UNIVERSITÀ DEGLI STUDI DI PADOVA

Data Structures
Hash Tables

Gianmaria Silvello

Department of Information Engineering
University of Padua

gianmaria.silvello@unipd.it

http://www.dei.unipd.it/~silvello/

Outline

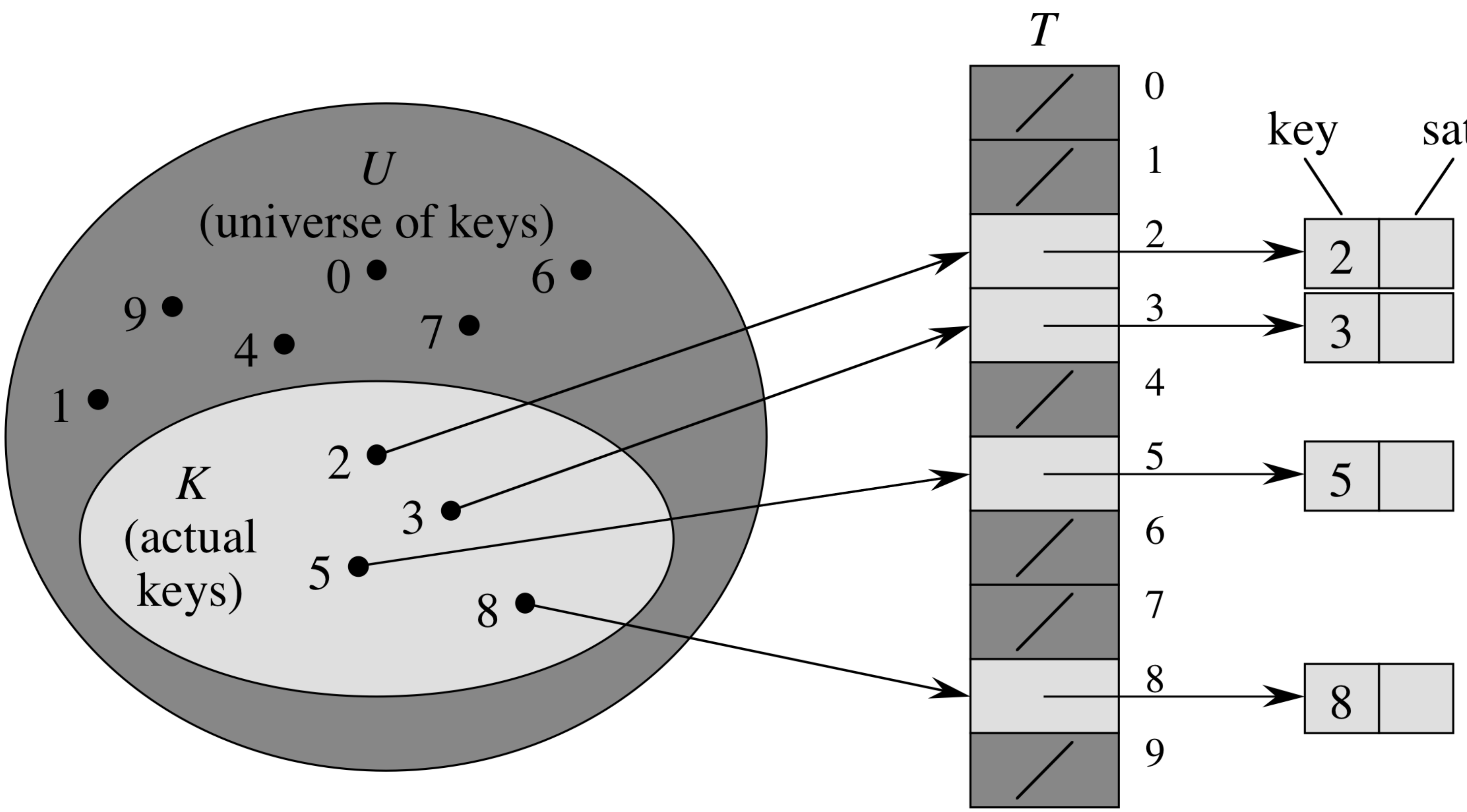
- Direct addressing
- Hash Table
- Hash functions
- Collisions
  - Chaining
  - Open addressing
- Rehashing
- Python hash tables…

Reference: Chapter 11 of CLRS

Reference: Chapter 10 of Goodrich, Tamassia and Goldwasser

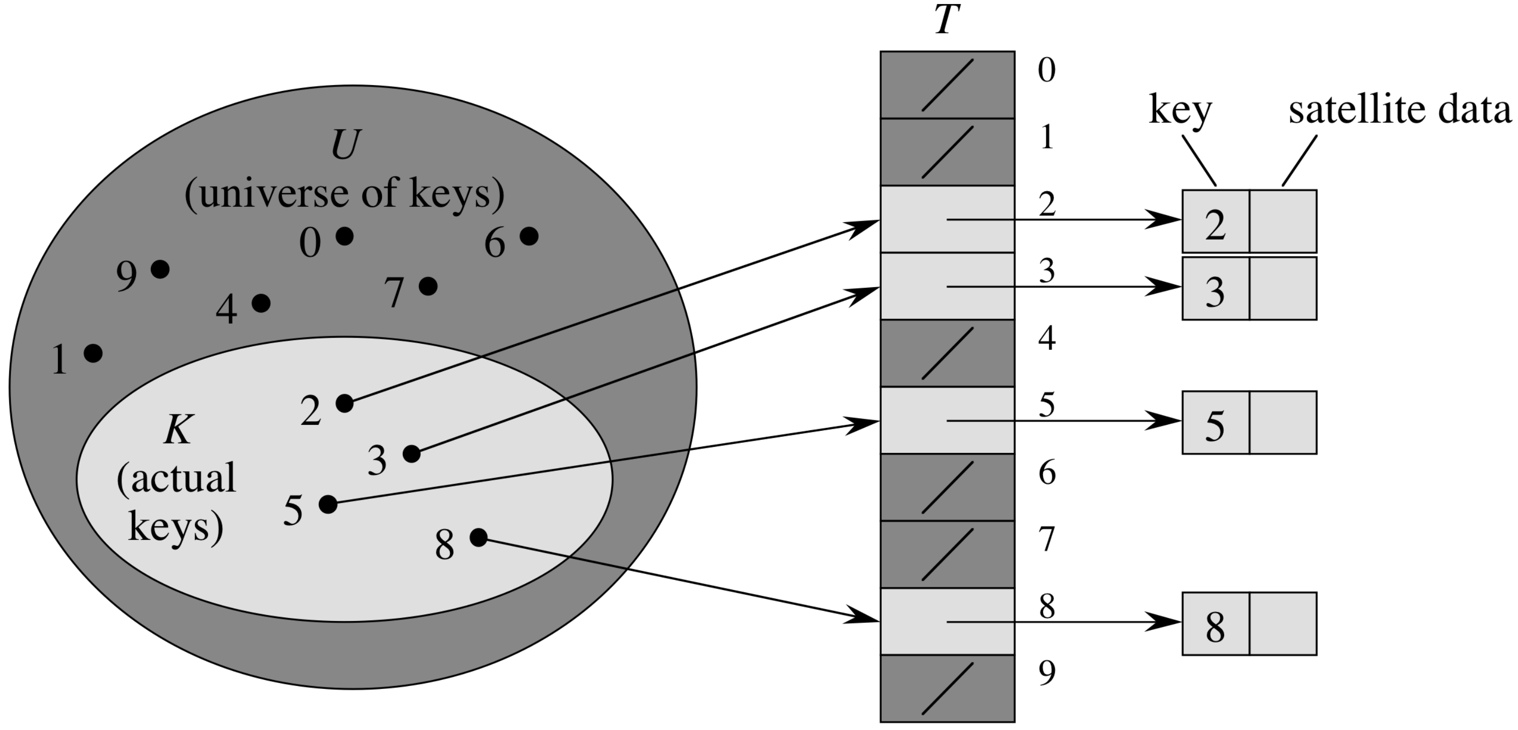
Direct-address table

- Direct addressing is a simple technique that works well when the universe U of possible keys is relatively small.
- Given a universe `U = {0,1,…,m-1}` a direct-address table is an array `T[0,…,m-1]` where `T[i]=i`∈`U`

