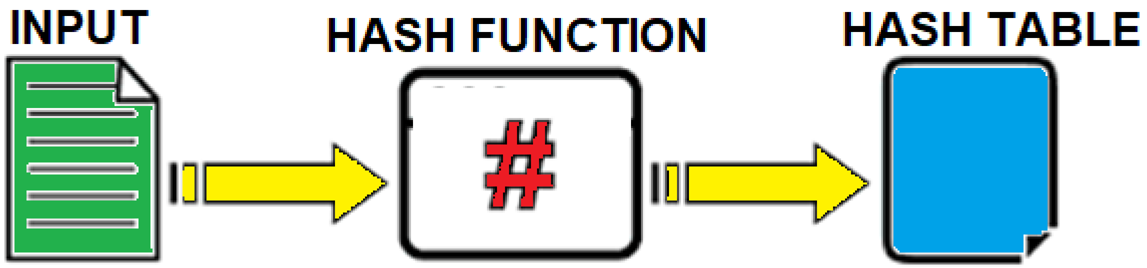
Hashing

Hashing

Hashing is a technique that can perform *insertions, deletions, and Search* in **Hash Table** in *O(1) average time.*



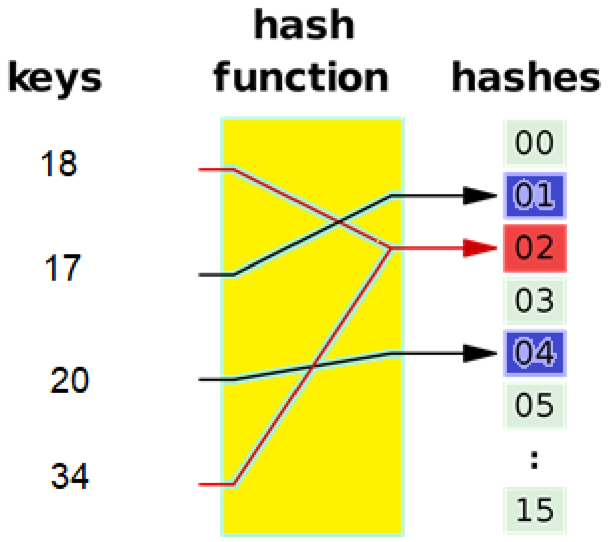
The goal is O(1) time

Use a function of the key value to identify its location in the list.

The function of the key value is called a hash function.

Hashing

- **Whats the Catch ?**
  - Operations that require any ordering information among the elements are not supported efficiently.
    - Thus, operations such as ***findMin, findMax, and the printing of the entire table in sorted order in linear time*** are not supported.



Hash Functions

A good hash function should

Minimize collisions.

Be *easy* and *quick* to compute.

Distribute key values *evenly* in the hash table.

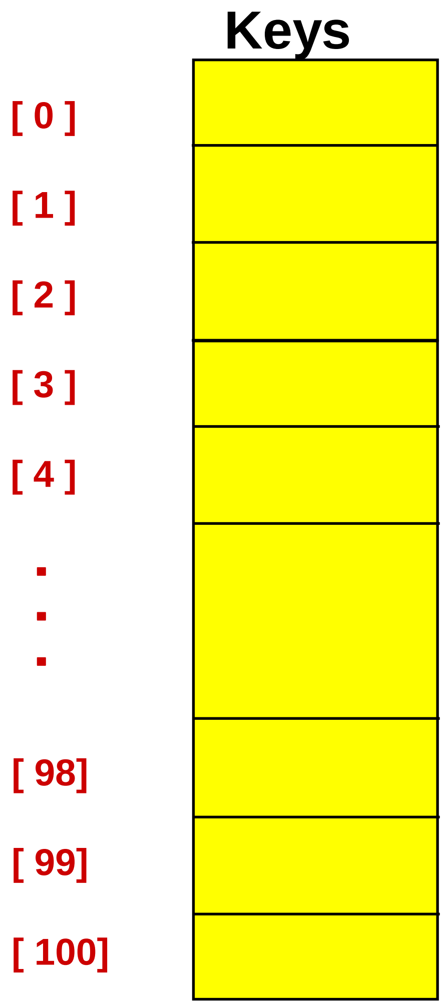
Use *all the information* provided in the key.

Hash Function

- When the input keys are random integers, then ***Key*** **mod** ***TableSize*** **function is not only very simple to compute but also distributes the keys evenly.**

- It is usually a good idea to ensure that the table size is prime.

Using a hash function



Let suppose we can have at most 100 students in a class & Roll number is a 3 digit number

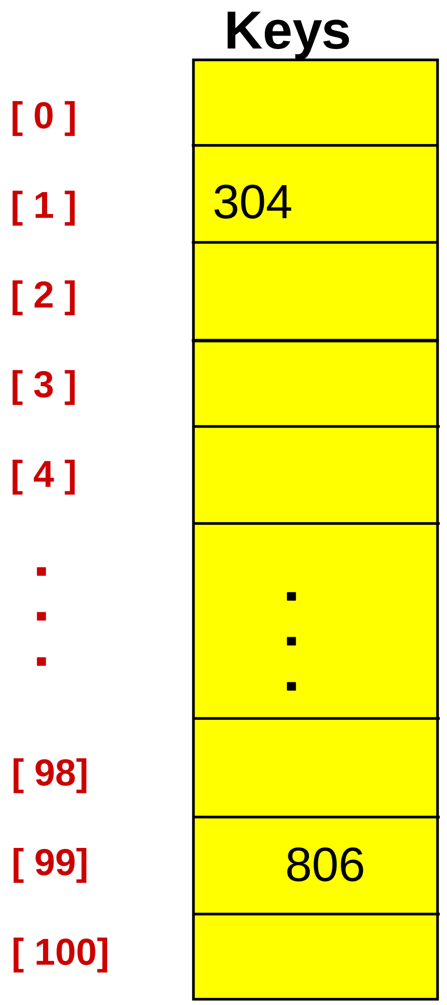
We look for first prime after 100

We create an hashtable (array of size 101)

This hash function can be used to store and retrieve parts in an array.

