Hashing

Dictionary

- **Dictionary:**
  - Dynamic-set data structure for storing items indexed using *keys*.
  - Supports operations Insert, Search, and Delete.
  - Applications:
    - Symbol table of a compiler.
    - Memory-management tables in operating systems.
    - Large-scale distributed systems.
- **Hash Tables:**
  - Effective way of implementing dictionaries.
  - Generalization of ordinary arrays.

Direct-address Tables

- Direct-address Tables are ordinary arrays.
- Facilitate direct addressing.
  - Element whose key is k is obtained by indexing into the k^{th} position of the array.
- Applicable when we can afford to allocate an array with one position for every possible key.
  - i.e. when the universe of keys U is small.
- Dictionary operations can be implemented to take $O(1)$ time.

Hash Tables

- **Notation:**
  - U – Universe of all possible keys.
  - K – Set of keys actually stored in the dictionary.
  - $|K| = n$.
- When U is very large,
  - Arrays are not practical.
  - $|K| << |U|$.
- 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.

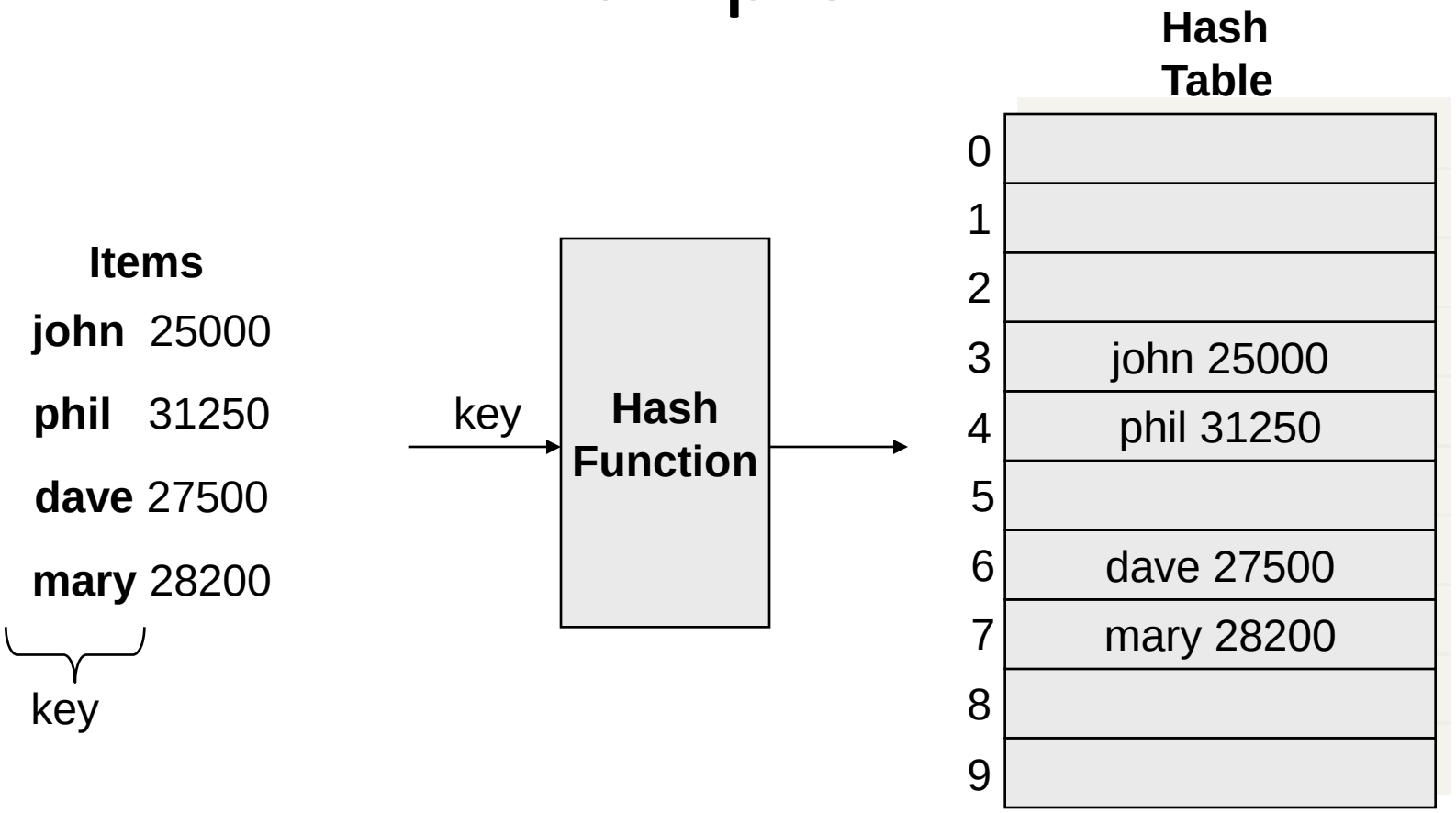
Hashing

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

$$h : U \to \{0,1,\ldots, m-1\}$$

- With arrays, key k maps to slot $A[k]$.
- With hash tables, key k maps or "hashes" to slot $T[h[k]]$.
- $h[k]$ is the *hash value* of key k.