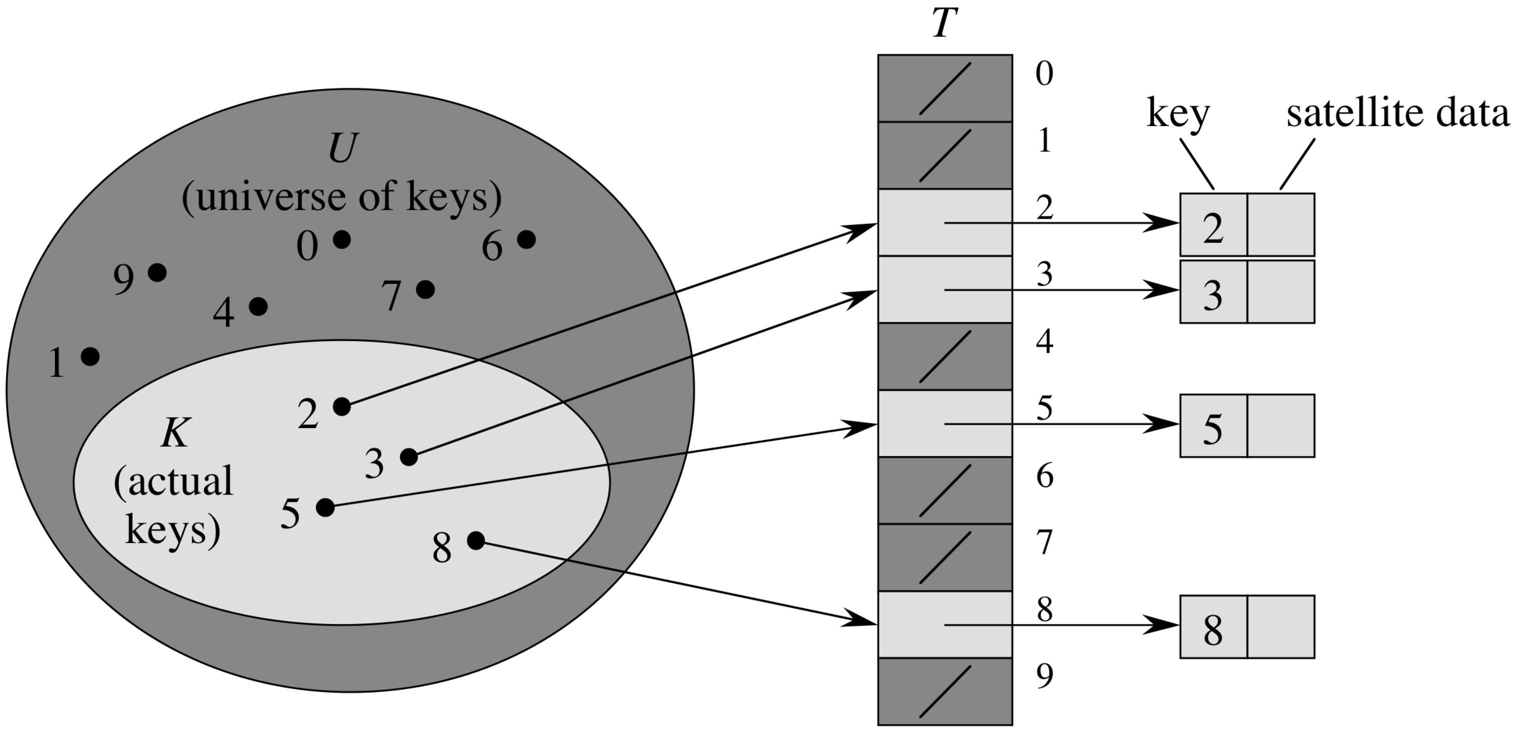


To store 4 keys we need to have |U| =10 slots

Direct-address table

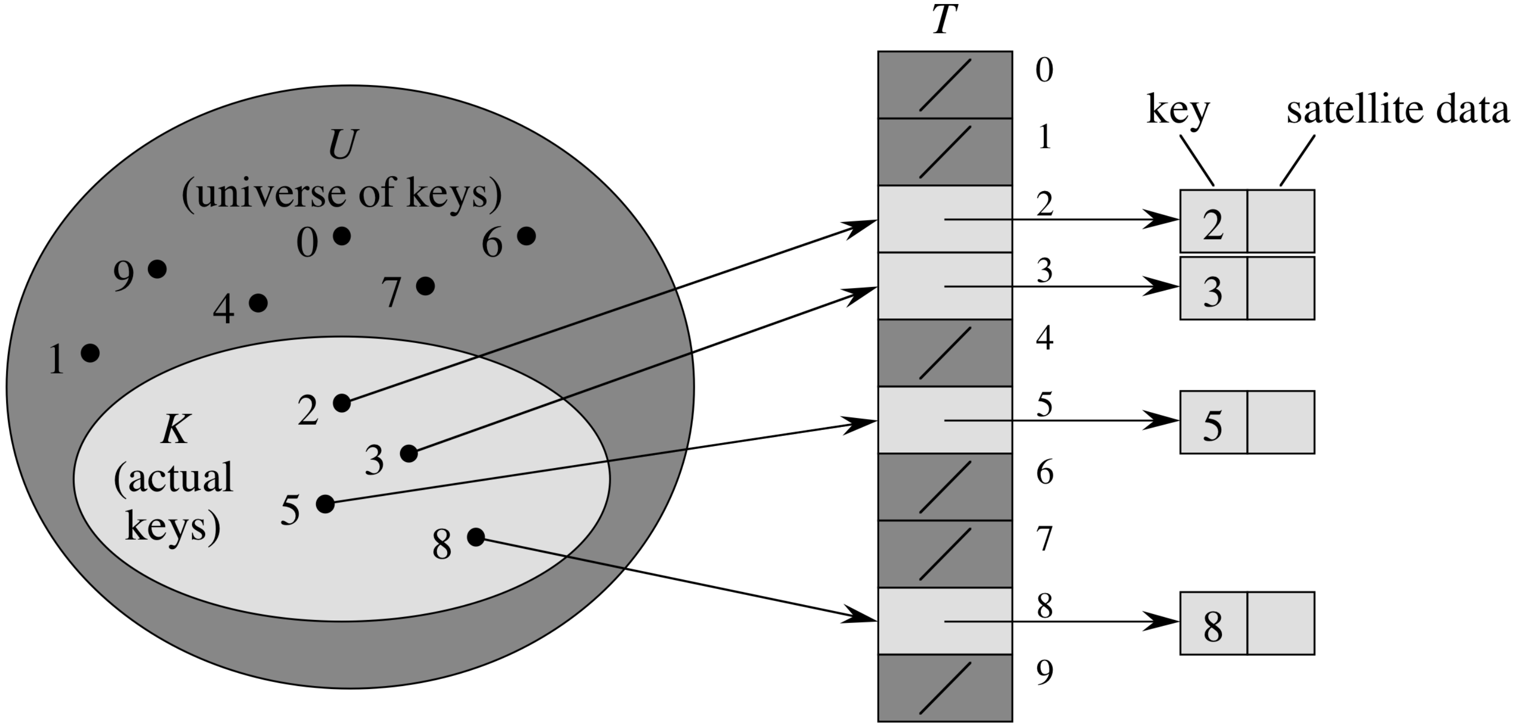


Direct-address table




```
Direct-Table-Search(T, k)
return T[k]
```

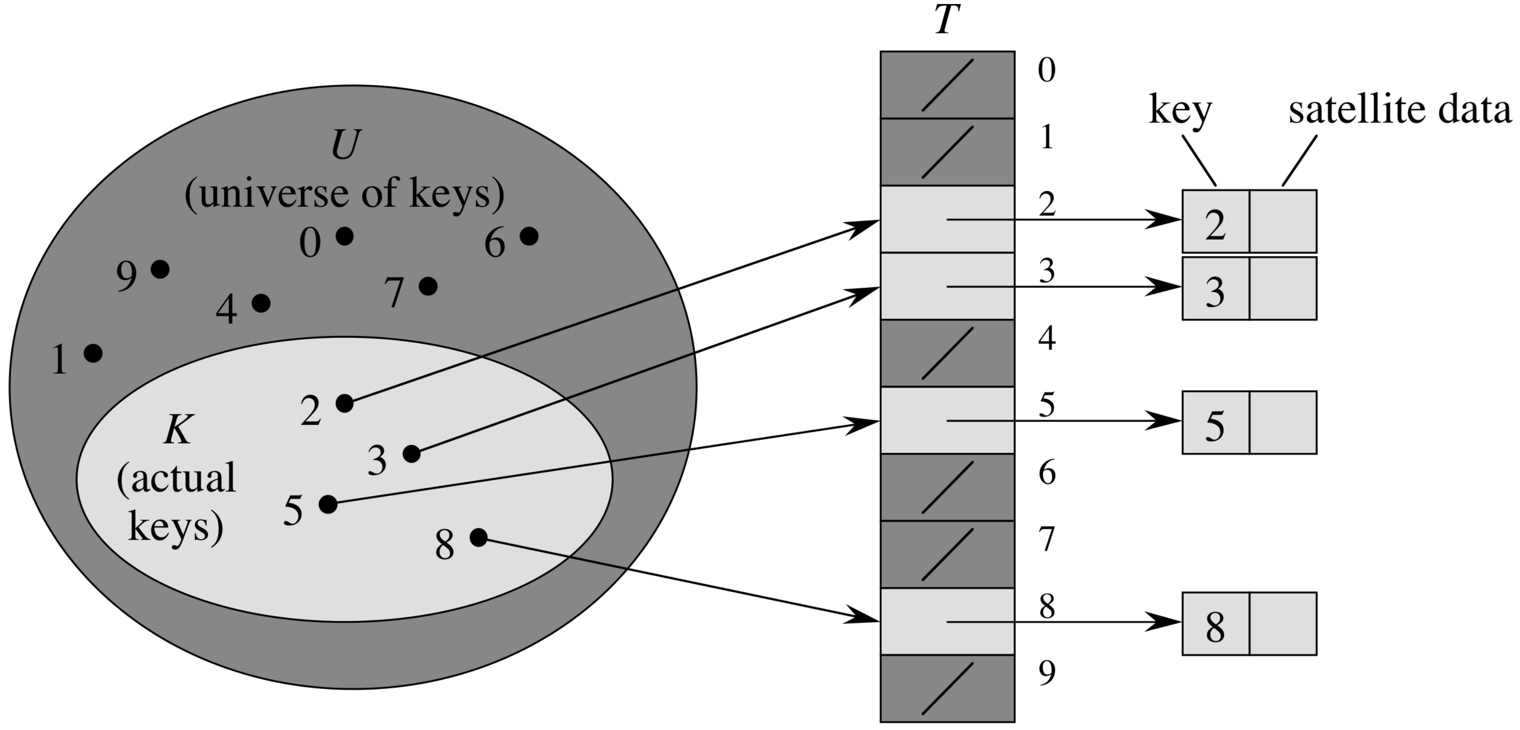
Direct-address table




```
