Hash Functions

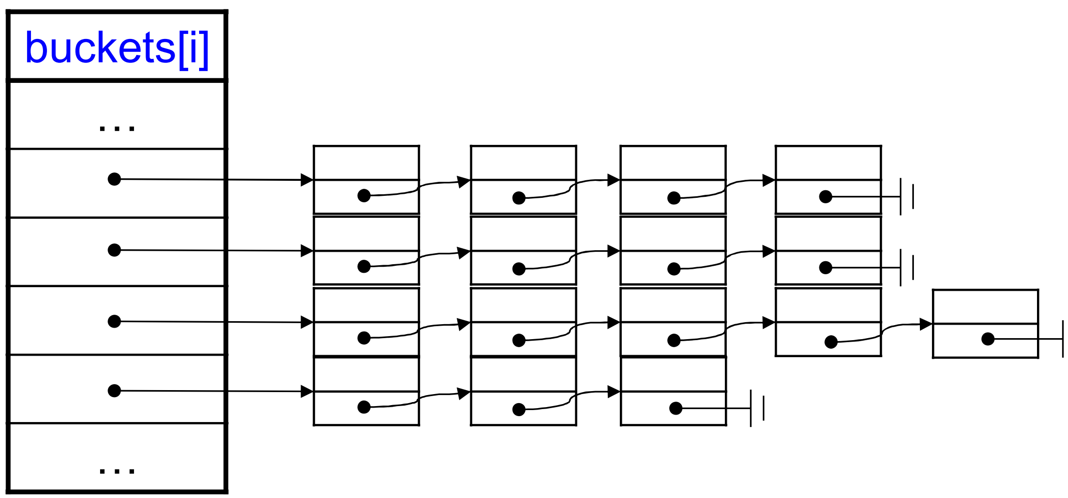
- The performance of a hashtable depends significantly on how often keys collide.
  - More collisions ➔ poorer efficiency.

- Poorly designed hash functions often map similar keys to the same bucket.
  - A correct but *extremely poor* hash function:

```
int Hash(char *s, int nBuckets) {
   return 0;
}
```

Hash Functions

- A good hash function should
  - reduce the number of collisions;
  - evenly distributes the keys into buckets;
  - quick to compute.



Deciding the Number of Buckets

- The ***load factor*** λ is the average length of each bucket chain.

$$\lambda = \frac{N_{entries}}{N_{buckets}}$$

- For good performance, the value of λ should remain relatively small.
  - λ too large (too few buckets) → collision inevitable.
  - λ too small (too many buckets) → waste memory.
- $N_{buckets}$ is usually *prime* to improve performance. (Again, too advanced to explain.)

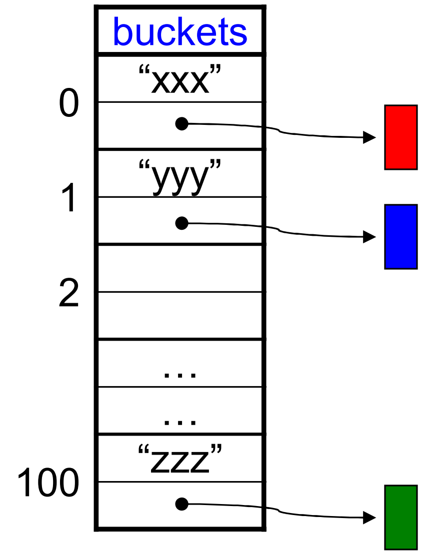
Resolving Collisions

- Besides chaining, other methods exist to resolve collisions.
  - Linear Probing
  - Quadratic Probing
  - Double Hashing
  - Rehashing
  - …

Open Addressing

- In ***open addressing***, an entry is stored within the table.

