# Hash Table

**IT5003:** Data Structures and Algorithms

(**AY2019/20 Semester 1**)

# Lecture Outline

- **Direct Addressing Table**

- **Hash Table**
  - Definition
  - Hash Function
    - Modulo Method
    - Multiplicative Method
  - Collision Resolution
    - Restructuring of Hash Table
    - Open Addressing

# Table ADT: Recap and Preview

| | Unsorted Array/List | Sorted Array | Sorted LinkedList |
|---|---|---|---|
| insert | **O(1)** | O(N) | O(N) |
| delete | **O(N)** | **O(N)** | **O(N)** |
| search | O(N) | **$O(\log_2 N)$** | O(N) |

| | BST | Balanced BST | Hash Table |
|---|---|---|---|
| insert | O(h) | **$O(\log_2 N)$** | better? |
| delete | O(h) | **$O(\log_2 N)$** | better? |
| search | O(h) | **$O(\log_2 N)$** | better? |

# Table ADT: Recap and Preview

- BST is dependent on height of tree:
  - Inconsistent behaviour ranging from O(log N) to O(N)
- Balanced BST (e.g. AVL Tree) provides consistent O(log N) performance
  - But we can do even better!
- **Hash table** can support Table ADT in **constant time on average!**

Back to Basics?

# DIRECT ADDRESSING TABLE

# Example: The SBS Bus Problem

- Consider a system to manage information about **bus services** for the bus companies SBS and Tower Transit
- The main operations are:

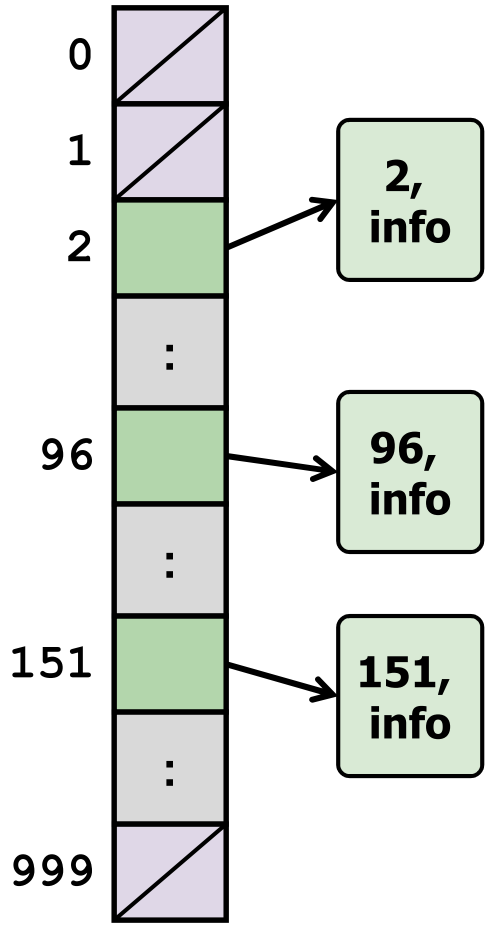
| Operations | Functionality |
| --- | --- |
| `Find(N)` | **Does bus service N exists?** |
| `Insert(N)` | **Add bus service N** |
| `Delete(N)` | **Remove bus service N** |

# Observations: **The SBS Bus Problem**

- The bus service are indicated by an integer between **[1 … 999]**

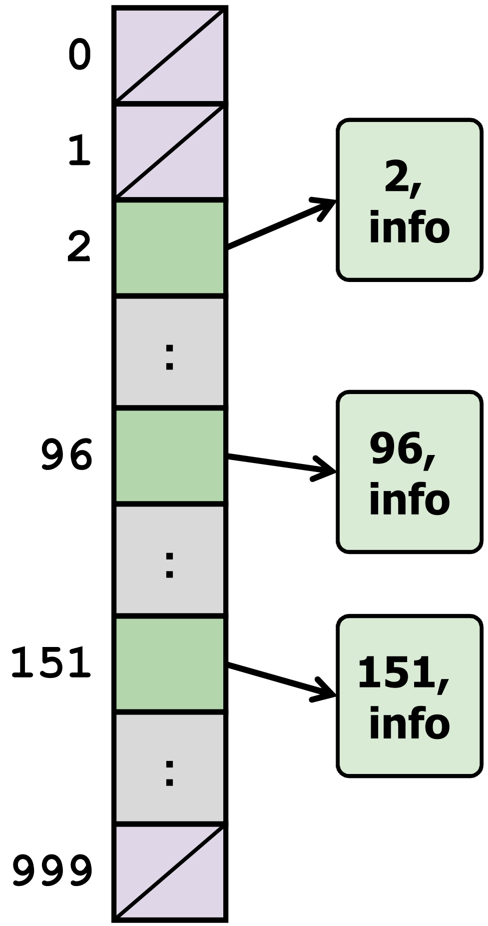
- **Efficient Solution:**
  - Use an array of 1000 elements
  - Element at index ***N*** represent the bus service ***N***
    - Can store an object with information about the bus service

