Hash Tables (Personal Notes-Muhammad Ahmad Saif)

Difficulty	Challenging
Materials	https://www.hackerearth.com/practice/data-structures/hash-tables/basics-of-hash-tables/tutorial/
Completed	☐
Property	

Mu-Ahmad - Overview

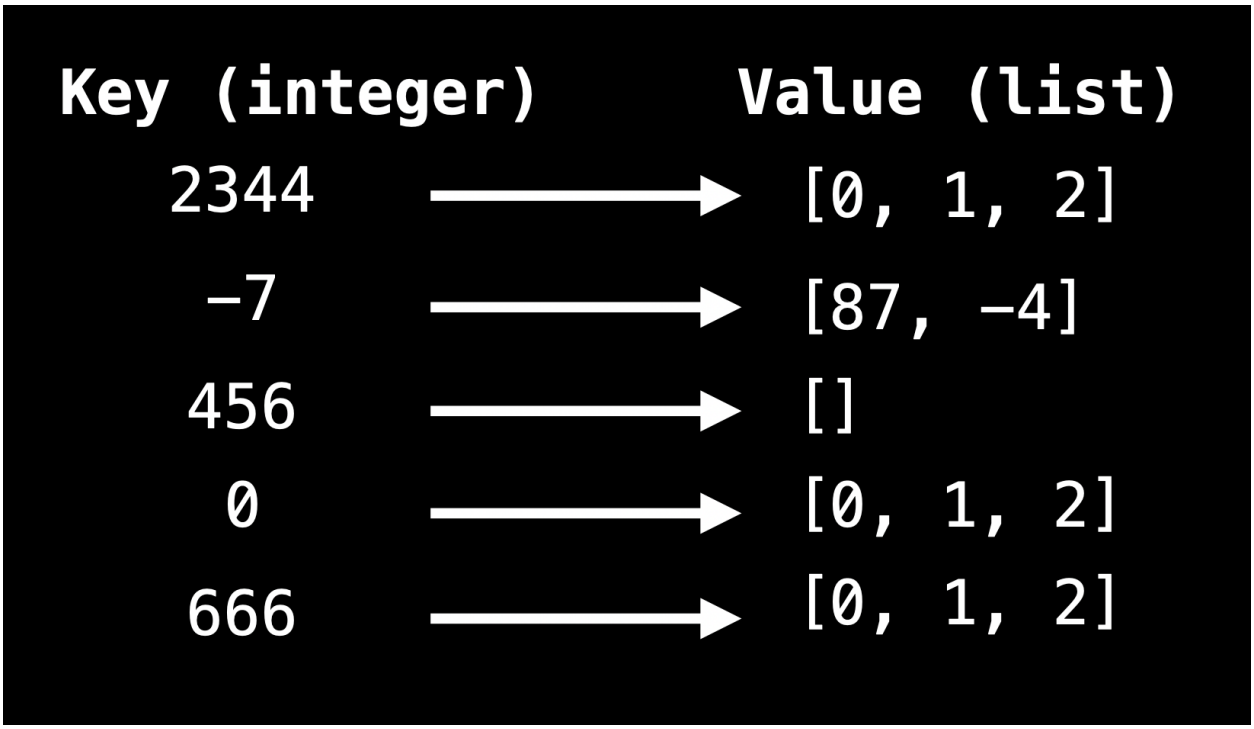
You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or window. Reload to refresh your session. Reload to refresh your session.

https://github.com/Mu-Ahmad

You can find implementation of this and other datastructures on my github.

Hash Table

> A Hash table (HT) is a data structure that provides a mapping from keys to values using a technique called hashing. We refer to these as key-value pairs. Keys must be unique, but values can be repeated. The key-value pairs you can place in a HT can be of any type not just strings and numbers, but also objects! However, the keys needs to be hashable (more on that later).



Hash Table/Map/Dictionary can be thought of as a general form of array where are key is primarily integer that we used to index. In hash-map we can use any object i.e., string, character, custom objects etc as index to store data.

⚠️ To be able to understand how a mapping is constructed between key-value pairs we first need to talk about hash functions.

Hash Function

A hash function H(x) is a function that maps a key ‘x’ to a whole number in a fixed range.

