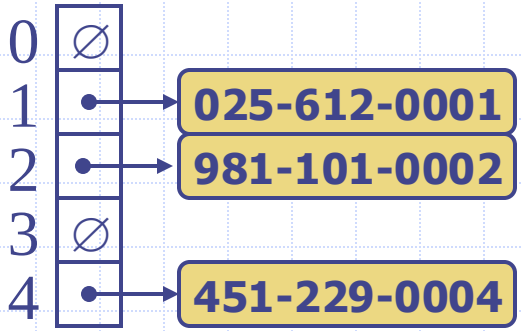
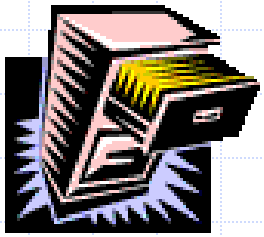
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

# Hash Tables

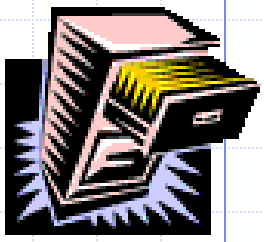


# Recall the Map ADT

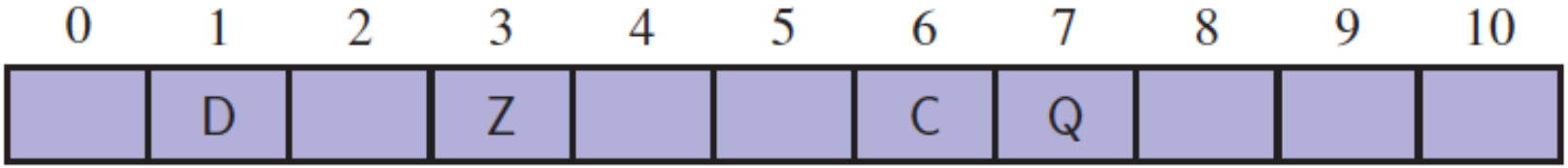


- get(k): if the map M has an entry with key k, return its associated value; else, return null
- put(k, v): if key k is not already in the map M, then *add* entry (k, v) into M and *return* null; else *replace* the old entry with the same key with the new entry and *return* the old value associated with k.
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- entrySet(): return an iterable collection of the entries in M
- keySet(): return an iterable collection of the keys in M
- values(): return an iterator of the values in M
- size(), isEmpty()

# Intuitive Notion of a Map

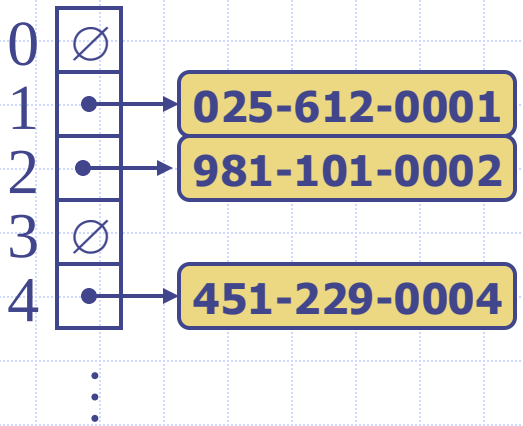


- Intuitively, a map M supports the abstraction of using keys as "addresses" that help locate an entry.
- As a mental warm-up, consider a restricted setting in which a map with $n$ items uses keys that are known to be integers in a range from $0$ to $N - 1$, for some $N \geq n$.



# More General Kinds of Keys

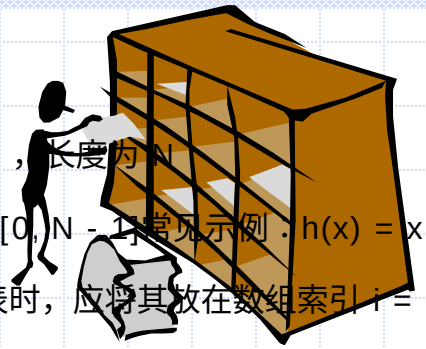
- But what should we do if our keys are not integers in the range from $0$ to $N - 1$?
  - Use a **hash function** to map ***general keys*** to corresponding indices in a table.
  - For instance, the last four digits of a Social Security number.



# Hash Tables

- Hash tables are a concrete data structure which is suitable for implementing maps.
- Basic idea: convert each key into an index.
- Look-up of keys, insertion and deletion in a hash table usually runs in $O(1)$ time.

