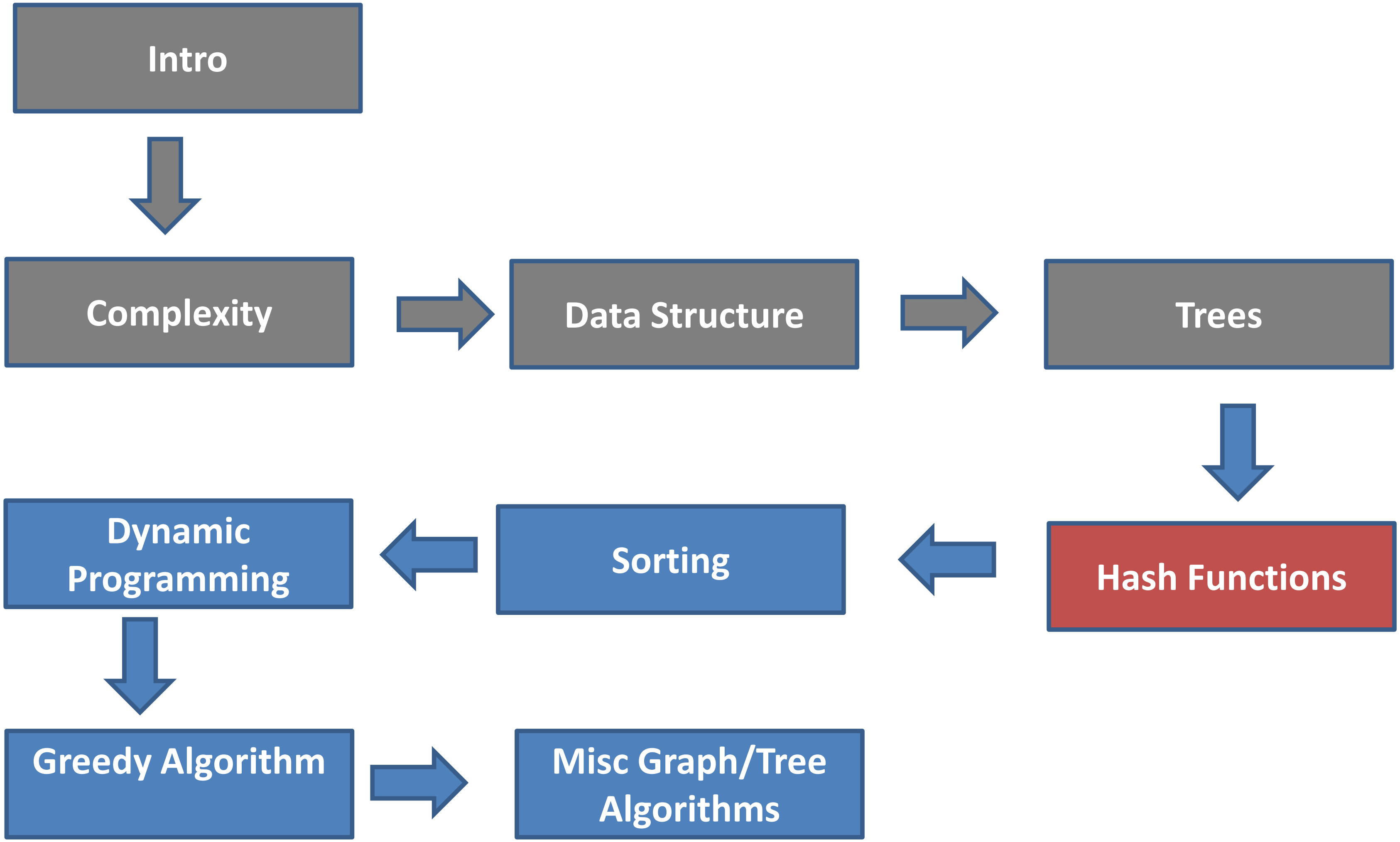
An Introduction to Algorithms

By

Hossein Rahmani

h_rahmani@iust.ac.ir
http://webpages.iust.ac.ir/h_rahmani/


Intro
Complexity
Data Structure
Trees
Hash Functions
Sorting
Dynamic Programming
Greedy Algorithm
Misc Graph/Tree Algorithms

Motivation

- Arrays provide an indirect way to access a **set**.
- Many times we need an association between two sets, or a set of **keys** and associated data.
- Ideally we would like to access this data directly with the keys.
- We would like a data structure that supports fast search, insertion, and deletion.
  - Do not usually care about sorting.
- The abstract data type is usually called a **Dictionary, Map** or **Partial Map**
  - **float googleStockPrice = stocks[“Goog”].CurrentPrice;**

Dictionaries

- What is the best way to implement this?
  - Linked Lists?
  - Double Linked Lists?
  - Queues?
  - Stacks?
  - Multiple indexed arrays (e.g., data[key[i]])?
- To answer this, ask what the <u>complexity</u> of the <u>operations</u> are:
  - Insertion
  - Deletion
  - Search

Direct Addressing

- Let's look at an easy case, suppose:
  - The range of keys is 0..m-1
  - Keys are distinct
- Possible solution
  - Set up an array T[0..m-1] in which
    - T[i] = x if $x \in T$ and key[x] = i
    - T[i] = NULL otherwise
  - This is called a *direct-address table*
    - Operations take O(1) time!
    - *So what's the problem?*

Direct Addressing

- Direct addressing works well when the range m of keys is relatively small
- But what if the keys are 32-bit integers?
  - Problem 1: direct-address table will have 2^{32} entries, more than 4 billion
  - Problem 2: even if memory is not an issue, the time to initialize the elements to NULL may be
- Solution: map keys to smaller range 0..p-1
  - Desire $p = \boldsymbol{O}(m)$.

