Hash Tables

Chapter 11

Comparison-Based Searches

To locate an item, **the target key has to be compared against the other keys** in the collection.

- linear search → O(n)
- binary search -> O(log n)

O(log n) is the best that can be achieved.

- to do better than O(log n) we need a different approach

Hashing

Hashing: the process of mapping (transforming) a search key to a number in range 0-(n-1) that can be used as a list or array index.

- the number of possible search keys might be huge:

  e.g. if the key is a string

- the number of slots (n) in a list/table is much smaller

The goal is to provide *immediate* access to the keys:

- **hash table** – the list containing the keys.
- **hash function** – maps a key to an array index.

Using mod (%) in the Hash function

Suppose we have the following set of keys

```
765, 431, 96, 142, 579, 226, 903, 388
```

and a hash table, T, with M = 13 elements.

We can define a simple hash function h()

```
h(key) = key % M
```

The **%** (mod) operator is an easy way to reduce any large number so it can be used as a table index in the range 0 to M-1

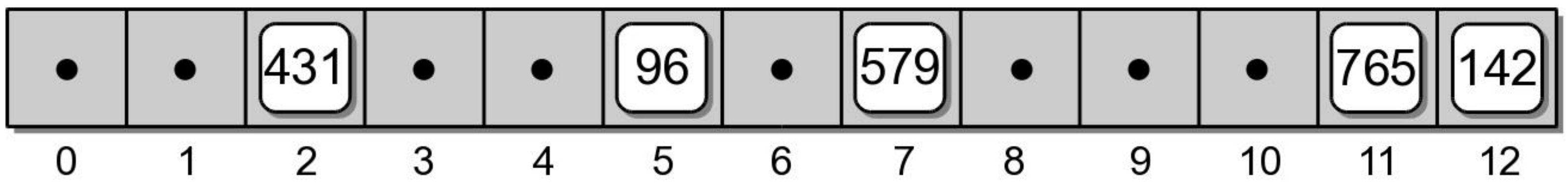
Adding Keys

To add a key to the hash table:

- Use the hash function to find the array index in which the key should be stored.

```
h(765) => 11
h(431) => 2
h(96) => 5
h(142) => 12
h(579) => 7
```

- Store the key in the given slot.

