Hash Tables

Motivation

- 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** or **Map**

Dictionaries

- What is the best way to implement this?
  - Linked Lists?
  - Doubly Linked Lists?
  - Queues?
  - Stacks?
- To answer this, ask what the complexity of the operations 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] = y   if $y \in T$ and key[y] = 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

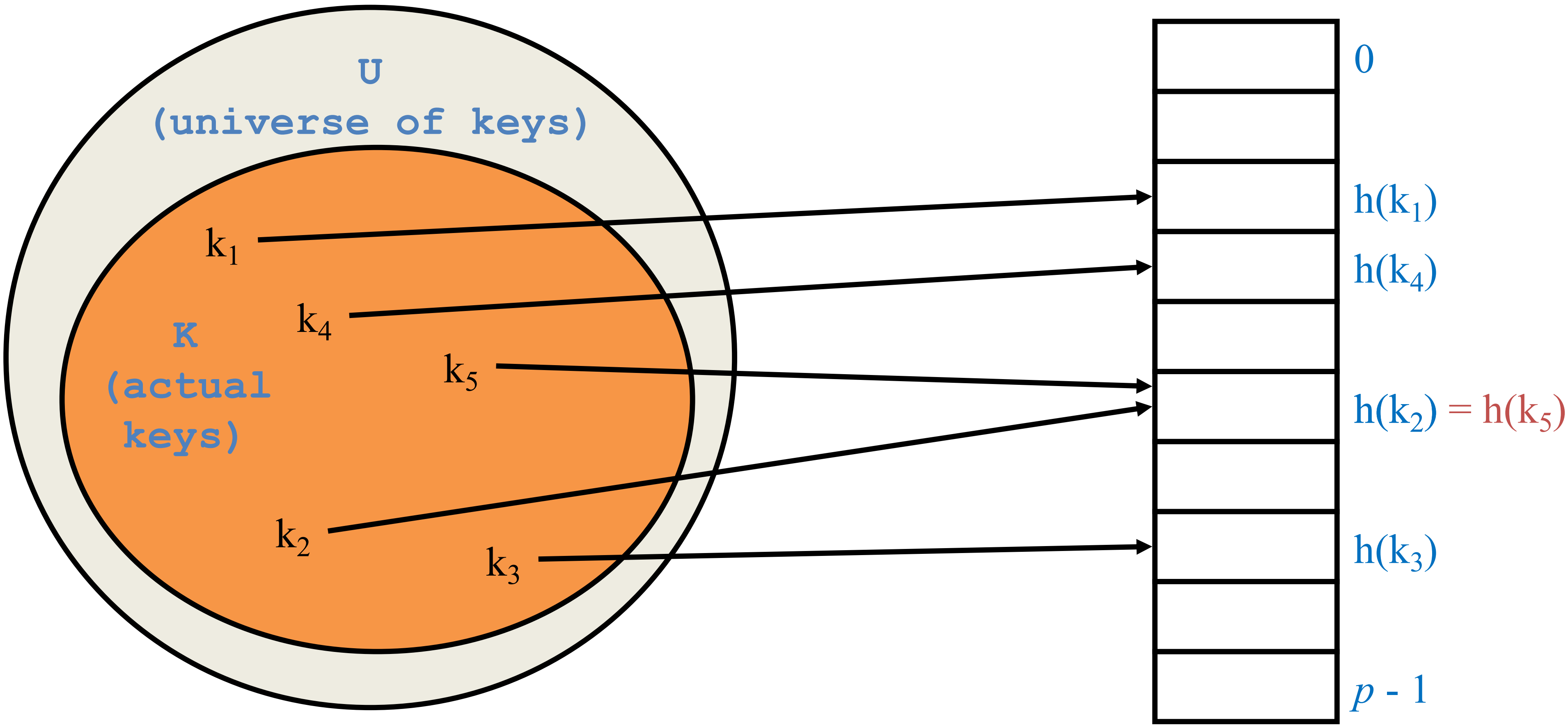
- Hash Tables provide $\boldsymbol{O}(1)$ support for all of these operations!
- The key is rather than index of array directly → index it through some function, $\boldsymbol{h}(x)$, called a *hash function*.
  - myArray[h(index)]
- Key questions:
  - What is the set that x comes from?
  - What is *h()*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