- Known as **direct addressing table**



# Direct Addressing Table: Complexity

| Operations | Basic Steps | Big-O |
| --- | --- | --- |
| `Find(N)` | | |
| `Insert(N,data)` | | |
| `Delete(N)` | | |



# Direct Addressing Table: Summary

- Direct addressing table is **very efficient**

- However, there are many **restrictions**
  - Key must be **integer**
    - e.g. how about bus service "NR10", "151e", "96A"?
  - Range of keys must be **small**
  - Keys must be **dense**
    - Most keys in the range are valid
    - Not many "gaps" in the key values

Improved Direct Addressing Table

# HASH TABLE

# Example: The IT-Bank Account

- Suppose the account number of **IT Bank** Accounts follow the format:

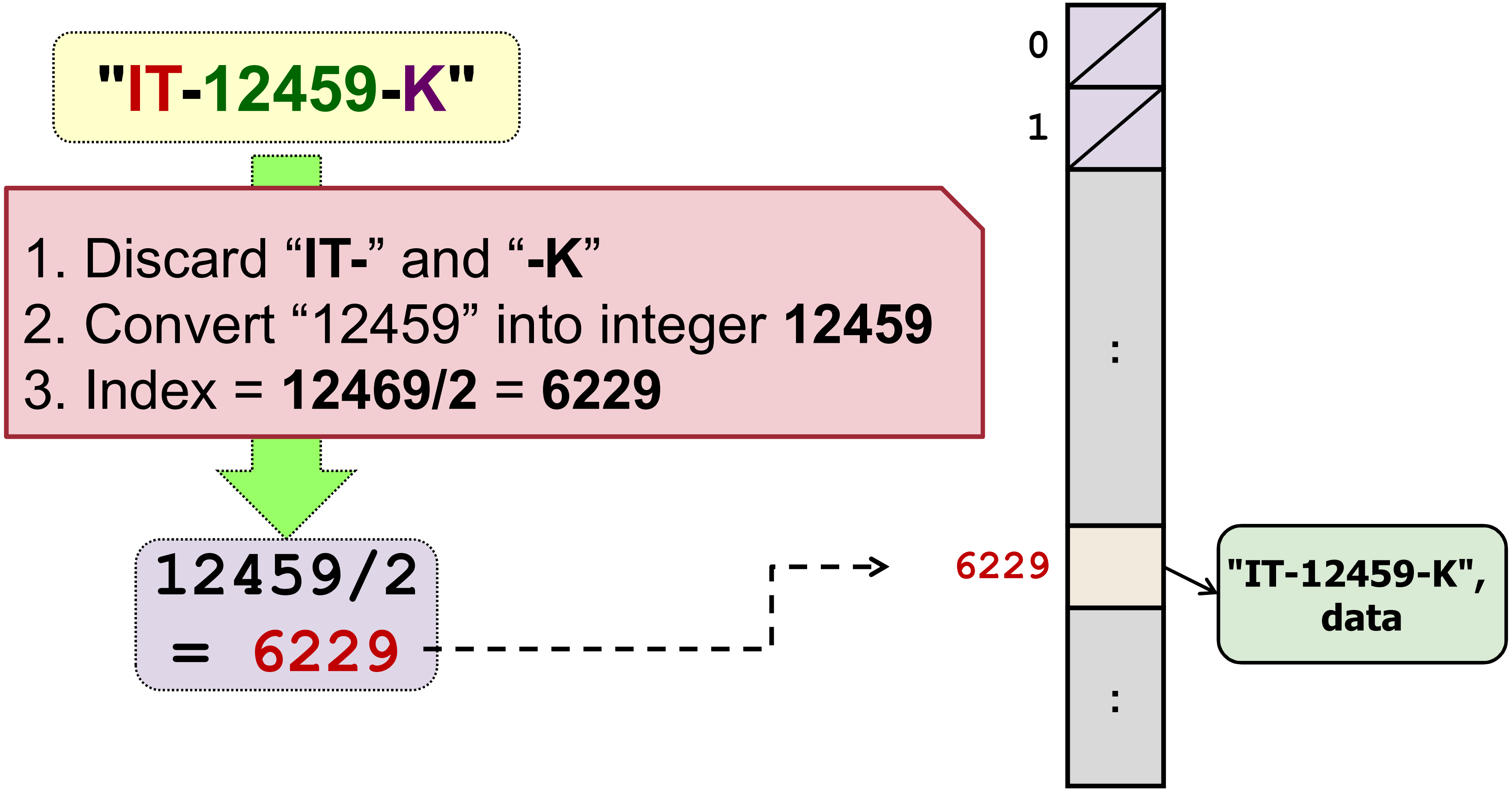
**IT-12459-K**

| Bank name prefix | Running sequence number | Check alphabet |
| --- | --- | --- |
| **Bank name prefix:** same for every account | **Running sequence number:** unique for each account, only odd numbers are used | **Check alphabet:** computed from the sequence number to detect fake account |

- Can we use direct addressing table?
  - Can we **convert** the account number to integer?