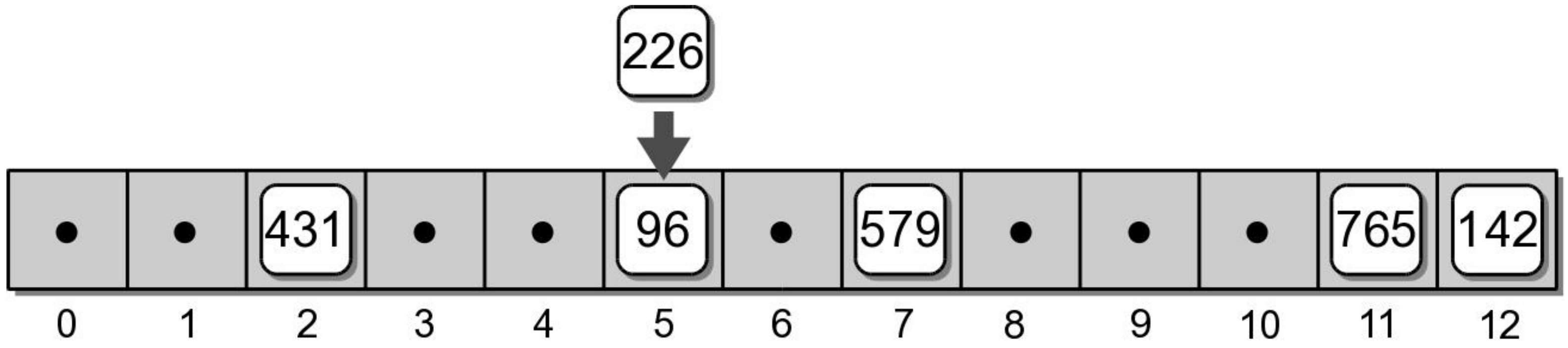


Collisions

When we tryto add key 226?

```
h(226) => 5
```

The key (226) maps to slot 5, but it's already in use

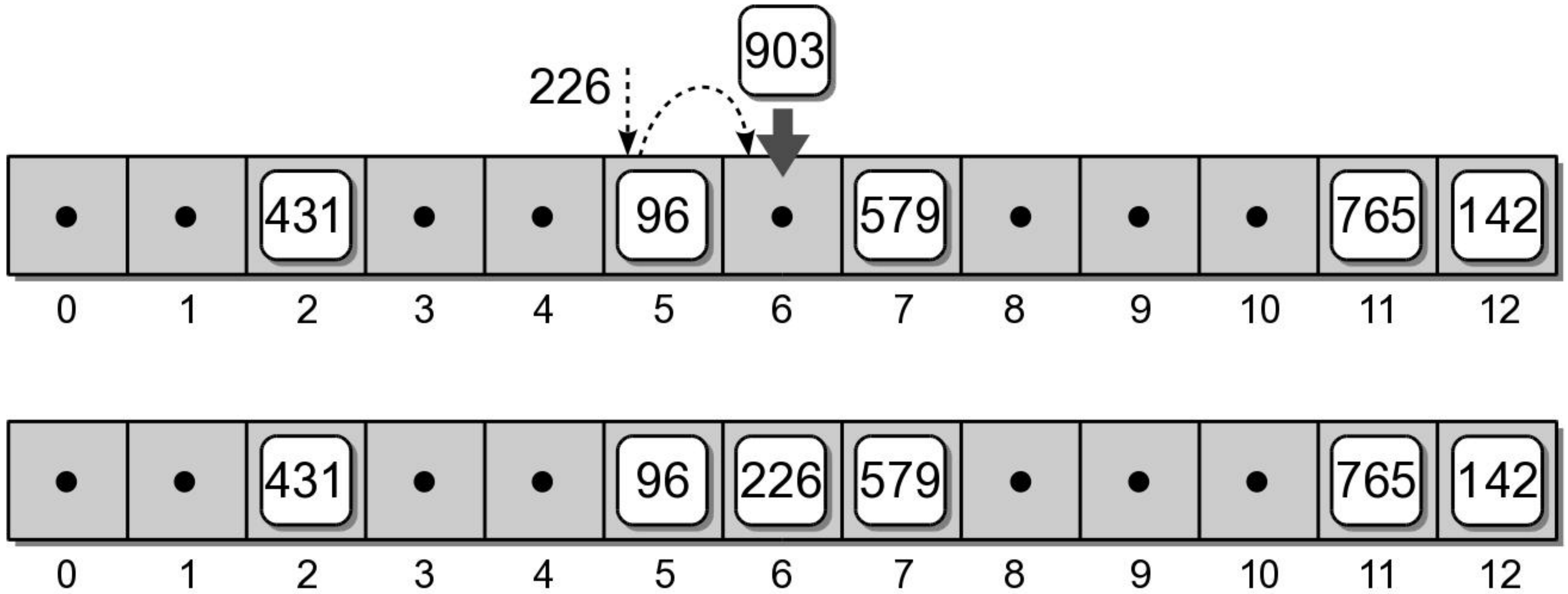


This is called a **collision** – when two or more keys map to the same hash location.

Resolving Collisions with Linear Probing

we must *resolve the collision* by finding another unused slot.