For example, H(x) = (x2 - 6x + 9) mod 10 maps all integer keys to the range [0,9]

H(4) = (16-24+9)mod10 = 1

H(-7) = (49+42+9)mod10 = 0

H(0) = (0- 0+9)mod10 = 9

H(2) = (4-12+9)mod10 = 1

H(8) = (64-48+9)mod10 = 5

> **We can also define hash functions for arbitrary objects such as strings, lists, tuples, multi data objects, etc...**

For a string s let H(s) be a hash function defined below where ASCII(x) returns the ASCII value of the character x.

```
def HashString(string):
  index = 0
  for character in string:
    index += ASCII(character)
  return index % 50
```

HashString(“”) = (0) mod 50 = 0
HashString(“ABC”) = (65 + 66 + 67) mod 50 = 48

HashString(“Z”) = (90) mod 50 = 40

Properties Of A Hash Function

1. If H(x) = H(y) then objects x and y might be equal, but if H(x) ≠ H(y) then x and y are certainly not equal. It must run in constant time.
2. A hash function H(x) must be deterministic.

3. It must satisfy SUHA (simple uniform hashing assumption).

⚠️ A hash collision is when two objects x, y hash to the same value (i.e. H(x) = H(y)).

💡 Q: What makes a key of type T hashable ?

Since we are going to use hash functions in the implementation of our hash table we need our hash functions to be deterministic. To enforce this behaviour, we demand that the keys used in our hash table are immutable data types. Hence, if a key of type T is immutable, and we have a hash function H(k) defined for all keys k of type T then we say a key of type T is hashable.

How does a hash table work?

- **Ideally we would like to have a very fast insertion, lookup and removal time for the data we are placing within our hash table.**

  Remarkably, we can achieve all this in O(1)* time using a hash function as a way to index into a hash table.

  💡 The constant time behaviour attributed to hash tables is only true if you have a good
  uniform hash function.

Q. What do we do if there is a hash collision?

A. We use one of many hash collision resolution techniques to handle this, the two most popular ones are separate chaining and open addressing.

▼ Separete Chaining

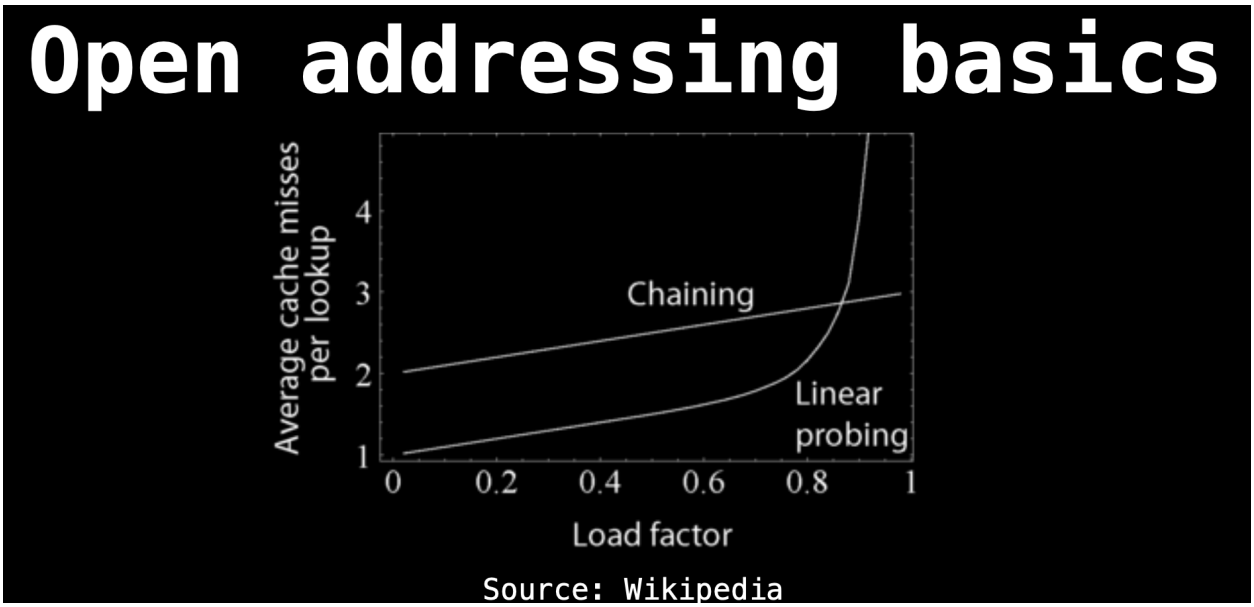
Separate chaining deals with hash collisions by maintaining a data structure (usually a linked list) to hold all the different values which hashed to a particular value.

