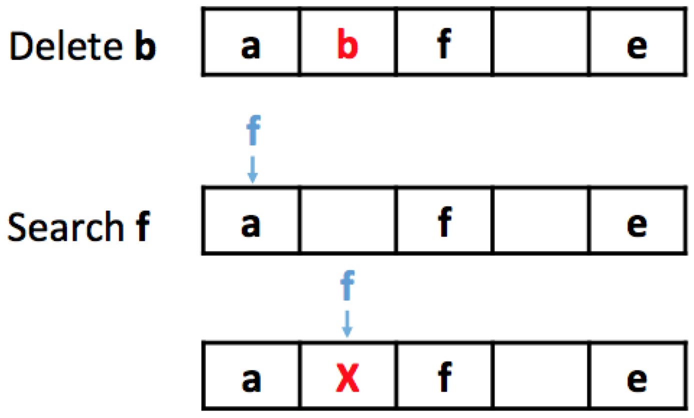
Step 8

Now that we know how the *find* operation works, let's look at how we might go about deleting an element in a **Hash Table** that uses **Linear Probing**. Intuitively, you might think that we should delete the key by simply emptying the slot. Would that work?

Suppose we have the following **Hash Table**:



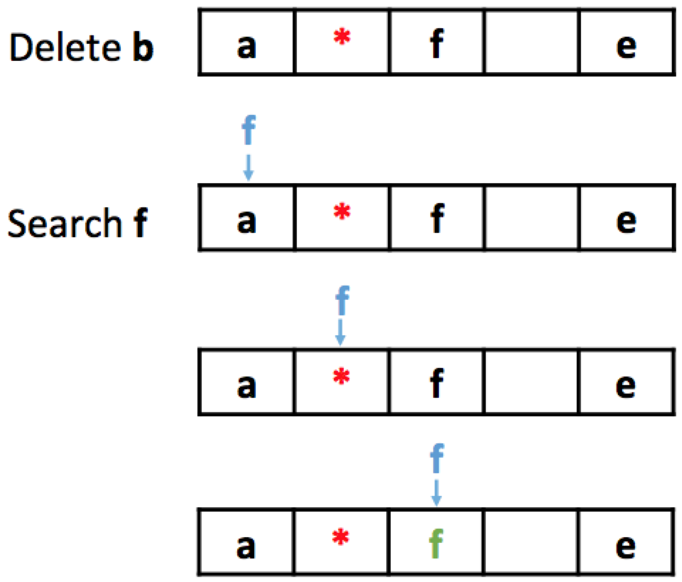
Now suppose we use the deletion method described above to delete 'b' and then search for 'f'. Let's see what happens:



As we see in the example above, the key 'f' would not have been found, which is clearly a problem.

Note: You might be thinking "Why don't we just edit the find method so that it doesn't stop when a slot is empty?". This would work, but if we were to do so, our find operations would become *significantly* less efficient. In practice, the find operation is used much more frequently than the remove operation, and as a result, it makes more sense to keep the find operation as optimized as possible.

Consequently, the solution to this conundrum is to use a "deleted" flag. In other words, we need to specify to the **Hash Table** that, even though there is in fact no element in that slot anymore, we still need to look beyond it since we have *deleted* from the slot.



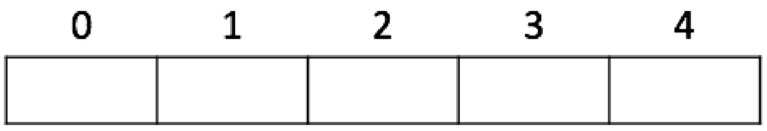
Now that we have discussed a proper remove algorithm, it is important to note that we face the problem of having these "deleted" flags piling up, which would make our find algorithm less and less efficient. Unfortunately, the only solution to this problem is to regenerate a new **Hash Table** and reinsert all valid (i.e., not deleted) keys into the new **Hash Table**.