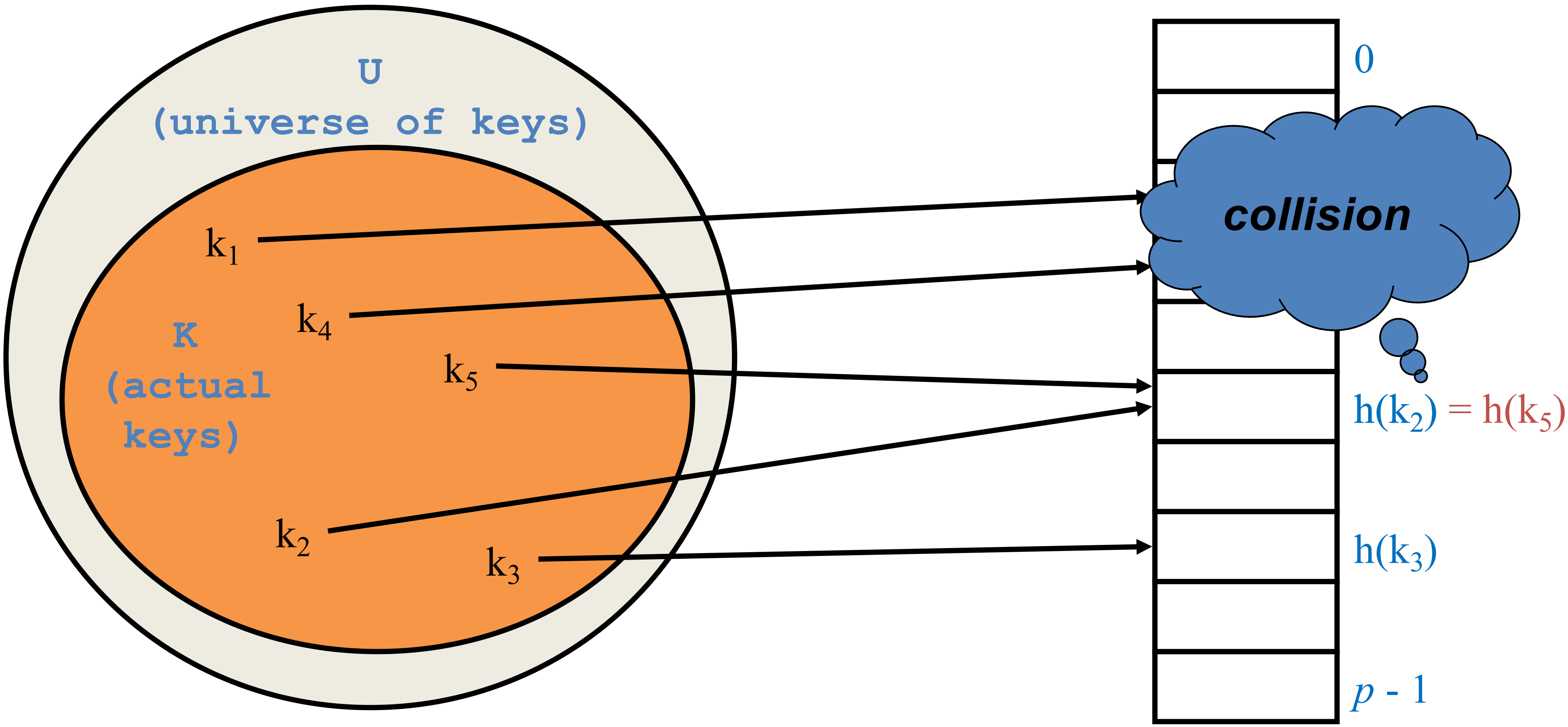
Hash Functions

- In general a difficult problem. Try something simpler.