Direct-Table-Search(T, k)
return T[k]

Direct-Table-Insert(T, x)
return T[x.key] = x
```

Direct-address table

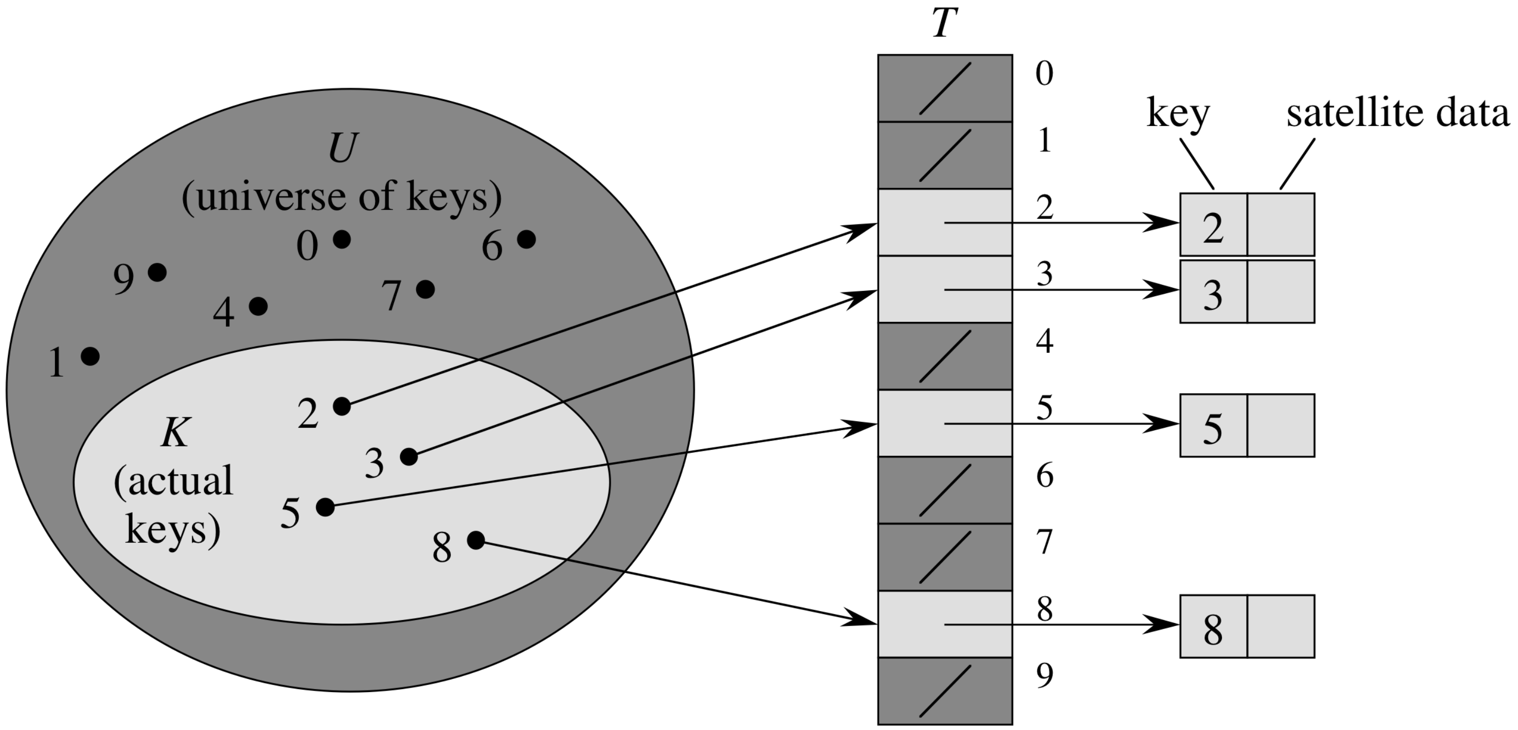



```
Direct-Table-Search(T, k)
return T[k]

Direct-Table-Insert(T, x)
return T[x.key] = x

Direct-Table-Delete(T, x)
return T[x.key] = NIL
```