Hash(key) = RollNo % 101

Using a hash function



Input Key is 304, 806

304 % 101 = 1

806 % 101 = 99

404 %101 = 1

This hash function can be used to store and retrieve parts in an array.

Hash(key) = partNum % 101

Collision

If, when an element is inserted, it hashes to the same value as an already inserted element, then we have a ***collision*** and need to resolve it.

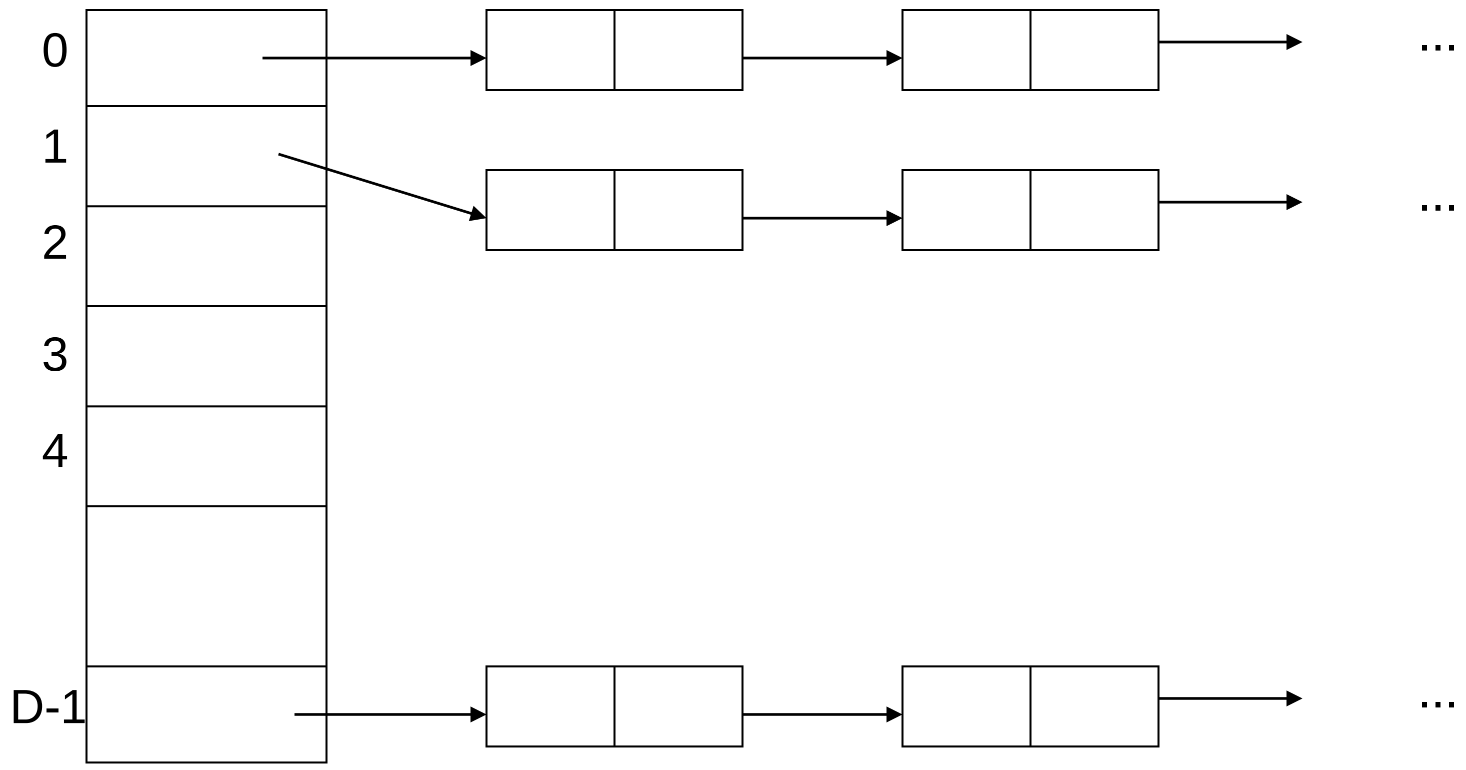
There are several methods for dealing with this.

We will discuss two of the simplest:

- **separate chaining**
- **open addressing**.

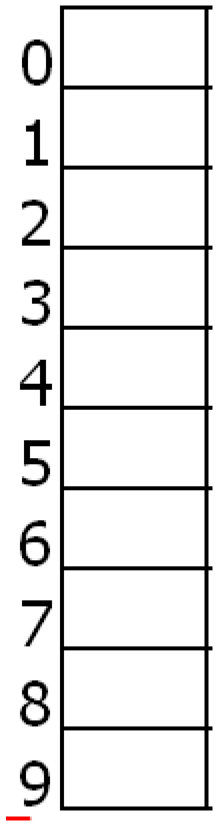
Chaining

- A linked list of elements that share the same hash location



Separate Chaining

The first strategy, separate chaining, is to keep a list of all elements that hash to the same value.