▼ Open Addressing

Open addressing deals with hash collisions by finding another place within the hash table for the object to go by offsetting it from the position to which it hashed to.

We define a term here:

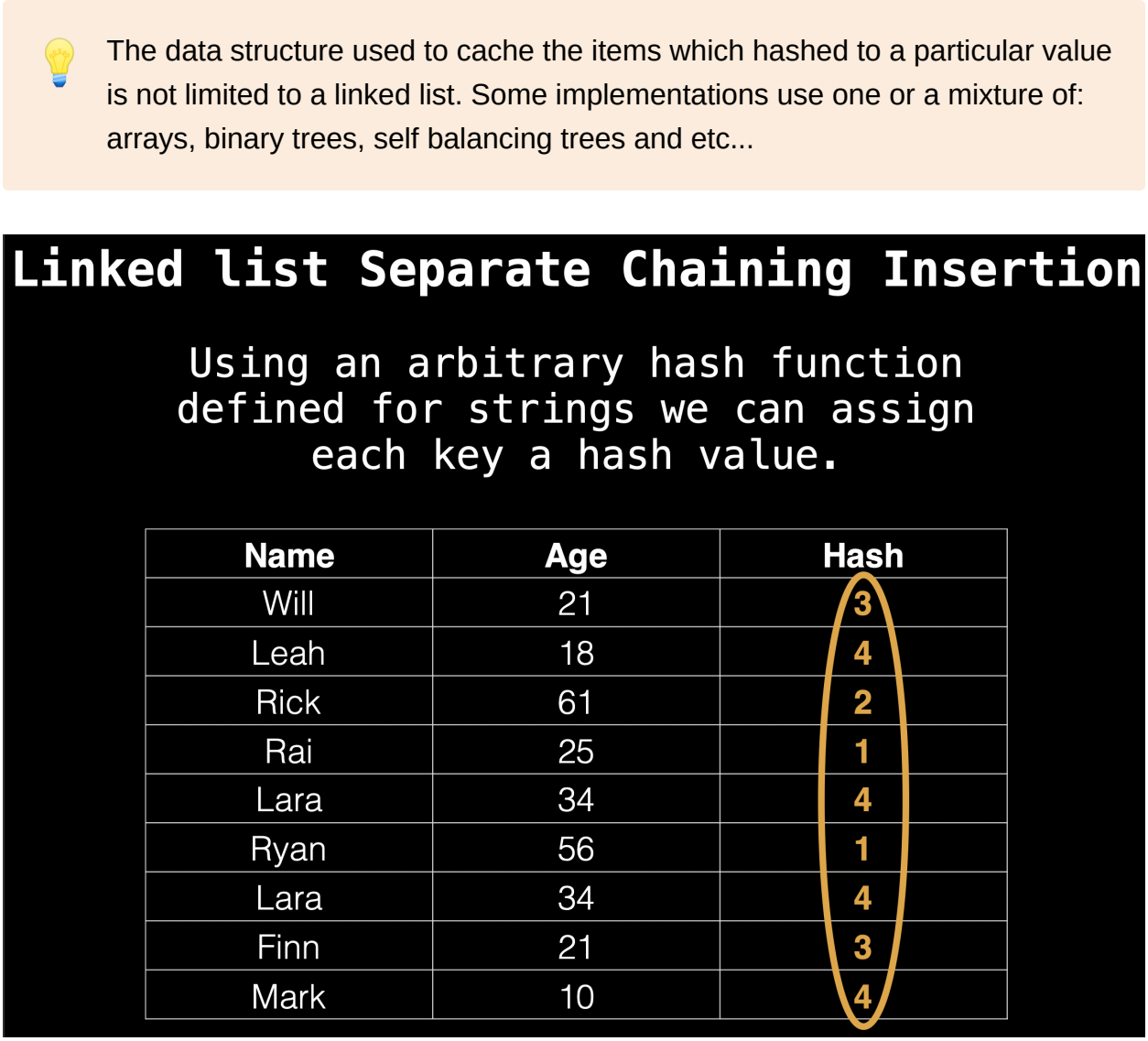
Load Factor = Items in table / size of table