Direct-address table




```
Direct-Table-Search(T, k)
return T[k]
```

O(1)

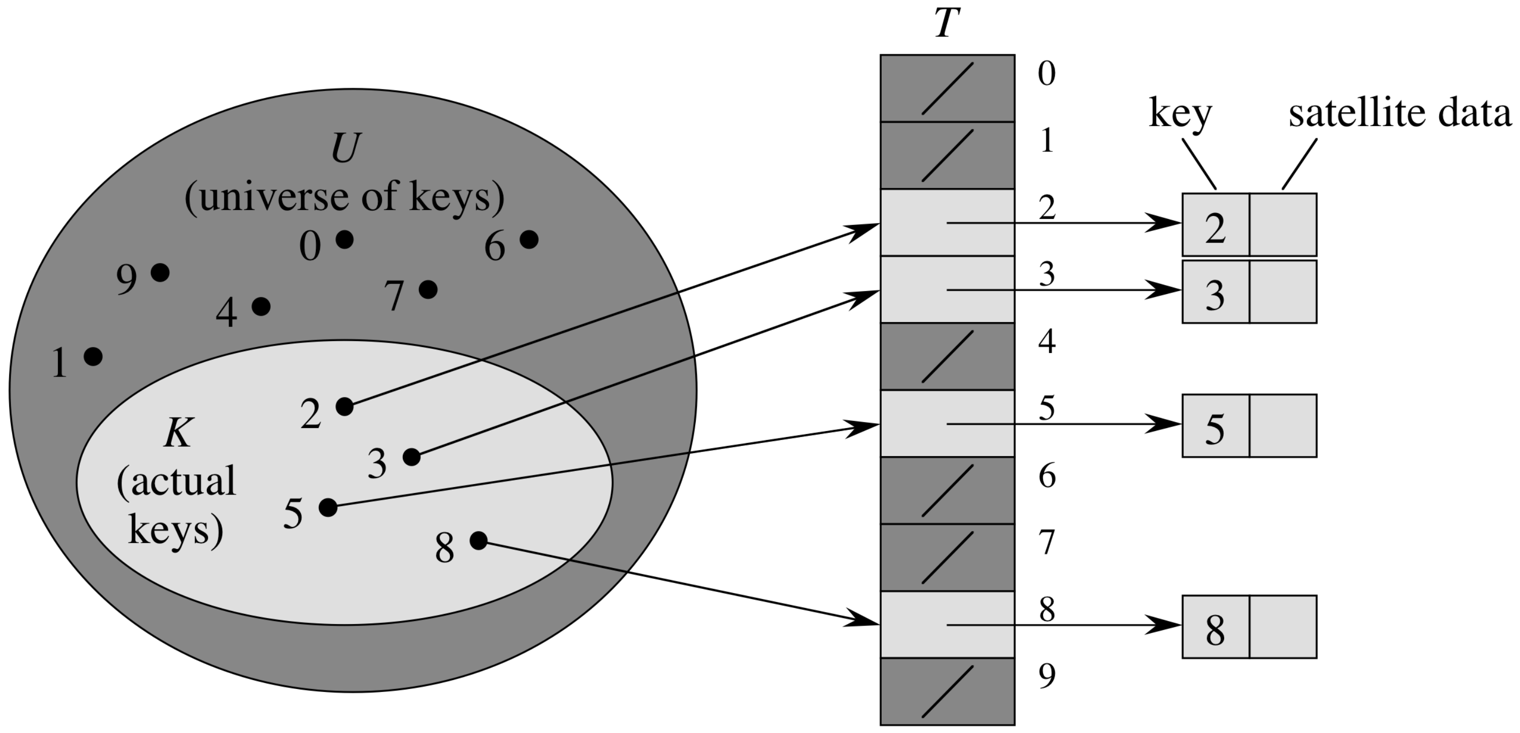
```
Direct-Table-Insert(T, x)
return T[x.key] = x
```

O(1)

```
Direct-Table-Delete(T, x)
return T[x.key] = NIL
```

O(1)

Direct-address table




```
Direct-Table-Search(T, k)
return T[k]
```

O(1)

Time efficient

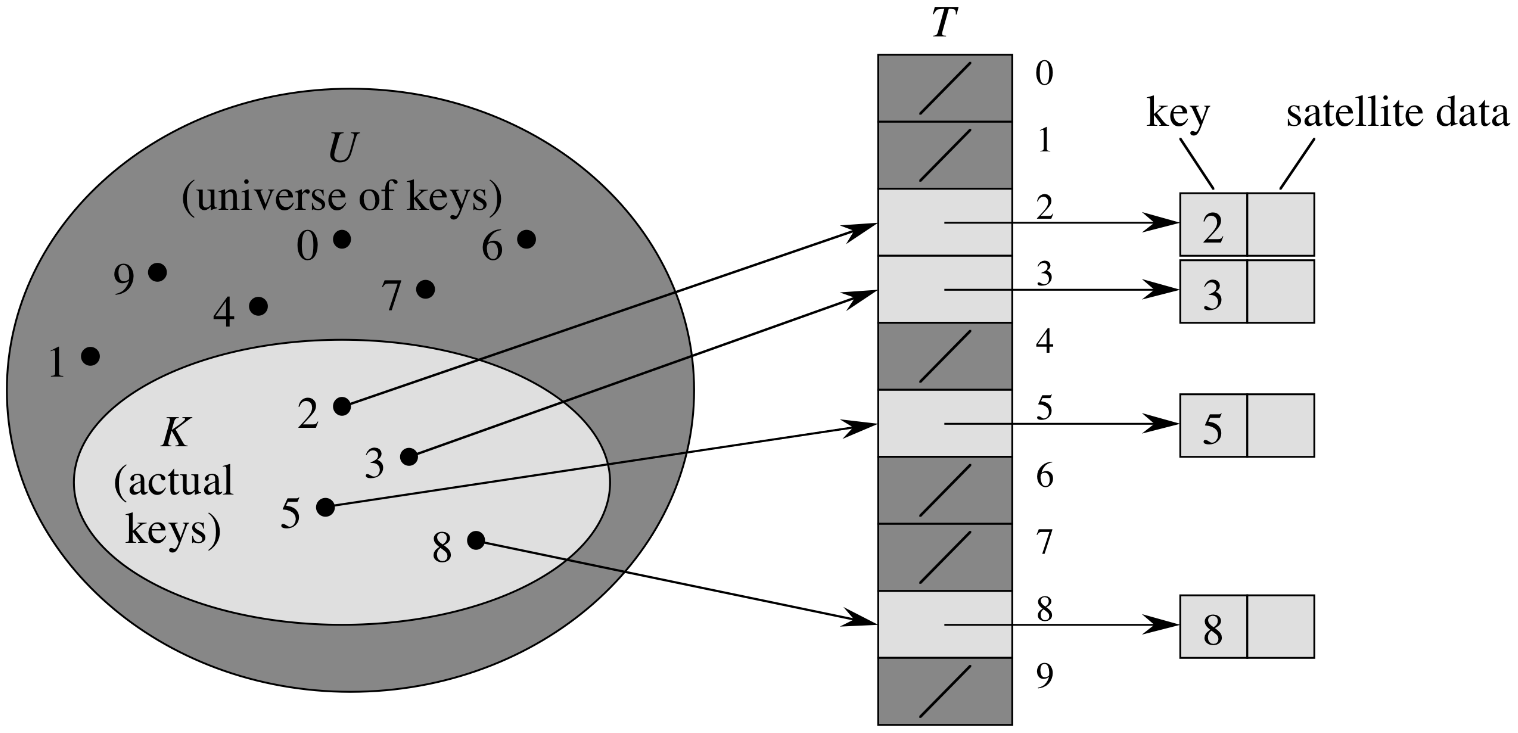
```
Direct-Table-Insert(T, x)
return T[x.key] = x
```