# Hash Tables and Hash Functions

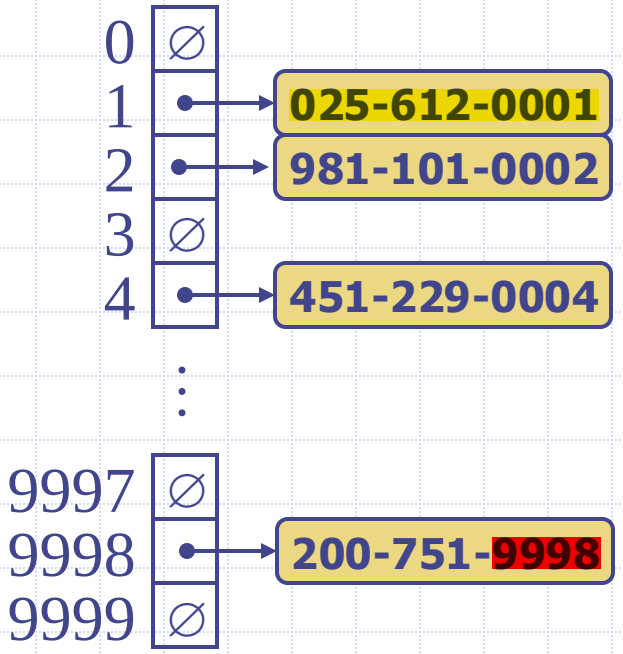


**哈希表的组成：**一个固定大小的数组 A，称为**表（table）**，长度为 N
一个**哈希函数**h，用于将“ 键（key） ”映射为数组索引
**哈希函数的作用：**将任意键 x 映射为整数 h(x)，范围在 [0, N - 1]常见示例：h(x) = x \ mod N用于整数键，取模操作产生索引
**映射与存储目标：**当你想存储一对键值对 (k, o) 到哈希表时，应将其放在数组索引 i = h(k) 处，即：A[h(k)] = (k, o)

- A hash table for a given key type consists of
  - Array $A$ (called table) of size $N$
  - Hash function $h$
- 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 key $x$
- When implementing a map with a hash table, the goal is to store item $(k, o)$ at index $i = h(k)$ in the array $A$

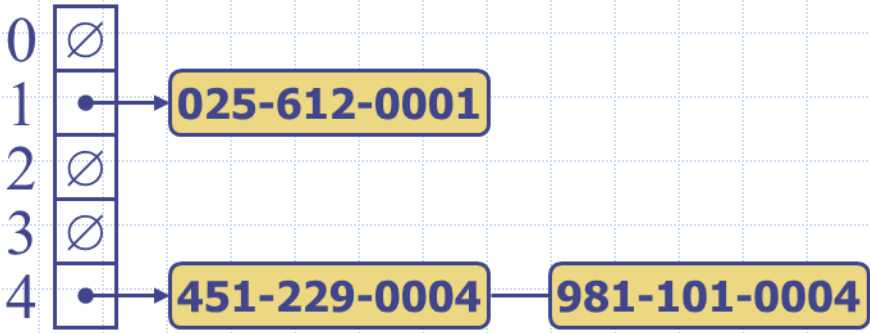
# Example

- We design a hash table for a map storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function
  $h(x) = \text{last four digits of } x$



# Collision Handling 哈希冲突

- Collisions occur when different elements are mapped to the same cell.
- A lot of theories and practices of hashing consist of devising better ways to *avoid* or *handle* collisions.



# Hash Functions (Sec.10.2)

- A hash function is usually specified as the *composition* of two functions:

  Hash code:

  $h_1$: keys $\rightarrow$ integers

  Compression function:

  $h_2$: integers $\rightarrow [0, N-1]$

- The hash code is applied first, and the compression function is applied next on the result, i.e.,

  $h(x) = h_2(h_1(x))$

- The goal of the hash function is to "disperse" the keys in an apparently random way.

# Thinking & Self-Study

- What is the advantage of separating the hash function into two components, namely, hash code and compression function? Why?

# Thinking & Self-Study

- The goal of the hash function is to "disperse" the keys in an apparently random way.
- Why disperse?
- Why random?

# Hash Codes [Not assessed]

- Memory address:
  - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
  - Good in general, except for numeric and string keys
- Integer cast:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)
- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)

# Hash Codes (cont.) [Not assessed]

- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)
    $$a_0 a_1 \ldots a_{n-1}$$
  - We evaluate the polynomial
    $$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$$
    at a fixed value $z$, ignoring overflows
  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)
- Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner's rule:
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time
    $$p_0(z) = a_{n-1}$$
    $$p_i(z) = a_{n-i-1} + z p_{i-1}(z) \quad (i = 1, 2, \ldots, n-1)$$
- We have $p(z) = p_{n-1}(z)$

# Compression Functions

- Division:
  - $h_2(y) = y \bmod N$
  - The size $N$ of the hash table is usually chosen to be a prime

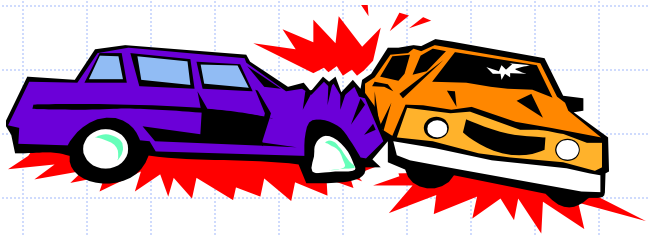
  (hash codes will be spread better)

| 方法 | 优点 | 注意事项 |
|---|---|---|
| Division | 简单、常用 | $N$ 必须是素数 |
| MAD | 更灵活，适合组合使用 | 需小心选择 $a, b$，避免退化 |

