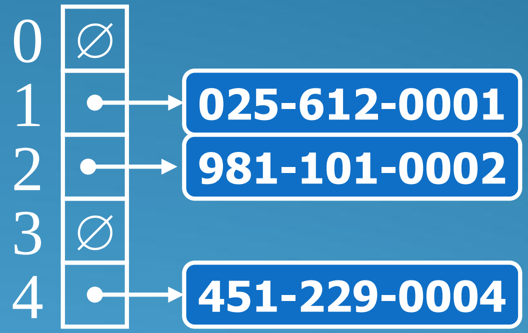
Part E

Hash Tables



Motivations of Hash Tables

- We have n items, each contains a key and value (k, value).
  - The key uniquely determines the item.
- Each key could be anything, e.g., a number in $[0, 2^{32}]$, a string of length *32*, array of numbers, etc.
- How to store the n items such that

  given the key k, we can find the position of the item with *key*= *k* in *O(1)* time.
  - Another constraint: space required is *O(n).*
- Linked list? Space *O(n)* and Time *O(n)*.
- Array? Time *O(1)* and space: too big, e.g.,
  - If the key is an integer in $[0, 2^{32}]$, then the space required is 2^{32}.
  - if the key is a string of length 30, the space required is 26^{30}.
- Hash Table: space *O(n)* and time *O(1)*.

Basic ideas of Hash Tables

- A hash function $\boldsymbol{h}$ maps keys of a given type with a wide range to integers in a fixed interval $[0, \mathbf{N}-1]$, where N is the size of the hash table such that

Problem:

It is hard to design a function *h* such that (1) holds.

What we can do:

- We can design a function *h* so that with high chance, (1) holds.
- i.e., (1) may not always holds, but (1) holds for most of the *n* keys.

Let’s Hash Strings

```
int StringHashFunc(string str, int TABLE_SIZE)
{
    int sum = 0;

    for(int i = 0; i < str.size(); i++)
            sum += str[i]-’a’;

    return sum % TABLE_SIZE;                    // Compression
}
```

Save Entry (Name, Age)

- ab, 27
- cab, 15
- bdb, 9
- ddb, 36

Hashing

- 1
- 2
- 5
- 7

Table size = 7

- 1
- 2
- 5
- 0

0	1	2	3	4	5	6
36	27	15			9	

What if another entry (bbd, 19)?
bbd =bdb = dbb = 5
Hash could give same index for different keys

A better hash function will deal with string as number
E.g. bdb= 1 * 26 * 26 + 3 * 26 + 1 = 755 % 7 = 6
E.g. bbd= 1 * 26 * 26 + 1 * 26 + 3 = 705 % 7 = 5

Hash Functions and Hash Tables

- A hash function $\boldsymbol{h}$ maps keys of a given type to integers in a fixed interval $[0, N-1]$
- Example:

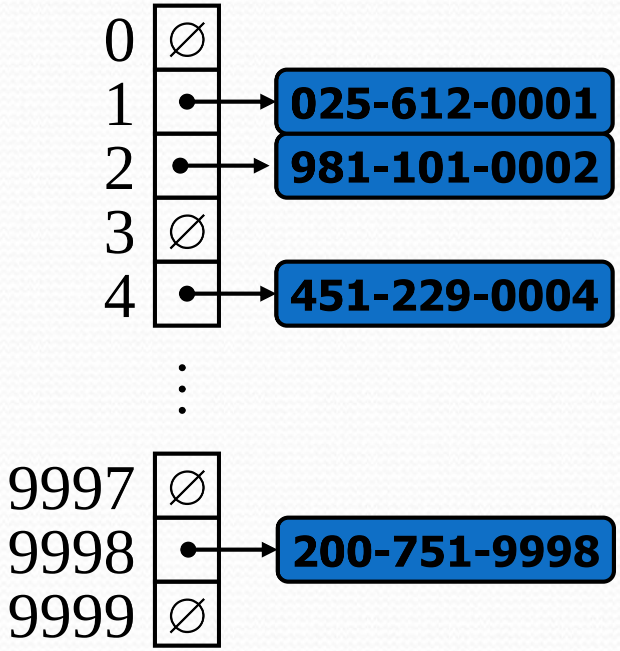
  $$\boldsymbol{h}(\text{int } \boldsymbol{x}) = \boldsymbol{x} \bmod \boldsymbol{N}$$

  is a hash function for integer keys
- The integer $\boldsymbol{h}(\boldsymbol{x})$ is called the hash value of key $\boldsymbol{x}$