O(1)

```
Direct-Table-Delete(T, x)
return T[x.key] = NIL
```

O(1)

Direct-address table




```
Direct-Table-Search(T, k)
return T[k]
```

O(1)

Time efficient

```
Direct-Table-Insert(T, x)
return T[x.key] = x
```

O(1)

Space inefficient if
|U| >> |K|

```
Direct-Table-Delete(T, x)
return T[x.key] = NIL
```

O(1)

Hash Table

- Use a table of size proportional to |K| – The hash tables.
  - However, we lose the direct-addressing ability.
  - Define functions that map keys to slots of the hash table.

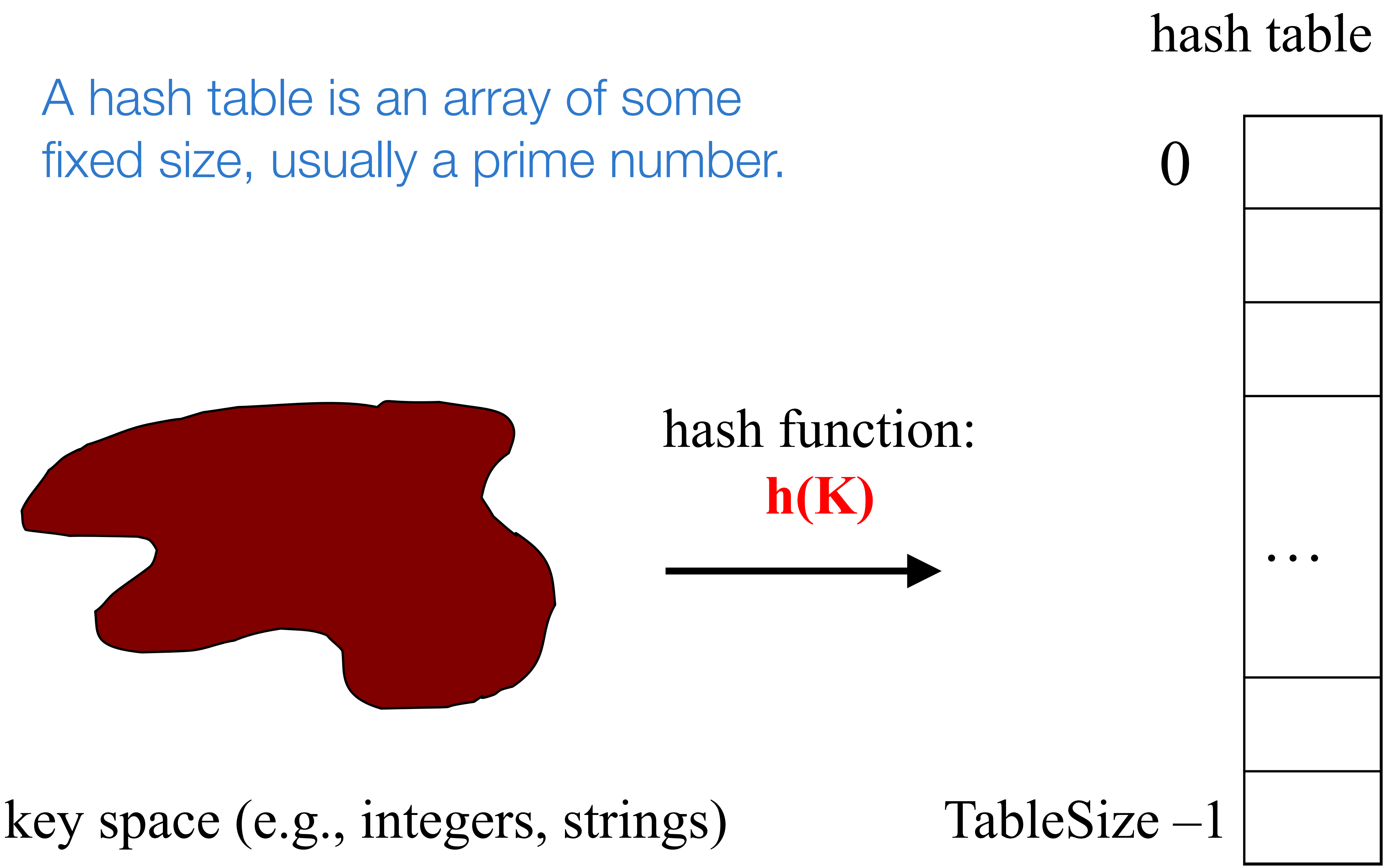
- Hash function h: Mapping from `U` to the slots of a hash table `T[0..m-1]`.

`h:U->{0,1,…,m-1}`

- With arrays, key k maps to slot `T[k]`.
- With hash tables, key k maps or “hashes” to slot `T[h[k]]`.
- `h[k]` is the hash value of key k.

Hash Table

A hash table is an array of some fixed size, usually a prime number.