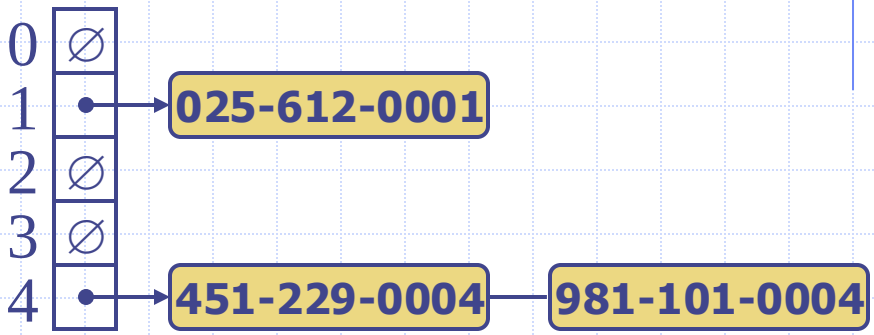
- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - $a$ and $b$ are nonnegative integers such that

    $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value $b$

# Collision Handling



- Collisions occur when different elements are mapped to the same cell
- Separate Chaining: let each cell in the table point to a list of entries that map there




- Separate chaining is simple, but requires additional memory outside the table

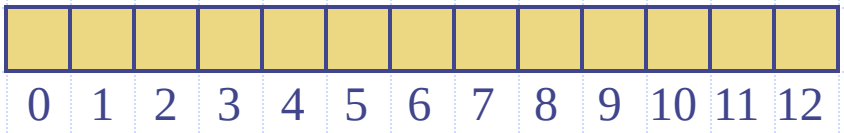
开放寻址（Open Addressing）：冲突发生时，不使用链表，而是将元素放到哈希表中**其他空位置**。
线性探测（Linear Probing）：从冲突位置开始，**顺序**（或循环）查找下一个空位进行插入。

- 哈希函数：h(x) = x mod 13，表大小为13。
- 插入顺序：18, 41, 22, 44, 59, 32, 31, 73
- 过程：
  - 18 mod 13 = 5 → 放在 index 5
  - 41 mod 13 = 2 → 放在 index 2
  - 22 mod 13 = 9 → 放在 index 9
  - 44 mod 13 = 5 → 冲突！→ 试 6 → 放在 index 6
  - 59 mod 13 = 7 → 放在 index 7
  - 32 mod 13 = 6 → 冲突！→ 试 7（已占）→ 试 8 → 放在 index 8
  - 31 mod 13 = 5 → 冲突！试 6、7、8、9 都被占了 → 试 10 → 放在 index 10
  - 73 mod 13 = 8 → 冲突！→ 找下一个空位：11 → 放在 index 11

# Linear Probing

- Open addressing: the colliding item is placed in a different cell of the table
- Linear probing: handles collisions by placing the colliding item in the next (circularly) available table cell
- Accessing a cell of the array is referred to as a “probe”
- Disadvantage: Colliding items lump together, causing future collisions to cause a longer sequence of probes

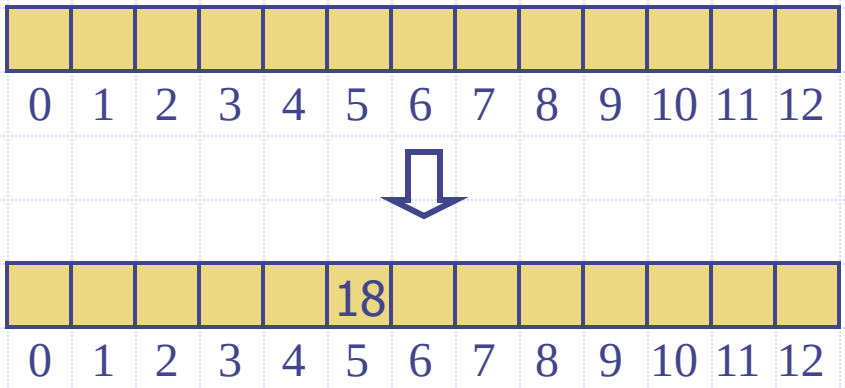
- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



# Linear Probing

- Open addressing: the colliding item is placed in a different cell of the table
- Linear probing: handles collisions by placing the colliding item in the next (circularly) available table cell
- Accessing a cell of the array is referred to as a "probe"
- Disadvantage: Colliding items lump together, causing future collisions to cause a longer sequence of probes

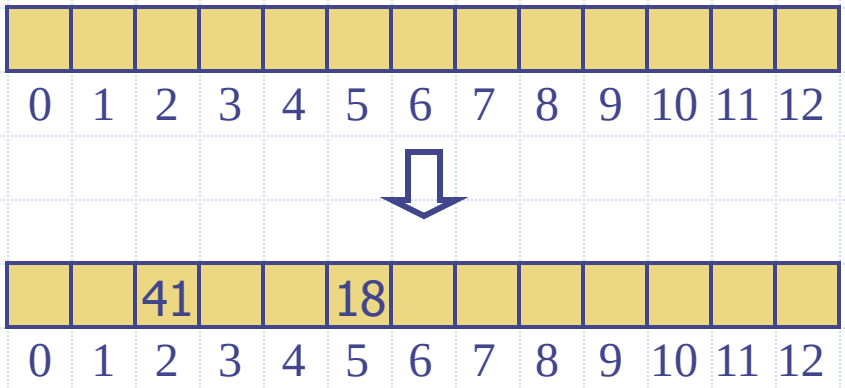
- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