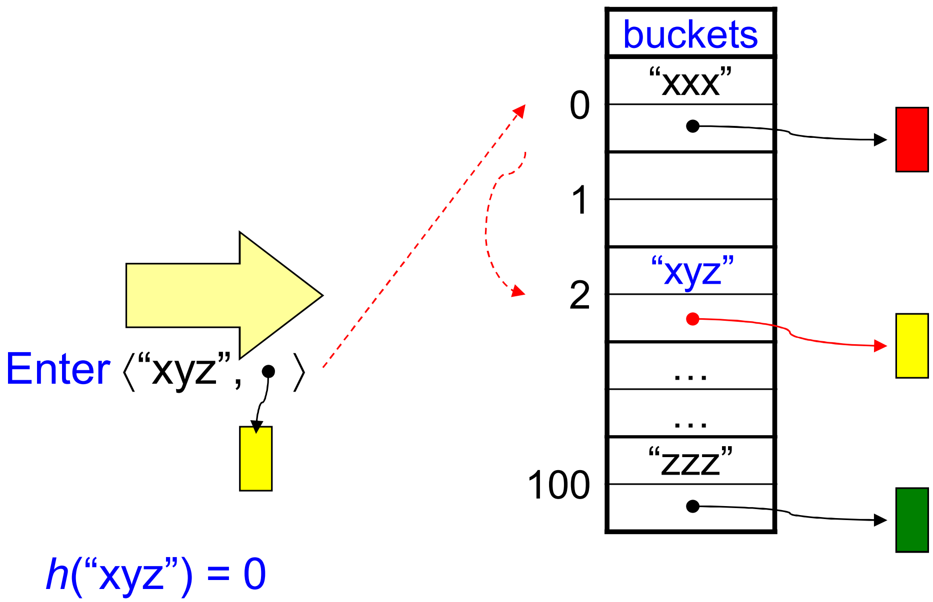
```
typedef struct {
   char *key;
   void *value;
} entryT;

struct hashtableCDT {
   entryT buckets[101];
};
```



- λ is *always* < 1.0.

Open Addressing

- When collision occurs, we simply ***probe*** another bucket in the hashtable.



Linear Probing

- Let $h_0 = Hash(key, N_{buckets})$
- If h_0 collides,
  - probe $h_1 = (h_0 + F(1)) \% N_{buckets}$.
- If h_1 collides,
  - probe $h_2 = (h_0 + F(2)) \% N_{buckets}$.
- If h_2 collides,
  - probe $h_3 = (h_0 + F(3)) \% N_{buckets}$.
- …
- $F(i)$ is a ***linear*** function. Typically, $F(i) = i$.

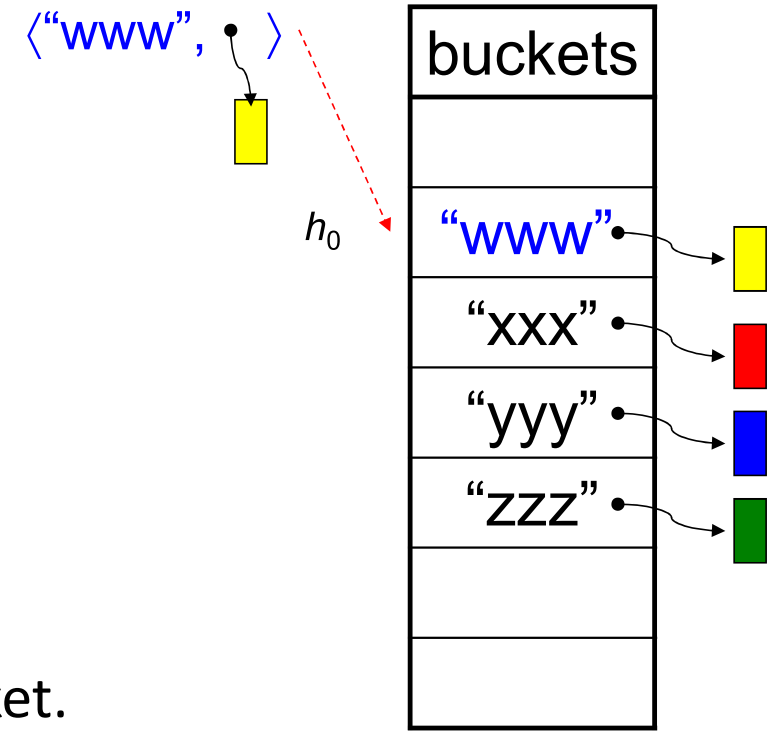
F is the ***collision resolution strategy***.