- A hash table for a given key type consists of
  - Hash function $\boldsymbol{h}$
  - Array (called table) of size $\boldsymbol{N}$
- the goal is to store item $(\boldsymbol{k}, \boldsymbol{o})$ at index $\boldsymbol{i} = \boldsymbol{h}(\boldsymbol{k})$

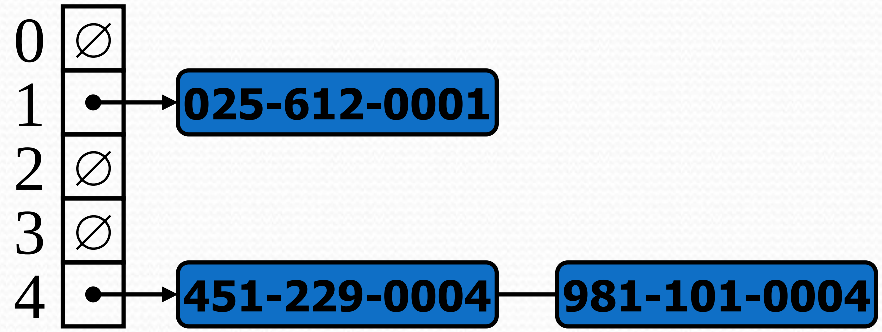
Example

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



Collision Handling

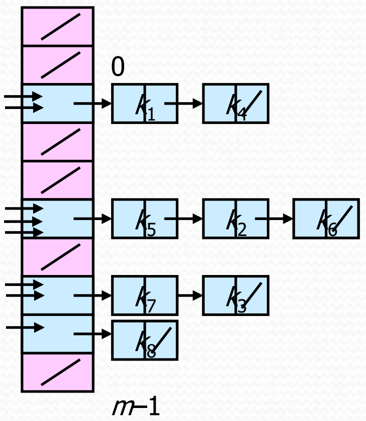
- Collisions occur when different elements are mapped to the same cell
- **Separate Chaining**: let each cell in the table point to a linked list of entries that map there



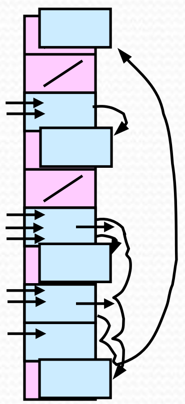
- Separate chaining is simple, but requires additional memory outside the table