# Linear Probing

- Open addressing: the colliding item is placed in a different cell of the table
- 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"
- Disadvantage: Colliding items lump together, causing future collisions to cause a longer sequence of probes

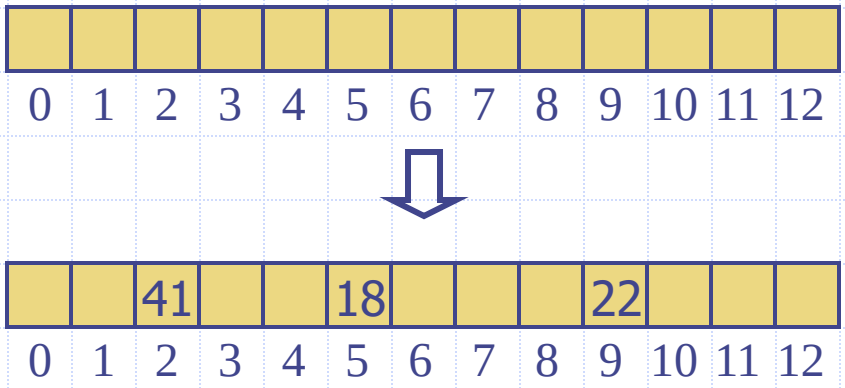
- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



# Linear Probing

- Open addressing: the colliding item is placed in a different cell of the table
- 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"
- Disadvantage: Colliding items lump together, causing future collisions to cause a longer sequence of probes

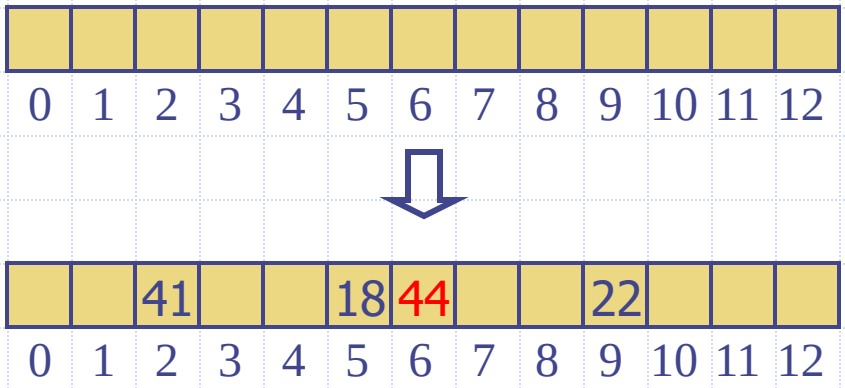
- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



# Linear Probing

- Open addressing: the colliding item is placed in a different cell of the table
- 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"
- Disadvantage: Colliding items lump together, causing future collisions to cause a longer sequence of probes

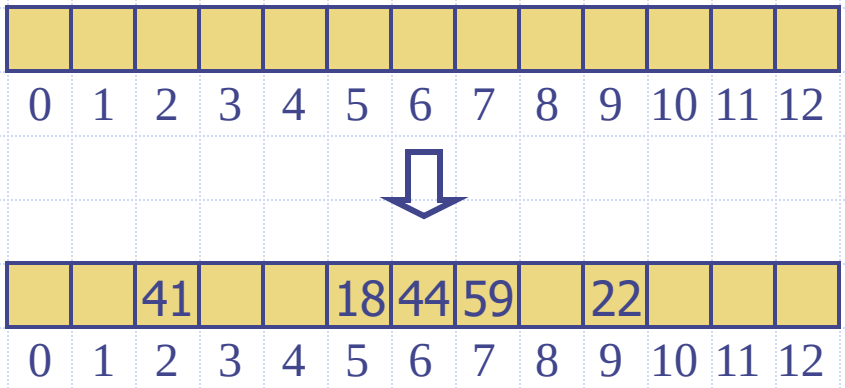
- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



# Linear Probing

- Open addressing: the colliding item is placed in a different cell of the table
- 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"
- Disadvantage: Colliding items lump together, causing future collisions to cause a longer sequence of probes

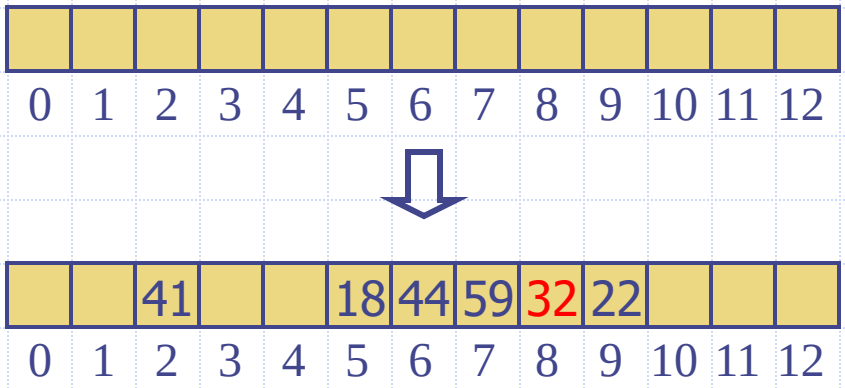
- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