Hash Functions

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

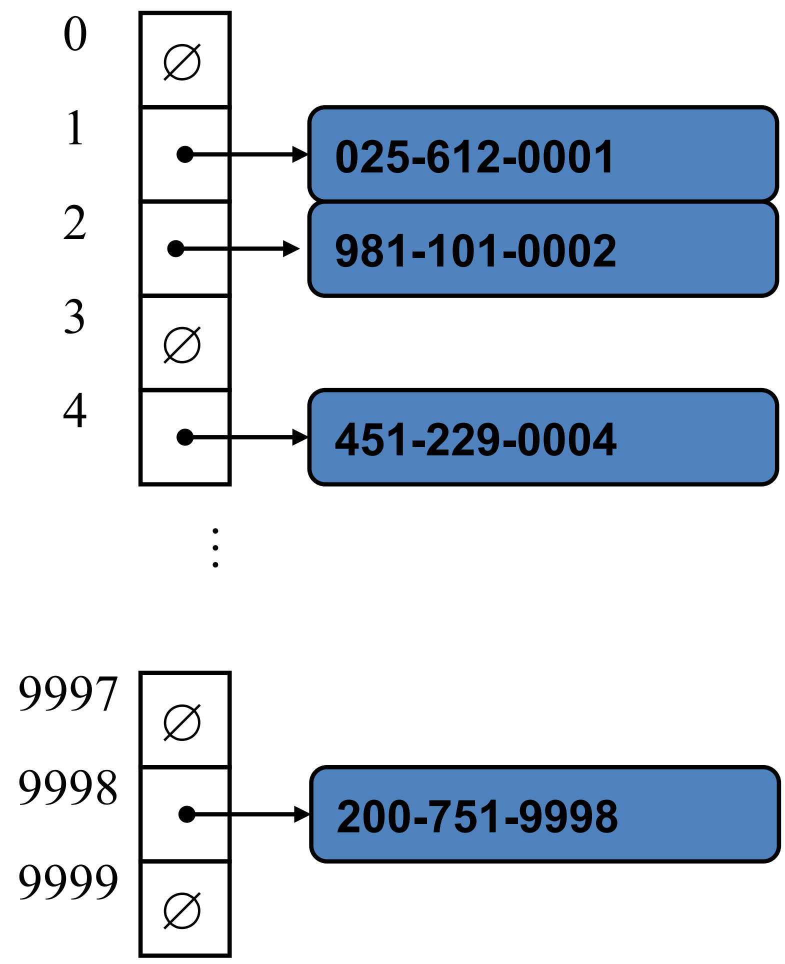


Hash Functions

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

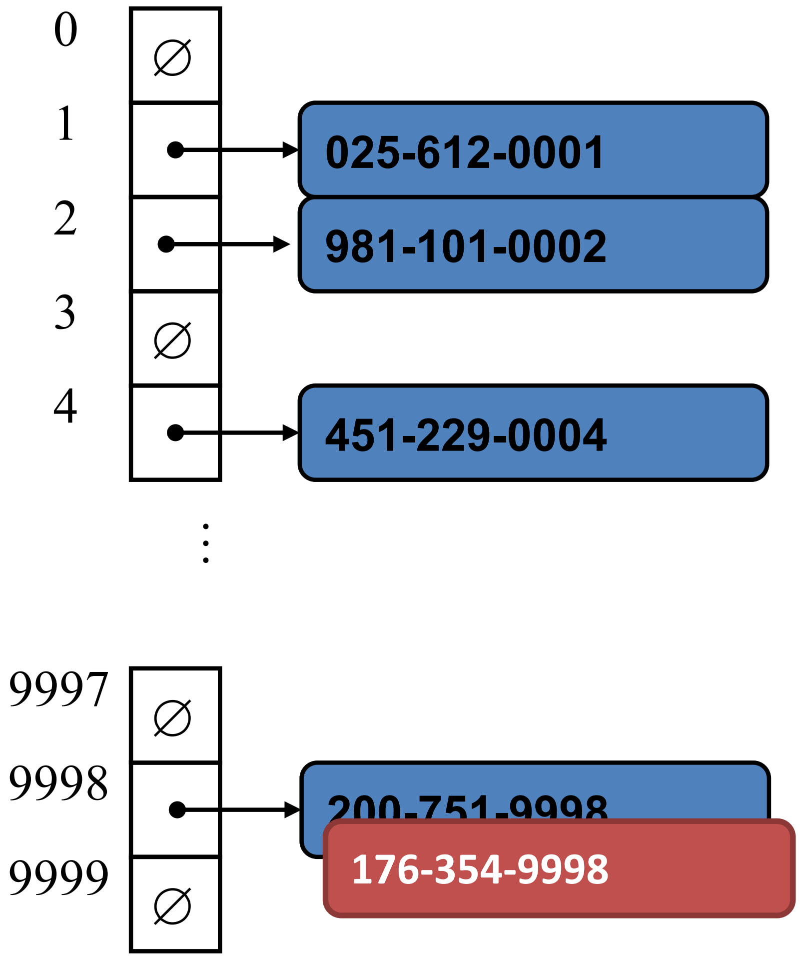
Example

- 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 $\boldsymbol{m} = 10{,}000$ and the hash function $\boldsymbol{h}(\boldsymbol{x})$ = last four digits of $\boldsymbol{x}$



Example

- Our hash table uses an *array* of size **m = 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 **m = 1000** and looking at the last four digits will *reduce* the chance of collision.