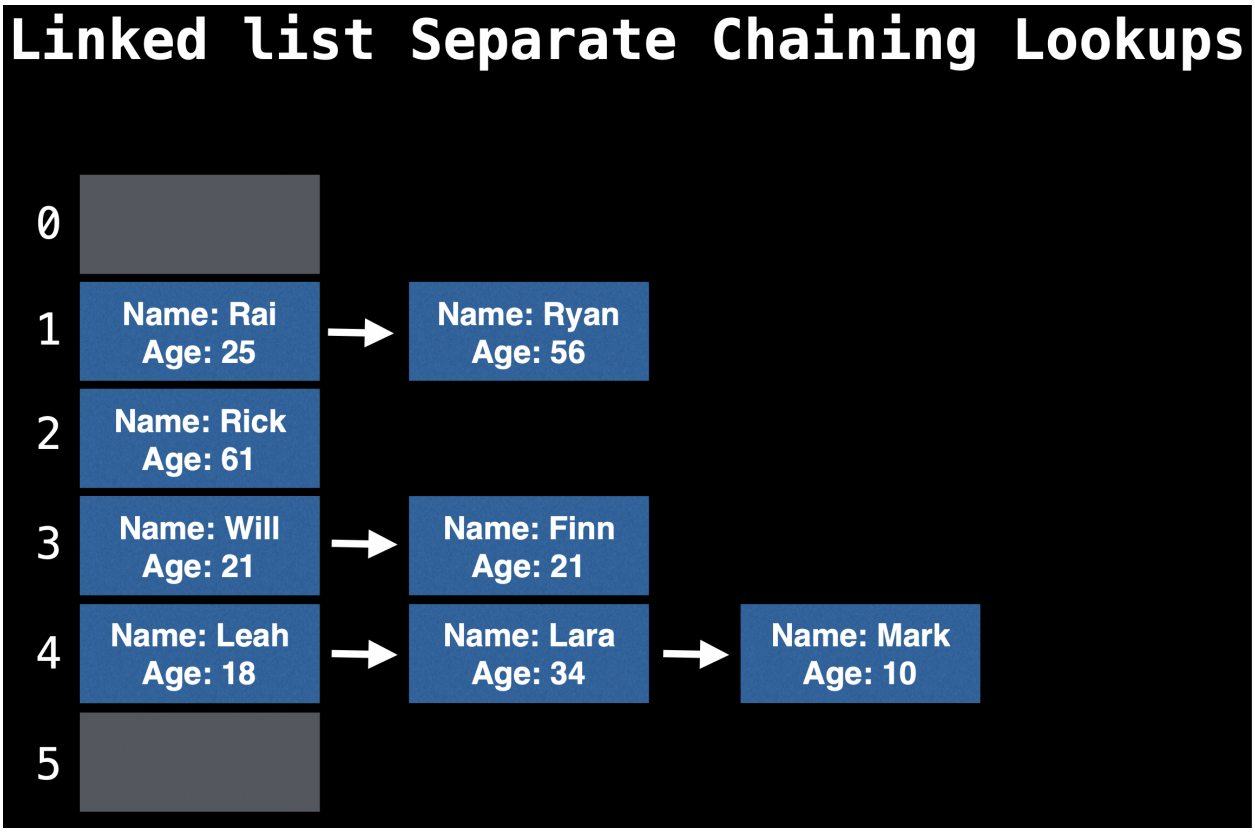


The O(1) constant time behaviour attributed to hash tables assumes the load factor (α) is kept below a certain fixed value. This means once α > threshold we need to grow the table size (ideally exponentially, e.g. double).

Separate Chaining

Separate chaining is one of many strategies to deal with hash collisions by maintaining a data structure (usually a linked list) to hold all the different values which hashed to a particular value.