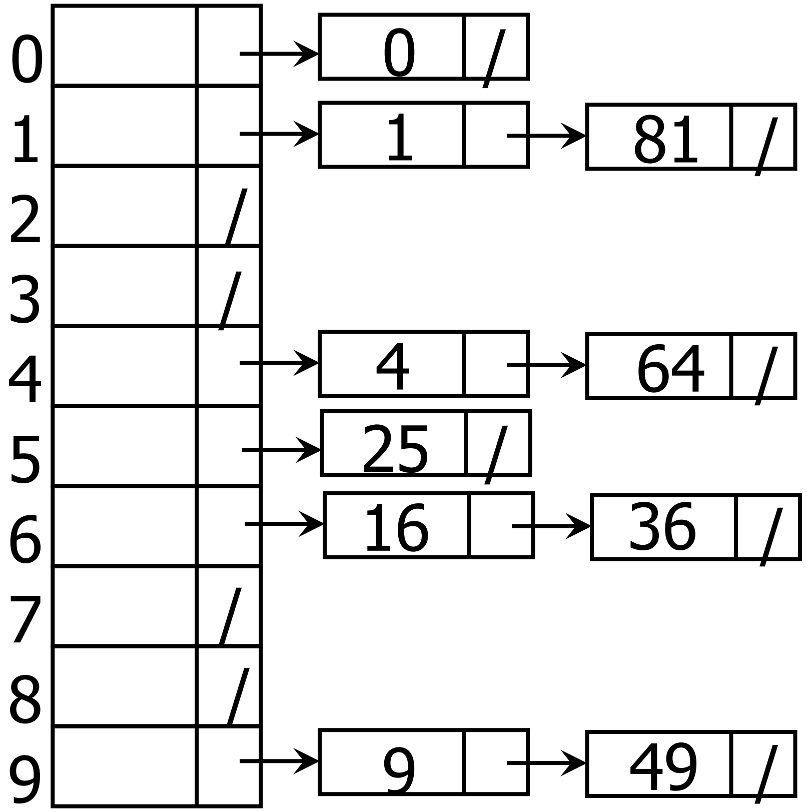
We assume that the keys are the first 10 perfect squares, namely 0, 1, 4, 9, 16, 25, 36, 49, 64, 81.

The hash function is simply
Hash(X)=X mod 10

(The table size is not prime but is used here for simplicity).

Separate Chaining

The first strategy, separate chaining, is to keep a list of all elements that hash to the same value.



The effort required to perform a search is

the constant time required to evaluate the hash function

plus

the time to traverse the list.

Separate Chaining

Any scheme could be used besides linked lists to resolve the collisions;

- a binary search tree or
- even another hash table would work,

But we expect that if the table is large and the hash function is good,

- all the lists should be short,
- so basic separate chaining makes no attempt to try anything complicated

Open Addressing

- In an open addressing hashing system, if a collision occurs, alternative cells are tried until an empty cell is found.

- More formally, cells $h_0(X)$, $h_1(X)$, $h_2(X)$, … are tried in succession, where

  $h_i(X)=(Hash(X)+F(i))$ mod TableSize, where F(i)=i.

- **The function, *F*, is the collision resolution strategy.**

Linear Probing

$h_i(X)=(Hash(X)+F(i)) \bmod TableSize$, where $F(i)=i$.

- Example: Keys are {89, 18, 49, 58, 69}.

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Linear Probing

$h_i(X)=(Hash(X)+F(i)) \bmod TableSize$, where $F(i)=i$.

- Example: Keys are {89, 18, 49, 58, 69}.

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

Linear Probing

$h_i(X)=(Hash(X)+F(i)) \bmod TableSize$, where $F(i)=i$.

- Example: Keys are {89, 18, 49, 58, 69}.

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Linear Probing

$h_i(X) = (Hash(X) + F(i)) \bmod TableSize$, where $F(i) = i$.

- Example: Keys are {89, 18, 49, 58, 69}.

0	49
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Linear Probing

$h_i(X)=(Hash(X)+F(i)) \bmod TableSize$, where $F(i)=i$.

- Example: Keys are {89, 18, 49, 58, 69}.

0	49
1	58
2	
3	
4	
5	
6	
7	
8	18
9	89

Linear Probing

$h_i(X)=(Hash(X)+F(i)) \bmod TableSize$, where F(i)=i.

- Example: Keys are {89, 18, 49, 58, 69}.

0	49
1	58
2	69
3	
4	
5	
6	
7	
8	18
9	89

Linear Probing

- $h_i(X) = (Hash(X) + F(i)) \bmod TableSize$

- In linear probing, *F* is a linear function of *i*, typically $F(i)=i$.

- This amounts to trying cells sequentially (with wraparound) in search of an empty cell.

In open addressing it is required that Hash table is half filled.

Linear Probing

- As long as the table is big enough, a free cell can always be found, but the time to do so can get quite large.

0	49
1	58
2	69
3	
4	
5	
6	
7	
8	18
9	89

Linear Probing

- Worse, even if the table is relatively empty, blocks of occupied cells start forming.

- This effect, known as **primary clustering**,
  - means that any key that hashes into the cluster will require several attempts to resolve the collision, and then it will add to the cluster.

0	49
1	58
2	69
3	
4	
5	
6	
7	
8	18
9	89

Linear Probing

- The empty cells following clusters have a much greater chance to be filled than other positions.
- This probability is equal to (*sizeof*(*cluster*) + 1)/*TSize.* Other empty cells have only 1/*TSize* chance of being filled.

0	49
1	58
2	69
3	
4	
5	
6	
7	
8	18
9	89

Linear Probing

- If a cluster is created, it has a tendency to grow, and the larger a cluster becomes, the larger the likelihood that it will become even larger.

- This fact undermines the performance of the hash table for storing and retrieving data.

- The problem at hand is how to avoid cluster buildup. An answer can be found in a more careful choice of the probing function *p*.