Hash Table

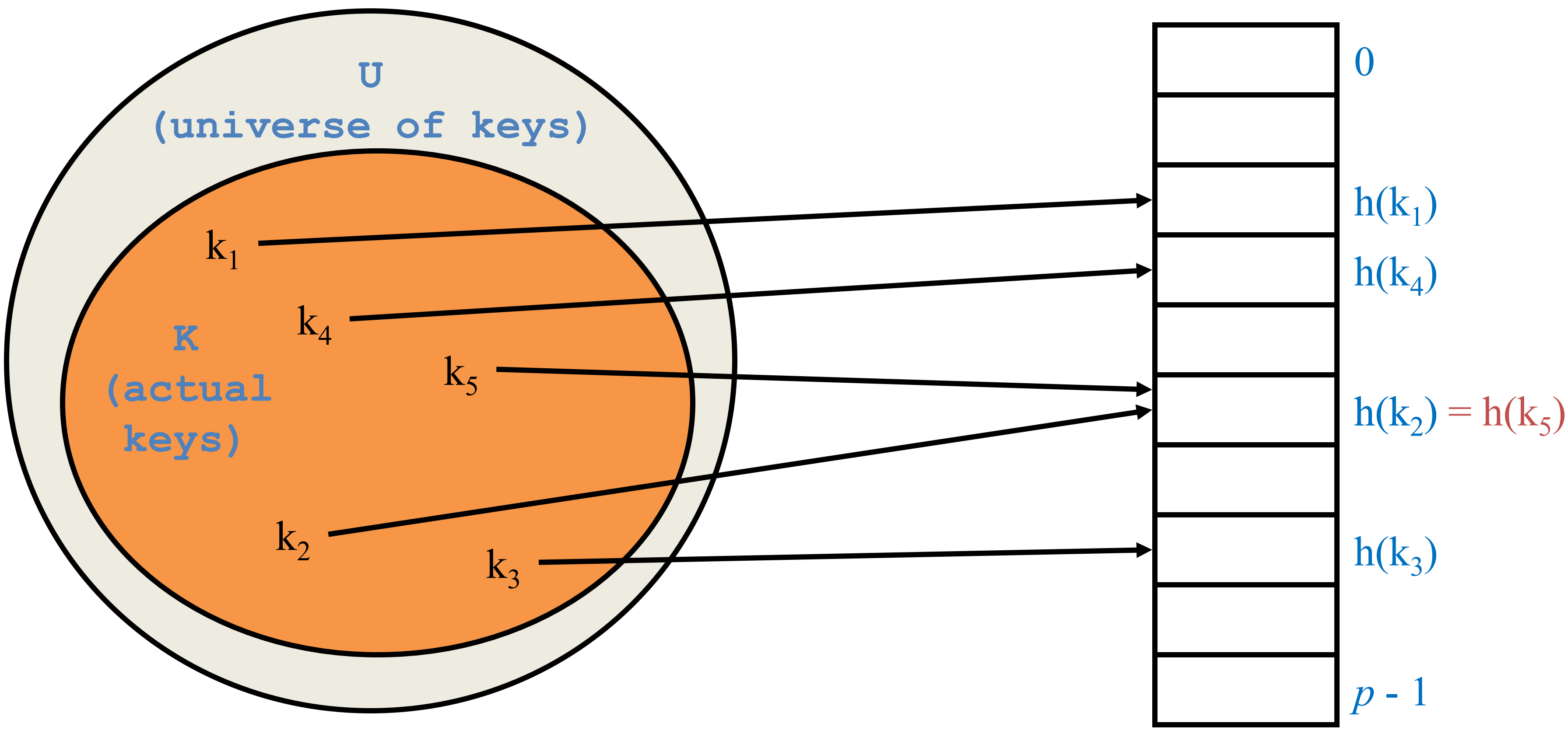
- Hash Tables provide ***O***(1) support for all of these operations!
- The key is rather than index an array directly, index it through some function, ***h***(*x*), called a *hash function*.
  - myArray[*h*(index)]
- Key questions:
  - What is the set that the *x* comes from?
  - What is *h()* and what is its range?

Hash Table

- Consider this problem:
  - If I know a priori the *p* keys from some finite set **U**, is it possible to develop a function *h(x)* that will uniquely map the *p* keys onto the set of numbers 0..*p*-1?