Methods of Resolution

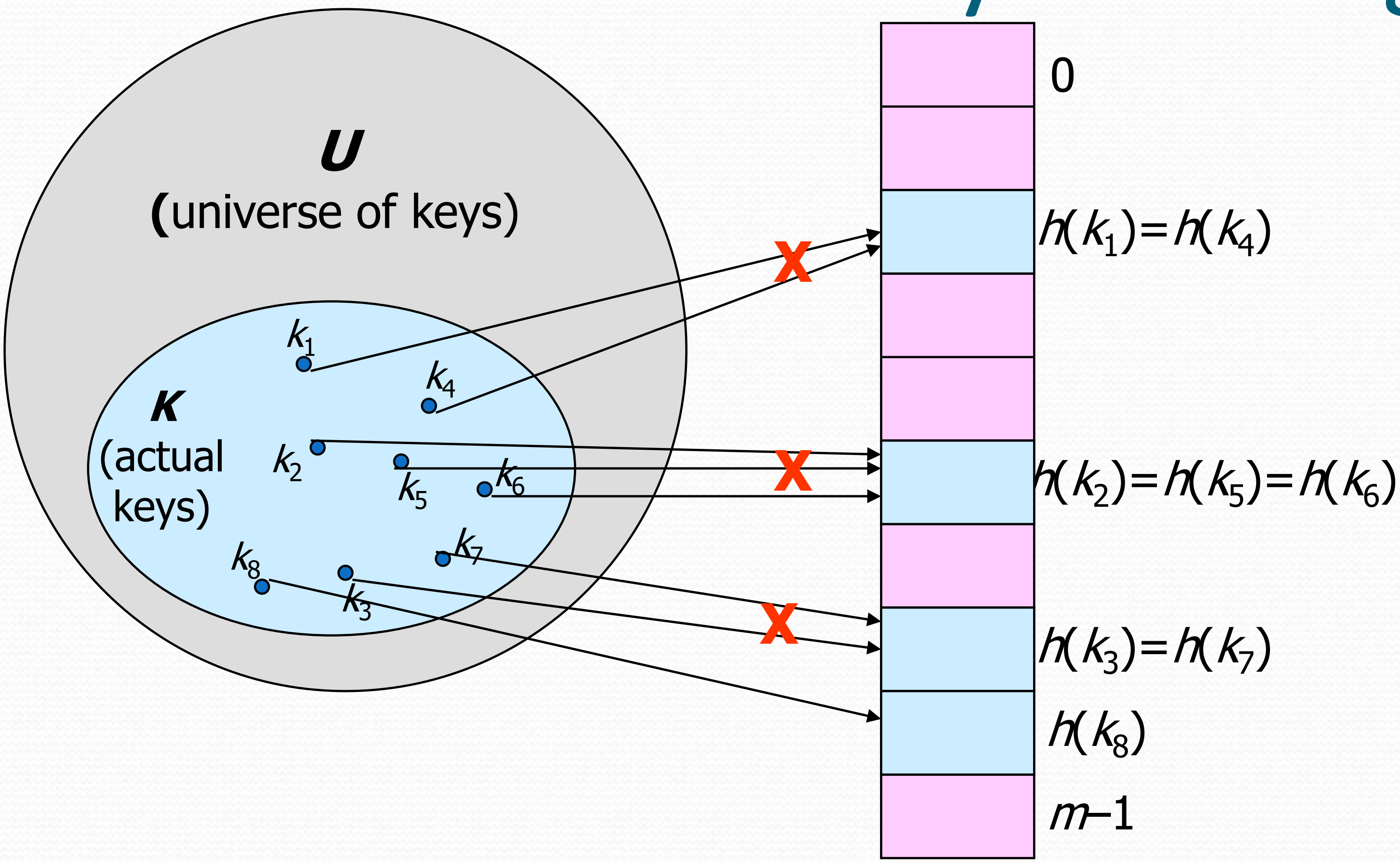
- Chaining:
  - Store all elements that hash to the same slot in a linked list.
  - Store a pointer to the head of the linked list in the hash table slot.