0	49
1	58
2	69
3	
4	
5	
6	
7	
8	18
9	89

Quadratic Probing

- Quadratic probing is a collision resolution method that eliminates the primary clustering problem of linear probing.
- Quadratic probing the collision function is quadratic.
- The popular choice is **$F(i)=i^2$.**

$$h_i(X)=(\mathrm{Hash}(X)+F(i)) \bmod \mathrm{TableSize}$$

Quadratic Probing

- Example: Keys are {89, 18, 49, 58, 69}.

$h_i(X)=(Hash(X)+F(i)) \bmod TableSize$

	Keys
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

Quadratic Probing

- Example: Keys are {89, 18, 49, 58, 69}.

$h_i(X)=(Hash(X)+F(i)) \bmod TableSize$

	Keys
0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Quadratic Probing

- Example: Keys are {89, 18, 49, 58, 69}.

$h_i(X)$=(Hash(X)+F(i)) mod TableSize

	Keys
0	49
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Quadratic Probing

- Example: Keys are {89, 18, 49, 58, 69}.

$h_i(X)=(Hash(X)+F(i)) \bmod TableSize$

	Keys
0	49
1	
2	58
3	
4	
5	
6	
7	
8	18
9	89

Quadratic Probing

- Example: Keys are {89, 18, 49, 58, 69}.

$h_i(X)=(Hash(X)+F(i)) \bmod TableSize$

	Keys
0	49
1	
2	58
3	69
4	
5	
6	
7	
8	18
9	89

Quadratic Probing

- For linear probing it is a bad idea to let the hash table get nearly full, because performance degrades.

- For quadratic probing, the situation is even more drastic:

- There is no guarantee of finding an empty cell once the table gets more than half full, or even before the table gets half full if the table size is not prime.

- **This is because at most half of the table can be used as alternative locations to resolve collisions.**

In class Exercise

- **Question:** You task is to hash some keys in a **table of size 11** using the

Hash Function = Key % TableSize.

For collision resolution use the technique of **Linear Probing**.

- **Insert the following keys in the given hash table 16, 66, 99, 11, 88,132**

0	605
1	
2	
3	
4	484
5	643
6	
7	84
8	
9	
10	

In class Exercise

- **Question:** You task is to hash some keys in a **table of size 11** using the
Hash Function = Key % TableSize.

For collision resolution use the technique of **Quadratic Probing**.

- **Insert the following keys in the given hash table 16, 66, 99, 11, 88,132**

0	605
1	
2	
3	
4	484
5	643
6	
7	84
8	
9	
10	

In class Exercise

- **Question:** You task is to hash some keys in a **table of size 11** using the

Hash Function = Key % TableSize.

For collision resolution use the technique of **quadratic probing**. The probing function that you will use is

- ***h*(*K*) - i^2, *h*(*K*) + i^2** **for *i* = 1, 2, . . . , (*TableSize* – 1)/2**

- For first collision of an Input_key K use *h*(*K*) - *1,* for second collision of K use *h*(*K*) + 1, for third use h(K) - 4, for fourth use h(k) + 4 so on….

- **Insert the following keys in the given hash table 16, 66, 99, 11, 88,132**

0	605
1	
2	
3	
4	484
5	643
6	
7	84
8	
9	
10	

In class Exercise

- **Question:** You task is to hash some keys in a **table of size 11** using the **Hash Function = Key % TableSize.**

For collision resolution use the technique of **quadratic probing**. The probing function that you will use is

- $h(K) - i^2, h(K) + i^2$ **for i = 1, 2, . . . , ($TableSize$ – 1)/2**

- For first collision of an Input_key K use $h(K) - 1$, for second collision of K use $h(K) + 1$, for third use h(K) - 4, for fourth use h(k) + 4 so on....

- **Insert the following keys in the given hash table 16, 66, 99, 11, 88,132**

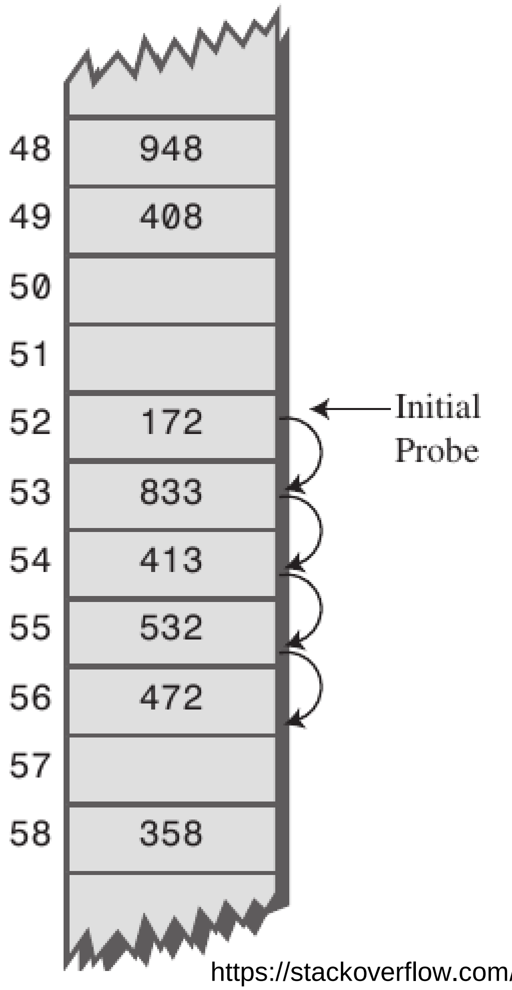
Open Addressing

- In an open addressing hashing system, if a collision occurs, alternative cells are tried until an empty cell is found.
- More formally, cells $h_0(X)$, $h_1(X)$, $h_2(X)$, … are tried in succession, where $h_i(X)=(Hash(X)+F(i)) \bmod TableSize$, with $F(0)=0$.
- There are three common collision resolution strategies, namely
  - Linear Probing,
  - Quadratic Probing, and
  - Double Hashing.