$[h_0, h_1, h_2, \ldots]$ is called a ***probing sequence***.

Linear Probing: Enter

- Let $F(i) = i$.

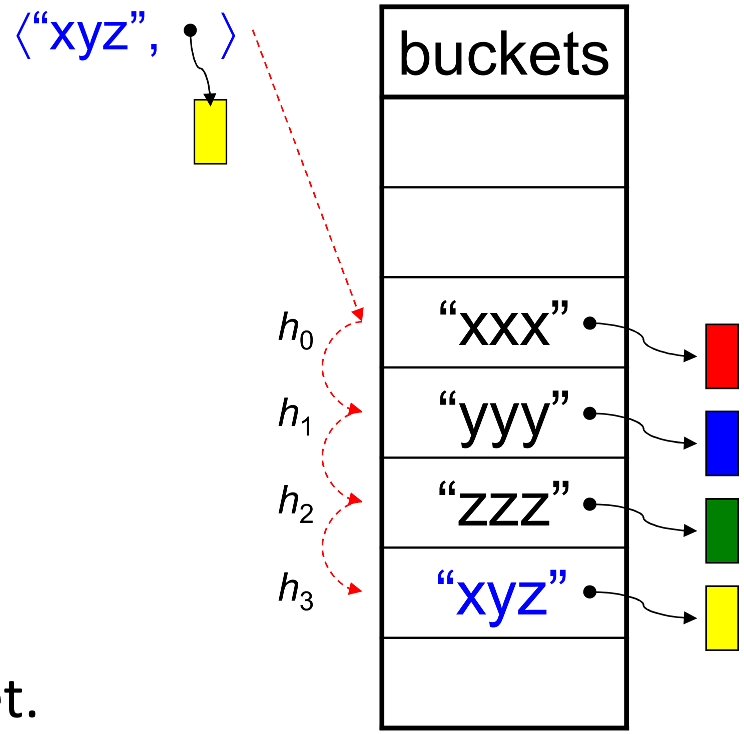
- Probe $h_0 = Hash(key, N_{buckets})$.
- Probe $h_1 = (h_0 + 1)$ % $N_{buckets}$.
- Probe $h_2 = (h_0 + 2)$ % $N_{buckets}$.
- Probe $h_3 = (h_0 + 3)$ % $N_{buckets}$.
- …
- Until *an empty bucket is found*.
- Insert the entry at the empty bucket.



Linear Probing: Enter

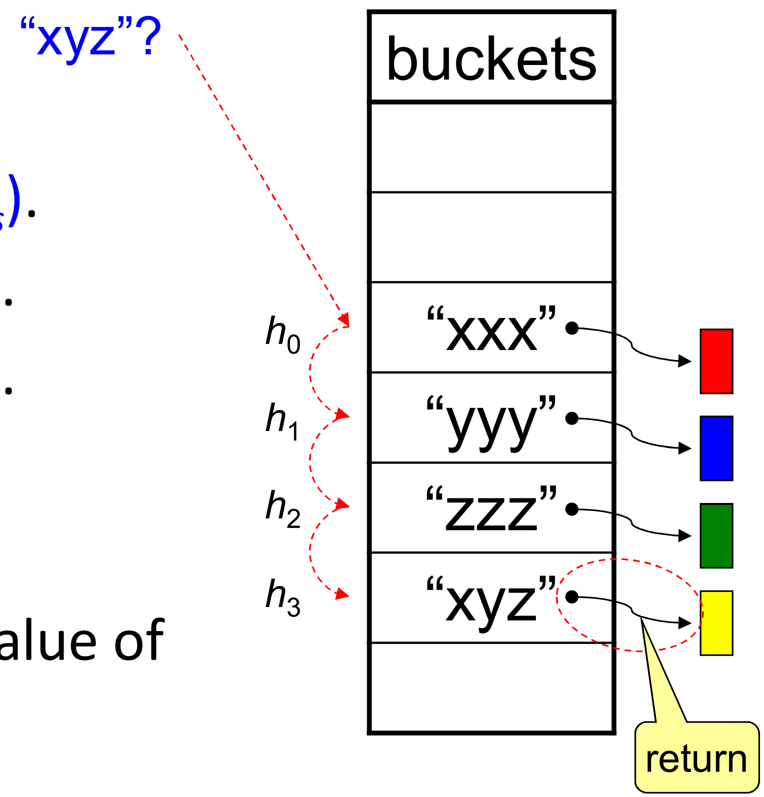
- Let $F(i) = i$.