Example


Items
john 25000
phil 31250
dave 27500
mary 28200
key
key
Hash Function
Hash Table
0
1
2
3
john 25000
4
phil 31250
5
6
dave 27500
7
mary 28200
8
9

Load Factor

- Load factor $\alpha = n/m$ = average keys per slot.
  - m – number of slots.
  - n – number of elements stored in the hash table.

- The load factor of a non-empty hash table is the number of items stored in the table divided by the size of the table.
- This is the decision parameter used when we want to rehash or expand the existing hash table entries.
- This also helps us in determining the efficiency of the hashing function. That means, it tells whether the hash function is distributing the keys uniformly or not.

Hash Function

- The good hash function:
  - must be simple to compute.
  - must distribute the keys evenly among the cells.
  - Minimize collision
  - Have a high load factor for a given set of keys
- If we know which keys will occur in advance we can write *perfect* hash functions, but we don't.

Hash function

Problems:

- Keys may not be numeric.
- Number of possible keys is much larger than the space available in table.
- Different keys may map into same location
  - Hash function is not one-to-one => collision.
  - If there are too many collisions, the performance of the hash table will suffer dramatically.

Hash Functions

- If the input keys are integers then simply *Key* mod *TableSize* is a general strategy.
  - Unless key happens to have some undesirable properties. (e.g. all keys end in 0 and we use mod 10)
- If the keys are strings, hash function needs more care.
  - First convert it into a numeric value.

Some methods

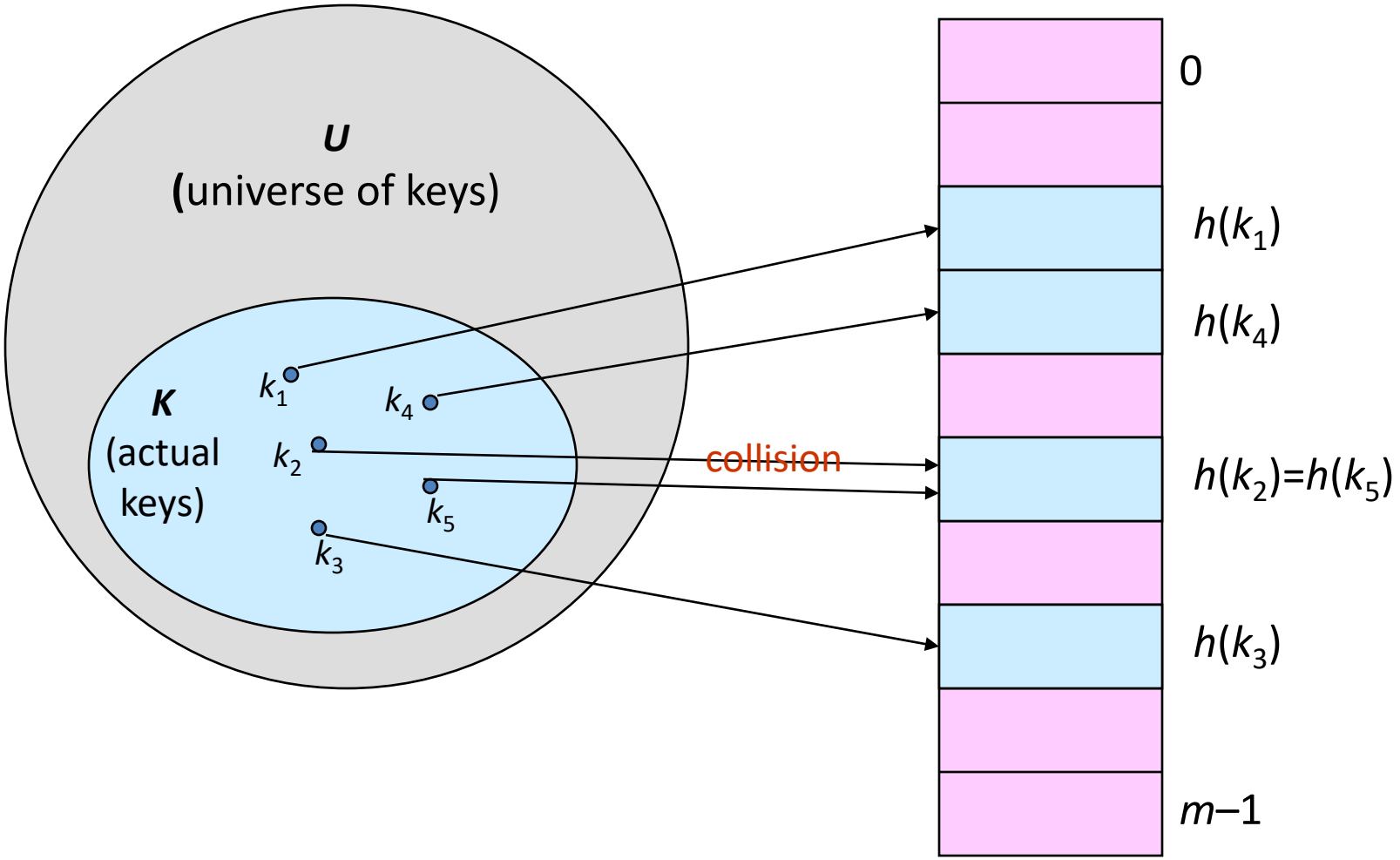
- **Truncation:**
  - e.g. 123456789 map to a table of 1000 addresses by picking 3 digits of the key.
- **Folding:**
  - e.g. 123|456|789: add them and take mod.
- **Key mod N:**
  - N is the size of the table, better if it is prime.
- **Squaring:**
  - Square the key and then truncate
- **Radix conversion:**
  - e.g. 1 2 3 4 treat it to be base 11, truncate if necessary.