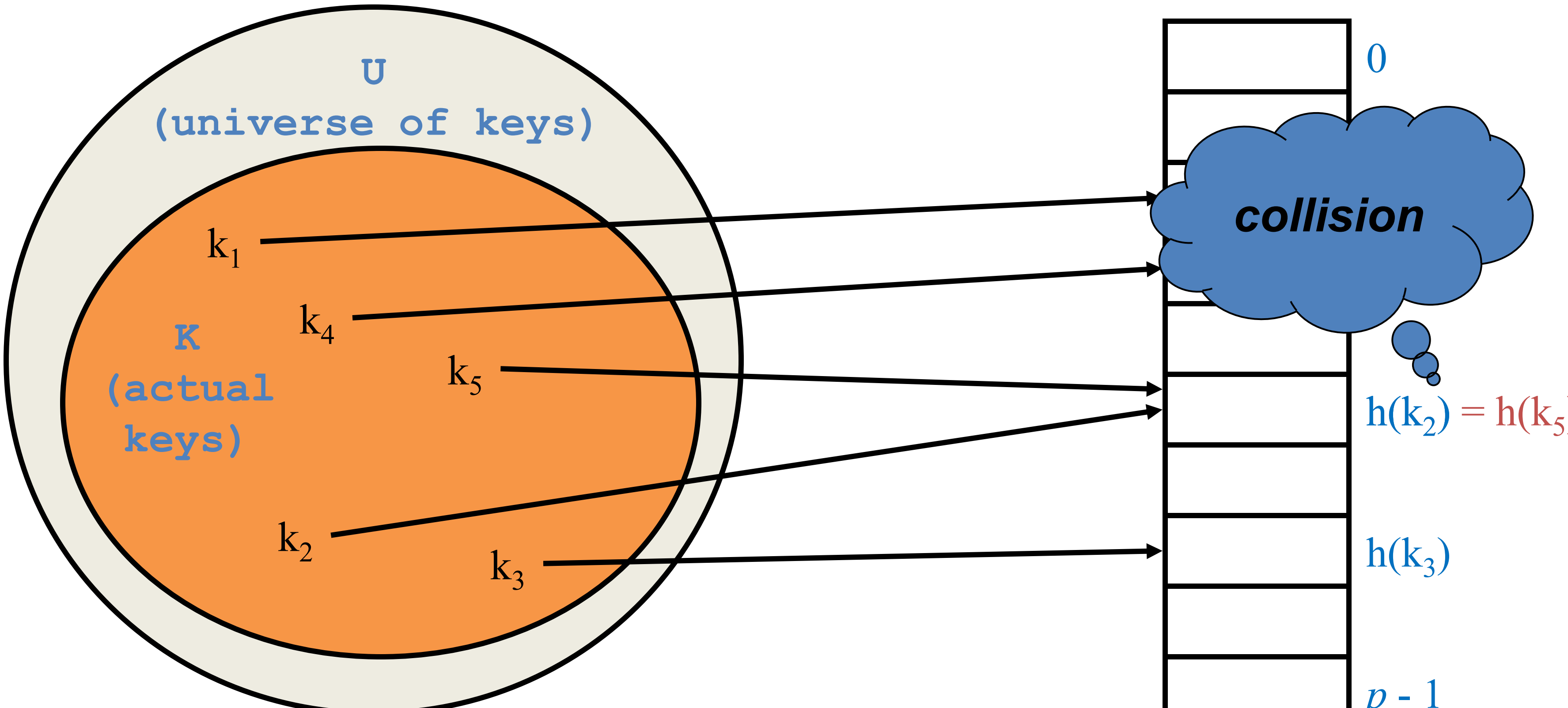
Hash Functions

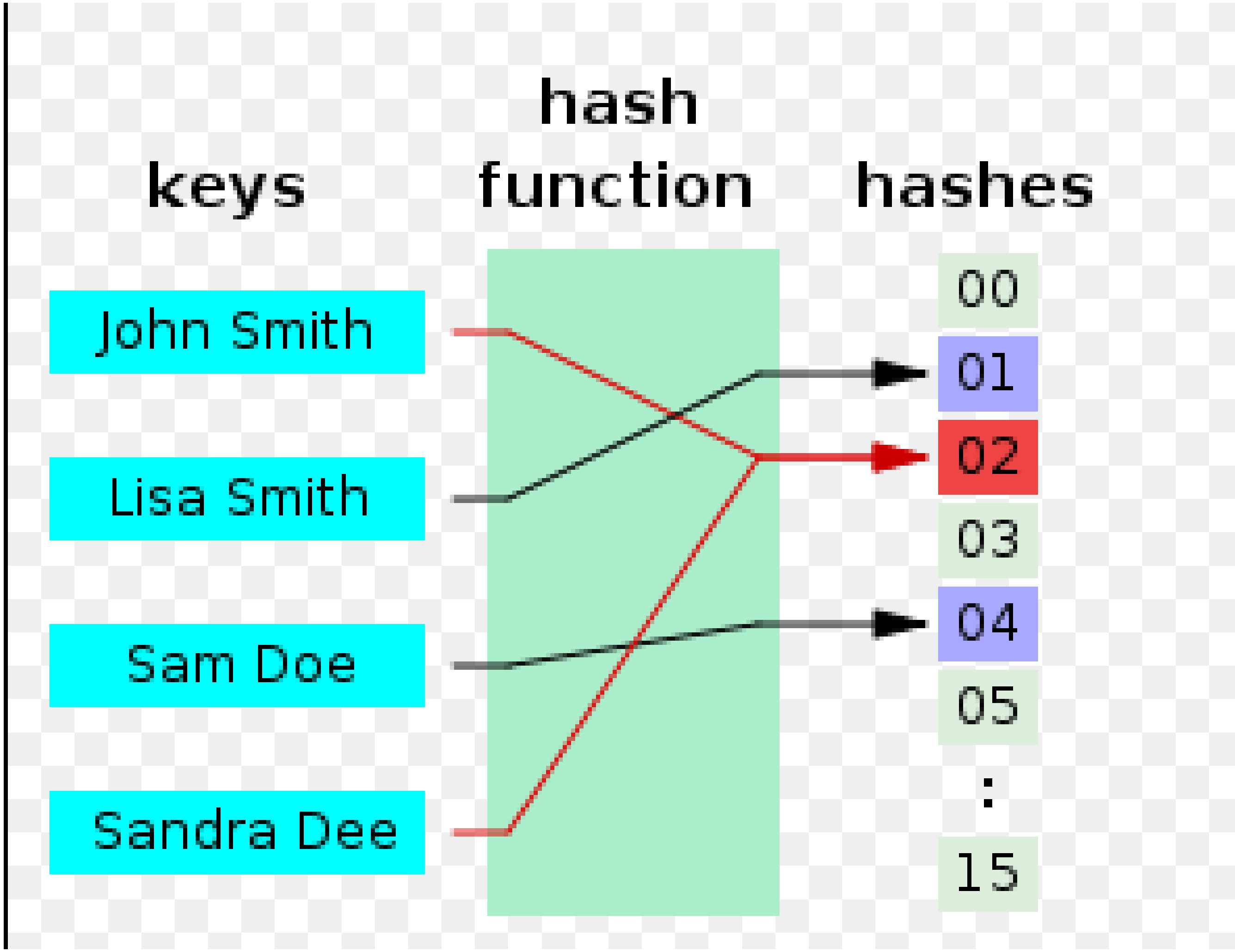
- In general a difficult problem. Try something simpler.



Hash Functions

- A **collision** occurs when *h(x)* maps two keys to the same location.



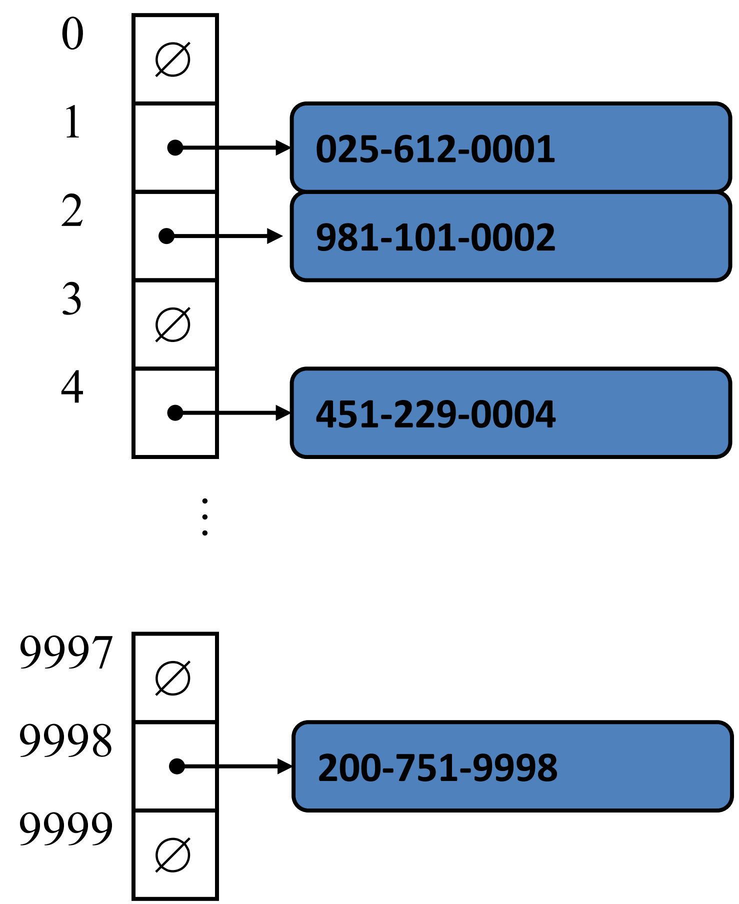

keys
hash function
hashes
John Smith
Lisa Smith
Sam Doe
Sandra Dee
00
01
02
03
04
05
:
15