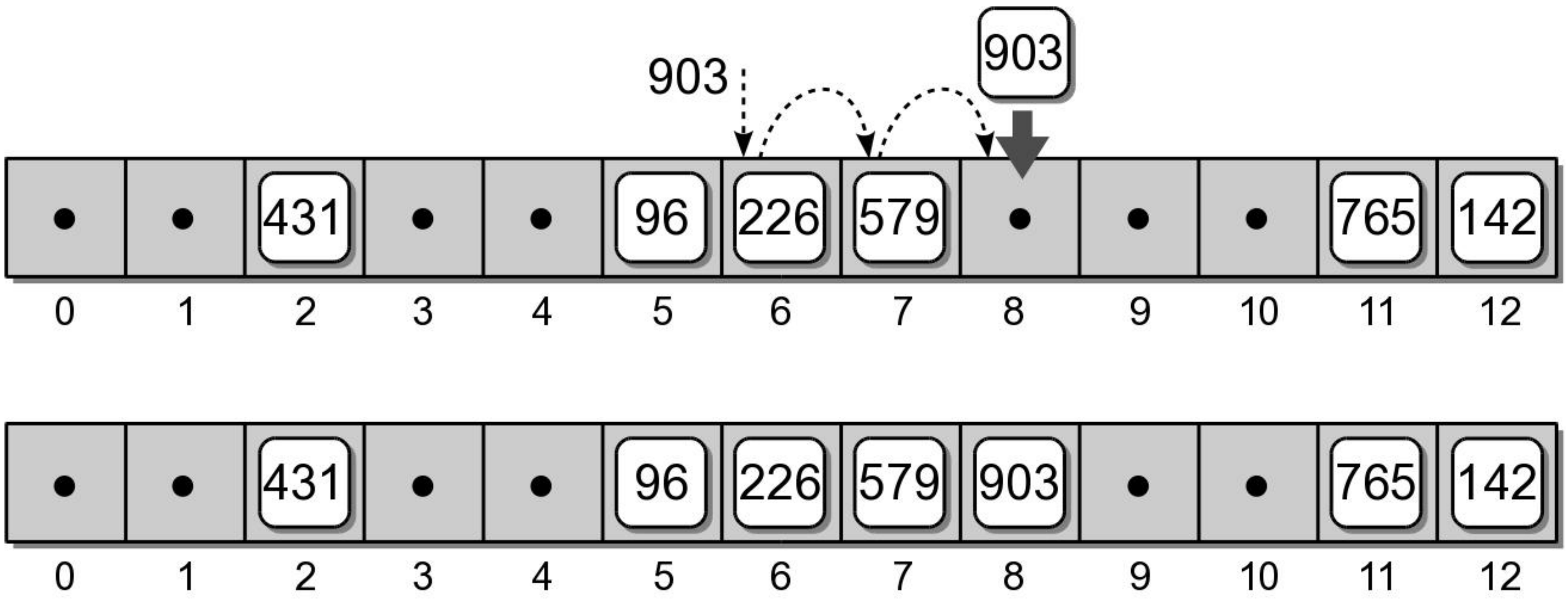
- **linear probe** – simplest approach which examines the table entries in sequential order.