Step 9

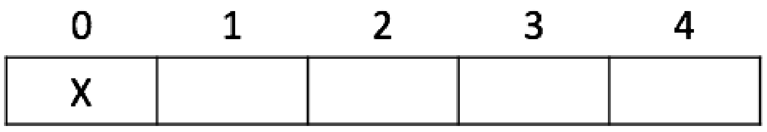
As you hopefully agree, **Linear Probing** is a fairly simple and straightforward solution to collision resolution. What's the catch?

We had originally introduced **Hash Tables** because we wanted to achieve extremely fast find, insert, and remove operations. Unfortunately, **Linear Probing** slows us down extensively. As you might have noticed while doing the previous exercises, **Linear Probing** can resort to having to *linearly scan* the **Hash Table** to find an open slot to insert a key. As a result, our **worst-case** time complexity for an insert operation becomes **O(*N*)**. The same goes for attempting to *find* a key in the **Hash Table**: in the worst-case scenario, we must linearly scan the entire **Hash Table** to see if the key exists. However, with a table that is not very full, we can in fact achieve an ***average-case* O(1)** time complexity for our operations.

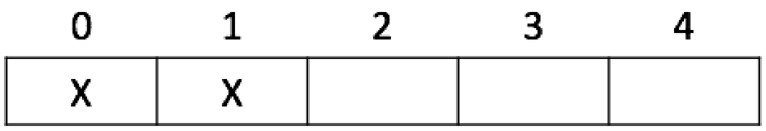
Another negative quality about **Linear Probing** that you may have noticed is that it results in clusters, or "clumps," of keys in the **Hash Table**. It turns out that clumps are not just "bad luck": probabilistically, they are actually *more likely* to appear than not! To demonstrate this, we will use the following empty **Hash Table**:



If we were to insert a new key, it would have a $\frac{1}{5}$ chance of landing in any of the 5 slots. Let's say that, by chance, it landed in slot 0:



Now, if we were to insert a new key again, how likely is it to land in any of the four remaining slots? Intuitively, you might say that it would have a $\frac{1}{4}$ chance of landing in any of the 4 remaining slots, but unfortunately, this is not correct. Remember: even though it can't be inserted in slot 0, it can still *index* to slot 0! For slots 2, 3, and 4, each has a $\frac{1}{5}$ chance of being filled by our new key (if the key indexes to 2, 3, or 4, respectively). What about slot 1? If the new key indexes to slot 1, it will be inserted into slot 1. However, remember, if the element indexes to slot 0, because of **Linear Probing**, we would shift over and insert it into the next open slot, which is slot 1. As a result, there are two ways for the new key to land in slot 1, making the probability of it landing in slot 1 become $\frac{2}{5}$, which is **twice as likely** as any of the other three open slots! Because this probability is twice as large as the others, let's say we ended up inserting into slot 1:



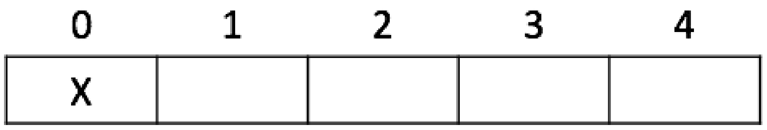
Now, what if we were to insert another key? Just as before, slots 3 and 4 each have a $\frac{1}{5}$ chance of being filled (if we were to index to 3 or 4, respectively). To fill slot 2, we could index to 2 with $\frac{1}{5}$ chance, but what would happen if we were to index to 1? Again, because of **Linear Probing**, if we index to 1, we would see the slot is filled, and we would shift over to slot 2 to perform the insertion. Likewise, if we index to 0, we would see the slot is filled and shift over to slot 1, see the slot is filled and shift over to slot 2, and *only then* perform the insertion. In other words, the probability of landing in slot 2 is $\frac{3}{5}$, which is **three times as likely** as any of the other slots!

We explained earlier *why* clumps are bad (because we potentially have to do a linear scan across the clump, which is slow), but now we've actually shown that clumps are *more likely to appear and grow* than not! What can we do to combat this issue?