Hash Functions

- A hash function, h, maps keys of a given type to integers in a fixed interval $[0, N - 1]$
- Example:
  $$h(x) = x \bmod N$$
  is a hash function for integer keys
- The integer $h(x)$ is called the hash value of x.
- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- **The goal is to store item (k, o) at index $i = h(k)$**

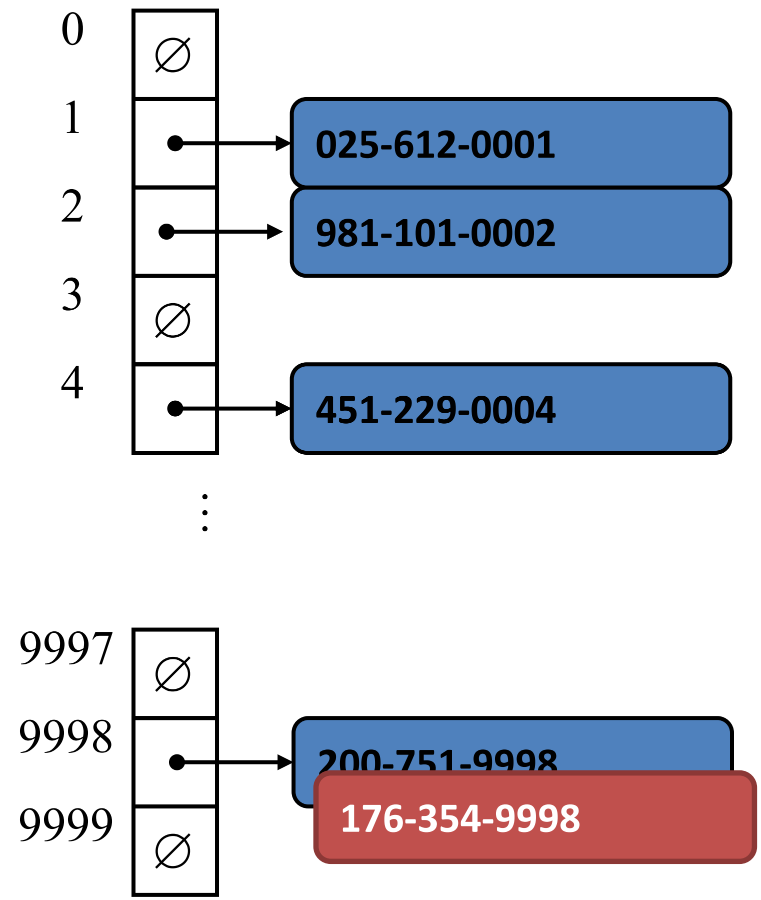
Example

- We design a hash table storing employees records using their social security number, SSN as the key.
  - SSN is a nine-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function $h(x) =$ last four digits of x



Example

- Our hash table uses an *array* of size **N = 100.**
- We have ***n* = 49** employees.
  - Need a method to handle ***collisions***.
- As long as the chance for collision is low, we can achieve this goal.
- Setting **N = 1000** and looking at the last four digits will <u>*reduce* the chance of collision</u>.

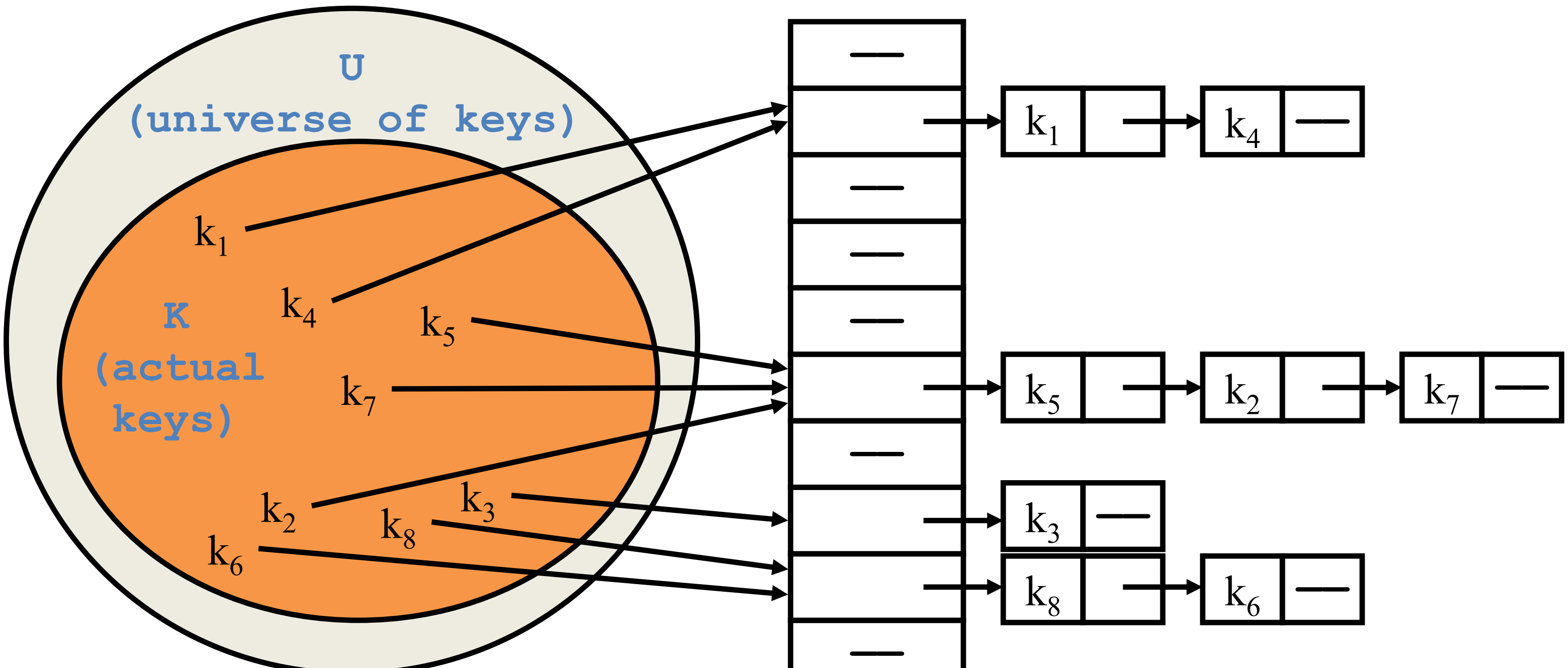


Collisions

- Can collisions be avoided?
  - If my data is immutable, yes
    - See *perfect hashing* for the case were the set of keys is static (not covered).
  - In general, no.
- Two primary techniques for resolving collisions:
  - **Chaining** – keep a collection at each key slot.
  - **Open addressing** – if the current slot is full use the ***next open*** one.