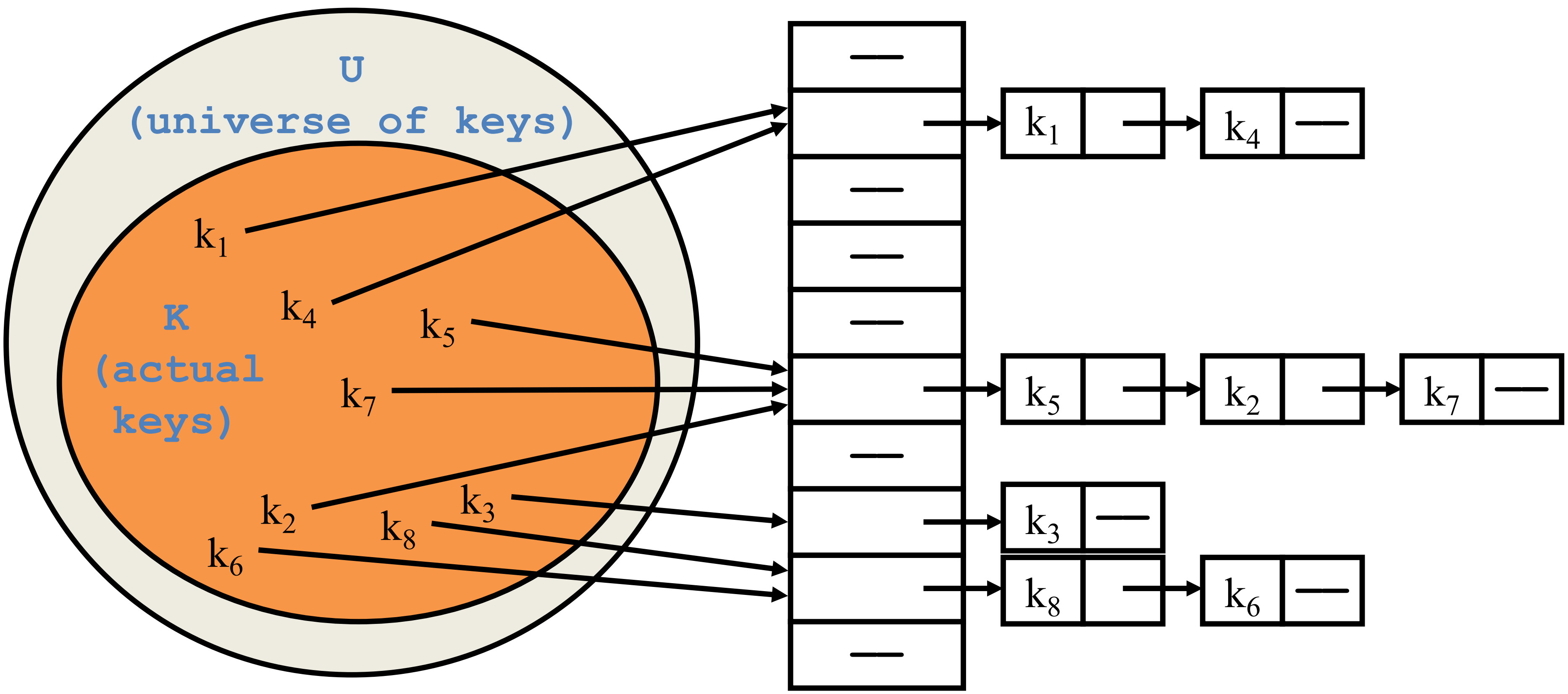


Collisions

- Can collisions be avoided?
  - 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.