# Example: **A possible conversion**

- Treat the account number as a string and then:



# Hash Table: Generalized Idea

- Use a **conversion function** to map:
  - Non-integer ➔ integer
  - Sparse integers in a large range ➔ a dense integers in a smaller range

- This conversion function is known as **hash function**
  - The fundamental idea behind hash table!
  - **Hash Table** (or **Hash Map**)

    = **Direct Addressing Table** + **Hash Function**

# Hash Table: Operations

- One additional step:
  - Apply hash function *h*() to the key value
  - *h*(key) gives the **home address** of the key

| Operations | Basic Steps |
|---|---|
| `Find(N)` | `return a[ h(N) ]` |
| `Insert(N,data)` | `a[ h(N) ] = data` |
| `Delete(N)` | `a[ h(N) ] = None` |

- Time complexity now depends on the performance of **the hash function *h*()**

# Hash Tables: Problems

- In the IT-Bank example:
  - The result of the hash function is **unique**
  - Each key is mapped to a difference home address
  - known as **perfect hash function**

- Unfortunately, this is **not always true!**
  - Given two different keys, it is possible for a hash function to give the **same result!**

```
Hash(key1) == Hash(key2)
but key1 != key2
```

- This problem is known as **collision**

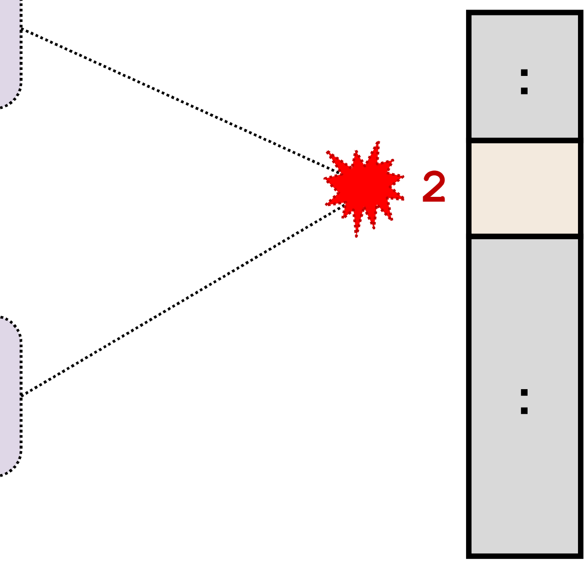
# Example: Hashing Collision

- Given the hash function:

```
h( key ) = key % 17
```

```
h( 19 ) = 19 % 17 = 2
```

```
h( 87 ) = 87 % 17 = 2
```



# Hash Table: **Important Issues**

- How to define good **hash function**?
  - What are the properties of a good hash function?