- Probe $h_0 = Hash(key, N_{buckets})$.
- Probe $h_1 = (h_0 + 1) \% N_{buckets}$.
- Probe $h_2 = (h_0 + 2) \% N_{buckets}$.
- Probe $h_3 = (h_0 + 3) \% N_{buckets}$.
- …
- Until *an empty bucket is found*.
- Insert the entry at the empty bucket.



Linear Probing: Lookup

- Let $F(i) = i$.
- Probe $h_0 = Hash(key, N_{buckets})$.
- Probe $h_1 = (h_0 + 1) \% N_{buckets}$.
- Probe $h_2 = (h_0 + 2) \% N_{buckets}$.
- …
- Until *the key is matched*.
- Return the corresponding value of the bucket.



Example

- Inserting keys {89, 18, 49, 58, 69} into a hash table using the hash function *hash(x) = x mod 10* and the collision resolution strategy *f(i)=i*.

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

Example

- Inserting keys {89, 18, 49, 58, 69} into a hash table using the hash function *hash(x) = x mod 10* and the collision resolution strategy *f(i)=i*.

h_0 = $Hash(49)$=9.
h_1 = $(h_0 + 1)$ % $10 = 0$.

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

Example

- Inserting keys {89, 18, 49, 58, 69} into a hash table using the hash function *hash(x) = x mod 10* and the collision resolution strategy *f(i)=i*.

$$h_0 = Hash(58) = 8.$$
$$h_1 = (h_0 + 1) \,\%\, 10 = 9.$$
$$h_2 = (h_0 + 2) \,\%\, 10 = 0.$$
$$h_3 = (h_0 + 3) \,\%\, 10 = 1.$$

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

Example

- Inserting keys {89, 18, 49, 58, 69} into a hash table using the hash function *hash(x) = x mod 10* and the collision resolution strategy *f(i)=i*.

$$h_0 = Hash(69) = 9.$$
$$h_1 = (h_0 + 1) \% 10 = 0.$$
$$h_2 = (h_0 + 2) \% 10 = 1.$$
$$h_3 = (h_0 + 3) \% 10 = 2.$$

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

Primary Clustering in Linear Probing

- Linear probing tends to create long sequences of filled buckets in a hash table.
- This effect is called ***primary clustering***.
- Primary clustering degrades the performance of a hash table.
- ***Quadratic probing*** can be used to address the primary clustering problem.