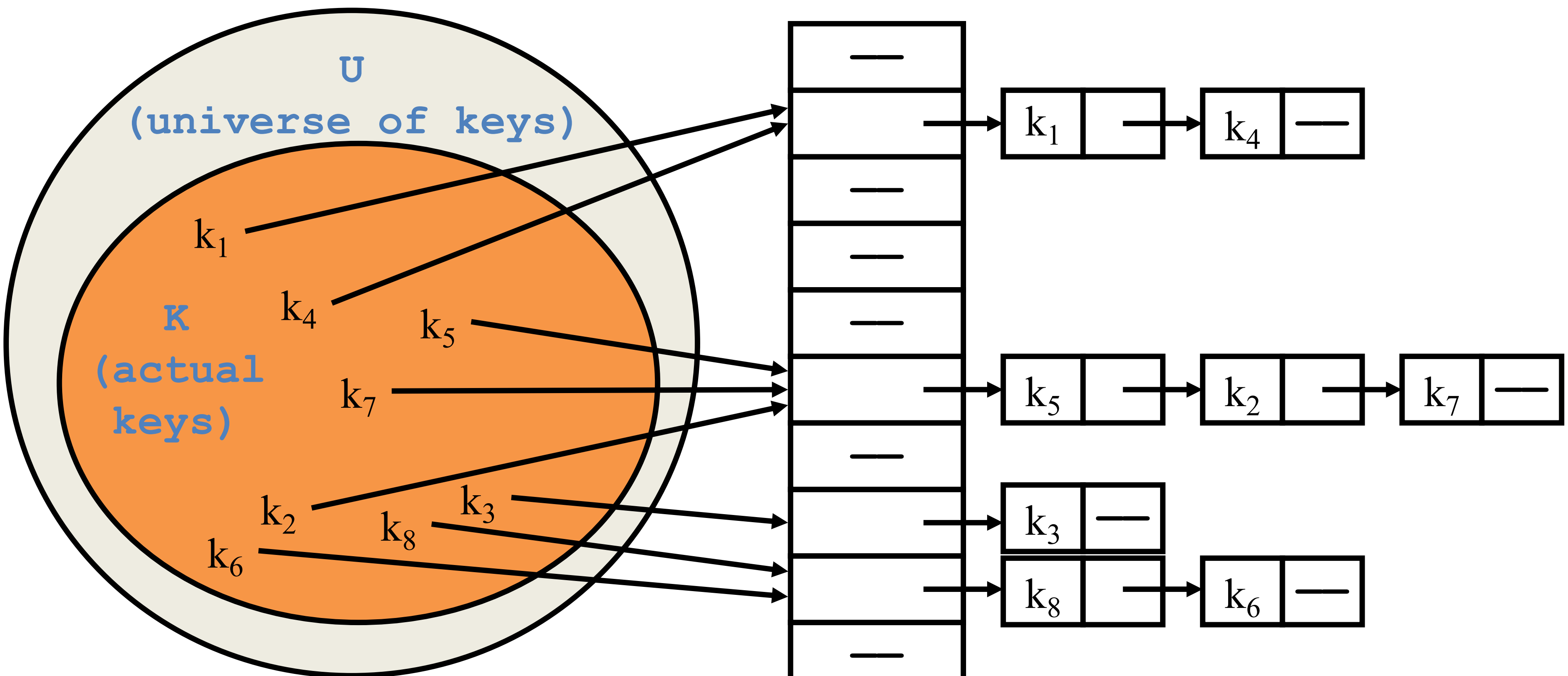
Chaining

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



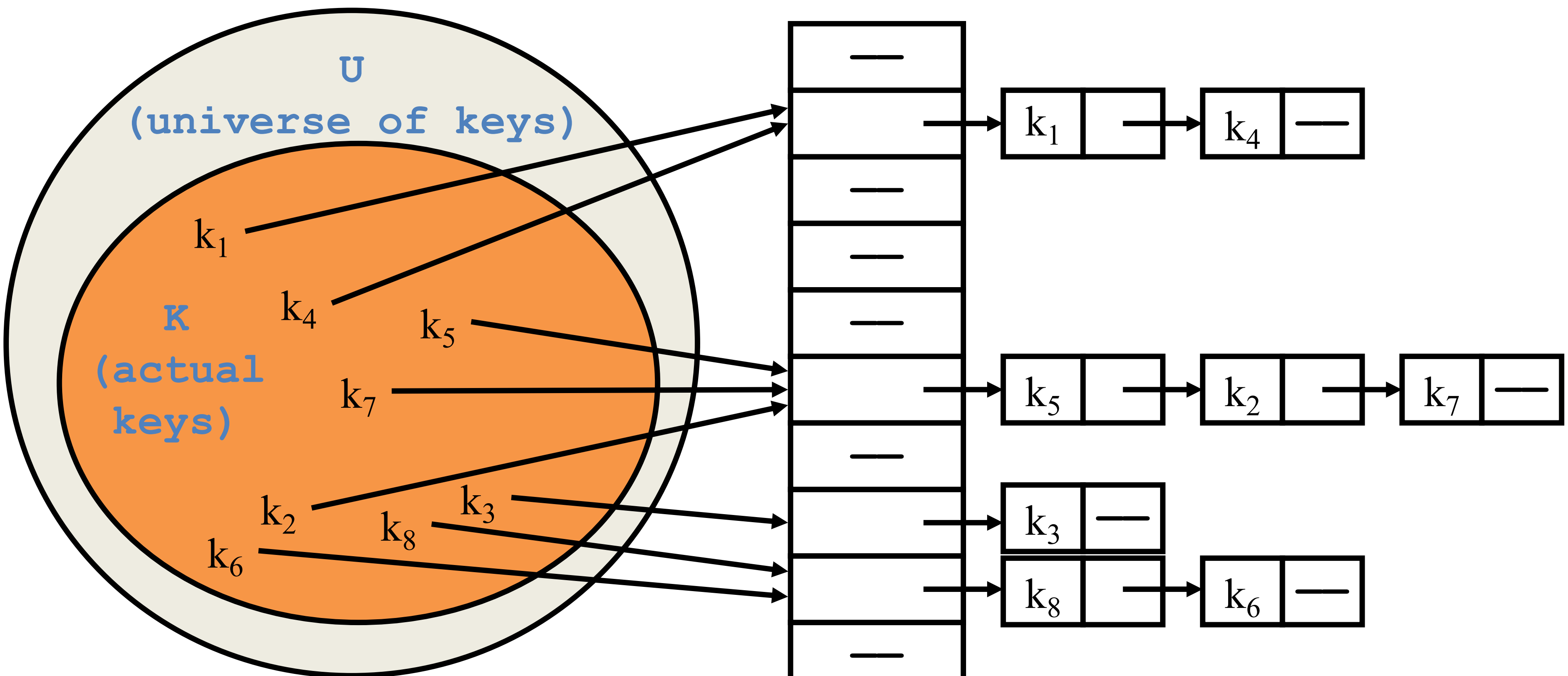
Chaining

- *How do we insert an element?*
  - Insert at Head. Time: O(1)
- *Searching:* O(list length) = O(n).