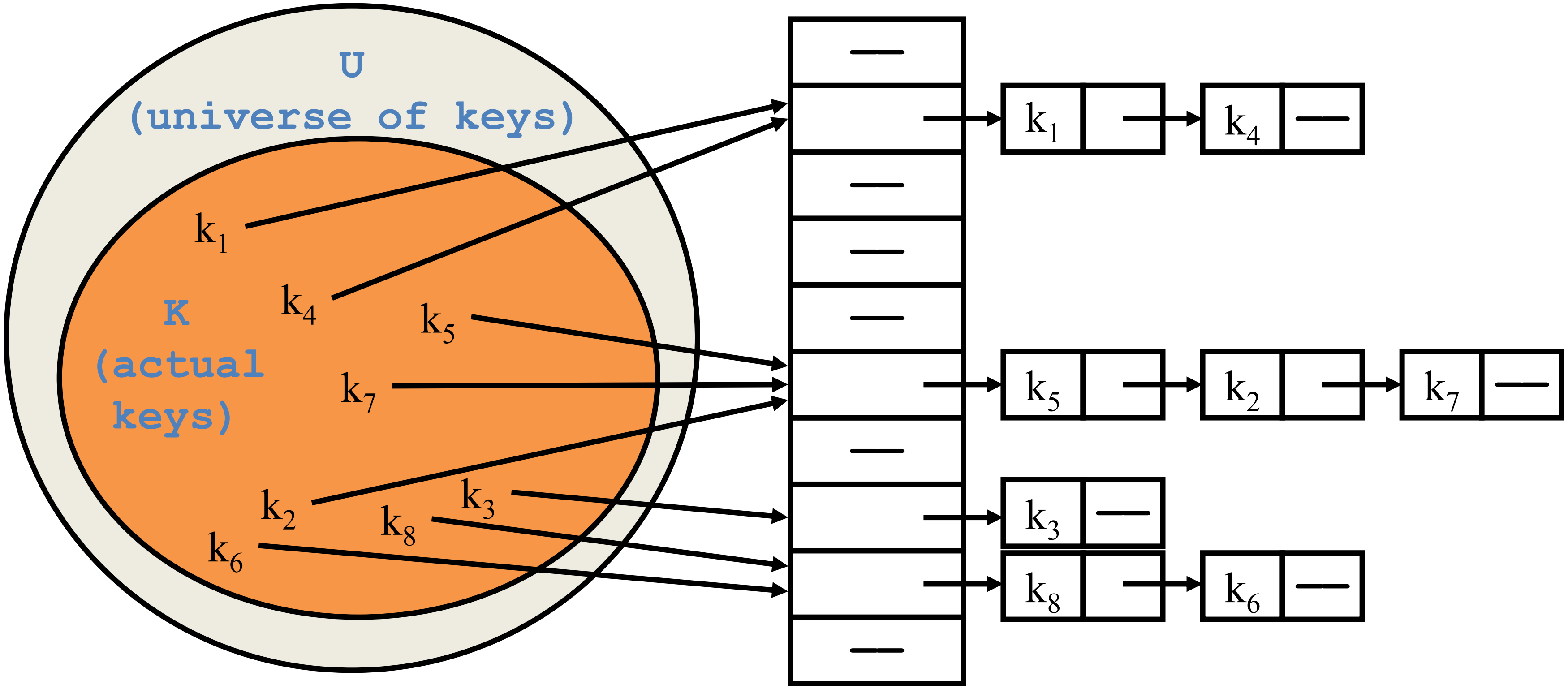
Chaining

- Chaining puts elements that hash to the same slot in a linked list:



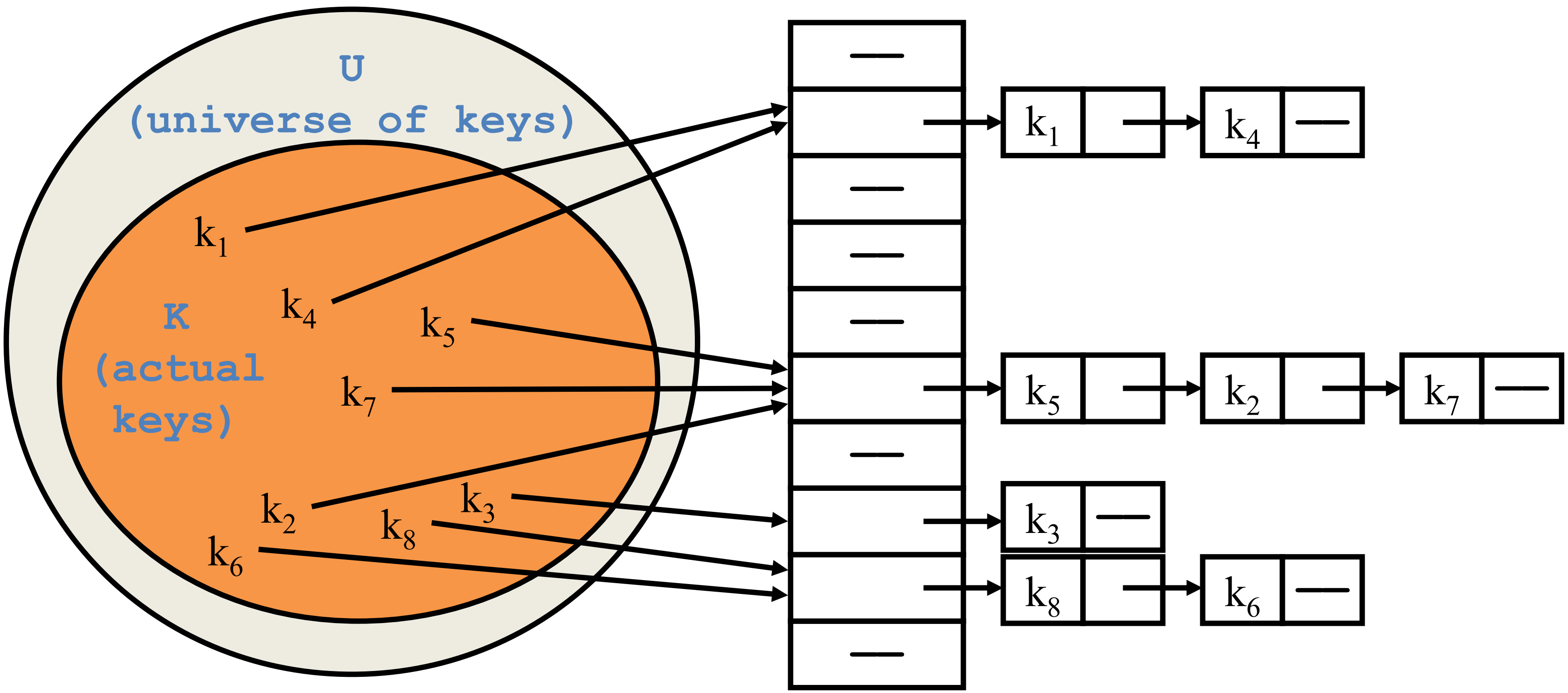
Chaining

- *How do we insert an element?*



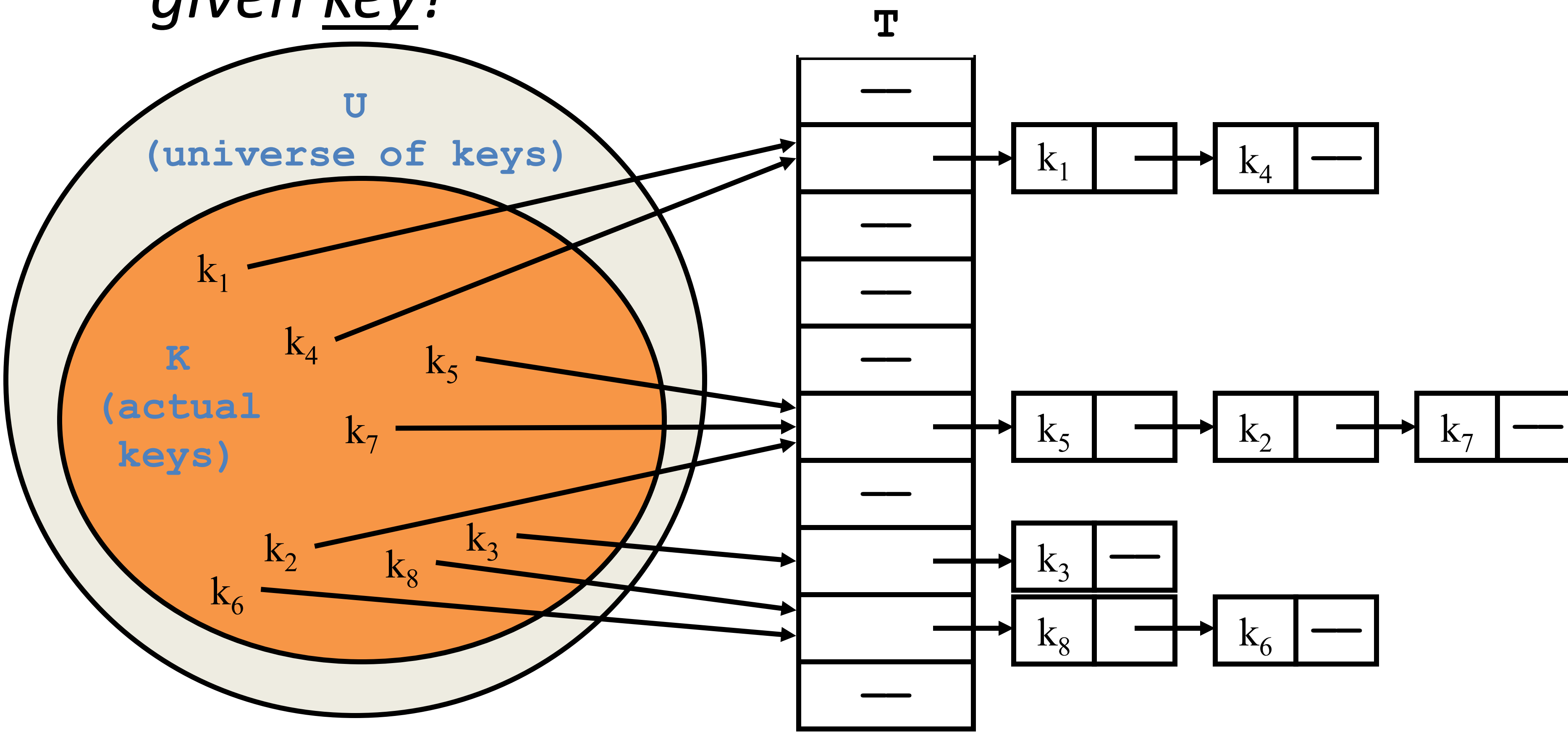
Chaining

- *How do we delete an element?*



Chaining

- *How do we __search__ for a element with a given __key__?*



Quiz 1

- Given the following input (4322, 1334, 1471, 9679, 1989, 6171, 6173, 4199) and the hash function x mod 10, which of the following statements are true? (i.) 9679, 1989, 4199 hash to the same value (ii.) 1471, 6171 has to the same value (iii.) All elements hash to the same value (iv.) Each element hashes to a different value

A	i only
B	ii only
C	i and ii only
D	iii or iv

Quiz 2

- Consider a hash table with 100 slots. Collisions are resolved using chaining. Assuming simple uniform hashing, what is the probability that the first 3 slots are unfilled after the first 3 insertions?

A	$(97 \times 97 \times 97)/100^3$
B	$(99 \times 98 \times 97)/100^3$
C	$(97 \times 96 \times 95)/100^3$
D	$(97 \times 96 \times 95)/(3! \times 100^3)$

Quiz 3

- Which one of the following hash functions on integers will distribute keys most uniformly over 10 buckets numbered 0 to 9 for i ranging from 0 to 2020?

A	$h(i) = i^2 \bmod 10$
B	$h(i) = i^3 \bmod 10$
C	$h(i) = (11 * i^2) \bmod 10$
D	$h(i) = (12 * i) \bmod 10$

Quiz 4

- Consider a hash function that distributes keys uniformly. The hash table size is 20. After hashing of how many keys will the probability that any new key hashed collides with an existing one exceed 0.5?

Quiz 5

- Given an initially empty hash table with capacity 13 and hash function H (x) = x % 13, insert these values in the order given: 5, 17, 4, 22, 31, 43, 44

Open Addressing

- Basic idea:
  - To insert: <u>if slot is full, try another slot</u>, …, until an open slot is found (***probing***)
  - To <u>search</u>, follow same sequence of <u>probes</u> as would be used when inserting the element
    - If reach element with <u>correct key</u>, return it
    - If reach a NULL pointer, element is not in table

Open Addressing

- The colliding item is placed in a different cell of the table.
  - No dynamic memory.
  - Fixed Table size.
- **Load factor:** n/N, where n is the number of items to store and N the size of the hash table.
  - Cleary, $n \le N$, or $n/N \le 1$.
- To get a reasonable performance, $n/N<0.5$.