How to Choose Hash Function?

The basic problems associated with the creation of hash tables are:

- An efficient hash function should be designed so that it distributes the index values of inserted objects uniformly across the table.
- An efficient collision resolution algorithm should be designed so that it computes an alternative index for a key whose hash index corresponds to a location previously inserted in the hash table.
- We must choose a hash function which can be calculated quickly, returns values within the range of locations in our table, and minimizes collisions.

Hashing

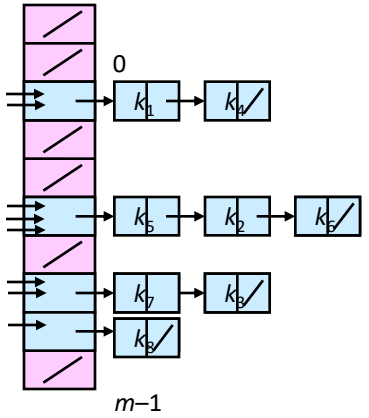

U
(universe of keys)
K
(actual
keys)
k_1
k_4
k_2
k_5
k_3
collision
0
$h(k_1)$
$h(k_4)$
$h(k_2)=h(k_5)$
$h(k_3)$
$m-1$

Issues with Hashing

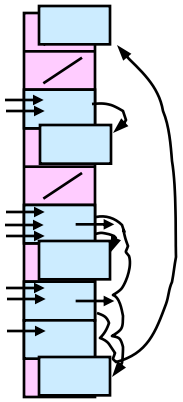
- Multiple keys can hash to the same slot – collisions are possible.
  - Design hash functions such that collisions are minimized.
  - But avoiding collisions is impossible.
    - Design collision-resolution techniques.
- Search will cost $\Theta(n)$ time in the worst case.
  - However, all operations can be made to have an expected complexity of $\Theta(1)$.