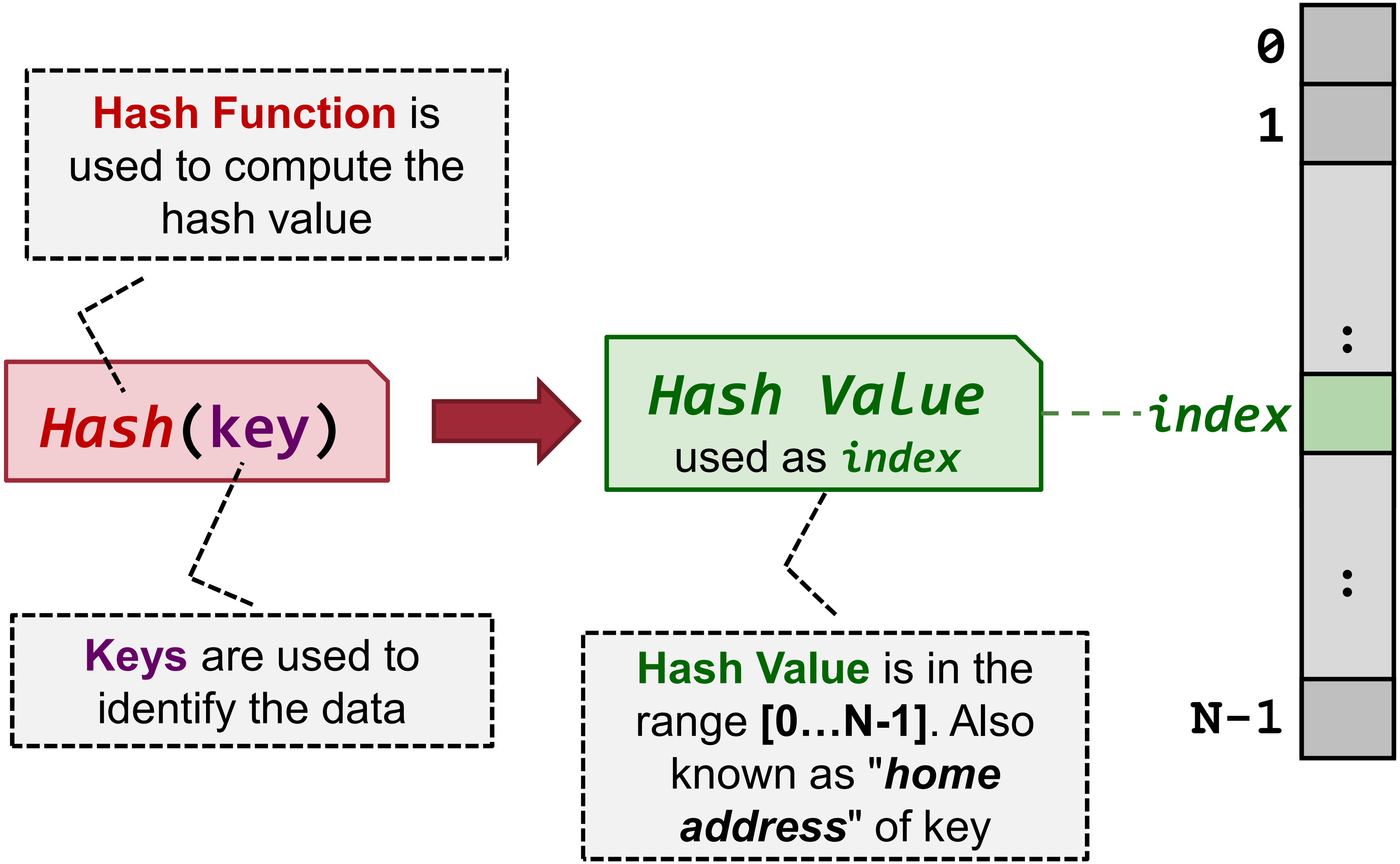
- How to **resolve collision**?

My hash is better than your hash ☺

# HASH FUNCTIONS

# Terminology

- Given a **hash table** of size ***N***



# Good Hash Functions: Properties

- Fast to compute
  - O(1) if possible
- Scatter keys **evenly** throughout the hash table
- Result in none of few collisions
- Allow the hash table to be small
  - As compared to the entire range of possible keys

# Bad Hash Functions: Example

- Select Digits:

$$\mathit{Hash}(\ d_0 d_1 d_2 \ldots d_7\ ) = d_2 d_7$$

  - *hash*(67<u>**5**</u>4737<u>**8**</u>) = **58**
  - *hash*(34<u>**5**</u>0810<u>**8**</u>) = **58**

- What happen when you hash Singapore's house phone numbers by selecting the ***first three digits***?

# Perfect Hash Function

- **Perfect Hash Function:**
  - One-to-one mapping between the keys and array indices
  - → **NO collision**
  - Possible if we know all keys in advance

- **Example:**
  - The IT-Bank account number example
  - When a compiler search for keywords

- **Minimal perfect hash function**:
  - The table size is the same as the number of keywords supplied

# Hash Function: How To

- We will cover the following approaches in this course:

  a. Uniform hash function

  b. Division method

  c. Multiplication method

  d. Hashing of strings

# Uniform Hash Function

- **Uniform Hash Function:**
  - Distribute the keys **evenly throughout** the hash table

- **Formal definition:**
  - Given **K keys** and **M locations** in hash table
  - ***H*( K )** is uniform if each location receive no more than $\left\lceil \frac{K}{M} \right\rceil$ keys

# Uniform Hashing Function: **Example**

- **Given:**
  - Keys are integers uniformly distributed in [0,X)
  - Hash table of size ***m*** ( m < X )

- We can hash the keys uniformly into the table by:

$$k \in [0, X)$$

$$hash(k) = \left\lfloor \frac{km}{X} \right\rfloor$$

> *k* is the key value
> [ ]: close interval
> ( ): open interval
> Hence, $0 \le k < X$
> $\lfloor\ \rfloor$ is the ***floor*** function

# Hash Function: Modulo Method

- Given a hash table of **m slots**
  - Modulo operator "%" maps an integer to a value between **0** and **m-1**:

$$hash(k) = k \,\%\, m$$

- One of the most popular methods

- Behavior of the hash function depends on:
  - Key distribution
  - Table size **m**

# Modulo Method: Table Size m

- Generally, we want the hash function to generate "random-like" home addresses even if the keys are in continuous range
  - Some table size should be avoided in modulo method **due to commonly encountered key sequence**

- **Example:**
  - **$m = 10^n$**
    - Hash function returns the last n-digits of the key!
  - **$m = 2^n$**
    - Hash function returns the last n-bits of the key!