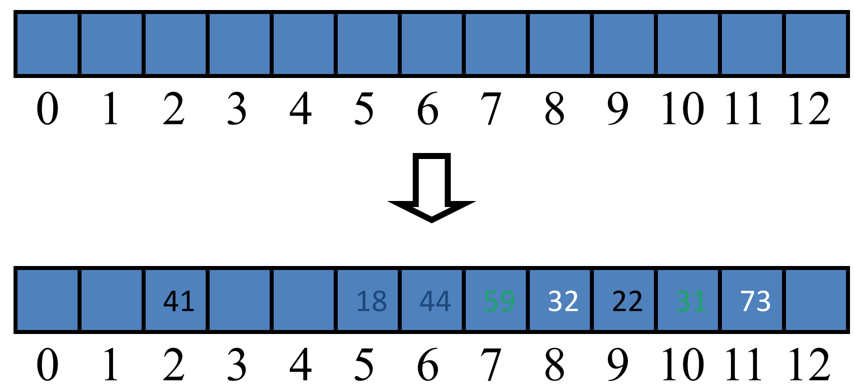
Probing

- They key question is what should the next cell to try be?
- Random would be great, but we need to be able to repeat it.
- Three common techniques:
  - Linear Probing (useful for discussion only)
  - Quadratic Probing
  - Double Hashing

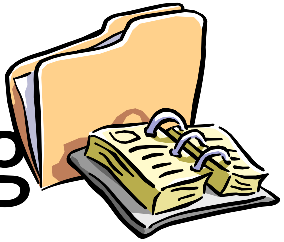
Linear Probing

- **Linear probing** handles collisions by placing the colliding item in the ***next*** (circularly) available table cell.
- Each table cell inspected is referred to as a ***probe.***
- Colliding items lump together, causing future collisions to cause a longer sequence of probes.

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



Search with Linear Probing



- Consider a hash table A that uses linear probing
- get(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - *An empty cell is found*, or
    - N cells have been unsuccessfully probed
  - To ensure the efficiency, if *k* is not in the table, we want to find an empty cell as soon as possible. The load factor can NOT be close to 1.

Algorithm *get*(k)
    $i \leftarrow h(k)$
    $p \leftarrow 0$
    repeat
        $c \leftarrow A[i]$
        if $c = \varnothing$
            return *null*
        else if $c.key() = k$
            return $c.element()$
        else
            $i \leftarrow (i + 1) \bmod N$
            $p \leftarrow p + 1$
    until $p = N$
    return *null*

Linear Probing

- Search for key=20.
  - h(20)=20 mod 13 =7.
  - Go through rank 8, 9, …, 12, 0.
- Search for key=15
  - h(15)=15 mod 13=2.
  - Go through rank 2, 3 and return null.

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, 12, 20 in this order

0	1	2	3	4	5	6	7	8	9	10	11	12

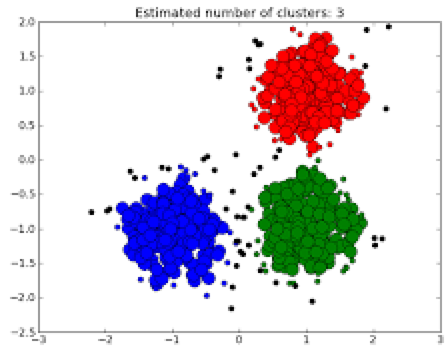
⇩

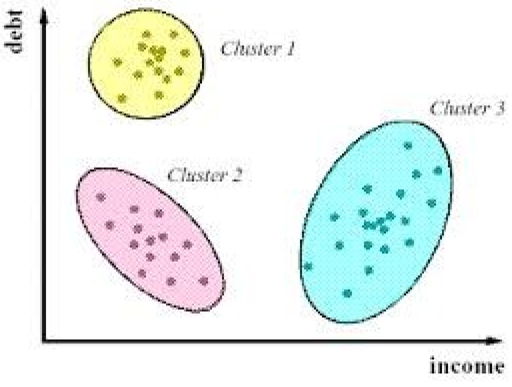
20		41			18	44	59	32	22	31	73	12
0	1	2	3	4	5	6	7	8	9	10	11	12

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called ***AVAILABLE***, which replaces deleted elements
- remove($\boldsymbol{k}$)
  - We search for an entry with key $\boldsymbol{k}$
  - If such an entry $(\boldsymbol{k}, \boldsymbol{o})$ is found, we replace it with the special item ***AVAILABLE*** and we return element $\boldsymbol{o}$
  - ***Have to modify other methods to skip available cells.***
- put($\boldsymbol{k}, \boldsymbol{o}$)
  - We throw an exception if the table is full
  - We start at cell $\boldsymbol{h}(\boldsymbol{k})$
  - We probe consecutive cells until one of the following occurs
    - A cell $\boldsymbol{i}$ is found that is either empty or stores ***AVAILABLE***, or
    - $\boldsymbol{N}$ cells have been unsuccessfully probed
  - We store entry $(\boldsymbol{k}, \boldsymbol{o})$ in cell $\boldsymbol{i}$

(Clustering)


Estimated number of clusters: 3
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3
−2
−1
0
1
2
3


debt
Cluster 1
Cluster 2
Cluster 3
income

Primary Clustering

- Primary clustering is one of major failure modes of open addressing based hash tables, especially those using linear probing.
- It occurs after a hash collision causes two of the records in the hash table to hash to the same position, and causes one of the records to be moved to the next location in its probe sequence.
- Once this happens, the cluster formed by this pair of records is more likely to grow by the addition of even more colliding records, regardless of whether the new records hash to the same location as the first two.
- This phenomenon causes searches for keys within the cluster to be longer

Quadratic Probing

- Primary <u>clustering</u> occurs with <u>linear probing</u> because the same linear pattern:
  - if a bin is inside a cluster, then the <u>next</u> bin must either:
    - also be <u>in</u> that <u>cluster</u>, or
    - <u>expand</u> the <u>cluster</u>
- Instead of searching forward in a linear fashion, try to <u>jump far enough out of the current</u> (unknown) cluster.