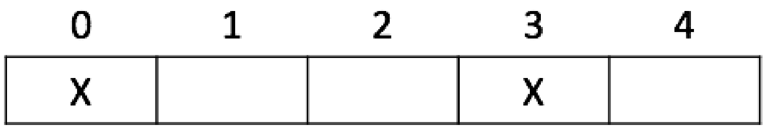
Step 10

Intuitively, you might think that, instead of probing one slot over during each step of our scan, we could just choose a larger "skip": why not jump *two* slots over? Or *three* slots over? This way, the the elements would be more spaced out, so we would have solved the issue of clumps, right? Unfortunately, it's not that simple, and the following example will demonstrate why simply changing the skip from 1 to some larger value doesn't actually change anything.

Let's start with the following **Hash Table** that contains a single element, but this time, let's use a skip of 3:



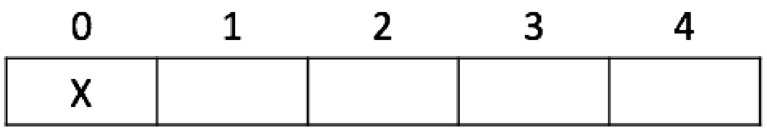
What would happen if we were to try to insert a new element? Like before, it has a $\frac{1}{5}$ chance of indexing to 0, 1, 2, 3, or 4. If it happens to land in slots 1, 2, 3, or 4 (each with $\frac{1}{5}$ probability), we would simply perform the insertion. However, if it were to land in slot 0 (with probability $\frac{1}{5}$), we would have a collision. Before, in traditional **Linear Probing**, we had a skip of 1, so we shifted over to slot 1, so the probability of inserting in slot 1 was elevated (to $\frac{2}{5}$). Now, if we index to 0, slot 3 has an elevated probability of insertion (to $\frac{2}{5}$, like before). In other words, by using a skip of 3, we have the exact same predicament as before: one slot has a higher probability of insertion than the others! All we've done is *changed* which slot it is that has a higher probability!



Even though it might *appear* as though the clumps have disappeared because there's more space between filled slots, it turns out that the clumps still do in fact exist: they are just harder to see. The reason *why* clumps tend to grow is the exact probabilistic issue we saw above (and on the previous step): with the initial indexing, all slots are equally likely to be filled, but upon collision, we deterministically increase the probability of filling certain slots, which are the slots that expand a clump. Instead, we need to think of a clever way to somehow *evenly distribute* the insertion probabilities over the open slots, but in a way that is deterministic such that we would still be able to find the key if we were to search for it in the future.

Step 11

The solution to avoid keys having a higher probability to insert themselves into certain locations—thereby avoiding the creation of clumps—is pretty simple: designate a *different* offset for each particular key. Once we do this, we ensure that we have an *equal* probability for a key to hash to any slot. For example, suppose we start with the **Hash Table** below:



Our initial problem in **Linear Probing** was that slot 1 had an unequal probability of getting filled (a $\frac{2}{5}$ chance). This was because slot 1 had two ways of getting filled: if key *k* initially hashed to index 1 *or* index 0, thereby guaranteeing that the key *k* would also hash to index 1. By designating a *different* offset for each particular key, however, we no longer have the guarantee that just because key *k* initially hashed to index 0, it will also hash to index 1. As a result, there is no guarantee for a *particular* area in the **Hash Table** to face a higher probability for keys to hash there and we thus consequently eliminate the inevitable creation of clumps.

We do what is described above using a technique called **Double Hashing**. The concept behind the collision resolution strategy of **Double Hashing** is to use two hash functions: $H_1(k)$ to calculate the hashing index and $H_2(k)$ to calculate the *offset* in the probing sequence.

More formally, $H_2(k)$ should return an integer value between 1 and *M-1*, where *M* is the size of the **Hash Table**. You can think of **Linear Probing** as originally having $H_2(k) = 1$ (i.e., we always moved the key 1 index away from its original location). A common choice in **Double Hashing** is to set $H_2(K) = 1 + \frac{K}{M} \ \% \ (M-1)$).