The load factor for a hash table in open addressing is that it should be less than half filled.

Quadratic Probing

Quadratic probing eliminates primary clustering.

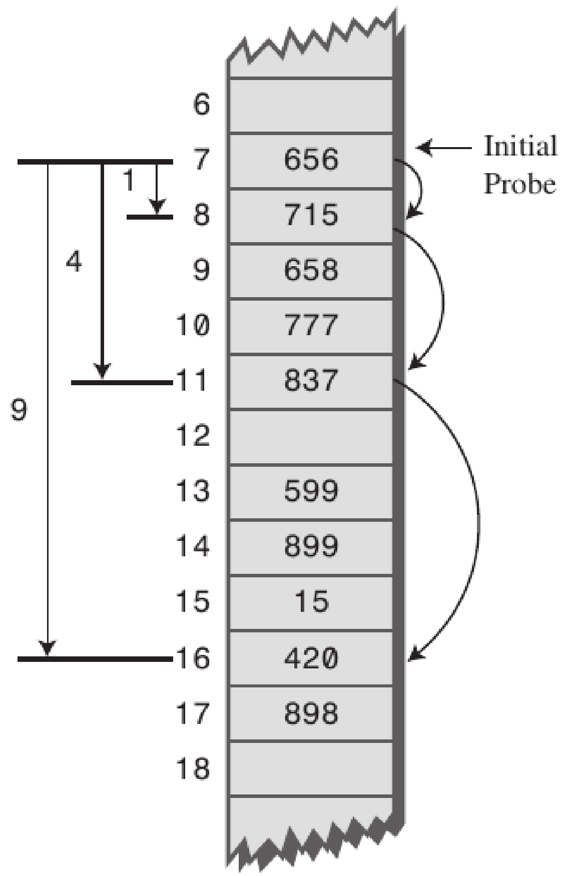


Primary clustering is the tendency for a collision resolution scheme to create long runs of filled slots near the **hash** position of keys.

https://stackoverflow.com/questions/27742285/what-is-primary-and-secondary-clustering-in-hash

Quadratic Probing

- Quadratic probing eliminates primary clustering but causes **secondary clustering**.



Secondary clustering is the tendency for a collision resolution scheme to create long runs of filled slots away from the **hash** position of keys.

If the primary **hash** index is x , probes go to x+1 , x+4 , x+9 , x+16, x+25 and so on,

Issue with quadratic formula

- H(K)+1, H(K)+4, H(K)+9, . . . , H(K)+ $(\text{TSize}-1)^2$
- Covers only half of the table

- The second half of the sequence repeats the first half in the reverse order

- For example, if TSize = 19 and H(K) = 9, then the sequence is
- 9,10,13,18,6,15,7,1,16,14,14,16,1,7,15,6,18,13,10

$$h(K) + i^2,\ h(K) - i^2 \text{ for } i = 1, 2, \ldots, (TSize - 1)/2$$

Including the first attempt to hash *K*, this results in the sequence:

$$h(K),\ h(K) + 1,\ h(K) - 1,\ h(K) + 4,\ h(K) - 4, \ldots, h(K) + (TSize - 1)^2/4,$$

$$h(K) - (TSize - 1)^2/4$$

- For example, if TSize = 19 and H(K) = 9, then the sequence is
- 9,10,8,13,5,18,0,6,12,15,3,11,1,17,16,2,14,4

Quadratic Probing

- Secondary clustering is a slight theoretical blemish.
- Simulation results suggest that it generally causes less than an extra half probe per search.
- Double Hashing eliminates it but at the cost of computing an extra hash function.

Double Hashing

- The last collision resolution method we will examine is Double hashing.
- For double hashing, one popular choice is $F(i)=i*hash_2(X)$
  - This formula says that we apply a second hash function to X and probe at a distance $hash_2(X)$, $2hash_2(X)$, …, and so on.
- A function such as $hash_2(X)=R-(X \bmod R)$, with R a prime smaller than TableSize, will work well. Here, R is set to 7.

Double Hashing

It works on a similar idea to linear and quadratic probing.

Use a big table and hash into it.

Whenever a collision occurs, choose another spot in table

to put the value.

The difference here is that instead of choosing next opening, a second hash function is used to determine the location of the next spot.

Double Hashing

For example, given hash function H1 and H2 and key. do the following:

- Check location $hash_1$(key). If it is empty, put record in it.
- If it is not empty calculate $hash_2$(key).
- Check if $hash_1$(key)+$hash_2$(key) is empty, if it is, put it in
- Repeat with $hash_1$(key)+2$hash_2$(key), $hash_1$(key)+3$hash_2$(key) and so on, until an opening is found.
- Like quadratic probing, you must take care in choosing $hash_2$
  - $hash_2$ CANNOT return 0 it should be such that all cells will be probed eventually.

Double Hashing

Example: Keys are {89, 18, 49, 58, 69}.

	Empty
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash(X)+F(i)) mod TableSize

$F(i)=i*hash_2(X)$

$hash_1(X)=$ X mod TableSize
$hash_2(X)=R-(X \bmod R)$, with R a prime smaller than TableSize,

Double Hashing

Example: Keys are {89, 18, 49, 58, 69}.

	Empty
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

Hash(X)+F(i)) mod TableSize

$F(i)=i*hash_2(X)$

$hash_1(X)= X \bmod TableSize$
$hash_2(X)=R-(X \bmod R)$, with R a prime smaller than TableSize,

Double Hashing

Example: Keys are {89, 18, 49, 58, 69}.