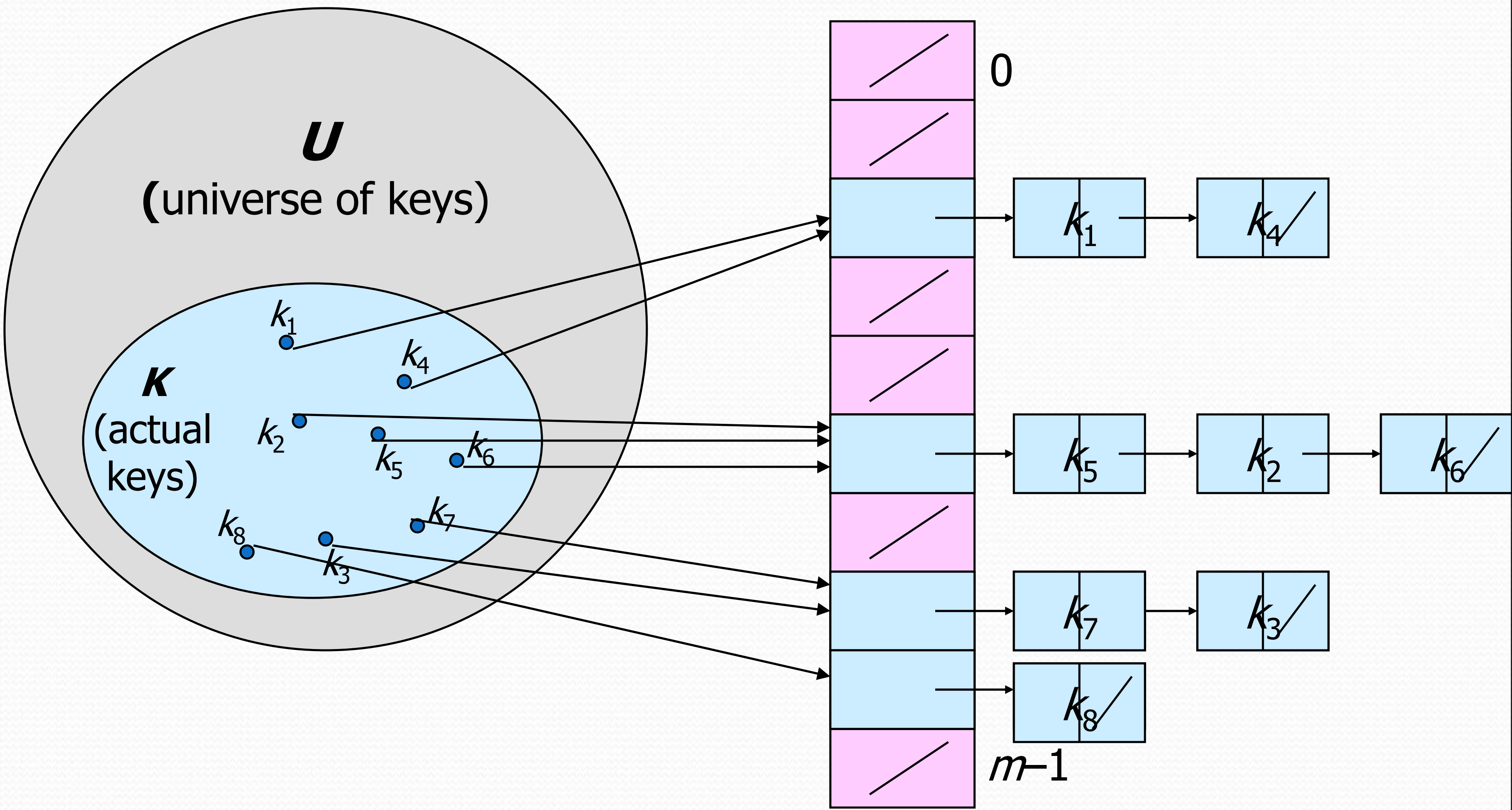
- Open Addressing:
  - All elements stored in hash table itself.
  - When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table.



Collision Resolution by Chaining



Collision Resolution by Chaining



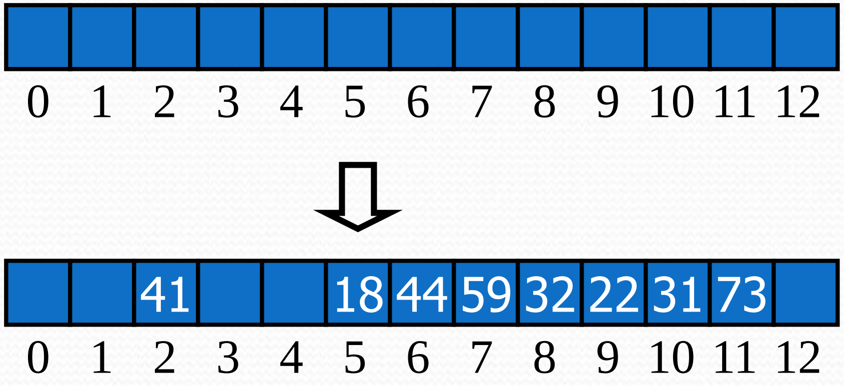
Open Addressing (closed hashing)

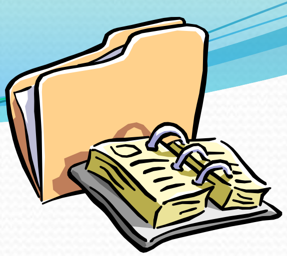
- We will use just 1D array. Elements either Null or has a pair (key, value)
- When add entry, check if its hash is empty or not
- If empty, then add it.
- If not empty, use some magic (probing) to determine another cell to set the pair. If not repeat until finding cell or declare full table.
- **Load factor:** n/N, where n is the number of items to store and N the size of the hash table = average keys per slot.
- $n/N \le 1$. To get a reasonable performance, $n/N < 0.5$.