Below is the pseudocode to implement **Double Hashing** when inserting a key `k` into a **Hash Table** of capacity `M` with a backing array called `arr`, using a **hash function** `H(k)`. Note that the pseudocode for **Double Hashing** is extremely similar to that of **Linear Probing** (the changes are highlighted with the comment `NEW`):

```
insert_DoubleHash(k): // Insert k using Double Hashing as the collision resolution strategy

    index = H1(k)
    offset = H2(k) // NEW

    Loop infinitely:

        // check for duplicate insertions (not allowed)
        if arr[index] == k:
            return

        // check if the slot of the array is empty (i.e., it is safe to insert)
        else if arr[index] == NULL:
            arr[index] = k
            return

        // there is a collision, so re-calculate index
        else:
            index = (index + offset) % M // NEW

        // we have tried all possible indices and we are now going in a circle
        if index == H(k):
            throw an exception OR enlarge table
```

STOP and Think: Is **Double Hashing** an **open addressing** collision resolution strategy?

Step 12

Another collision resolution strategy that solves **Linear Probing's** clumping problem is called **Random Hashing**. The idea behind **Random Hashing** is that we use a pseudorandom number generator **seeded by the key** to produce a **sequence** of hash values. Once an individual hash value is returned, the algorithm just mods it by the capacity of the **Hash Table**, *M*, to produce a valid index that the key can map to. If there is a collision, the algorithm just chooses the next hash value in the pseudorandomly-produced sequence of hash values.

Below is pseudocode for implementing **Random Hashing** when inserting a key `k` into a **Hash Table** of capacity `M` with a backing array called `arr`:

```
insert_RandomHash(k): // Insert k using Random Hashing as the collision resolution strategy

    RNG = new Pseudorandom Number Generator seeded with k
    nextNumber = next pseudorandom number generated by RNG
    index = nextNumber % M

    Loop infinitely:

        // check for duplicate insertions (not allowed)
        if arr[index] == k:
            return

        // check if the slot of the array is empty (i.e., it is safe to insert)
        else if arr[index] == NULL:
            arr[index] = k
            return

        // there is a collision, so re-calculate index
        else:
            nextNumber = next pseudorandom number generated by RNG
            index = nextNumber % M

        // we have tried all possible indices and we are now going in a circle
        if all M locations have been probed:
            throw an exception OR enlarge arr and rehash all existing elements
```

An important nuance of **Random Hashing** is that we must seed the pseudorandom number generator by the key. Why? Because we need to make sure that our **hash function** is deterministic (i.e., that it will *always* produce the same hash value sequence for the same key). Therefore, the only way we can do that is to guarantee to always use the same seed: the key we are currently hashing.

In practice, **Random Hashing** is considered to work just as well as **Double Hashing**. However, *very good* pseudo random generation can be an inefficient procedure (i.e., polynomial time) and as a result, it is more common to just stick with **Double Hashing**.

Step 13

By merely introducing a second offset **hash function**, **Double Hashing** turns out to outperform **Linear Probing** in practice because we no longer face the inevitable fast deterioration of performance resulting from clumps of keys dominating the **Hash Table**. By using a deterministic second **hash function** that depends upon the key being inserted, we are able to get a more uniform distribution of the keys while still having the ability to figure out where we expect to find the key. The same applies for **Random Hashing**. However, the remove algorithm for **Double Hashing** and for **Random Hashing** still turn out to be as wasteful as for **Linear Probing**, so we still have to worry about reinserting elements into the **Hash Table** periodically to clean up the "delete" flags. Also, in all the open-addressing methods discussed so far, the probability of *future* collisions increase each time an inserting key faces a collision.

Consequently, in the next lesson, we will explore an alternative collision resolution strategy called **Separate Chaining**, which vastly simplifies the process of deleting elements, and more importantly, doesn't necessary require the probability of *future* collisions to increase each time an inserting key faces a collision in our **Hash Table**.