Hash function: Example

key space = integers

TableSize = 6

`h(K) = K mod 6`

0	
1	
2	
3	
4	
5	

Hash function: Example

key space = integers

TableSize = 6

`h(K) = K mod 6`

Insert 7 7 mod 6 = 1

0	
1	7
2	
3	
4	
5	

Hash function: Example

key space = integers

TableSize = 6

`h(K) = K mod 6`

Insert 7 7 mod 6 = 1

Insert 18 18 mod 6 = 0

0	18
1	7
2	
3	
4	
5	

Hash function: Example

key space = integers

TableSize = 6

`h(K) = K mod 6`

Insert 7 7 mod 6 = 1

Insert 18 18 mod 6 = 0

Insert 41 41 mod 6 = 5

Insert 34 34 mod 6 = 4

0	18
1	7
2	
3	
4	34
5	41

Hash function: Example

key space = integers

TableSize = 6

`h(K) = K mod 6`

0	18
1	7
2	
3	
4	34 10
5	41

Insert 7 7 mod 6 = 1

Insert 18 18 mod 6 = 0

Insert 41 41 mod 6 = 5

Insert 34 34 mod 6 = 4

Insert 10 10 mod 6 = 4

Hash function: Example

key space = integers

TableSize = 6

`h(K) = K mod 6`

Insert 7   7 mod 6 = 1

Insert 18   18 mod 6 = 0

Insert 41   41 mod 6 = 5

Insert 34   34 mod 6 = 4

Insert 10   10 mod 6 = 4

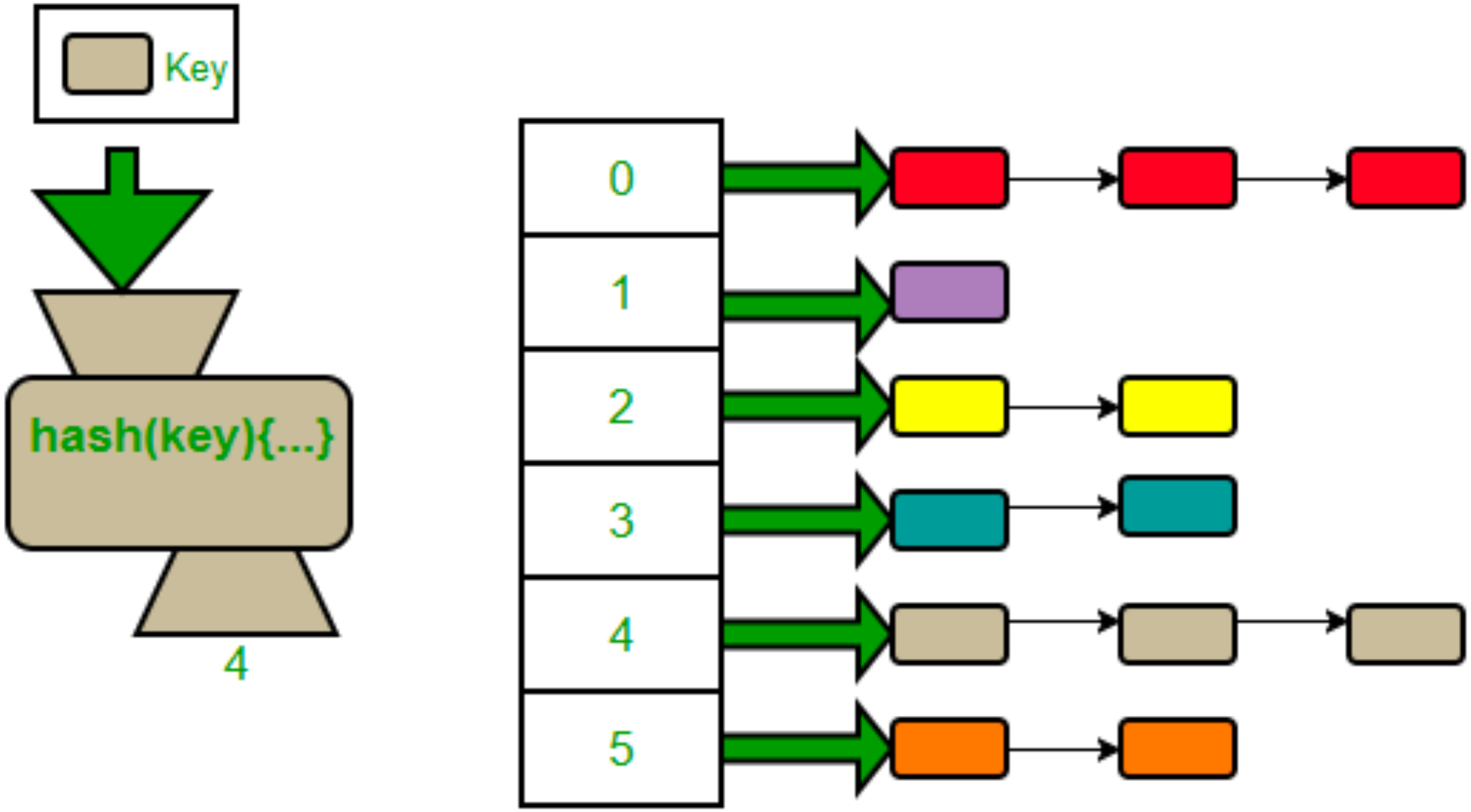
0	18	
1	7	
2		
3		
4	34 10	collision
5	41	

Hash function

- A hash function should:
  - be simple/fast to compute;
  - avoid collisions;
  - have keys distributed evenly among cells.

- Given that `|U|>>|K|`, collisions cannot be avoided, but they can be handled…

Avoiding collisions: Separate chaining



https://www.geeksforgeeks.org/wp-content/uploads/implementing-own-hash-table.png

Avoiding collisions: Separate chaining

- INSERT into the hash table requires O(1) if we are sure that the element is not already present in the hash table, otherwise it takes O(1+SEARCH)
- SEARCH worst-case running time is proportional to the length of the list associated with the hash
- DELETE is O(1+SEARCH)

- How long does SEARCH takes?

Loading factor

- Given a hash table T with m slots that stores n elements, the load factor $\alpha = n/m$
- The load factor is the average number of elements stored in a hash table where $\alpha \in [0,1]$
- The worst-case search time is $\Theta(n)$!!!
- The average-case for searching depends on how well the hash function distributes the n keys in the m slots (on average).

Simple Uniform Hashing

- We assume that any given element is equally likely to hash into any of the m slots.
- For $j = 0, 1, \ldots, m-1$, the list $T[j]$ is long n_j, so that $n = n_0 + n_1 + \ldots + n_{m-1}$
  - The expected value of n_j is $E[n_j] = \alpha = n/m$
- The average running time for a successful search in a hash table solving collisions with chaining and adopting simple uniform hashing is $\Theta(1+\alpha)$
- If m is proportional to n, then $n = O(m)$ and, $\alpha = n/m = O(m)/m = O(1)$

Good hash functions

- We'd need to know the distributions of the keys
- For instance, if the keys are random real numbers $0 \leq k \leq 1$ then the hash function $h(k) = \lfloor nk \rfloor$ satisfies the condition of simple uniform hashing.
- Most hash functions assume that the universe of keys is the set of natural numbers
  - Strings: we can interpret strings as naturals. e.g. `pt` is $p=112$ and $t=116$ (ASCII codes), then expressed as radix-128 integer we have $(112 \cdot 128)+116=14452$

Division method

- `h(k) = k mod m`
- How do we choose `m`?
  - Often a prime number not too close to a power of 2 is a good choice
  - `m=2p` means that *h(k)* is just the *p* lowest-order bits of *k*
    - *Unless we know that the lower order bits are all equally likely, we'd better consider all the bits of the key*
  - For instance if n=2000 character strings (each char is 8 bits), we do not mind examining 3 elements in a list when conflict occur, then given that 2000/3 = 666.666 we may choose m=701 because it is a prime number close to 666 but not to close to a power of 2
- Other methods:
  - multiplication method: $h(k) = \lfloor m(kA \bmod 1) \rfloor,\ 0 \leq A \leq 1$
    - Which A? Knuth said that a reasonable A is $(\sqrt{5}-1)/2=0.61803\ldots$
  - squaring: square the key and then truncate

Division method

- `h(k) = k mod m`
- How do we choose `m`?
  - Often a prime number not too close to a power of 2 is a good choice
  - `m=2p` means that *h(k)* is just the *p* lowest-order bits of *k*
    - *Unless we know that the lower order bits are all equally likely, we'd better consider all the bits of the key*
  - For instance if n=2000 character strings (each char is 8 bits), we do not mind examining 3 elements in a list when conflict occur, then given that 2000/3 =
    6 66 but
    r

Unlike the division method, we don't need to avoid certain values of m here

In fact, we often set m to be a power of 2 (say m = 2p) -> easier computation

- Other methods:
  - multiplication method: `h(k) = ⌊m(kA mod 1)⌋, 0≤A≤1`
    - Which A? Knuth said that a reasonable A is (√(5)-1)/2=0.61803…
  - squaring: square the key and then truncate

Avoiding Collisions: Open Addressing

- Separate chaining has the disadvantage of using linked lists.
  - Requires the implementation of a second data structure.
- In an open addressing hashing system, all the data go inside the table.
  - Thus, a bigger table is needed.
  - Generally the load factor should be below 0.5.
- If a collision occurs, alternative cells are tried until an empty cell is found

Avoiding Collisions: Open Addressing

- 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 $f(0) = 0$.
- The function f is the collision resolution strategy.
- There are three common collision resolution strategies:
  - Linear Probing
  - Quadratic probing
  - Double hashing