# Modulo Method: Table Size m

- **Rule of thumb:**
  - Choose table size to be a **large prime number** close to a power of 2

- **Several reasons:**
  - We can get a "shuffling" effect by first multiplying the key with another prime number ***q***:

$$hash(k) = (k * q) \,\%\, m$$

  - Prime table size allows effective collision resolution method (more later)

# Hash Function: Multiplicative Method

- Hash function takes the following form:
  1. Multiply key with a real number A between [0..1]
  2. Extract the fractional part
  3. Multiply by hash table size, **m**

$$hash(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

- **Rationale:**
  - Fraction part of multiplication is "random-like" even for continuous key range
- A common choice for A is the reciprocal of **golden ratio**:

$$A = \frac{\sqrt{5}-1}{2}$$

# Hash Function: Handling Strings

- For non-integer keys:
    1. Convert the key into an integer
    2. Then apply hash function on the result
- Let us use string as illustration

```
def HashString( str ):
    sum = 0

    for ch in str:
        #ord() gives the ASCII encoding of char
        sum += ord(ch)

    return sum % tableSizeM
```

# Hashing String: Problems

- The method used is not very good:
  - Many strings converted to the **same sum**
  - ➔ Result in large number of collisions

- Example:
  - *Hash*( "abc" ) == *Hash*( "bac" ) == *Hash*( "cba" )

- **Problem:**
  - The conversion fails to take the **position of each character into account**
  
  ➔ Permutations of a strings get the same sum!

# Hashing Strings: **Better Conversion**

- **Idea:**
  - Associate a **weight** to each position in string
- **Common approach:**
  - Multiply each position by $X^{position}$, for a chosen X
- **Example (X = 17, m = 1023):**
  - Hash( "abc" )
    $= (97*17^2 + 98*17^1 + 99*17^0)$ % 1023
    = 29798 %1023 = **131**
  - Hash(" bac")
    $= (98*17^2 + 97*17^1 + 99*17^0 )$ % 1023 = **403**

# Hashing Strings: **Better Conversion**

- The idea can be implemented efficiently:
  - Using **Horner's Rule**

```
def HashStringPos( str ):
     sum = 0

     for ch in str:
        #ord() gives the ASCII encoding of char
        sum = sum * X + ord(ch)

    return sum % tableSizeM
```

- In actual implementations, popular choice of **X** is **31** or **37**

# Hash Function: Summary

- First convert non-integer key into integer
  - Quality of conversion affects the hashing
- Perform hashing using the integer key
  - Take note of the range and characteristics of the input when designing hash function
  - Try to meet the qualities of a good hash function
- Modulo method is one of the most common choices for hash function

Warning! Warning! Collision Imminent!

# COLLISION RESOLUTION

# Probability of Collision

**von Mises Paradox (The Birthday Paradox)**

"How many people must be in a room before the probability that **some share a birthday** becomes **at least 50 percent**?"

Q($n$) = Probability of **unique** birthday for ***n*** people

$$= \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \frac{362}{365} \cdots \frac{365-n+1}{365}$$

P($n$) = Probability of collisions (same birthday) for ***n*** people

= 1 – Q($\boldsymbol{n}$)

P(**23**) = **0.507** (i.e. you need only **23 people** in the room!)

# Probability of Collision

- In the hashing context:
  - If we insert 23 keys into a table with 365 slots, more than half of the time we will get collisions!
  - Such a result is quite counter-intuitive

- Since collision is very likely

  ➔ Any good hash table implementation needs to take collision into account!