# Linear Probing

- Open addressing: the colliding item is placed in a different cell of the table
- 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"
- Disadvantage: Colliding items lump together, causing future collisions to cause a longer sequence of probes

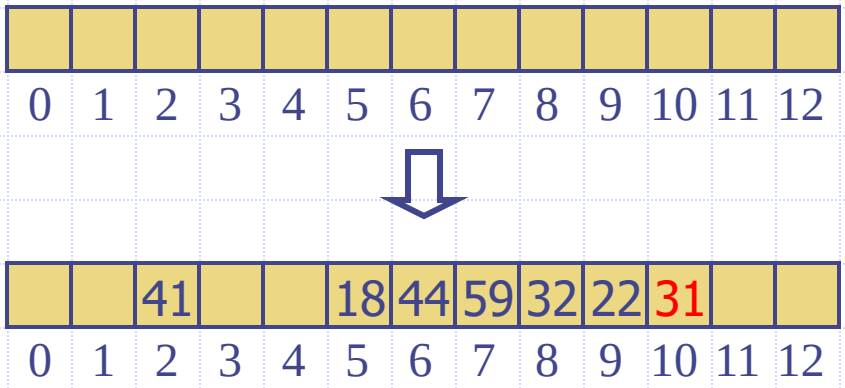
- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



# Linear Probing

- Open addressing: the colliding item is placed in a different cell of the table
- 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"
- Disadvantage: Colliding items lump together, causing future collisions to cause a longer sequence of probes

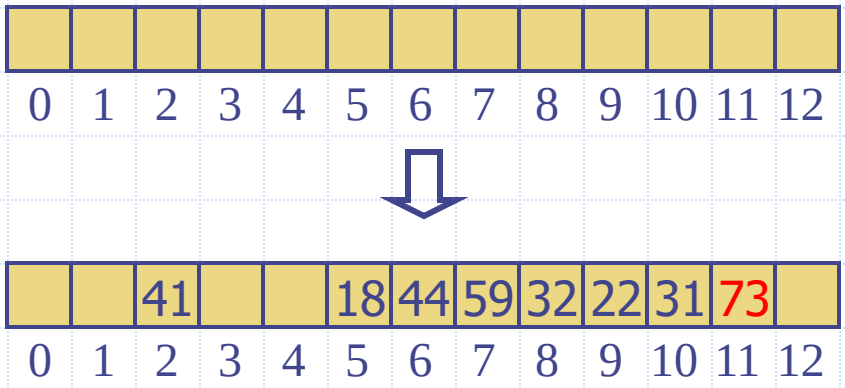
- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



# Linear Probing

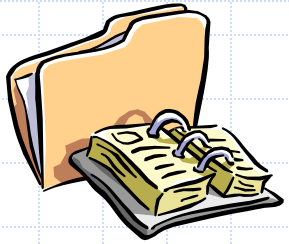
- Open addressing: the colliding item is placed in a different cell of the table
- Linear probing: handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Disadvantage: Colliding items lump together, causing 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



从 h(k) 开始：这是哈希函数计算出的初始索引。
连续探测（线性）：如果当前位置是空的（）：说明没找到，返回 null
如果当前位置的键是 k：找到，返回对应值
否则继续向后（模 N）找下一个位置
如果探测了 N 次都没找到，说明表中没有这个键，返回 null

# 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

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

# Exercise

**使用 get(x) 找到 x 的位置；将该位置设为 null 或空白**，即视为删除；**修复右侧的探测序列**（非循环地继续向右）：
**为什么要修复右边的探测序列？**如果右边的某个元素本来是由于冲突而线性探测插入的，那么它的位置依赖于之前的元素；**现在中间这个位置被设为空白了，get() 遇到空位就会停止，于是找不到这个元素了！**
**如何修复？**向右遍历，从删除点开始一个个检查后续元素；如果这些元素是通过线性探测插入的（即它们的原始位置不同于当前位置），就：临时删除它们；然后重新插入（重新哈希）进表中。

- How do you remove an element $x$?
- One answer:
  - Find $x$ using `get' and set the entry back to blank, i.e., null or empty
  - Fix the sequence on its `right-hand-side' (non-circular fashion)
    - Why?: If any entry on the right used linear probing then it might no longer be discovered by `get' because it will stop at the blank!
    - Fix: Move such entries, e.g., by removing them and then re-inserting them all.