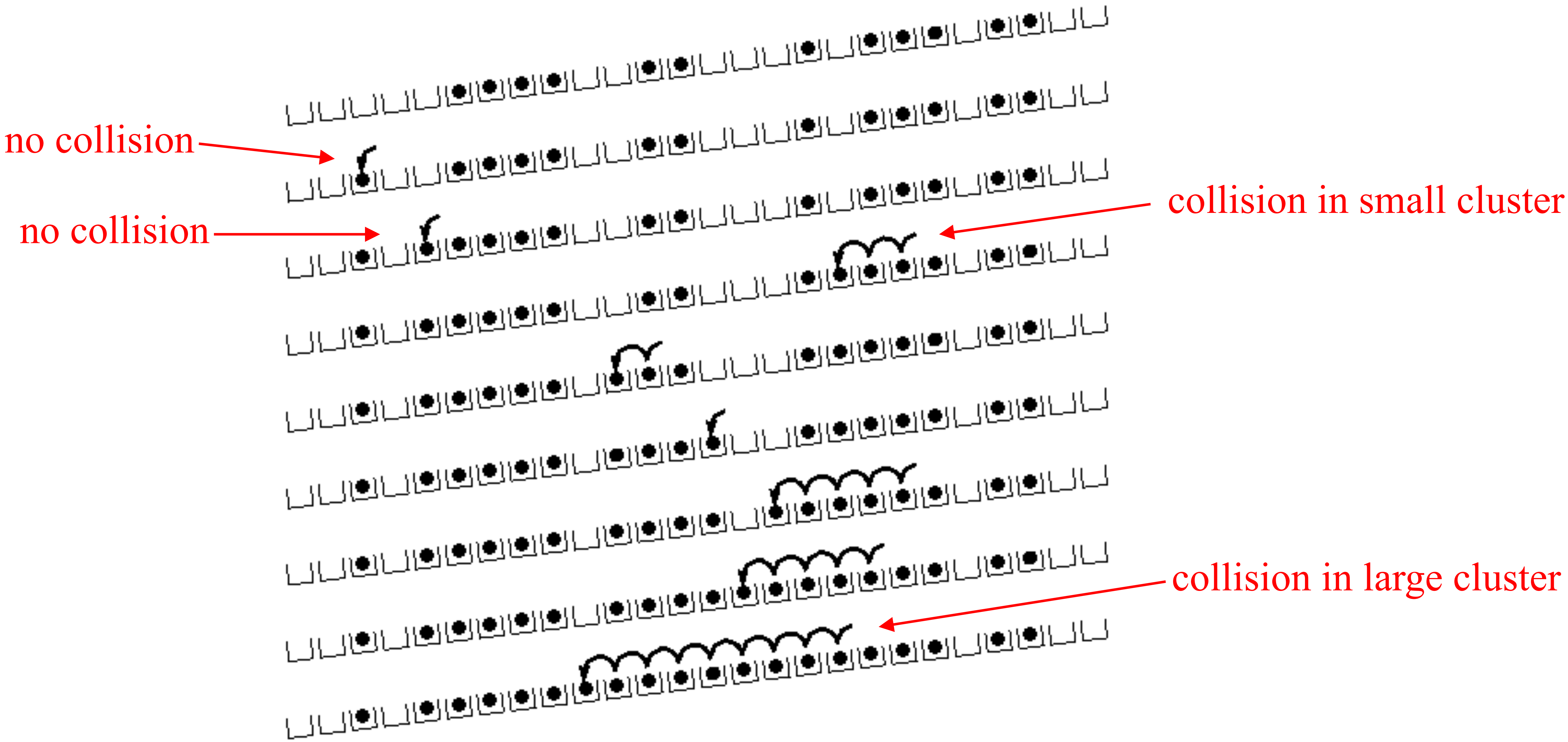
Linear Probing

- Probe sequence:
  - 0^{th} probe = $h(k) = k \bmod TableSize$
  - 1^{st} probe = $(h(k) + 1) \bmod TableSize$
  - 2^{nd} probe = $(h(k) + 2) \bmod TableSize$
  - . . .
  - i^{th} probe = $(h(k) + i) \bmod TableSize$
- Add a function of i to the original hash value to resolve the collision.
- Any key that hashes into the cluster 1) will require several attempts to resolve collision and 2) will then add to the cluster.

Primary Clustering

- It works pretty well for an empty table and gets worse as the table fills up
- If a bunch of elements hash to the same spot, they clash one with the others
- But, worse, if a bunch of elements hash to the same area of the table, they keep clashing!
  - (Even though the hash function isn't producing lots of collisions!)
- This phenomenon is called *primary clustering*.

Primary Clustering



[R. Sedgewick]

Open Addressing: Linear Probing

- In linear probing, collisions are resolved by sequentially scanning an array (with wraparound) until an empty cell is found.
  - i.e. f is a linear function of i, typically f(i)= i.
- Example:
  - Insert items with keys: 89, 18, 49, 58, 9 into an empty hash table.
  - Table size is 10.
- Hash function is h(x) = x mod 10.
  - f(i) = i;

Open Addressing: Linear Probing

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

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

Linear Probing: An Example

What is the average number of probes for a successful search and an unsuccessful search for this hash table?

–Hash Function: h(x) = x mod 11

0	9
1	
2	2
3	13
4	25
5	24
6	
7	
8	30
9	20
10	10

<u>Successful Search:</u>

–20: 9 -- 30: 8 -- 2 : 2 -- 13: 2, 3 -- 25: 3,4
–24: 2,3,4,5 -- 10: 10 -- 9: 9,10, 0
Avg. Probe = (1+1+1+2+2+4+1+3)/8=15/8

<u>Unsuccessful Search:</u>

–We assume that the hash function uniformly distributes the keys.
–0: 0,1 -- 1: 1 -- 2: 2,3,4,5,6 -- 3: 3,4,5,6
–4: 4,5,6 -- 5: 5,6 -- 6: 6 -- 7: 7 -- 8: 8,9,10,0,1
–9: 9,10,0,1 -- 10: 10,0,1
Avg. Probe = (2+1+5+4+3+2+1+1+5+4+3)/11=31/11

Open Addressing: Search and Insertion

- The SEARCH follows the same probe sequence as the insert algorithm.
  - SEARCH for 58 would involve 4 probes.
  - SEARCH find for 19 would involve 5 probes.
- The average number of cells that are examined in an insertion using linear probing is roughly
  - $(1 + 1/(1 - \alpha)^2) / 2$
- For a half full table we obtain 2.5 as the average number of cells examined during an insertion.
- Primary clustering is a problem at high load factors. For half empty tables the effect is not disastrous

Open Addressing: Search and Insertion

- The cost of a successful search of X is equal to the cost of inserting X at the time X was inserted.
- For $\alpha = 0.5$ the average cost of insertion is 2.5. The average cost of finding the newly inserted item will be 2.5 no matter how many insertions follow.
- The average number of cells that are examined in an unsuccessful search using linear probing is roughly
  - $(1 + 1/(1 - \alpha)^2) / 2$
- The average number of cells that are examined in a successful search is approximately
  - $(1 + 1/(1 - \alpha)) / 2$

Quadratic probing

- $f(i) = i^2$
- Probe sequence:
  - 0th probe = h(k) = k mod TableSize
  - 1st probe = (h(k) + 1) mod TableSize
  - 2nd probe = (h(k) + 4) mod TableSize
  - 3rd probe = (h(k) + 9) mod TableSize
  - . . .
  - ith probe = $(h(k) + i^2)$ mod TableSize

Quadratic probing

- Less likely to encounter primary clustering, but could run into secondary clustering
- Although keys that hash to the same initial location will still use the same sequence of probes (and conflict with each other)
- How big to make hash table? $\alpha = ½$, (hash table is twice as big as the number of elements expected.)
- Note: $(i+1)^2 - i^2 = 2i + 1$ Thus, to get to the NEXT step, you can add 2 times the current value of i plus one.