# Collision resolutions: Overview

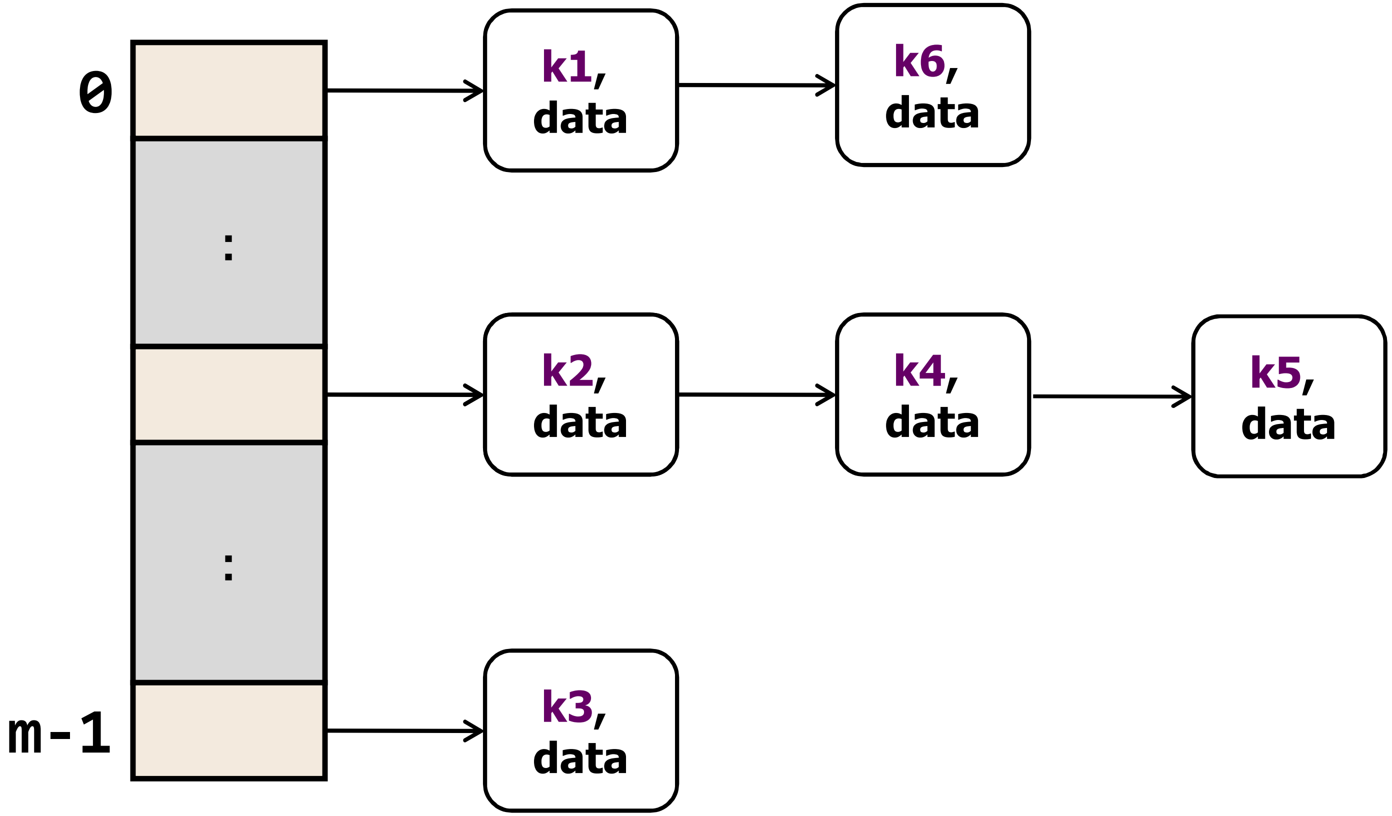
- There are two main approaches:

1. **Restructuring the Hash Table**
   - Change the way we store items in hash table to accommodate multiple items per slot

2. **Open Addressing**
   - Instead of looking at a single home address in the hash table, we look for another **suitable location** for the collided item

# Restructuring HT: **Separate Chaining**

- **Idea:**
  - Instead of storing one item per slot, we can store a **linked list of items** at each slot



# Separate Chaining: **Operations**

- Separate chaining requires modification of the hash table operations:

| Operations | Basic Steps |
|---|---|
| ***Find*(N)** | Search through the linked list at **a[ *h*(N) ]** |
| ***Insert*(N,data)** | Add data to the head of linked list at **a[ *h*(N) ]** |
| ***Delete*(N)** | Search through the linked list at **a[ *h*(N) ] and delete *N*** |

# Separate Chaining: **Analysis** (1/2)

- Let
  - ***N*** = number of keys in a hash table
  - ***m*** = size of hash table

- We can then define the **load factor α**
  - **α = *N* / *m***
  - Measures **how full is the hash table**

- As ***N*** increases, **α** also increases
  - Large **α** indicates higher chance of collision

  - For separate chaining, **α** can rise above 1.0

# Separate Chaining: **Analysis** (2/2)

- The operations are dependent on the **length of the linked list** at each slot

| Operations | Average Running Time |
|---|---|
| ***Find*(N)** | $O(1 + \alpha)$ |
| ***Insert*(N,data)** | $O(1)$ |
| ***Delete*(N)** | $O(1 + \alpha)$ |

- If $\alpha$ is bounded by a constant, then the running time gives O(1) for all operations
- Question:
  - What is the **worst case analysis**?