	Empty
0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Hash(X)+F(i)) mod TableSize

F(i)=i*hash$_2$(X)

hash$_1$(X)= X mod TableSize
hash$_2$(X)=R-(X mod R), with R a prime smaller than TableSize,

Double Hashing

Example: Keys are {89, 18, 49, 58, 69}.

	Empty
0	
1	
2	
3	
4	
5	
6	49
7	
8	18
9	89

Hash(X)+F(i)) mod TableSize

$F(i)=i*hash_2(X)$

$hash_1(X)$= X mod TableSize
$hash_2(X)$=R-(X mod R), with R a prime smaller than TableSize,

$hash_1(X)$= 49 mod 10
$hash_2(X)$= 7 - (49 mod 7),
with R =7

Hash(49) = 9 + 7 = 16%10 =6

Double Hashing

Example: Keys are {89, 18, 49, 58, 69}.

	Empty
0	
1	
2	
3	58
4	
5	
6	49
7	
8	18
9	89

Hash(X)+F(i)) mod TableSize

$F(i)=i*hash_2(X)$

$hash_1(X)$= X mod TableSize
$hash_2(X)$=R-(X mod R), with R a prime smaller than TableSize,

$hash_1(X)$= 58 mod 10
$hash_2(X)$= 7 - (58 mod 7),
with R =7

Hash(49) = 8 + 5 = 13%10 = 3

Double Hashing

Example: Keys are {89, 18, 49, 58, 69}.

	Empty
0	69
1	
2	
3	58
4	
5	
6	49
7	
8	18
9	89

Hash(X)+F(i)) mod TableSize

F(i)=i*hash$_2$(X)

hash$_1$(X)= X mod TableSize
hash$_2$(X)=R-(X mod R), with R a prime smaller than TableSize,

hash$_1$(X)= 69 mod 10
hash$_2$(X)= 7 - (69 mod 7),
with R =7

Hash(69) = 9 + 1 = 10%10 = 0

Double Hashing

	Empty	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Hash Function

Hash Functions (cont'd)

- A good hash function should:
  - *Minimize* collisions.
  - Be *easy* and *quick* to compute.
  - Distribute key values *evenly* in the hash table.
  - Use *all the information* provided in the key.

Hash Function

- If hash function transforms different keys into different numbers, it is called a **perfect hash function.**
- To create a perfect hash function, the tableSize has to be at least equal to the number of elements being hashed.
- But the number of elements is not always known ahead of time.

Hash Function

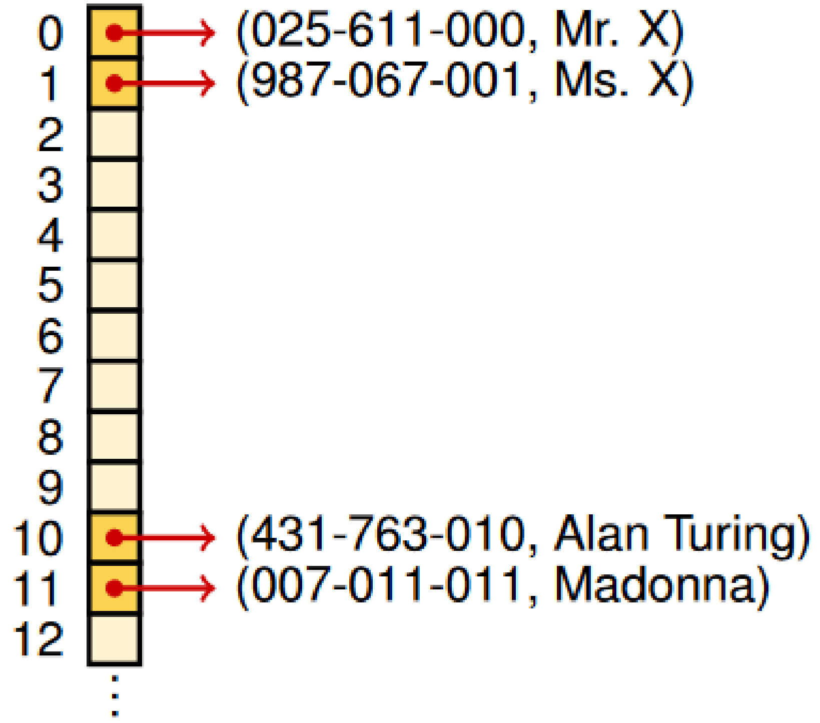
- If the input keys are integers, then ***Key* mod *TableSize*** is generally a reasonable strategy,

- Unless *Key* happens to have some undesirable properties.
  - In this case, the choice of hash function needs to be carefully considered.
  - For instance, if the table size is 10 and the keys all end in zero, then the standard hash function is a bad choice.

Example –Bad Hash

- hash function $h(x) = x$ as number mod 1000

Assume the last digit is always 0 or 1 indicating male/femal.



Hash Function

- To avoid situations like the one above, it is usually a good idea to ensure that the table size is prime.

- **When the input keys are random integers, then *Key* mod *TableSize* function is not only very simple to compute but also distributes the keys evenly.**

Common Hashing Functions

1. **Division Remainder** *(using the table size as the divisor)*

- Computes hash value from key using the % operator.

Common Hashing Functions

2. Truncation or Digit/Character Extraction

- Works based on the distribution of digits or characters in the key.
- More evenly distributed digit positions are extracted and used for hashing purposes.
- For instance, students IDs or ISBN codes may contain common subsequences that can increase the likelihood of collision.