Quadratic Probing

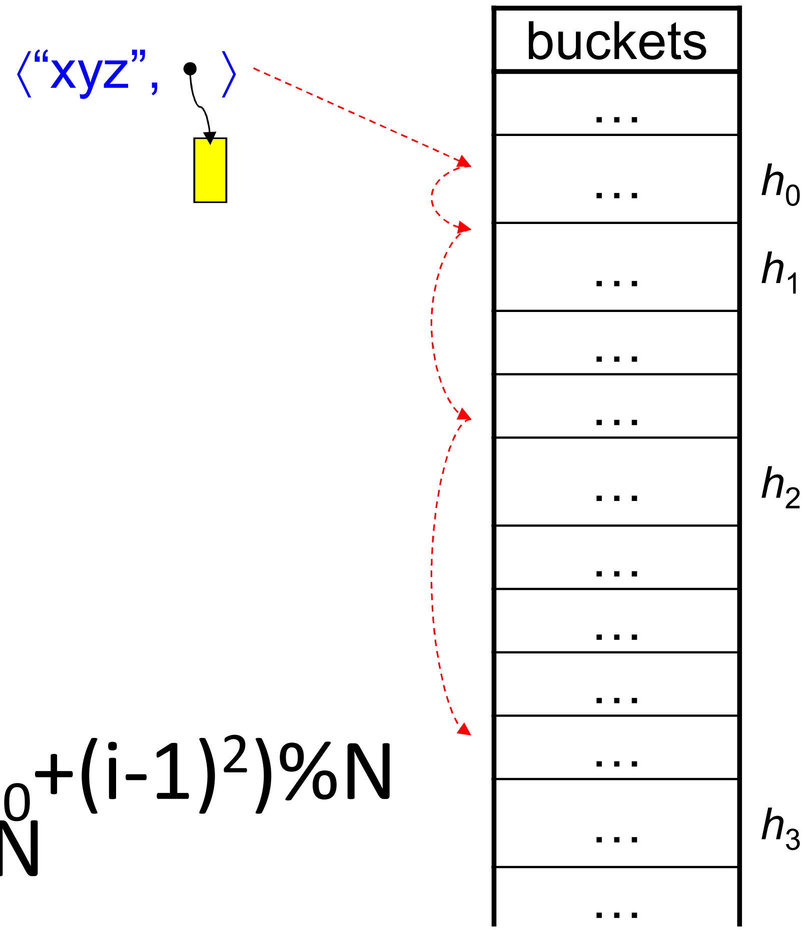
- Quadratic probing is mostly the same as linear probing.
- Let $h_0 = Hash(key, N_{buckets})$
- If h_0 collides, probe $h_1 = (h_0 + F(1)) \% N_{buckets}$.
- If h_1 collides, probe $h_2 = (h_0 + F(2)) \% N_{buckets}$.
- If h_2 collides, probe $h_3 = (h_0 + F(3)) \% N_{buckets}$.
- …
- $F(i)$ is a ***quadratic*** function. Typically, $F(i) = i^2$.

Quadratic Probing

- Let $F(i) = i^2$.
- Probe $h_0 = Hash(key, N_{buckets})$.
- Probe $h_1 = (h_0 + 1) \% N_{buckets}$.
- Probe $h_2 = (h_0 + 4) \% N_{buckets}$.
- Probe $h_3 = (h_0 + 9) \% N_{buckets}$.
- Probe $h_4 = (h_0 + 16) \% N_{buckets}$.
- …

$$h_i=(h_0+i^2)\%N \qquad h_{i-1}=(h_0+(i-1)^2)\%N$$

$$h_i=(h_{i-1}+2i-1)\%N$$



Example

- Inserting keys {89, 18, 49, 58, 69} into a hash table using the hash function *hash(x) = x mod 10* and the collision resolution strategy $f(i)=i^2$.

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

Example

- Inserting keys {89, 18, 49, 58, 69} into a hash table using the hash function *hash(x) = x mod 10* and the collision resolution strategy $f(i)=i^2$.

  h_0 = *Hash*(*49*)=9.
  $h_1 = (h_0 + 1^2)\ \%\ 10 = 0$.

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

Example

- Inserting keys {89, 18, 49, 58, 69} into a hash table using the hash function *hash(x) = x mod 10* and the collision resolution strategy $f(i)=i^2$.

h_0 = $Hash(58)=8$.
$h_1 = (h_0 + 1^2)$ % $10 = 9$.
$h_2 = (h_0 + 2^2)$ % $10 = 2$.

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

Example

- Inserting keys {89, 18, 49, 58, 69} into a hash table using the hash function *hash(x) = x mod 10* and the collision resolution strategy $f(i)=i^2$.

$$h_0 = Hash(69)=9.$$
$$h_1 = (h_0 + 1^2)\ \%\ 10 = 0.$$
$$h_2 = (h_0 + 2^2)\ \%\ 10 = 3.$$

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

Quadratic Probing

- While quadratic probing eliminates the primary clustering problem, it has its drawback as well.