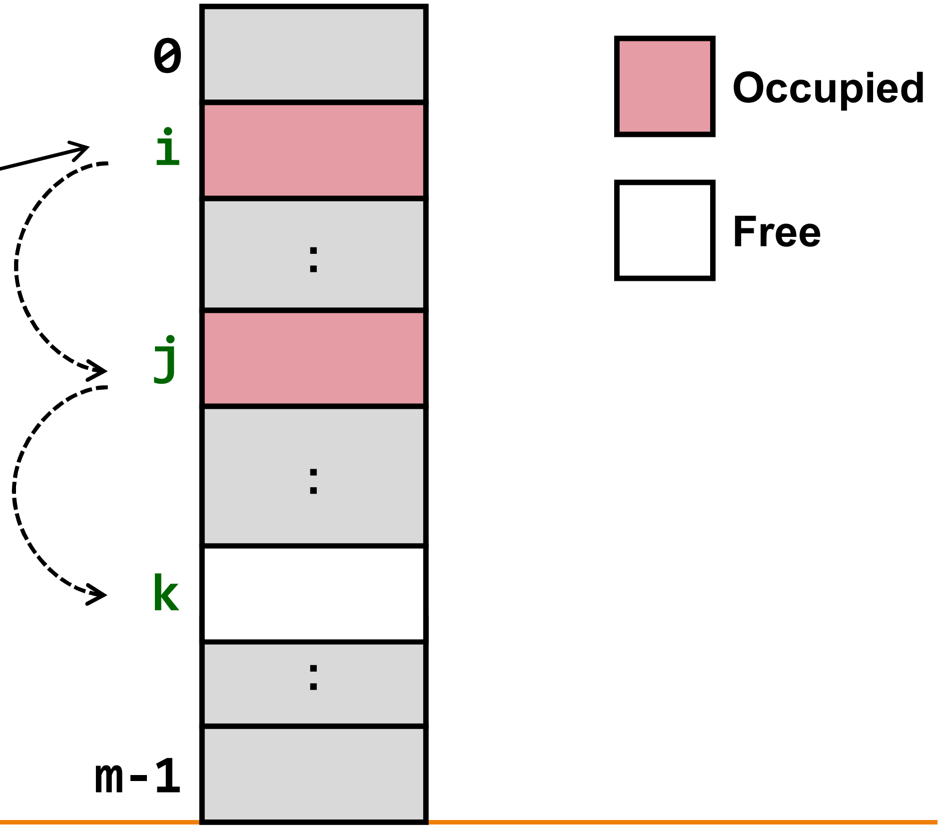
I'm not at home, perhaps you can try my office address?

# OPEN ADDRESSING

# Open Addressing: **Overview**

- If collision occurs:
  - We **probe** (try out) other **suitable locations**
  - The series of locations probed is known as the **probe sequence**

***Hash***( **key** ) = **i**




{ **i**, **j**, **k** } is the **probe sequence**

# Open Addressing: **Common Schemes**

- There are **three common schemes** to generate the probe sequence
  - Decides where to look for suitable slots when collision occurs

1. **Linear Probing**

2. **Quadratic Probing**

3. **Double Hashing**

# Linear Probing: **Overview**

**Linear Probe Sequence**

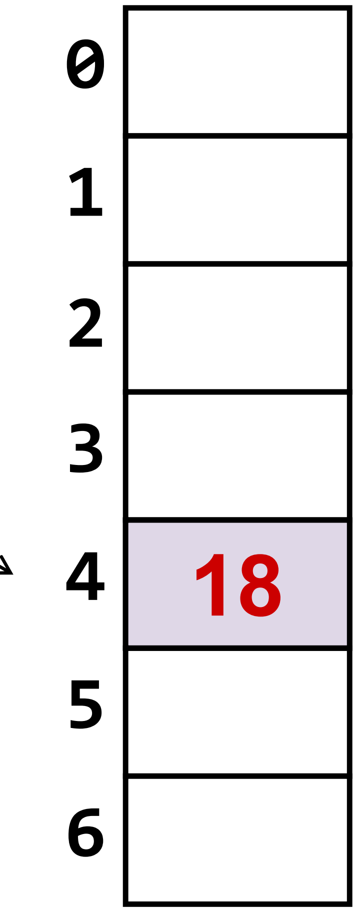
`idx, idx+1, idx+2, …, m-1, 0, 1, 2, …, idx-1`

- **Basic Idea:**
  - Whenever collision occurs, we look at the next slot in the table
    - Wraps around when the end of the array is encountered
  - **Continue until:**
    - An empty slot is found **OR**
    - Visited **every slot** and could not find an empty slot