Methods of Resolution

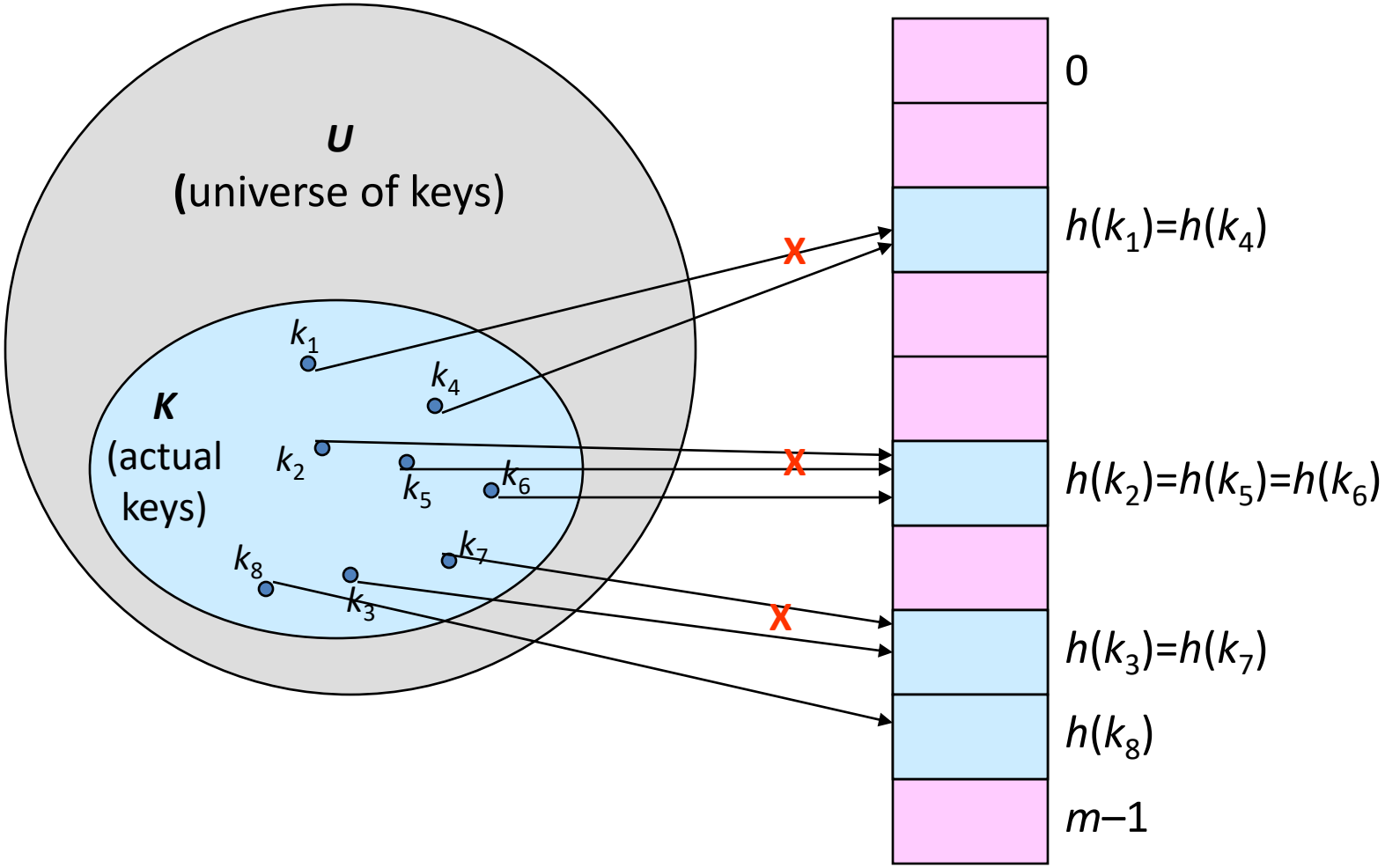
- Chaining:
  - Store all elements that hash to the same slot in a linked list.
  - Store a pointer to the head of the linked list in the hash table slot.



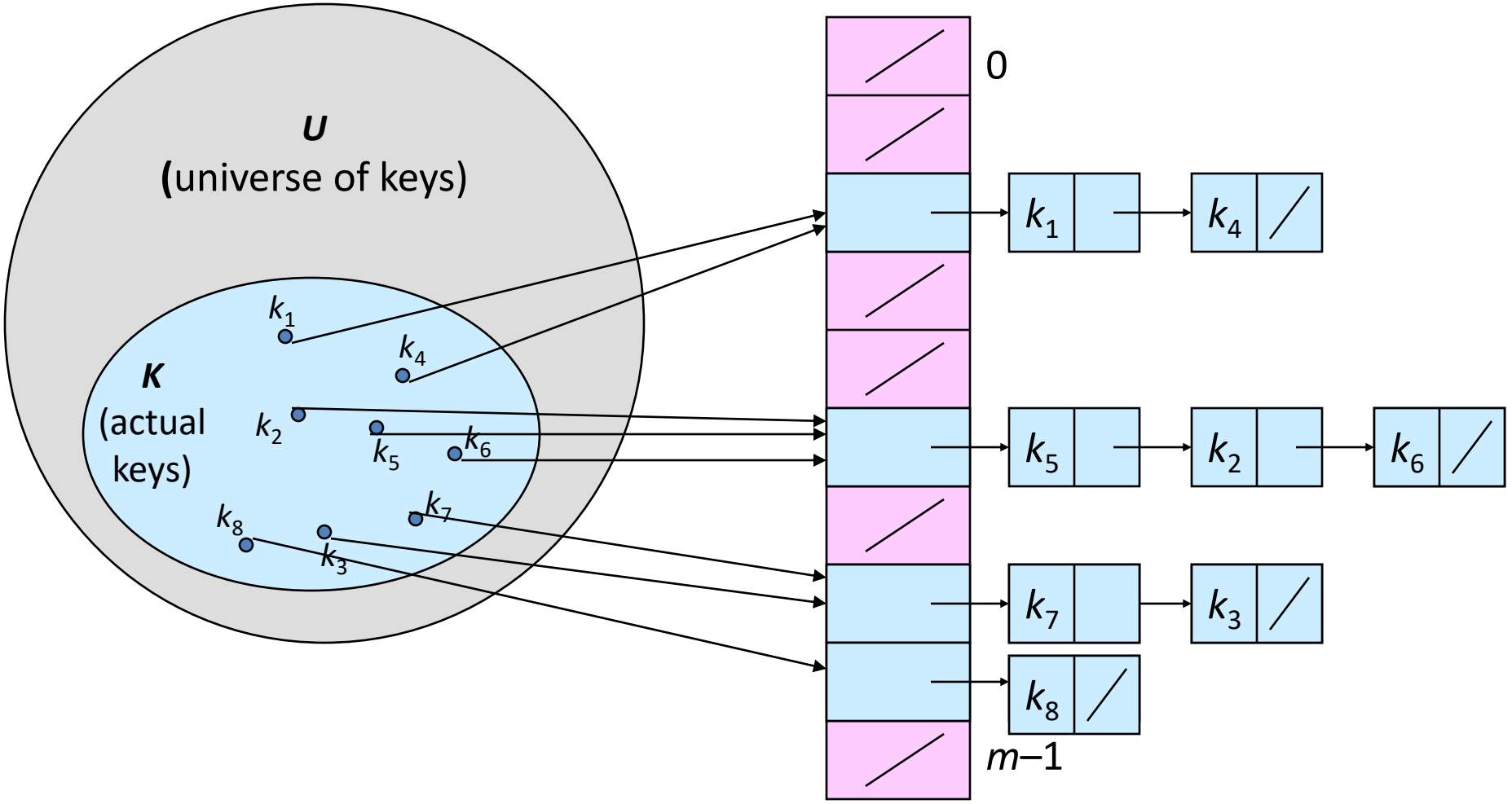
- Open Addressing:
  - All elements stored in hash table itself.
  - When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table.