Linear Probing

Consider adding key 903 to our hash table.

```
h(903) => 6
```

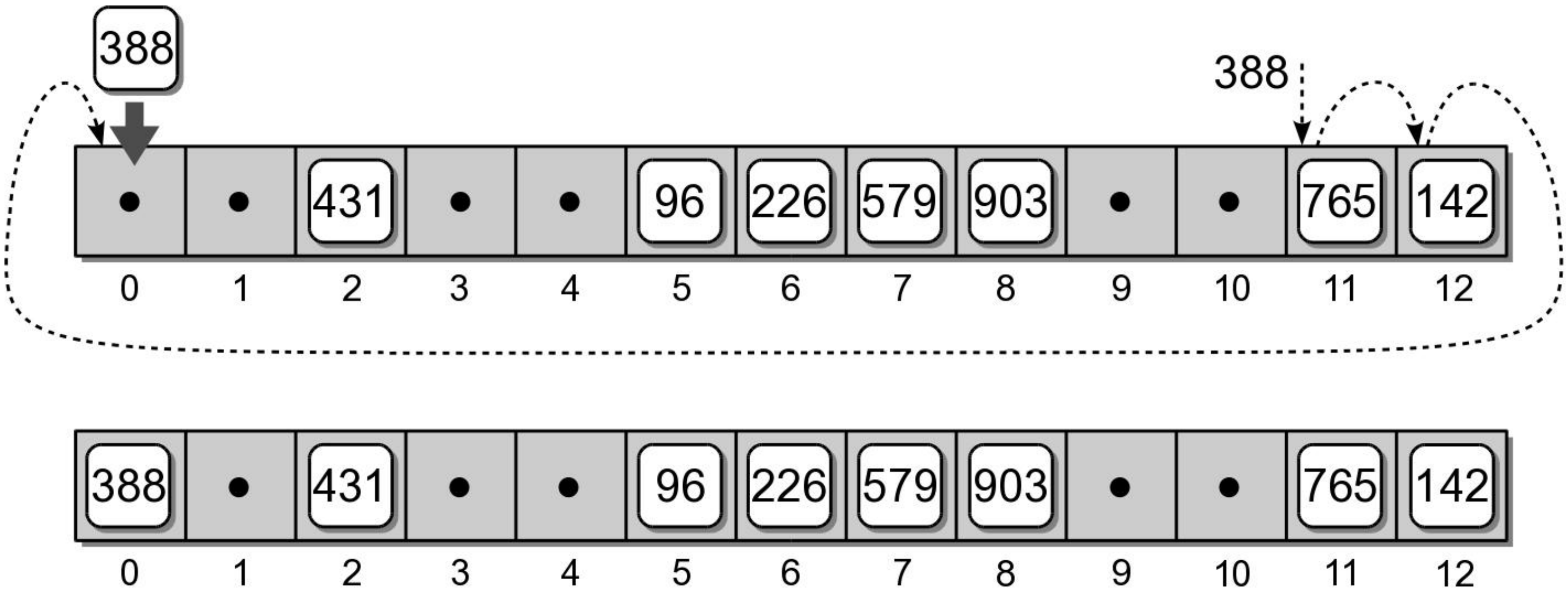


Linear Probing – and wrap-around

If the end of the array is reached during the probe, it wraps around to the first entry and continues.

- Consider adding key 388 to our hash table.

```
h(388) => 11
```

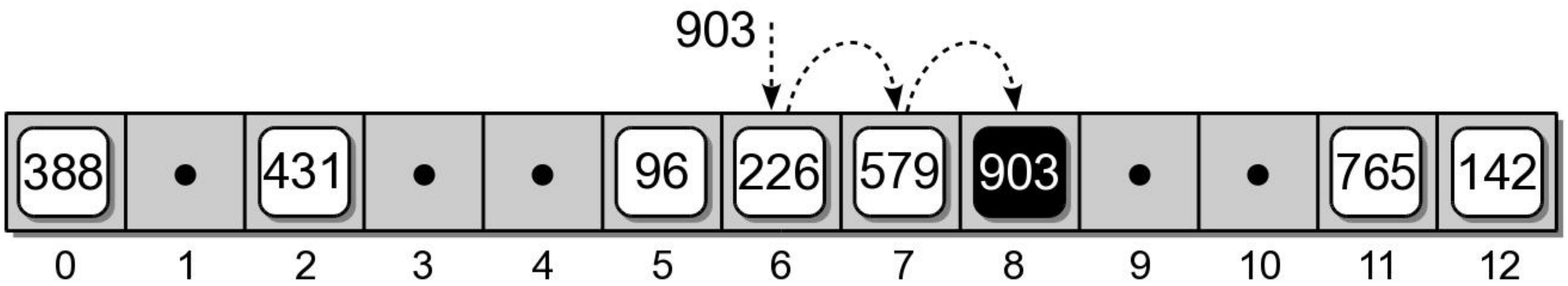


Searching for a key

Searching a hash table for a specific key is similar to the *add* operation.

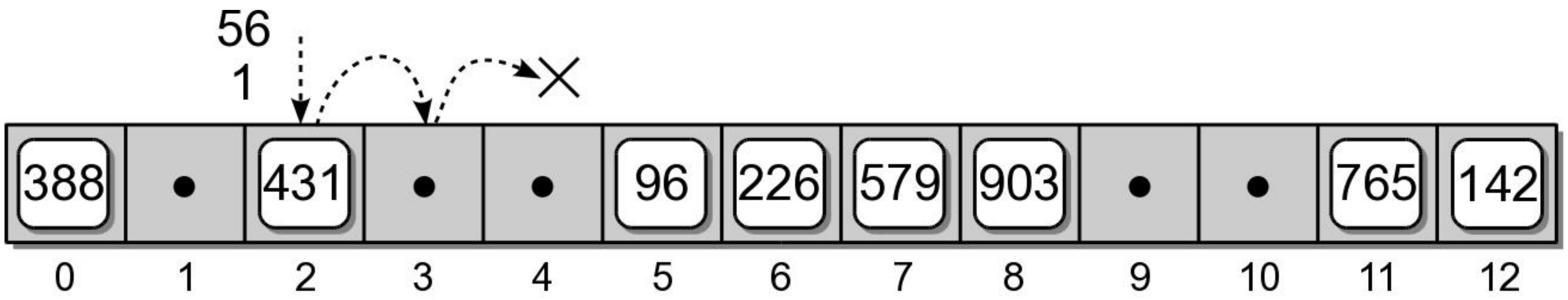
- the key is hashed to find an initial slot.
- does the slot contains the target?
- if not, apply the same probe sequence used to add keys to locate the target.