Linked list Separate Chaining Insertion
Using an arbitrary hash function defined for strings we can assign each key a hash value.
Name
Age
Hash
Will
21
3
Leah
18
4
Rick
61
2
Rai
25
1
Lara
34
4
Ryan
56
1
Lara
34
4
Finn
21
3
Mark
10
4



Q. How do we maintain O(1) insertion and lookup time complexity once the HT gets really full and we have long linked list chains?

A: **Once the HT contains a lot of elements we create a new HT with a larger capacity and rehash all the items inside the old HT and disperse them throughout the new HT at different locations.**

Code Implementation

Open Addressing

When using open addressing as a collision resolution technique the key-value pairs are stored in the table itself as opposed to a data structure like in separate chaining.

This means we need to care a great deal about the size of our hash table and how many elements are currently in the table.

▼ Main Idea

When we want to insert a key-value pair (k,v) into the hash table we hash the key and

obtain an original position for where this key-value pair belongs, i.e **H**(k).

If the position our key hashed to is occupied, try another position in the hash table by
offsetting the current position subject to a **probing sequence P**(x). Keep doing this until
an unoccupied slot is found.

There are an infinite amount of probing sequences you can come up with, here are a few:

Linear probing:

P(x) = ax + b where a, b are constants

Quadratic probing:

P(x)= ax^2 + bx + c, where a,b,c are constants

Double Hashing:

P(k,x) = x*H2(k), where H2(k) is a secondary hash function

Pseudo random number generator:

P(k,x) = x*RNG(H(k),x), where RNG is a random number generator function seeded with H(k).

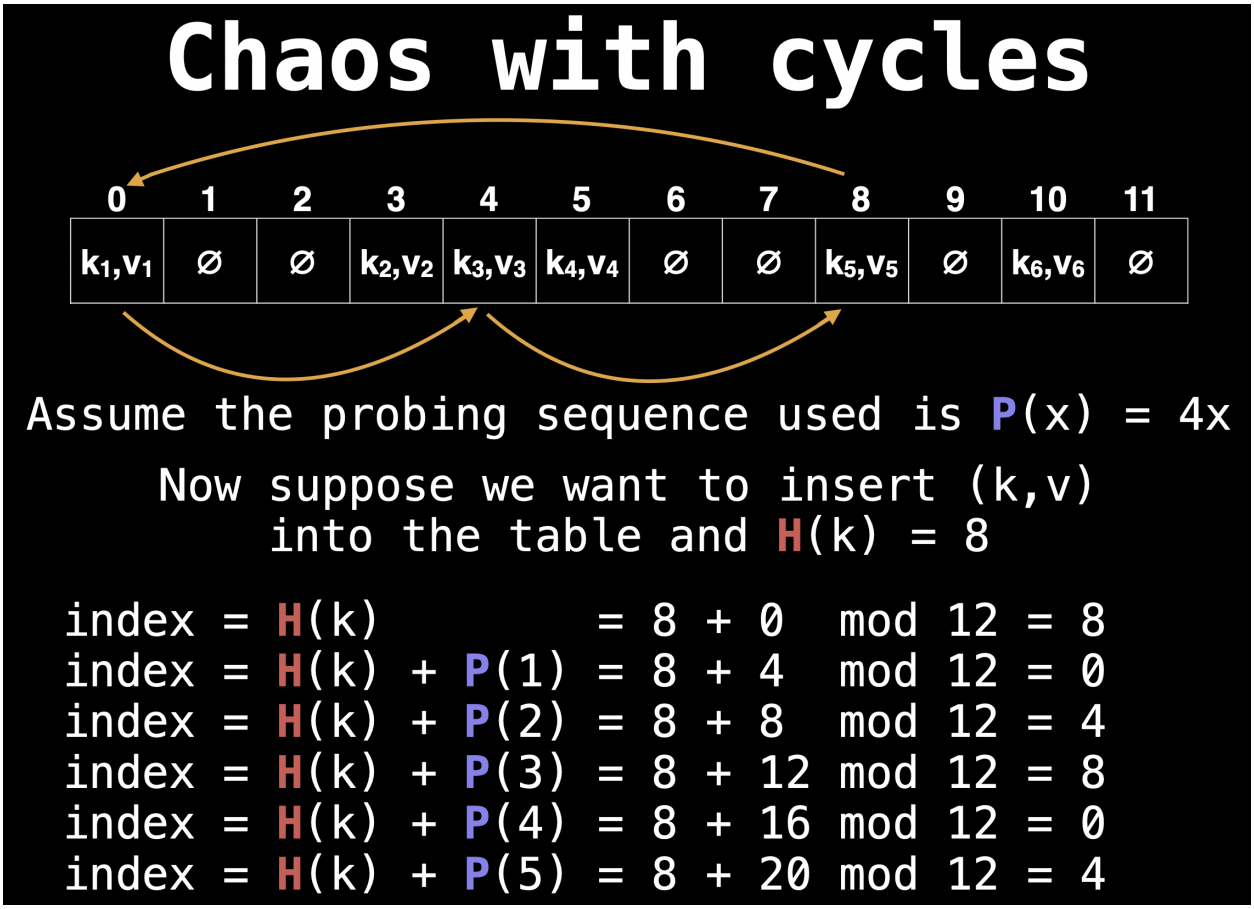
General insertion method for open addressing on a table of size N goes as follows:

```
x = 1
keyHash = H(k) % n
index = keyHash
while hash_table[index] != Null:
  index = (keyHash + P(x)) % n
  x += 1
#insert at index
hash_table[index] = (k, value)
```