- To map the key 25936715 to a range between 0 and 99
  - Take 4th and 7th digit → 3 and 1. Truncate the rest
  - Place the value at index 31

Common Hashing Functions

3. Folding

- It involves splitting keys into two or more parts and then combining the parts to form the hash addresses.
- To map the key 25936715 to a range between 0 and 9999, we can:
  - split the number into two as 2593 and 6715 and
  - add these two to obtain 9308 as the hash value.
- Very useful if we have keys that are very large.
- Fast and simple especially with bit patterns.
- A great advantage is ability to transform non-integer keys into integer values.

Common Hashing Functions

4. Radix Conversion

- Transforms a key into another number base to obtain the hash value.

- Typically use number base other than base10 and base 2 to calculate the hash addresses.

- To map the key 55354 in the range 0 to 9999 using base 11 we have:

  $55354_{10} = 38652_{11}$
- We may truncate the high-order 3 to yield 8652 as our hash address within 0 to 9999.

Common Hashing Functions

5. Mid-Square

- The key is squared and the middle part of the result is taken as the hash value.

- **To map the key 3121 into a hash table of size 1000**
  - **we square it $3121^2 = 9740641$**
  - **extract 406 as the hash value.**

- Works well if the keys do not contain a lot of leading or trailing zeros.
- Non-integer keys must be preprocessed to obtain corresponding integer values.

Hash Function

- Usually, the keys are strings; in this case, the hash function needs to be chosen carefully.
- One option is to add up the ASCII values of the characters in the string.

```
int Hash(char *Key, int TableSize)
{
    unsigned int HashVal=0;
    while (*Key!='\0')
        HashVal+=*Key++;
    return HashVal%TableSize;
}
```

Hash Function

- **If the table size is large, the function does not distribute the keys well.**

- For instance,
- suppose that TableSize =10007 (10007 is a prime number),
- All keys are eight or fewer characters long.
  - As a *char* has an integer value that is always at most 127,
  - the hash function can only assume values between 0 and 1016, which is 127*8=1016.

- **This is clearly not an equitable distribution!**

Hash Function

```
int Hash(char *Key, int TableSize){
    return (Key[0]+27*Key[1]+729*Key[2]) % TableSize;
}
```

- This hash function assumes that *Key* has at least two characters plus the *NULL* terminator.
- The value 27 represents the number of letters in the English alphabet, plus the blank, and 729 is 27^2.
- This function examines only the first three characters, but if these are random and the table size is 10,007, as before, then we would expect a reasonably equitable distribution.

Hash Function

- Unfortunately, English is not random.
- Although there are 26^3=17,576 possible combinations of three characters (ignoring blanks), a check of a reasonably large on-line dictionary reveals that the number of different combinations is actually only 2,851.
- Even if none of these combinations collide, only 28 percent of the table can actually be hashed to.

Hash Function

```
int Hash(char *Key, int TableSize){
    for( i = 0 to size(key))
       hash +=  (37^i)*Key[i];
    return hash % TableSize;
}
```

- This hash function involves all characters in the key and can generally be expected to distribute well

Hash Function

```
int Hash(char *Key, int TableSize){
    for( i =0 to size(key))
       hash +=  (37^i)*Key[i];
    return hash % TableSize;
}
```

- This hash function involves all characters in the key and can generally be expected to distribute well

$$\boxed{h_k = k_0 + 37k_1 + 37^2k_2}$$

Hash Function

```
/** A hash routine for string objects. */
unsigned int hash(const string & key, int tableSize){
    unsigned int hashVal = 0;
    for (char ch : key)
         hashVal = 37 * hashVal + ch;
    return hashVal % tableSize;
 }
```

- The code computes a polynomial function (of 37) by use of Horner’s rule

$h_k = k_0 + 37k_1 + 37^2k_2$ is by the formula $h_k = ((k_2) * 37 + k_1) * 37 + k_0$.

Hash Function

This hash function is not necessarily the best with respect to table distribution, but it does have the merit of ***extreme simplicity and is reasonably fast.***

If the keys are very long, the hash function will take too long to compute.

A common practice in this case is ***not to use all the characters***.

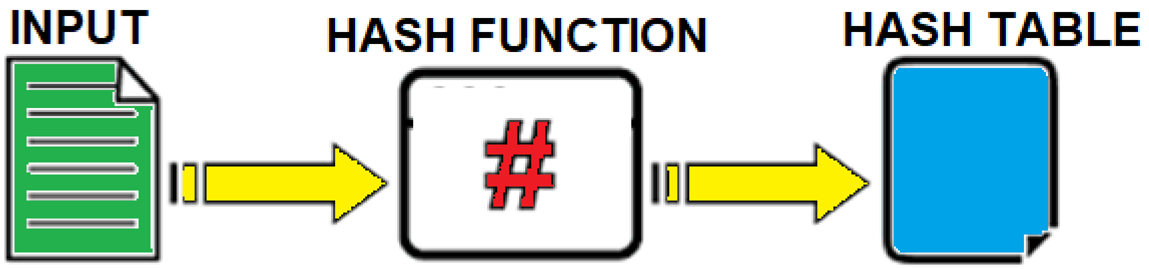
The length and properties of the keys would then influence the choice.

- For instance, the keys could be a complete street address. The hash function might include a couple of characters from the street address and perhaps a couple of characters from the city name and ZIP code.