Example: search for key 903.



Searching for missing values

What if the key is not in the hash table?



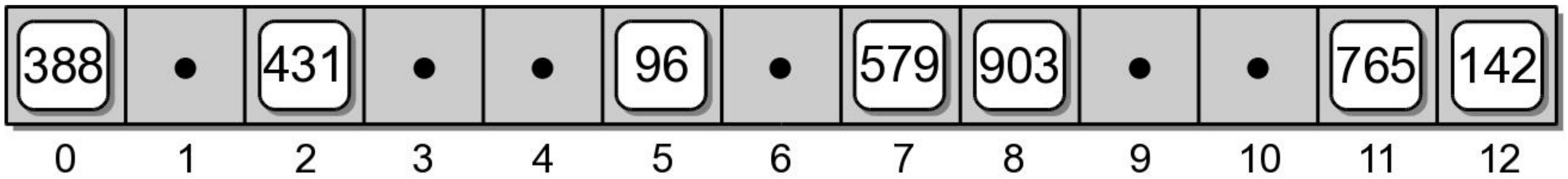
- the probe continues until either:
  - an **empty slot is reached**, or
  - **all slots have been examined.**

Deleting Keys

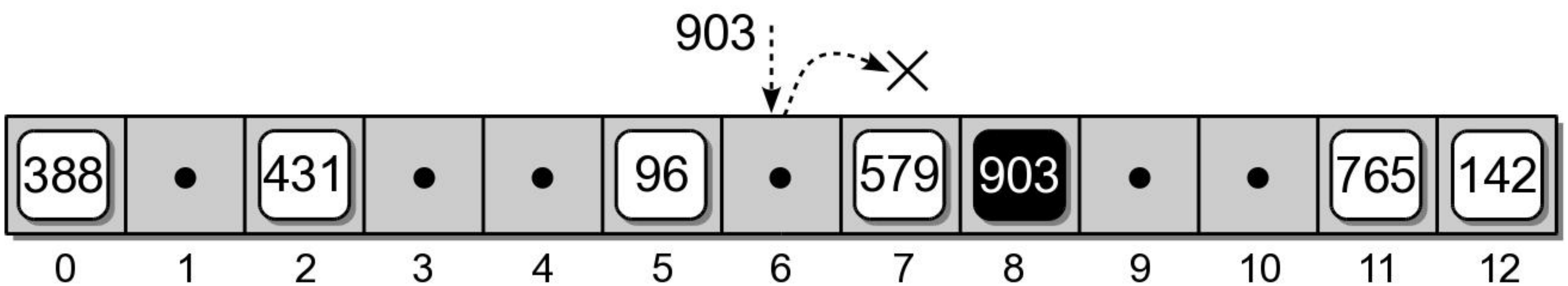
- Deleting a key from a hash table is a more complicated than adding keys.
  - we can search for the slot containing the key
  - But cannot simply delete it by setting the entry to **`None`.** *Why?*

Incorrect Deletion

- Suppose we simply remove key 226 from slot 6.

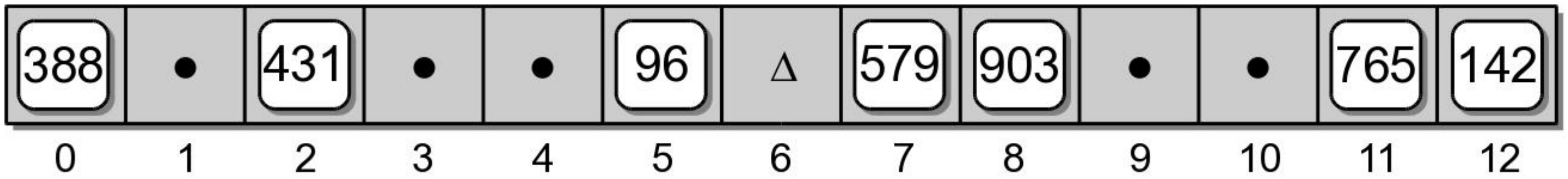


- What happens if we search for key 903?