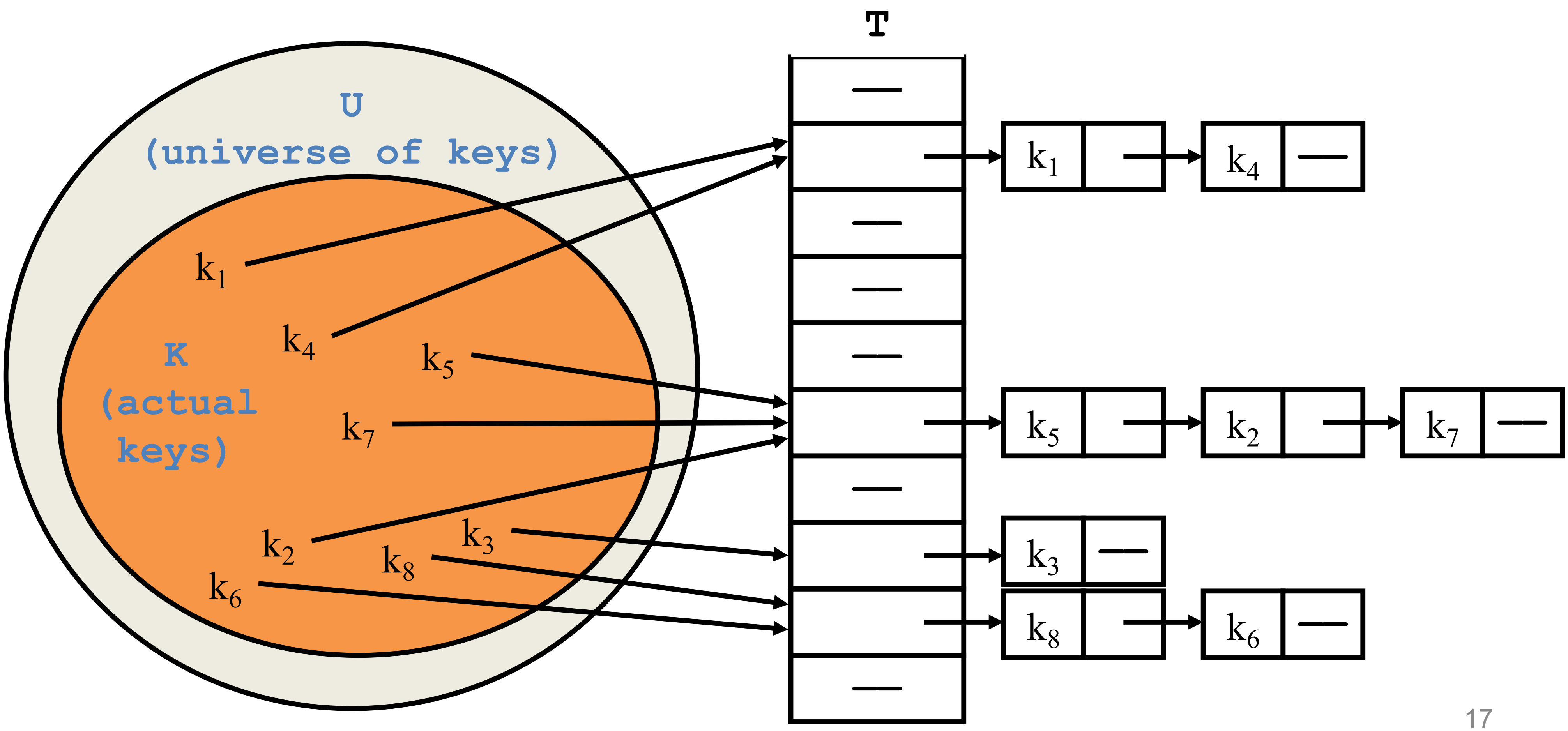
Chaining

- *How do we delete an element?*
  - Using singly linked list: O(n) as we need to find a node's previous node.
  - Using doubly-linked list: O(1) as both prev and next pointers are available in the node to be deleted (O(n) if the key is the input)



Chaining

- How do we search for an element with a given key?
  - Worst-case time O(n)



Open Addressing

- Basic idea:
  - To insert: if slot is full, try another slot, …, until an open slot is found (***probing***)
  - To search, follow same sequence of probes as would be used when inserting the element
    - If reach element with correct key, return it
    - If reach a NULL pointer, element is not in table
- Good for fixed sets (adding but no deletion)
  - Example: spell checking