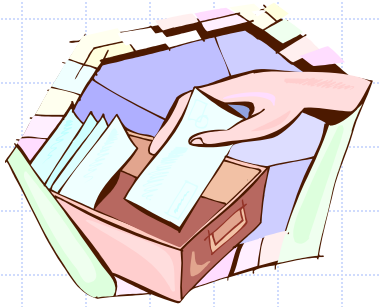
**删除操作 remove(k)**搜索键为 k 的项；
如果找到条目 (k, o)，则：将其替换为特殊对象 DEFUNCT（表示这个槽曾经被占用但现在已被删除）；
返回原值 o；
否则，返回 null。

# Updates with Linear Probing

**插入操作 put(k, o)**从哈希地址 h(k) 开始；向后探测，直到：
找到一个空槽；
或者找到一个 DEFUNCT 的槽；
或者找到同样的 k（则更新）；
或者探测满了 N 个槽（失败）；
将 (k, o) 存入该槽。

- To handle insertions and deletions, we introduce a special object, called ***DEFUNCT***, 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 ***DEFUNCT*** and we return element $o$
  - Else, we return ***null***
- put($k$, $o$)
  - 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 ***DEFUNCT***, or an entry with the key $k$
    - $N$ cells have been unsuccessfully probed
  - We store $(k, o)$ in cell $i$

# Double Hashing



- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series
  $(h(k) + jd(k)) \bmod N$
  for $j = 0, 1, \ldots, N-1$
- The secondary hash function $d(k)$ cannot have zero values
- Linear probing is a special case where $d(k) = 1$
- The table size $N$ must be a *prime* to allow probing of all the cells.

- Common choice of compression function for the secondary hash function:
  $d(k) = q - (k \bmod q)$
  where
  - $q < N$
  - $q$ is a *prime*
- The possible values for $d(k)$ are
  $1, 2, \ldots, q$

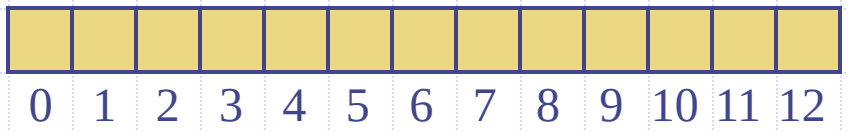
# Exercise & Self-Study

- The table size $N$ must be a prime to allow probing of all the cells.
- With a prime $N$, then eventually all table positions will be reached. Why?
- The positions are calculated using $(h(k) + jd(k)) \, mod \, N$, for $j = 0, 1, \ldots, N - 1$
- E.g., consider h(k) = 1, d(k) = 3,
  - If N = 12, what are the positions?
  - What if N = 13?

# Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - (k \bmod 7)$

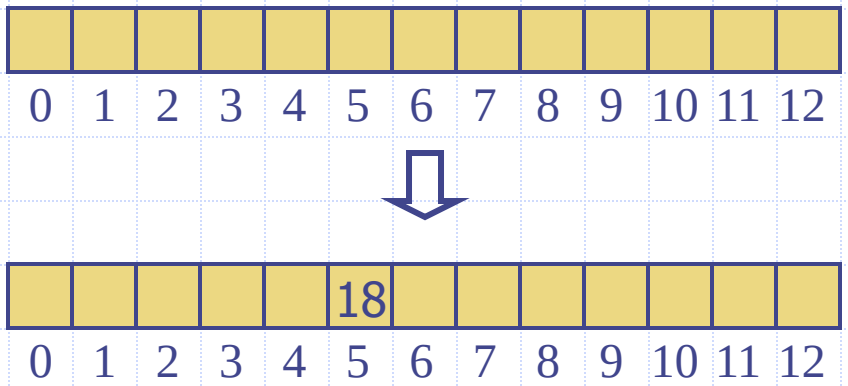
Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



# 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

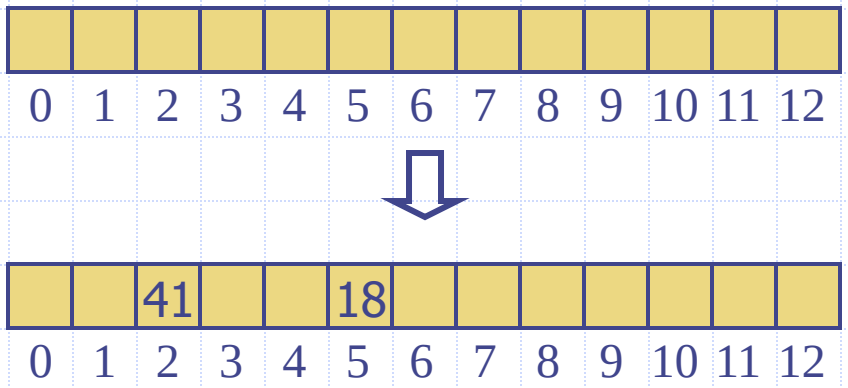
| $k$ | $h(k)$ | $d(k)$ | Probes |
|---|---|---|---|
| 18 | 5 | 3 | 5 |