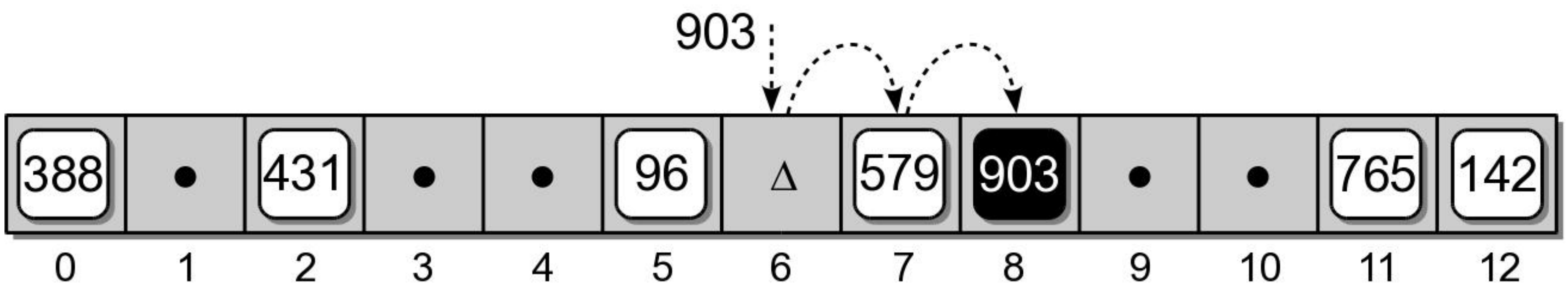


Correct Deletion

Use a **special flag** to indicate an entry that *was previously occupied, is now empty.*



When searching a hash table, the probe must continue past the slot(s) with the special flag.



Clustering

The grouping of keys in a common area.

- as the table fills, collisions are more likely to occur.
- clusters begin to form due to the probing required to find an empty slot.
- as a cluster grows larger, more collisions will occur.

- **primary clustering** is the clustering around the original hash position.

Probe Sequence

- The probe sequency is order in which the hash entries are visited during a probe.
  - a linear probe steps through the entries in sequential order.
  - the next array slot can be represented as

    `slot = (home + i) % M`

  - where
    - **`i` is the *i*th probe,** where *i* starts at zero.
    - home is the **home position**

Modified Linear Probe

To improve the linear probe, change the step size to some fixed constant ***i*c***

```
slot = (home + i*c) % M
```

- Suppose we set c = 3 to build the hash table.

```
h(765)  => 11          h(579)  => 7
h(431)  => 2           h(226)  => 5    => 8
h(96)   => 5           h(903)  => 6
h(142)  => 12          h(388)  => 11   => 1
```

0	1	2	3	4	5	6	7	8	9	10	11	12
•	388	431	•	•	96	903	579	226	•	•	765	142

Quadratic Probing

A better approach for reducing primary clustering.

```
slot = (home + i**2) % M
```

- this increases the distance between each probe in the sequence.
- Example:

```
h(765) => 11        h(579) => 7
h(431) => 2         h(226) => 5   => 6
h(96)  => 5         h(903) => 6   => 7   => 10
h(142) => 12        h(388) => 11  => 12 => 2   => 7 => 1
```

0	1	2	3	4	5	6	7	8	9	10	11	12
•	388	431	•	•	96	226	579	•	•	903	765	142

Quadratic Probing

Quadratic probing reduces the number of collisions.

but introduces the problem of **secondary clustering**.

- i.e. when two keys map to the same entry and have the same probe sequence.

- as both keys map to the same home slot and the probe sequence is the same, the probes for both keys follow one another.

Double Hashing

A way of avoiding this is to use **a secondary hash function** of the original key as an offset.