Collision Resolution by Chaining



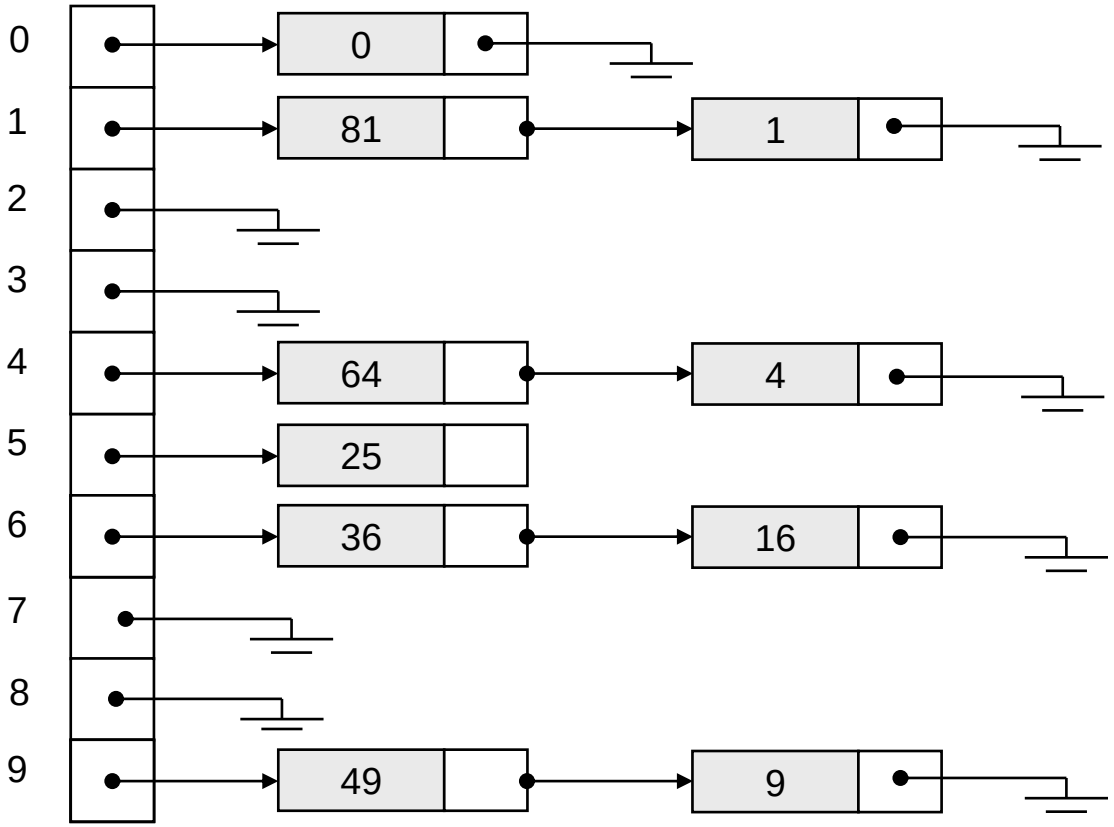
Collision Resolution by Chaining



Example

Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

hash(key) = key % 10.



Separate Chaining

- The idea is to keep a list of all elements that hash to the same value.
  - The array elements are pointers to the first nodes of the lists.
  - A new item is inserted to the front of the list.
- Advantages:
  - Better space utilization for large items.
  - Simple collision handling: searching linked list.
  - Overflow: we can store more items than the hash table size.
  - Deletion is quick and easy: deletion from the linked list.