Universal Hash Function

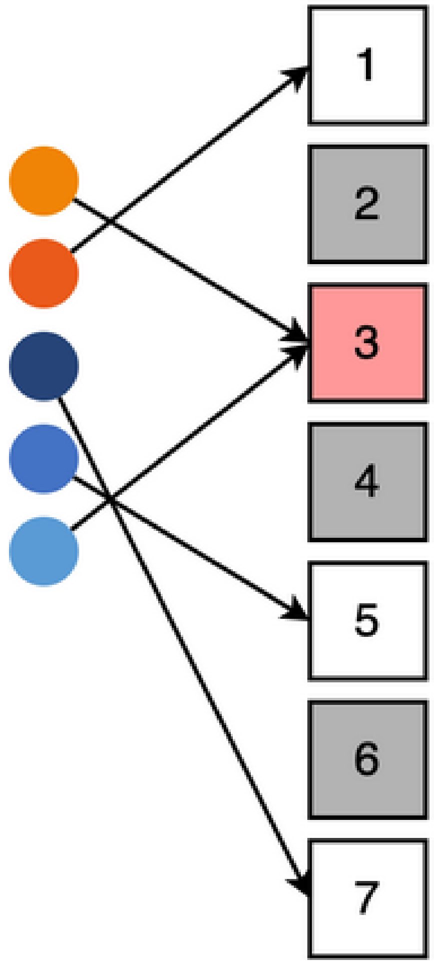
Universal Hash Functions

When very little is known about keys, a *universal class of hash functions* can be used



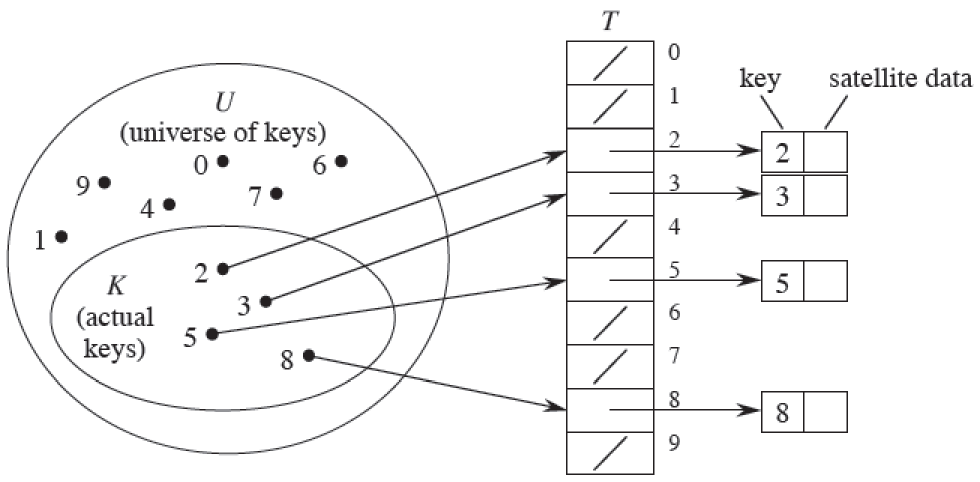
Universal Hash Functions

- A class of functions is universal when for any sample, a randomly chosen member of that class will be expected to distribute the sample evenly,
- and guarantee low probability of collisions



Universal Hash Functions

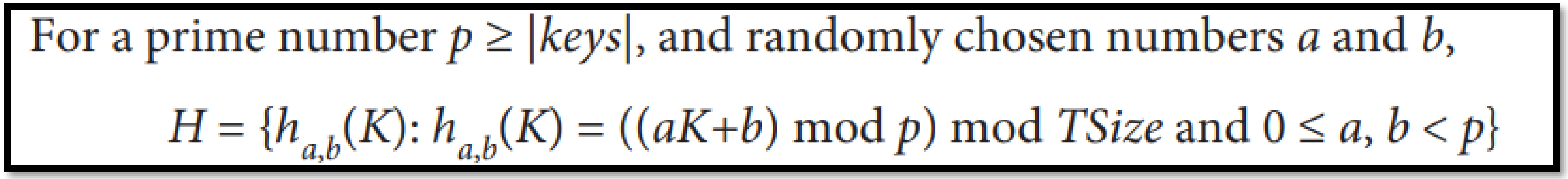
- *H* is universal if no pair of distinct keys are mapped into the same index by a **randomly chosen function *h*** with the probability equal to **1/*TSize*.**



- In other words, there is one chance in *TSize* that two keys collide when a randomly picked hash function is applied.

Universal Hash Functions

- One class of such functions is defined as follows.


For a prime number $p \geq |keys|$, and randomly chosen numbers a and b,

$$H = \{h_{a,b}(K) : h_{a,b}(K) = ((aK+b) \bmod p) \bmod TSize \text{ and } 0 \leq a, b < p\}$$

- We can choose **a and b** randomly and fix it for one hashtable.
- So if everyone is using this class of universal hash function, they can have their own hash function because of **different a and b**

Universal Hash Functions

One class of such functions is defined as follows.

For a prime number $p \geq |keys|$, and randomly chosen numbers a and b,

$$H = \{h_{a,b}(K): h_{a,b}(K) = ((aK+b) \bmod p) \bmod TSize \text{ and } 0 \leq a, b < p\}$$

- However, we can use pseudo-random number generator to generate a and b for each key.
- The pseudo-random number generator take a seed to generate random sequence.
- For a particular seed they generate the same sequences…so for each key their will be a different hash function

Lazy Deletion

- Standard deletion cannot be performed in a probing hash table, because the cell might have caused a collision to go past it.
- For instance, if we remove 89, then virtually all the remaining find operations will fail.
- Thus, probing hash tables require lazy deletion, although in this case there really is no laziness implied.

Applications of Hashing

- Compilers use hash tables to keep track of declared variables
- A hash table can be used for on-line spelling checkers — if misspelling detection (rather than correction) is important, an entire dictionary can be hashed and words checked in constant time
- Game playing programs use hash tables to store seen positions, thereby saving computation time if the position is encountered again
- In DB they are used in creating lock tables …
- Hash functions can be used to quickly check for inequality — if two elements hash to different values they must be different
- Storing sparse data