Quadratic Probing: Example

key space = integers

TableSize = 7

h(K) = K mod 7

- ith probe = $(h(k) + i^2)$ mod TableSize

Insert 76 $\quad$ 76 mod 7 = 6

Insert 40 $\quad$ 40 mod 7 = 5

0	
1	
2	
3	
4	
5	40
6	76

Quadratic Probing: Example

key space = integers

- ith probe = (h(k) + i^2) mod TableSize

TableSize = 7

h(K) = K mod 7

Insert 76 76 mod 7 = 6

Insert 40 40 mod 7 = 5

Insert 48 48 mod 7 = 6 (6 + 1) mod 7 = 0

0	48
1	
2	
3	
4	
5	40
6	76

Quadratic Probing: Example

- ith probe = $(h(k) + i^2) \bmod TableSize$

key space = integers

TableSize = 7

$h(K) = K \bmod 7$

Insert 76 $76 \bmod 7 = 6$

Insert 40 $40 \bmod 7 = 5$

Insert 48 $48 \bmod 7 = 6$ $(6 + 1) \bmod 7 = 0$

Insert 5 $5 \bmod 7 = 5$ $(5 + 1) \bmod 7 = 6$

$(5 + 2^2) \bmod 7 = 2$

0	48
1	
2	5
3	
4	
5	40
6	76

Quadratic Probing: Example

key space = integers

TableSize = 7

h(K) = K mod 7

- ith probe = $(h(k) + i^2) \bmod \text{TableSize}$

0	48
1	
2	5
3	55
4	
5	40
6	76

Insert 76 — 76 mod 7 = 6

Insert 40 — 40 mod 7 = 5

Insert 48 — 48 mod 7 = 6 — (6 + 1) mod 7 = 0

Insert 5 — 5 mod 7 = 5 — (5 + 1) mod 7 = 6; $(5 + 2^2) \bmod 7 = 2$

Insert 55 — 55 mod 7 = 6 — (6 + 1) mod 7 = 0; $(6 + 2^2) \bmod 7 = 3$

Quadratic Probing: Example

key space = integers

TableSize = 7

$h(K) = K \bmod 7$

- ith probe = $(h(k) + i^2) \bmod TableSize$

Insert 47 $47 \bmod 7 = 5$

Insert 76 $76 \bmod 7 = 6$

Insert 40 $40 \bmod 7 = 5$

Insert 48 $48 \bmod 7 = 6$ $(6 + 1) \bmod 7 = 0$

Insert 5 $5 \bmod 7 = 5$ $(5 + 1) \bmod 7 = 6$

$(5 + 2^2) \bmod 7 = 2$

Insert 55 $55 \bmod 7 = 6$ $(6 + 1) \bmod 7 = 0$

$(6 + 2^2) \bmod 7 = 3$

0	48
1	
2	5
3	55
4	
5	40
6	76

Quadratic Probing: Example

key space = integers

TableSize = 7

$h(K) = K \bmod 7$

- ith probe = $(h(k) + i^2) \bmod \text{TableSize}$

Insert 47 $47 \bmod 7 = 5$

$(5 + 1) \bmod 7 = 6$

$(5 + 2^2) \bmod 7 = 2$

$(5 + 3^2) \bmod 7 = 0$

$(5 + 4^2) \bmod 7 = 0$

…

Insert 76 $76 \bmod 7 = 6$

Insert 40 $40 \bmod 7 = 5$

Insert 48 $48 \bmod 7 = 6$ $(6 + 1) \bmod 7 = 0$

Insert 5 $5 \bmod 7 = 5$ $(5 + 1) \bmod 7 = 6$

$(5 + 2^2) \bmod 7 = 2$

Insert 55 $55 \bmod 7 = 6$ $(6 + 1) \bmod 7 = 0$

$(6 + 2^2) \bmod 7 = 3$

0	48
1	
2	5
3	55
4	
5	40
6	76

Quadratic probing

- For any $\alpha < ½$, quadratic probing will find an empty slot; for bigger α, quadratic probing *may* find a slot
- Quadratic probing does not suffer from primary clustering: keys hashing to the same area are not bad

- But what about keys that hash to the same spot?
  - Secondary Clustering!

Double Hashing

- $f(i) = i * g(k)$
  - where g is a second hash function
- Probe sequence:
  - 0^{th} probe = h(k) mod TableSize
  - 1^{th} probe = (h(k) + g(k)) mod TableSize
  - 2^{th} probe = (h(k) + 2*g(k)) mod TableSize
  - 3^{th} probe = (h(k) + 3*g(k)) mod TableSize
  - . . .
  - i^{th} probe = (h(k) + i*g(k)) mod TableSize

Double hashing example

i^{th} probe = (h(k) + i*g(k)) mod TableSize

h(k) = k mod 7 and g(k) = 5 – (k mod 5)

	76	93	40	47	10	55
0						
1				47	47	47
2		93	93	93	93	93
3					10	10
4						55
5			40	40	40	40
6	76	76	76	76	76	76
Probes	0	0	0	1	0	1

Double hashing example

i^{th} probe = (h(k) + i*g(k)) mod TableSize

h(k) = k mod 7 and g(k) = 5 – (k mod 5)

47 mod 7 = 5 -> collision
(5 + 1*(5-(47 mod 5)) mod 7 =
(5 + 1*(5-2) mod 7 = 8 mod 7 = 1

	76	93	40	47	10	55
0						
1				47	47	47
2		93	93	93	93	93
3					10	10
4						55
5			40	40	40	40
6	76	76	76	76	76	76
Probes	0	0	0	1	0	1

Rehashing

- When the table gets too full, create a bigger table (usually 2x as large) and hash all the items from the original table into the new table
- When to rehash?
  - half full ($\alpha = 0.5$)
  - when an insertion fails
  - some other threshold
- Cost of rehashing
  - Linear

Rehashing

- Go through old hash table, ignoring items marked deleted
- Recompute hash value for each non-deleted key and put the item in new position in new table
- Cannot just copy data from old table because the bigger table has a new hash function

- Running time: O(n) – but infrequent.
  - But not good for real-time safety critical applications

What about Python?

- In Python dictionaries are used as hash tables, but what actually happens is:
  - Python dictionaries are implemented as hash tables.
  - Each slot in the table can store one and only one entry.
  - Python dict uses open addressing to resolve hash collisions
  - A logical representation of a Python hash table:

```
-+-----------------+
0| <hash|key|value>|
-+-----------------+
1|      ...        |
-+-----------------+
.|      ...        |
-+-----------------+
i|      ...        |
-+-----------------+
.|      ...        |
-+-----------------+
n|      ...        |
-+-----------------+
```

What about Python?

- When a new dict is initialized it starts with 8 slots
- If the slot is occupied, Python compares the hash AND the key (== comparison not the is comparison) of the entry in the slot against the key of the current entry to be inserted. If both match, then it thinks the entry already exists, gives up and moves on to the next entry to be inserted. If either hash or the key don't match, it starts probing…
- Python uses *random probing*. In random probing, the next slot is picked in a pseudo random order. The entry is added to the first empty slot. —> https://hg.python.org/cpython/file/52f68c95e025/Objects/dictobject.c
- The dict will be resized if it is two-thirds full.