The offset will vary due to the different original keys so the probe sequence won't be the same.

```
slot = (home + i * SecondHash(key)) % M
```

- step size remains a constant throughout the probe.
- multiple keys that have the same home position, will have different probe sequences.

Double Hashing

A simple choice for the second hash function.

```
SecondHash(key) = 1 + key % P
```

- Example: let P = 8

```
h(765)  => 11            h(579)  => 7
h(431)  => 2             h(226)  => 5   => 8
h(96)   => 5             h(903)  => 6
h(142)  => 12            h(388)  => 11  => 3
```

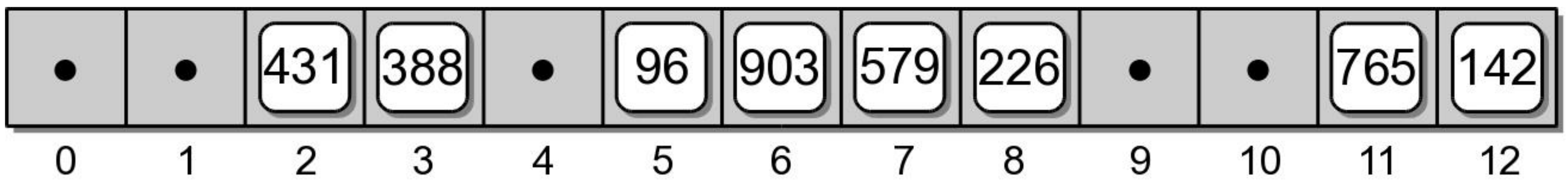


Table Size

How big should a hash table be?

- if we know the max number of keys.
  - make it big enough to hold all of keys.
- usually, we don't know the max. number of keys.

Most probing techniques **work best when the table size is a prime number.**

Expanding a Hash Table by *Rehashing*

To expand, create a new bigger table and move the data into it when required.

We can start with a small table and expand it as needed.

- **the load factor = used slots/total slots**

A hash table should be expanded before the load factor reaches 80%.

IMPORTANT

To move data from old table to new bigger table, it must be rehashed, not simply moved. *Why?*

Expansion Size

Size of the expansion depends on the application.

- good guideline is to at least double its size.
- Two common approaches:
    - double the size of the table, then search for the first larger prime number.
    - double the size of the table and add one to ensure M is odd.

Efficiency Analysis

Efficiency depends on:

- the hash function
- size of the table
- type of collision resolution probe

Once an empty slot is located, adding or deleting a key can be done in O(1) time.

The time required to perform the search is the main contributor to the overall time of all operations.

Efficiency Analysis

- **Best case: O(1)**
  - the key maps directly to the correct entry.
  - there are no collisions.
- **Worst case: O(m)**
  - assume there are *n* keys stored in a table of size *m*.
  - the probe has to visit every entry in the table.

Efficiency Analysis

In the worst case, hashing appears to be no better than a basic linear search, but on average, hashing is very efficient:

Load Factor	0.25	0.5	0.67	0.8	0.99
Successful search:					
Linear probe	1.17	1.50	2.02	3.00	50.50
Quadratic probe	1.66	2.00	2.39	2.90	6.71
Unsuccessful search:					
Linear probe	1.39	2.50	5.09	13.00	5000.50
Quadratic probe	1.33	2.00	3.03	5.00	100.00