Problem with cycles:

Most randomly selected probing sequences modulo N will produce a cycle shorter than the table size. This becomes problematic when you are trying to insert a key-value pair

and all the buckets on the cycle are occupied because you will get stuck in an infinite loop!



Q. So that's concerning... how do we handle probing functions which produce cycles shorter than the table size?

A. In general the consensus is that we don't handle this issue. Instead we avoid it altogether by restricting our domain of probing functions to those which produce a cycle of exactly length N.

> 💡 There are a few exceptions with special properties that can produce shorter cycles.

> Techniques such as linear probing, quadratic probing and double hashing are all subject to the issue of causing cycles which is why the probing functions used with these methods are very specific.

Linear Probing

(An in depth look at linear probing)

LP is a **probing method** which probes according to a linear formula, specifically: **P**(x) = ax + b where a(≠0), b are constants. However, as we previously saw not all linear functions are viable because they are unable to produce a cycle of order N. We will need some way to handle this.

If our linear function is: **P**(x) = 3x, **H**(k) = 4, and table size is nine (N = 9) we end up with the following cycle occurring:

H(k)+**P**(0) mod N = 4
H(k)+**P**(1) mod N = 7
H(k)+**P**(2) mod N = 1
H(k)+**P**(3) mod N = 4
H(k)+**P**(4) mod N = 7
H(k)+**P**(5) mod N = 1
H(k)+**P**(6) mod N = 4
H(k)+**P**(7) mod N = 7
H(k)+**P**(8) mod N = 1
...

The cycle {4,7,1} makes it impossible to reach buckets {0,2,3,5,6,8}!

This would cause an **infinite loop** in our hash table if all the bucket 4, 7 and 1 were already occupied!

Q. Which value(s) of the constant a in P(x) = ax produce a full cycle modulo N?

A. This happens when a and N are relatively prime. Two numbers are relatively prime if their Greatest Common Denominator (GCD) is equal to one. Hence, when GCD(a,N) =

1 the probing function P(x) be able to generate a complete cycle and we will always be able to find an empty bucket!

> ⚠️ A common choice for P(x) is P(x) = 1x since GCD(N,1) = 1 no matter the choice of N (table size)

When we reach the limit, we increase size of the hash table and rehash key value pair in it.

Quadratic Probing (QB)

QP is a **probing method** which probes according to a quadratic formula, specifically:

> **P**(x) = ax2 + bx + c where a,b,c are constants
> and a ≠ 0 (otherwise we have linear probing)

> ⚠️ However, as we previously saw not all quadratic functions are viable because they are unable to produce a cycle of order N. We will need some way to handle this.

Q. So how do we pick a probing function we can work with?

A. There are numerous ways, but three of the most popular approaches are:

1) Let **P**(x) = x2, keep the table size a prime number > 3 and also keep α ≤ 1⁄2

2) Let **P**(x) = (x2 + x)/2 and keep the table size a power of two

3) Let **P**(x) = (-1x)*x2 and keep the table size a prime N where N ≡ 3 mod 4

> That almost wraps it up but for the part where we delete from the open addressing.