- There is *no* guarantee of finding an empty bucket once the table gets more than half full (i.e., $\lambda > 0.5$).

Load Factor Restriction: an Example

- Let $N_{buckets} = 7$, $h_0 = 0$, and $F(i) = i^2$.
  - $h_0 = 0$
  - $h_1 = (0 + 1) \% 7 = 1$
  - $h_2 = (0 + 4) \% 7 = 4$
  - $h_3 = (0 + 9) \% 7 = 2$
  - $h_4 = (0 + 16) \% 7 = 2$
  - $h_5 = (0 + 25) \% 7 = 4$
  - $h_6 = (0 + 36) \% 7 = 1$
  - $h_7 = (0 + 49) \% 7 = 0$
  - $h_8 = (0 + 64) \% 7 = 1$
  - $h_9 = (0 + 81) \% 7 = 4$
  - $h_{10} = (0 + 100) \% 7 = 2$
  - $h_{11} = (0 + 121) \% 7 = 2$
  - $h_{12} = (0 + 144) \% 7 = 4$
  - $h_{13} = (0 + 169) \% 7 = 1$
  - $h_{14} = (0 + 196) \% 7 = 0$
  - …

The buckets 3, 5, and 6 are *never* probed, even if they are empty.

Double Hashing

- To eliminate the load factor restriction, we can use a second hash function $Hash_2$ and

$$F(i) = i \times Hash_2(key, N_{buckets})$$

  - $h_0 = Hash(key, N_{buckets})$
  - $h_1 = (h_0 + 1 \times Hash_2(key, N_{buckets}))\ \%\ N_{buckets}$
  - $h_2 = (h_0 + 2 \times Hash_2(key, N_{buckets}))\ \%\ N_{buckets}$
  - $h_3 = (h_0 + 3 \times Hash_2(key, N_{buckets}))\ \%\ N_{buckets}$
  - …

$Hash_2$ should return an integer in $[1, N_{buckets} - 1]$.

- The idea is that even if two keys hash to the same code using *Hash*, they will have different codes using $Hash_2$.

Second Hash Function

- One possible second hash function

```
int Hash2(char *s, int nBuckets) {
   int i;
   unsigned long hashCode;

   hashCode = 0;
   for (i = 0; s[i] != '\0'; i++)
      hashCode = hashCode * MULTIPLIER + s[i];
   return (hashCode % (nBuckets - 1) + 1);
}
```

```
int Hash2(char *s, int nBuckets) {
   int i;
   unsigned long hashCode;

   hashCode = 0;
   for (i = 0; s[i] != '\0'; i++)
      hashCode = hashCode * MULTIPLIER + s[i];
   return (hashCode % (nBuckets – 1) + 1);
}
```

- $h_0 = Hash(key, N_{buckets})$
- $h_1 = (h_0 + 1 \times Hash_2(key, N_{buckets}))\ \%\ N_{buckets}$
- $h_2 = (h_0 + 2 \times Hash_2(key, N_{buckets}))\ \%\ N_{buckets}$

Returns an integer in the range [1, nBuckets - 1].

```
int Hash(char *s, int nBuckets) {
   int i;
   unsigned long hashCode;

   hashCode = 0;
   for (i = 0; s[i] != '\0'; i++)
      hashCode = hashCode * MULTIPLIER + s[i];
   return (hashCode % nBuckets);
}
```

Returns an integer in the range [0, nBuckets - 1].

Rehashing

- No matter what collision resolution scheme is used, the performance of an open addressing hash table degrades when λ approaches 1 (i.e., the table is becoming full).

- We can dynamically increase $N_{buckets}$ when λ increases above a certain threshold.

Rehashing

- When $N_{buckets}$ is changed, the hash function has to be changed also. (Why?)
- All entries in the hash table have to be re-entered using the new hash function.
- This process is called ***rehashing***.
- Note: rehashing is *rarely necessary* for most applications.