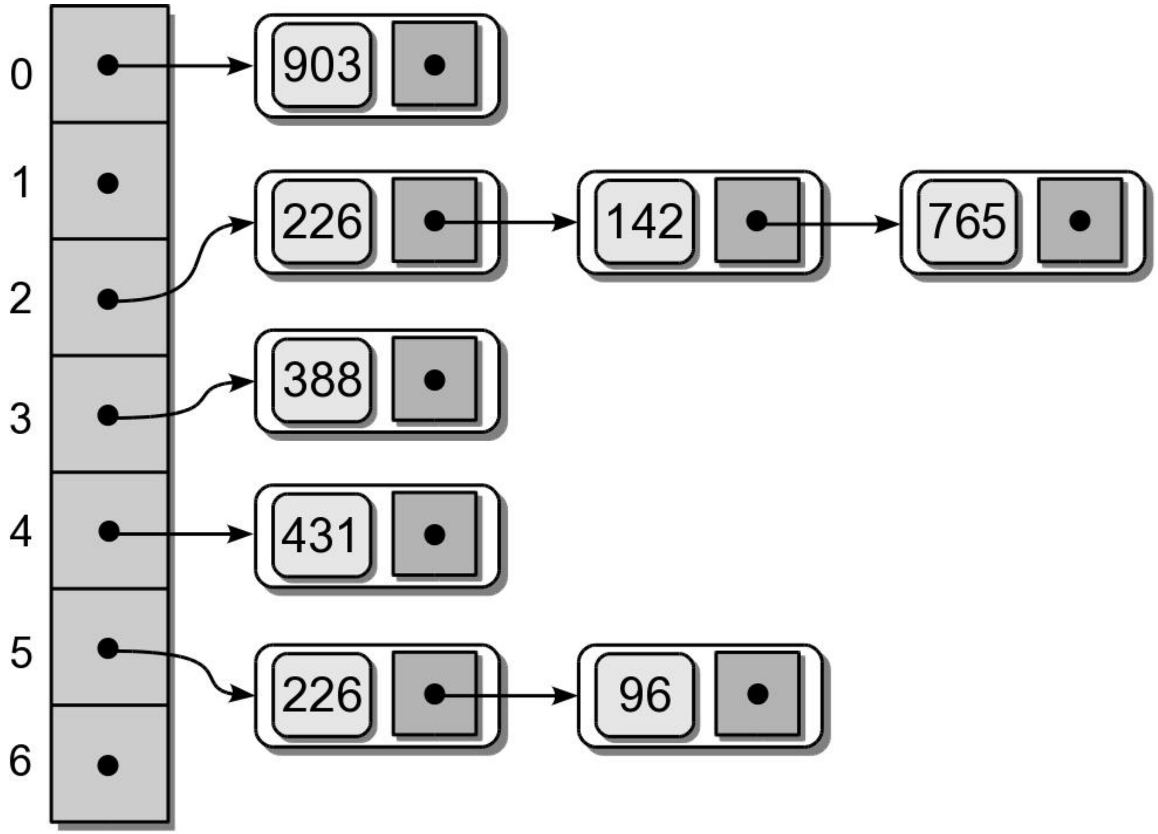
These are the number of probes

Separate Chaining

We can eliminate collisions if we store the keys outside the table.

- **chains** – use linked lists to store keys that map to the same entry.
- The hash table becomes an list of sublists.
- After mapping the key to an entry in the table, the list is searched for the key.

Separate Chaining



Efficiency: Separate Chaining

- Very efficient in the average case.
  - If there are *n* keys and *m* entries, the average list length is

$$\alpha = n/m$$

  - Successful search:

$$1 + \alpha/2$$

  - Unsuccessful search:

$$1 + \alpha$$

Hash Functions

- The efficiency of hashing depends in large part on the selection of a good hash function.
  - A “perfect” function will map every key to a different table entry.
    - This is seldom achieved except in special cases.
  - A “good” hash function distributes the keys evenly across the range of table entries.

Function Guidelines

- Important guidelines to consider in designing a hash function.
  - Computation should be simple.
  - Resulting index can not be random.
  - Every part of a multi-part key should contribute.
  - Table size should be a prime number.

Common Hash Functions

- **Division** – simplest for integer values.

```
h(key) = key % M
```

- **Truncation** – some columns in the key are ignored.
  - Example: assume keys composed of 7 digits.
  - Use the 1^{st}, 3^{rd}, 6^{th} digits to form an index (M = 1000).

Common Hash Functions

- **Folding** – key is split into multiple parts then combined into a single value.
  - Given the key value 4873152, split it into three smaller values (48, 73, 152).
  - Add the values together and use with division.

Hashing Strings

- Strings can also be stored in a hash table.
  - Convert to an integer value that can be used with the division or truncation methods.
- Simplest approach: sum the ASCII values of individual characters.
  - Short strings will not hash to larger table entries.
- Better approach: use a polynomial.

$$s_{0a}^{n-1} + s_{1a}^{n-2} + \cdots + s_{n-3}a^{2} + s_{n-2}a + s_{n-1}$$