Hashing with Chaining

Dictionary Operations:

- Chained-Hash-Insert (*T, x*)
  - Insert *x* at the head of list *T*[*h*(*key*[*x*])].
  - Worst-case complexity – *O*(1).
- Chained-Hash-Delete (*T, x*)
  - Delete *x* from the list *T*[*h*(*key*[*x*])].
  - Worst-case complexity – proportional to length of list with singly-linked lists. *O*(1) with doubly-linked lists.
- Chained-Hash-Search (*T, k*)
  - Search an element with key *k* in list *T*[*h*(*k*)].
  - Worst-case complexity – proportional to length of list.

Analysis on Chained-Hash-Search

- Load factor $\alpha = n/m$ = average keys per slot.
  - m – number of slots.
  - n – number of elements stored in the hash table.
- Worst-case complexity: $\Theta(n)$ + time to compute $h(k)$.

- Average depends on how h distributes keys among m slots.
- **Assume**
  - *Simple uniform hashing*.
    - Any key is equally likely to hash into any of the m slots, independent of where any other key hashes to.
  - $O(1)$ time to compute $h(k)$.
- Expected length of a linked list = load factor = $\alpha = n/m$.
- Search takes expected time $\Theta(1+\alpha)$.

Expected Cost – Interpretation

- If $n = O(m)$, then $\alpha = n/m = O(m)/m = O(1)$.

  $\Rightarrow$ Searching takes constant time on average.

- Insertion is $O(1)$ in the worst case.
- Deletion takes $O(1)$ worst-case time when lists are doubly linked.
- Hence, all dictionary operations take $O(1)$ time on average with hash tables with chaining.

Hashing: Open Addressing

Collision Resolution with 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.
  - If a collision occurs, alternative cells are tried until an empty cell is found.