# 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

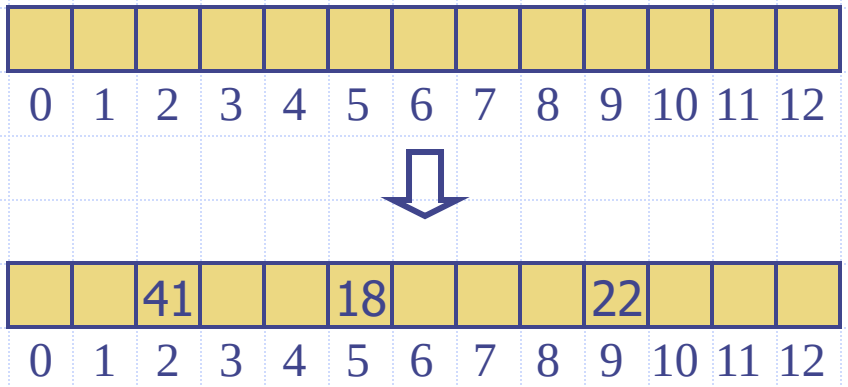
| $k$ | $h(k)$ | $d(k)$ | Probes |
|---|---|---|---|
| 18 | 5 | 3 | 5 |
| 41 | 2 | 1 | 2 |



# 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

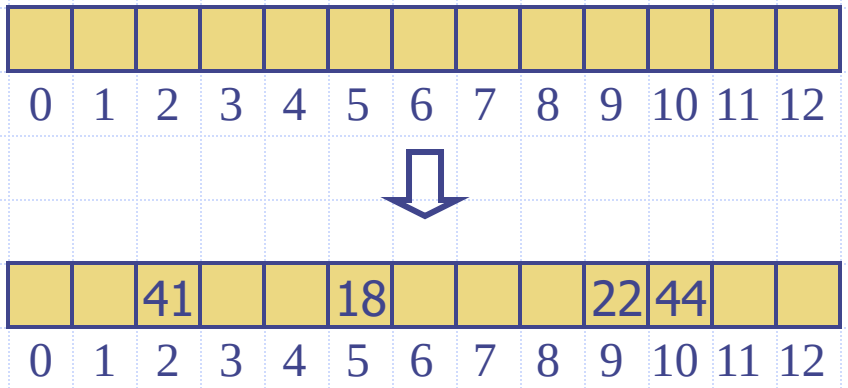
| $k$ | $h(k)$ | $d(k)$ | Probes |
|---|---|---|---|
| 18 | 5 | 3 | 5 |
| 41 | 2 | 1 | 2 |
| 22 | 9 | 6 | 9 |



# 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

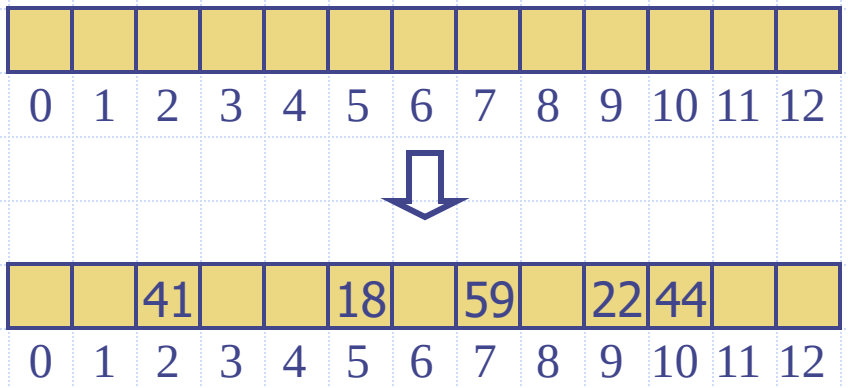
| $k$ | $h(k)$ | $d(k)$ | Probes | |
|---|---|---|---|---|
| 18 | 5 | 3 | 5 | |
| 41 | 2 | 1 | 2 | |
| 22 | 9 | 6 | 9 | |
| 44 | 5 | 5 | 5 | 10 |



# 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

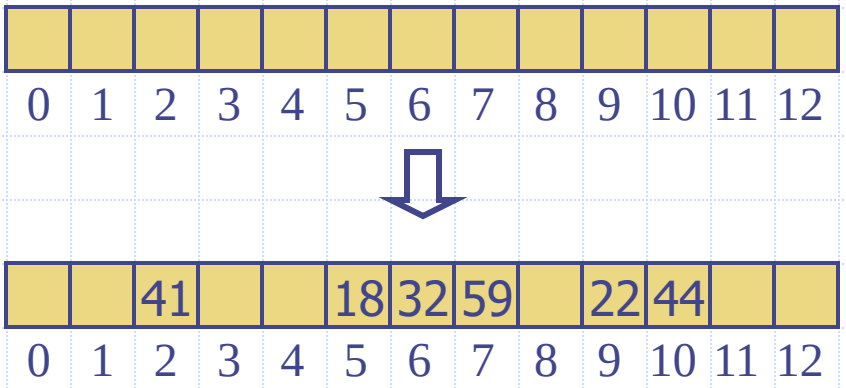
| $k$ | $h(k)$ | $d(k)$ | Probes | |
|---|---|---|---|---|
| 18 | 5 | 3 | 5 | |
| 41 | 2 | 1 | 2 | |
| 22 | 9 | 6 | 9 | |
| 44 | 5 | 5 | 5 | 10 |
| 59 | 7 | 4 | 7 | |