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
  - h(x)=5, 2, 9, 5, 7, 6, 5, 8





Search with Linear Probing

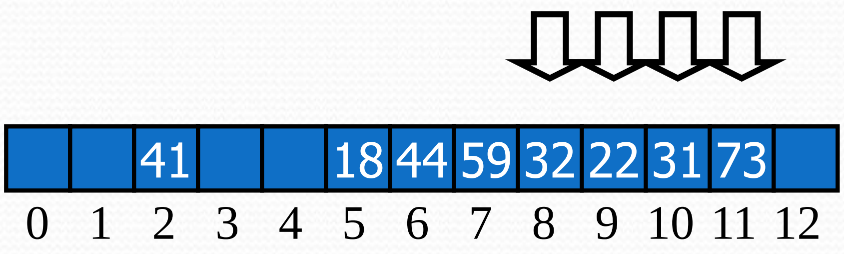
- Consider a hash table $\boldsymbol{A}$ that uses linear probing
- get($\boldsymbol{k}$)
  - We start at cell $\boldsymbol{h}(\boldsymbol{k})$
  - We probe consecutive locations until one of the following occurs
    - An item with key $\boldsymbol{k}$ is found, or
    - An empty cell is found, or
    - $\boldsymbol{N}$ 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*($\boldsymbol{k}$)
    $\boldsymbol{i} \leftarrow \boldsymbol{h}(\boldsymbol{k})$
    $\boldsymbol{p} \leftarrow 0$
    repeat
        $\boldsymbol{c} \leftarrow \boldsymbol{A}[\boldsymbol{i}]$
        if $\boldsymbol{c} = \varnothing$
            return ***null***
        else if $\boldsymbol{c}.\boldsymbol{key}\,() = \boldsymbol{k}$
            return $\boldsymbol{c}.\boldsymbol{element}()$
        else
            $\boldsymbol{i} \leftarrow (\boldsymbol{i} + 1) \bmod \boldsymbol{N}$
            $\boldsymbol{p} \leftarrow \boldsymbol{p} + 1$
    until $\boldsymbol{p} = \boldsymbol{N}$
    return ***null***

Search for 73

- Example:
  - $h(73) = 8$

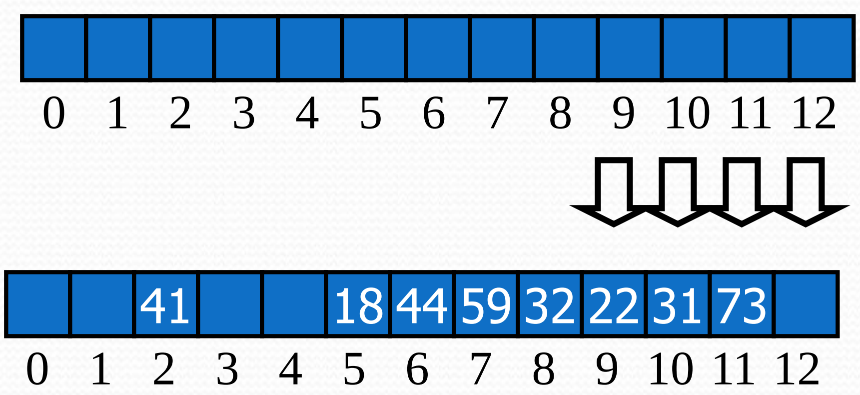
Yes, value array = 73



Search for 35

- Example:
  - ***h*(*35*)** =9

Empty Cell! Not found



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***
  - Else, we return ***null***
- 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***

Remove 31

- Get Mod
- Search for 31
- If found Mark it deleted

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

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

Remove 59

- Get Mod
- Search for 59
- If found Mark it deleted

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

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

Insert 57

- H(57) = 57%13 = 5

First Empty Slot

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

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44	57	32	22	X	73	

Performance of Hashing

- 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
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is

$$1 / (1 - \alpha)$$

- The expected running time of all the 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

What else in hashing?

- Hash Functions
  - How to select the Mod? Prime? Power of 2?..
  - Compression Function
  - Good hashing for: Arrays, Strings, Big Numbers, ..
- Probing
  - Issues with Probing
  - Quadratic Probing
- Double Hashing / Perfect Hashing
- Theories: E.g. Expected Cost for Search