Open Addressing

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

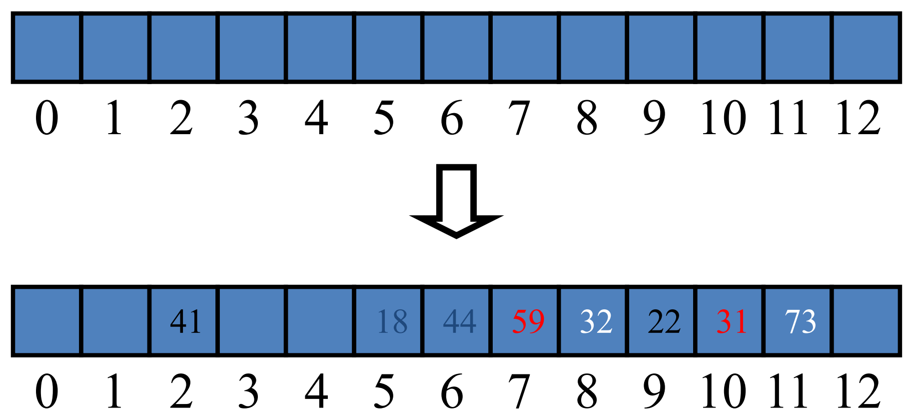
Probing

- 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

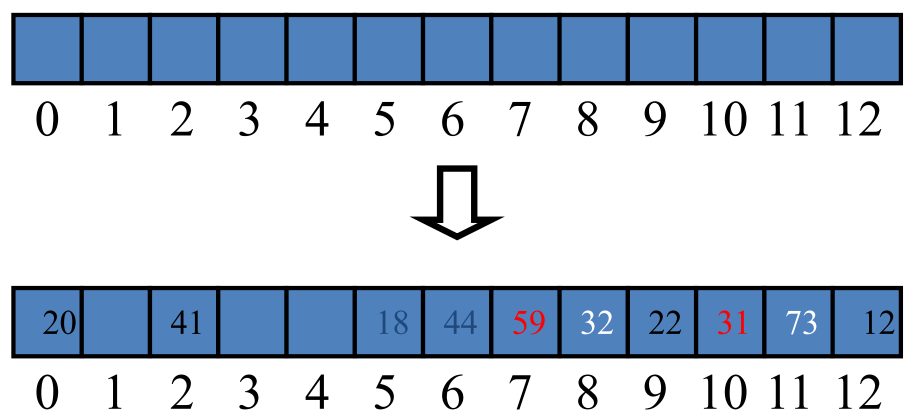
- Consider a hash table A that uses linear probing
- get(k): search
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - *An empty cell is found*, or
    - m cells have been unsuccessfully probed
  - To ensure the efficiency, if k is not in the table, we want to find an empty cell as soon as possible. The load factor canNOT 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
            $i \leftarrow (i + 1) \bmod m$
            $p \leftarrow p + 1$
    until $p = m$ // **table size**
    return *null*

Linear Probing

- Search for key=20.
  - h(20)=20 mod 13 =7.
  - Go through rank 7, 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



Updates with Linear Probing

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

Quadratic Probing

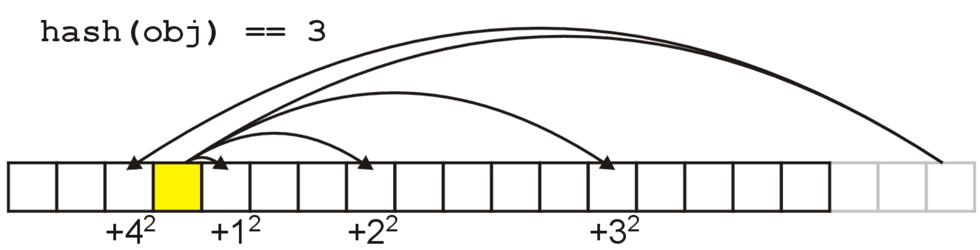
- Primary clustering occurs with linear probing because of the same linear pattern:
  - Items can get clustered in the same area, making search costly.
- Instead of searching forward in a linear fashion, try to jump far enough out of the current (unknown) cluster.

Quadratic Probing

- Suppose an element should appear in bin (slot) h:
  - if bin h is occupied, then check the following sequence of bins: $(h + i^2)$ mod m

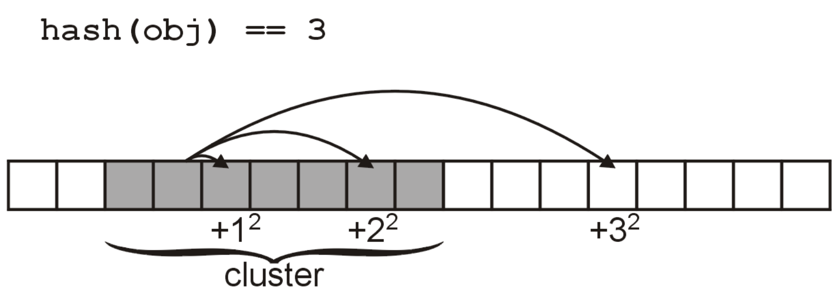
    $h + 1^2,\ \ h + 2^2,\ \ h + 3^2,\ \ h + 4^2,\ \ h + 5^2, \ldots$

    $h + 1,\ \ h + 4,\ \ h + 9,\ \ h + 16,\ \ h + 25, \ldots$
- For example, with m = 17 bins:



Quadratic Probing

- If one of $(h + i^2)$mod m 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

- Unfortunately, there is no guarantee that

  $$(h + i^2) \bmod m$$

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

Quadratic Probing

- Example with m = 11:

  0, 1, 4, 9, 16 ≡ 5, 25 ≡ 3, 36 ≡ 3

- With m = 13:

  0, 1, 4, 9, 16 ≡ 3, 25 ≡ 12, 36 ≡ 10, 49 ≡ 10

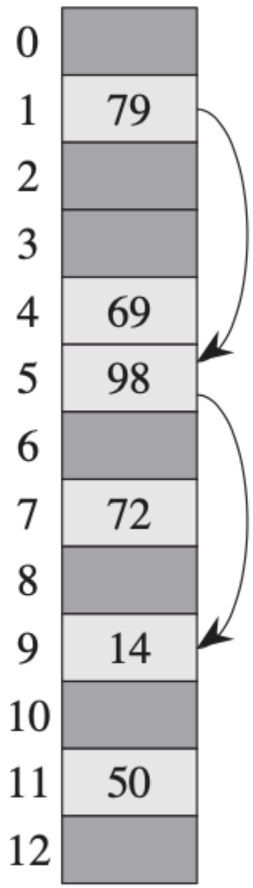
- With m = 17:

  0, 1, 4, 9, 16, 25 ≡ 8, 36 ≡ 2, 49 ≡ 15, 64 ≡ 13, 81 ≡ 13

Secondary Clustering

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

Double hashing



Probe sequence *i* is determined by

$$h(k,i) = (h_1(k) + i(h_2(k))) \text{ mod } m$$

i= 0 (initial prob), 1, 2, 3, ….

Figure 11.5 Insertion by double hashing. Here we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$, we insert the key 14 into empty slot 9, after examining slots 1 and 5 and finding them to be occupied.

Running time of insert and search for open addressing

Depends on the hash function/probe sequence

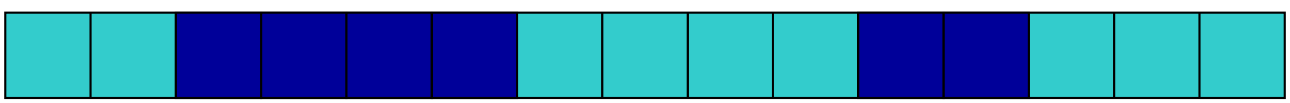
Worst case?

- O(n) – probe sequence visits every full entry first before finding an empty

Running time of insert and search for open addressing

Average case?

We have to make at least one probe

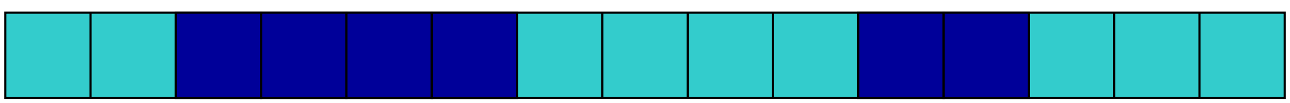


Running time of insert and search for open addressing

Average case?

What is the probability that the first probe will **not** be successful (assume uniform hashing function)?

$$\sim \alpha$$



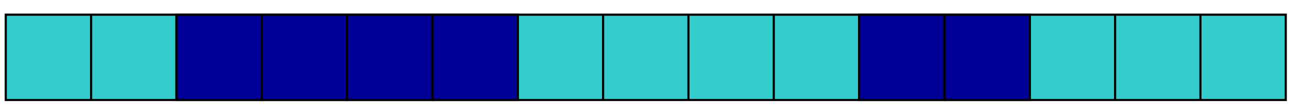
Running time of insert and search for open addressing

Average case?

What is the probability that the first **two** probed slots will **not** be successful?

$\sim\alpha^2$

why '~'?

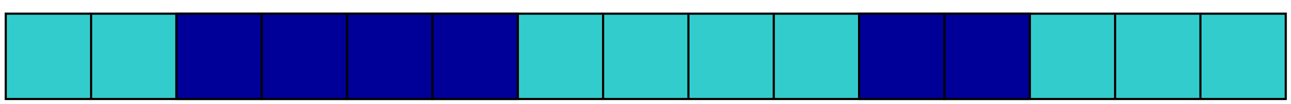


Running time of insert and search for open addressing

Average case?

What is the probability that the first **three** probed slots will **not** be successful?

$$\sim\alpha^3$$



Running time of insert and search for open addressing

Average case: expected number of probes
= sum of (each slot's prob * its probability of being probed)
= sum of the probability of making 1 probe, 2 probes, 3 probes, …

$$E[probes] = 1 + \alpha + \alpha^2 + \alpha^3 + \ldots$$

$$= \sum_{i=0}^{m} \alpha^i$$

$$< \sum_{i=0}^{\infty} \alpha^i$$

$$= \frac{1}{1-\alpha}$$

Average number of probes

$$E[probes] = \frac{1}{1-\alpha}$$

α	Average number of searches
0.1	$1/(1-.1) = 1.11$
0.25	$1/(1-.25) = 1.33$
0.5	$1/(1-.5) = 2$
0.75	$1/(1-.75) = 4$
0.9	$1/(1-.9) = 10$
0.95	$1/(1-.95) = 20$
0.99	$1/(1-.99) = 100$

How big should a hashtable be?

A good rule of thumb is the hashtable should be around half full.