# 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

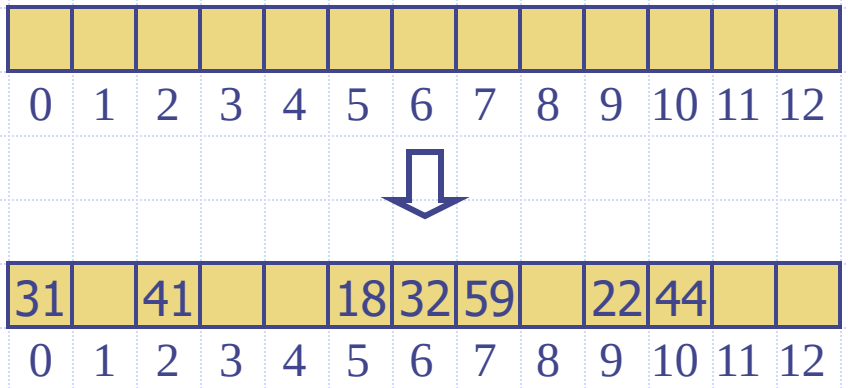
| $k$ | $h(k)$ | $d(k)$ | Probes | |
|---|---|---|---|---|
| 18 | 5 | 3 | 5 | |
| 41 | 2 | 1 | 2 | |
| 22 | 9 | 6 | 9 | |
| 44 | 5 | 5 | 5 | 10 |
| 59 | 7 | 4 | 7 | |
| 32 | 6 | 3 | 6 | |



# 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

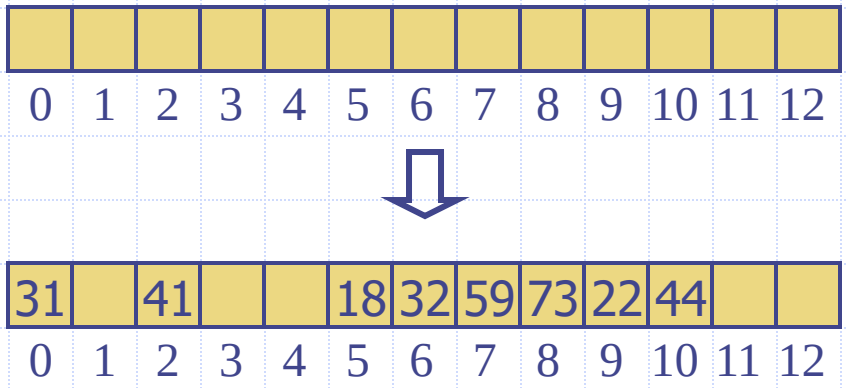
| $k$ | $h(k)$ | $d(k)$ | Probes | | |
|---|---|---|---|---|---|
| 18 | 5 | 3 | 5 | | |
| 41 | 2 | 1 | 2 | | |
| 22 | 9 | 6 | 9 | | |
| 44 | 5 | 5 | 5 | 10 | |
| 59 | 7 | 4 | 7 | | |
| 32 | 6 | 3 | 6 | | |
| 31 | 5 | 4 | 5 | 9 | 0 |



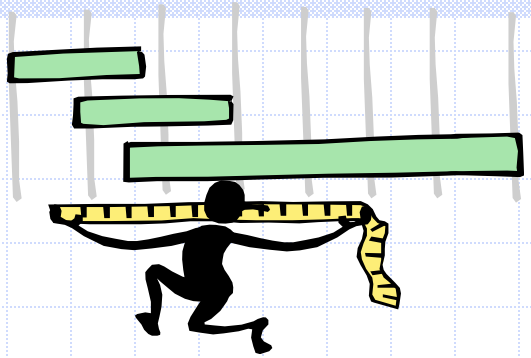
# Example of Double Hashing

- 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 | | |



# Performance of Hashing (Sec.10.2.3)



- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time.
- The worst case occurs when all the keys inserted into the map collide.
- The load factor $\alpha = n/N$ affects the performance of a hash table.
- In Java, maximal load factor is 0.75 (75%) – after that, rehashed
  - As for Vector, it is good to double the table size each rehash
- The expected running time of all the map ADT operations in a hash table is $O(1)$.
- In practice, hashing is very fast provided the load factor is not close to 100%.
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches

# Summary

Let $n$ denote the number of entries in the map, and we assume that the bucket array supporting the hash table is maintained such that its capacity is proportional to the number of entries in the map.

| Method | Unsorted List | Hash Table | |
|---|---|---|---|
| | | expected | worst case |
| get | $O(n)$ | $O(1)$ | $O(n)$ |
| put | $O(n)$ | $O(1)$ | $O(n)$ |
| remove | $O(n)$ | $O(1)$ | $O(n)$ |
| size, isEmpty | $O(1)$ | $O(1)$ | $O(1)$ |
| entrySet, keySet, values | $O(n)$ | $O(n)$ | $O(n)$ |

# Some Answers for Self-Study

- Why disperse?
  - To reduce numbers of collisions
- Why random?
  - Random means no pattern
  - If there is an obvious pattern, then the incoming data may have a matching pattern that leads to many collisions.