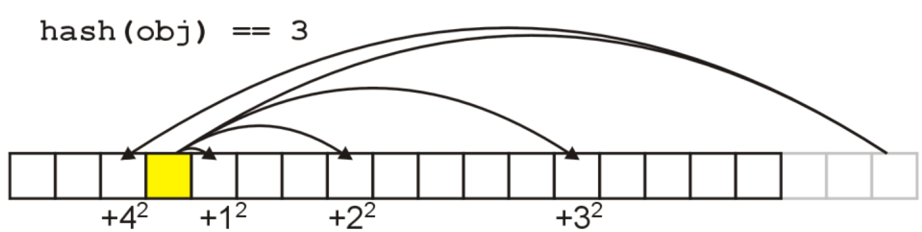
Quadratic Probing

- Suppose that an element should appear in bin h:
  - if <u>bin h</u> is occupied, then check the <u>following sequence</u> of bins:

$$h+1^2,\ h+2^2,\ h+3^2,\ h+4^2,\ h+5^2, \ldots$$

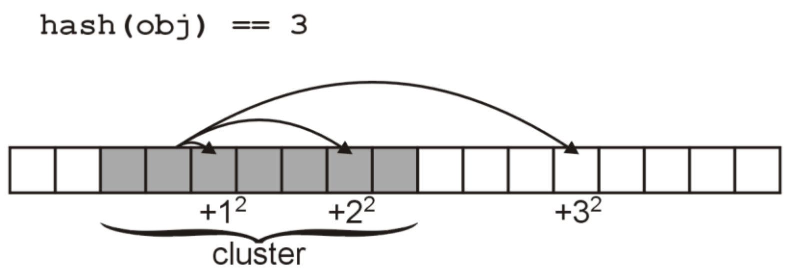
$$h+1,\ h+4,\ h+9,\ h+16,\ h+25, \ldots$$

- For example, with $\mathrm{M} = 17$:



Quadratic Probing

- If one of $h + i^2$ falls into a cluster, this does not imply the next one will



Quadratic Probing

- For example, suppose an element was to be inserted in bin 23 in a hash table with 31 bins
- The sequence in which the bins would be checked is:

23, 24, 27, 1, 8, 17, 28, 10, 25, 11, 30, 20, 12, 6, 2, 0

Quadratic Probing

- Even if two bins are initially close, the sequence in which subsequent bins are checked varies greatly
- Again, with $M = 31$ bins, compare the first 16 bins which are checked starting with 22 and 23:

22, 23, 26, 0, 7, 16, 27, 9, 24, 10, 29, 19, 11, 5, 1, 30
23, 24, 27, 1, 8, 17, 28, 10, 25, 11, 30, 20, 12, 6, 2, 0

Quadratic Probing

- Thus, quadratic probing solves the problem of primary clustering
- Unfortunately, there is a second problem which must be dealt with
- Suppose we have $\mathrm{M} = 8$ bins:

$$1^2 \equiv 1,\ 2^2 \equiv 4,\ 3^2 \equiv 1$$

- In this case, we are checking bin $h + 1$ twice having checked only one other bin

Quadratic Probing

- Unfortunately, there is no guarantee that

  $$h + i^2 \bmod \mathrm{M}$$

  will cycle through $0, 1, \ldots, \mathrm{M} - 1$
- Solution:
  - require that M be prime
  - in this case, $h + i^2 \bmod \mathrm{M}$ for $i = 0, \ldots, (\mathrm{M} - 1)/2$ will cycle through exactly $(\mathrm{M} + 1)/2$ values before repeating

Quadratic Probing

- Example with $M = 11$:

  $0, 1, 4, 9, 16 \equiv 5, 25 \equiv 3, 36 \equiv 3$

- With $M = 13$:

  $0, 1, 4, 9, 16 \equiv 3, 25 \equiv 12, 36 \equiv 10, 49 \equiv 10$

- With $M = 17$:

  $0, 1, 4, 9, 16, 25 \equiv 8, 36 \equiv 2, 49 \equiv 15, 64 \equiv 13, 81 \equiv 13$

Quadratic Probing

- Thus, quadratic probing avoids primary clustering
- Unfortunately, we are not guaranteed that we will use all the bins

Secondary Clustering

- The phenomenon of <u>primary</u> clustering will <u>not</u> occur with <u>quadratic probing</u>
- However, if <u>multiple items</u> all hash to the <u>same</u> <u>initial</u> bin, the <u>same</u> <u>sequence</u> of numbers will be followed
- This is termed <u>*secondary clustering*</u>
- The effect is <u>less significant</u> than that of primary clustering

Double Hashing

- Use two hash functions
- If **M** is prime, eventually will examine every position in the table
- double_hash_insert(K)
  if(table is full) error
  probe = h1(K)
  offset = h2(K)
  while (table[probe] occupied)
      probe = (probe + offset) mod M
  table[probe] = K

Double Hashing

- Many of same (dis)advantages as linear probing
- Distributes keys more <u>uniformly</u> than linear probing does
- Notes:
  - h2(x) should <u>never</u> return <u>zero</u>.
  - M should be prime.

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

Double Hashing: Example

$h_1(k) = k \bmod 13$

$h_2(k) = 1 + (k \bmod 11)$

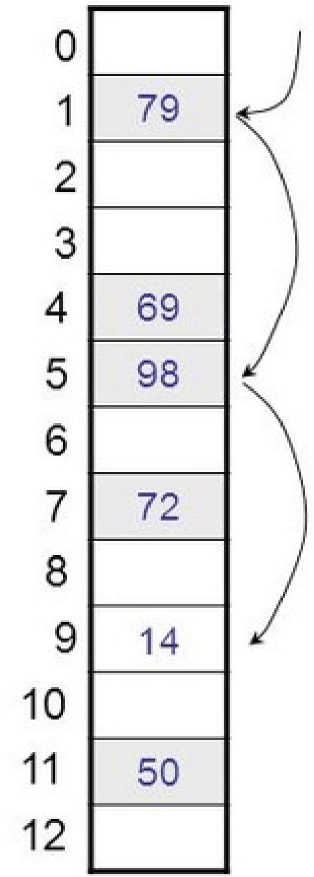
$$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod 13$$

- Insert key 14:

$h_1(14,0) = 14 \bmod 13 = 1$

$$h(14,1) = (h_1(14) + h_2(14)) \bmod 13 = (1 + 4) \bmod 13 = 5$$

$$h(14,2) = (h_1(14) + 2\, h_2(14)) \bmod 13 = (1 + 8) \bmod 13 = 9$$



Choosing A Hash Function

- Clearly choosing the hash function well is crucial.
  - *What will a worst-case hash function do?*
  - *What will be the time to search in this case?*
- *What are desirable features of the hash function?*
  - Should distribute keys uniformly into slots
  - Should not depend on patterns in the data

Quiz

- Consider inserting the keys 59, 10, 31, 88, 22, 4, 68, 28, 15, 34, 17 into a hash table of size m = 11 using open addressing with linear probing with the hash function h(k) = k mod m. What is the content of the hash tables applying the following strategies?
- A- Linear probing
- B- Quadratic probing
- C- Double hashing with hash function:

  $h(k) = (k \bmod 11 + i\,(1 + (k \bmod 8))) \bmod 11$
- D- Random probing (Discussable)