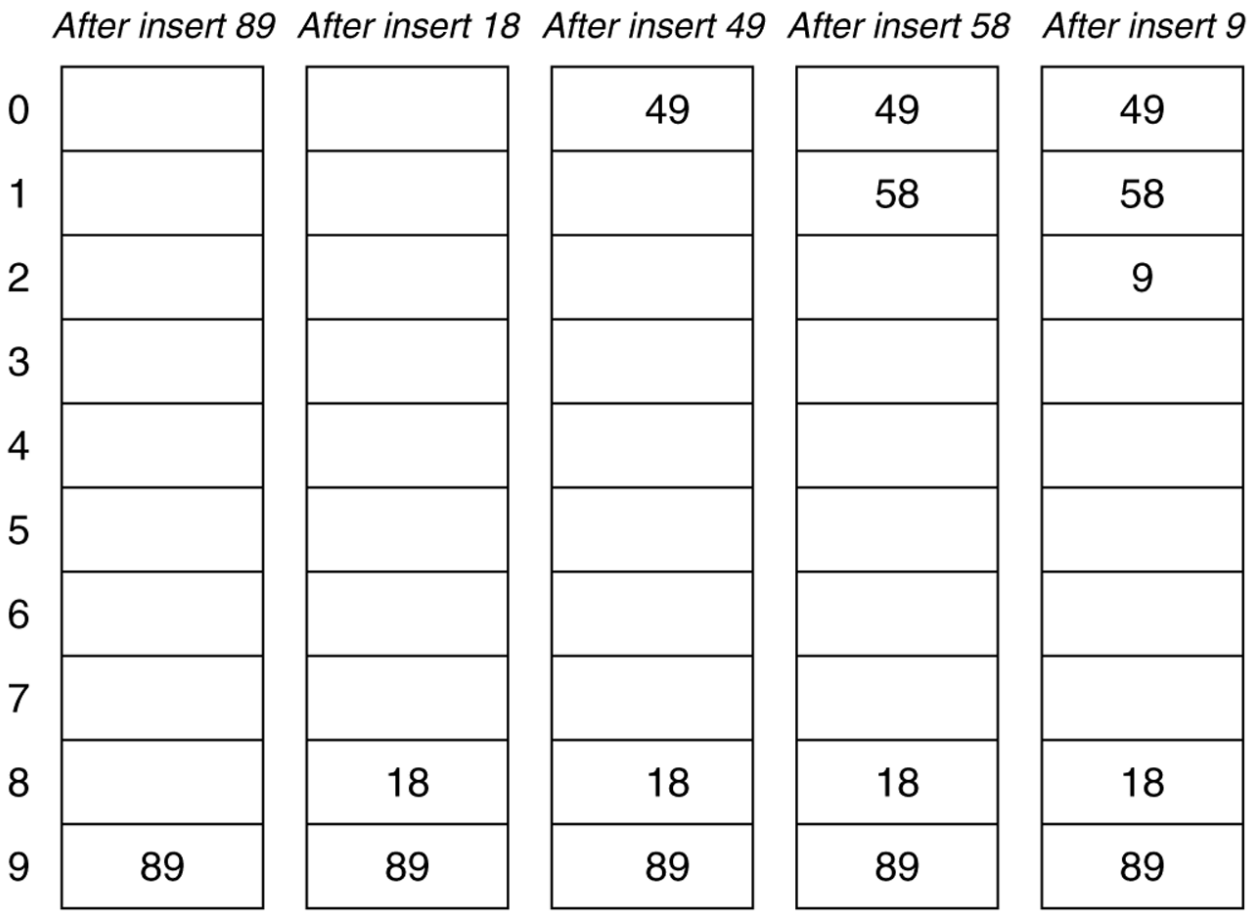
Open Addressing

- 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,* 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

- 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 hash(x) = x mod 10.
    - *f(i) = i*;

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



Linear Probing Example

Insertion Sequence: 34,55,12,8,45,37,32,88,98,54,21,42,56,74,52,33,16 del 56 Ins 75

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
74	21	42	52	33	45	16	-1	8	88	-1	-1	12	32	34	55	54	37	98	56/75
7	1	1	12	12	1	11	-1	1	2	-1	-1	1	2	1	1	3	1	1	4

Find and Delete

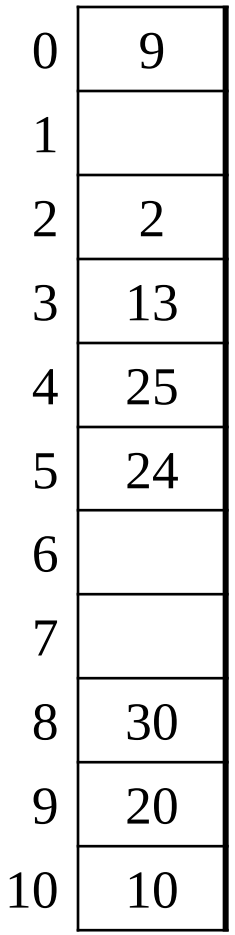
- The find algorithm follows the same probe sequence as the insert algorithm.
  - A find for 54 would involve 3 probes.
  - A find for 74 would involve 7 probes.
- We must use *lazy deletion* (i.e. marking items as deleted)
  - Standard deletion (i.e. physically removing the item) cannot be performed.
  - e.g. remove 89 from hash table.

Clustering Problem

- As long as table is big enough, a free cell can always be found, but the time to do so can get quite large.
- Worse, even if the table is relatively empty, blocks of occupied cells start forming.
- This effect is known as *primary clustering*.
- Any key that hashes into the cluster will require several attempts to resolve the collision, and then it will add to the cluster.

Linear Probing – Analysis -- 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



Successful Search:

- 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 for SS = (1+1+1+2+2+4+1+3)/8=15/8

Unsuccessful Search:

- 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

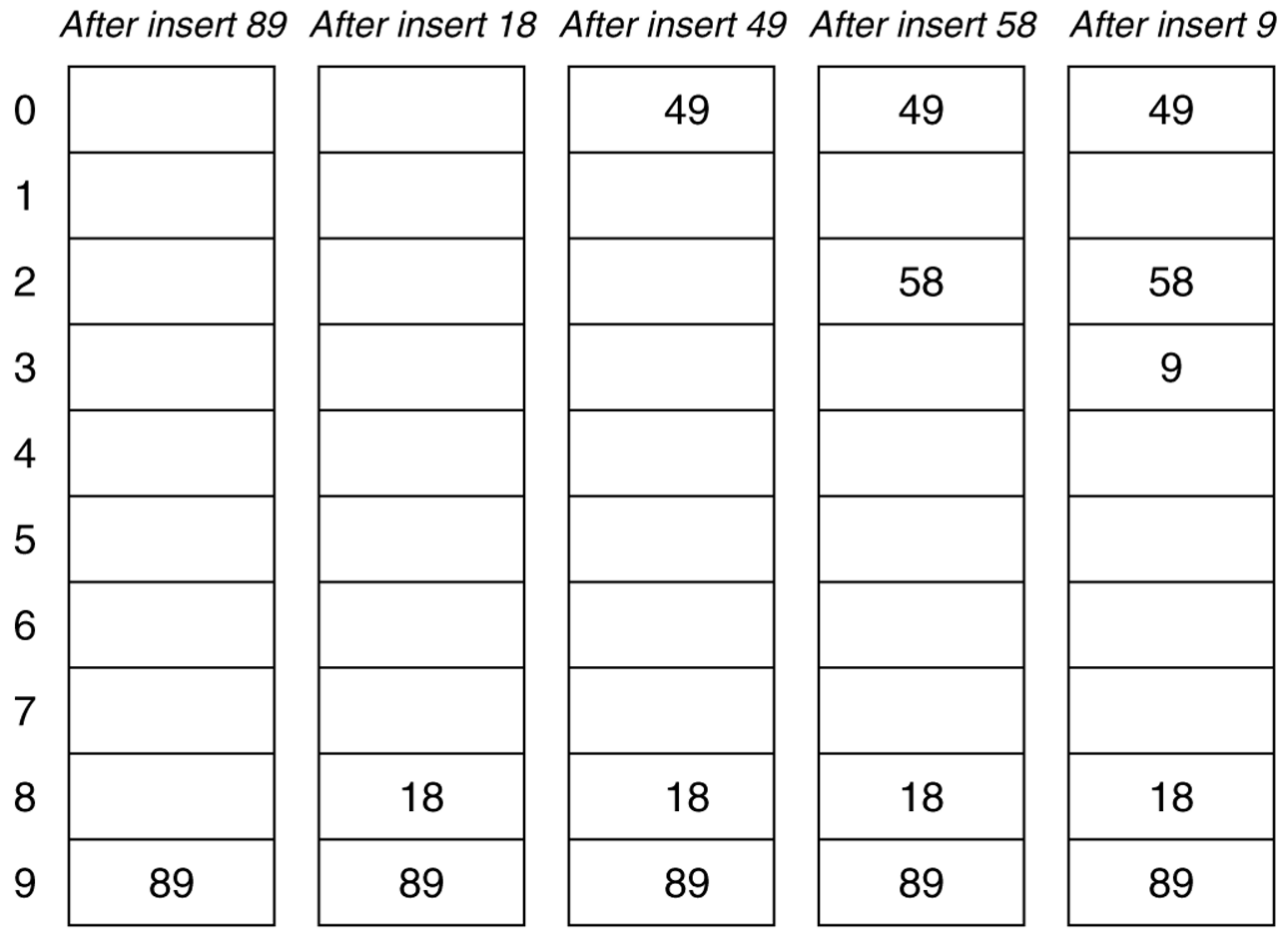
Avg. Probe for US =

(2+1+5+4+3+2+1+1+5+4+3)/11=31/11

Quadratic Probing

- Quadratic Probing eliminates primary clustering problem of linear probing.
- Collision function is quadratic.
  - The popular choice is $f(i) = i^2$.
- If the hash function evaluates to h and a search in cell h is inconclusive, we try cells $h + 1^2$, $h+2^2$, … $h + i^2$.
  - i.e. It examines cells 1,4,9 and so on away from the original probe.
- Remember that subsequent probe points are a quadratic number of positions from the *original probe point*.

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



Quadratic Probing

- Problem:
  - We may not be sure that we will probe all locations in the table (i.e. there is no guarantee to find an empty cell if table is more than half full.)
  - If the hash table size is not prime this problem will be much severe.
- However, there is a theorem stating that:
  - If the table size is *prime* and load factor is not larger than 0.5, all probes will be to different locations and an item can always be inserted.

Example

Insertion Sequence: 34,55,12,8,45,37,32,88,98,54,21,42,56,74,52

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
-1	21	42	54	-1	45	-1	-1	8	88	74	-1	12	32	34	55	56	37	98	-1
-1	1	1	4	-1	1	-1	-1	1	2	5	-1	1	2	1	1	1	1	1	-1

Array Index
Key Values
Hops

52 cannot be inserted because probe sequence is already full

Analysis of Quadratic Probing

- Quadratic probing has not yet been mathematically analyzed.
- Although quadratic probing eliminates primary clustering, elements that hash to the same location will probe the same alternative cells. This is know as *secondary clustering*.
- Techniques that eliminate secondary clustering are available.
  - the most popular is *double hashing*.

Double Hashing

- A second hash function is used to drive the collision resolution.
  - $f(i) = i * hash_2(x)$
- We apply a second hash function to x and probe at a distance $hash_2(x)$, $2*hash_2(x)$, … and so on.
- The function $hash_2(x)$ must never evaluate to zero.
  - e.g. Let $hash_2(x) = x \bmod 9$ and try to insert 99 in the previous example.
- A function such as $hash_2(x) = R - (x \bmod R)$ with R a prime smaller than Table Size will work well.
  - e.g. try R = 7 for the previous example.(7 - x mode 7)

Double Hashing Example

Insertion Sequence: 34,55,12,8,45,37,32,88,98,54,21,42,56,74,52,16

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
74	98	42	-1	-1	45	16	52	8	21	-1	-1	12	88	34	55	54	37	32	56
3	2	1	-1	-1	1	3	4	1	3	-1	-1	1	2	1	1	3	1	3	2

Array Index
Key Values
Hops

f1(k) = k%20 f2k= k%6+1
for 32 32%20=12 is full so 32%6=2+1, we try every 3rd position

Thank you