# Example: Insert 18

```
Hash( key )
= key % 7
```

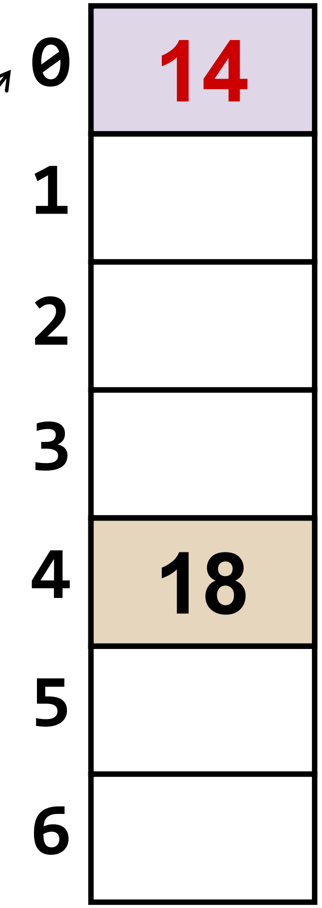
```
Hash( 18 )
= 18 % 7
= 4
```



# Example: Insert 14

*Hash*( key )
= key % 7

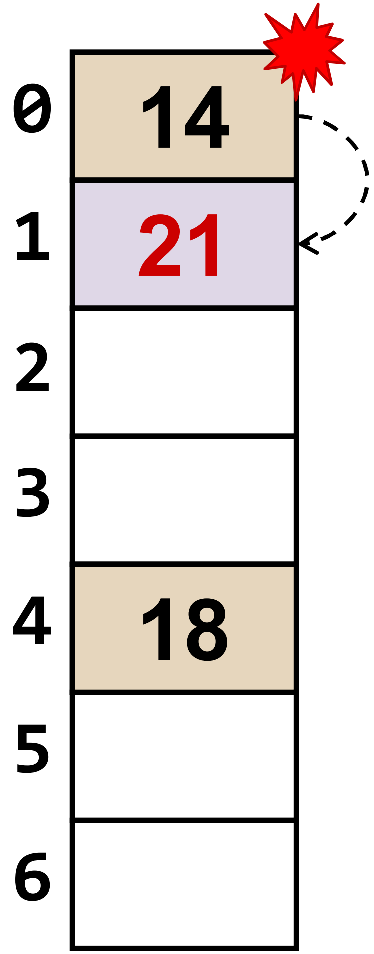
*Hash*( 14 )
= 14 % 7
= 0



# Example: Insert 21

```
Hash( key )
= key % 7
```

```
Hash( 21 )
= 21 % 7
= 0
```

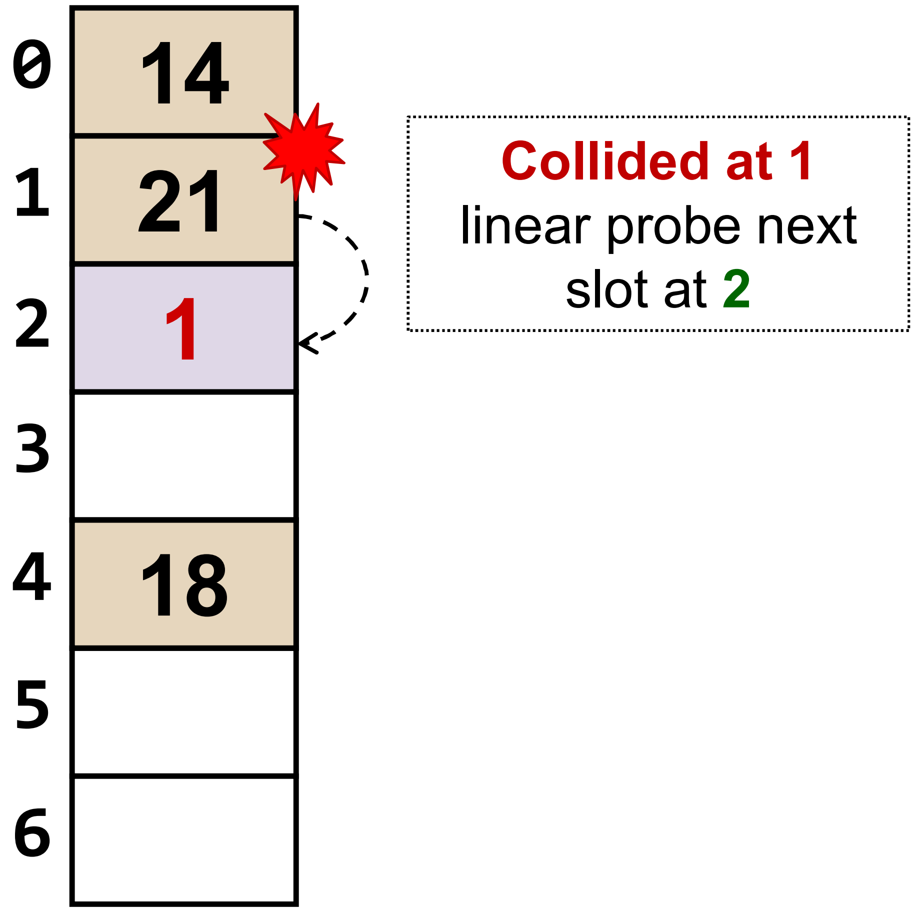



**Collided at 0**
linear probe next slot at 1

# Example: Insert 1

```
Hash( key )
= key % 7
```

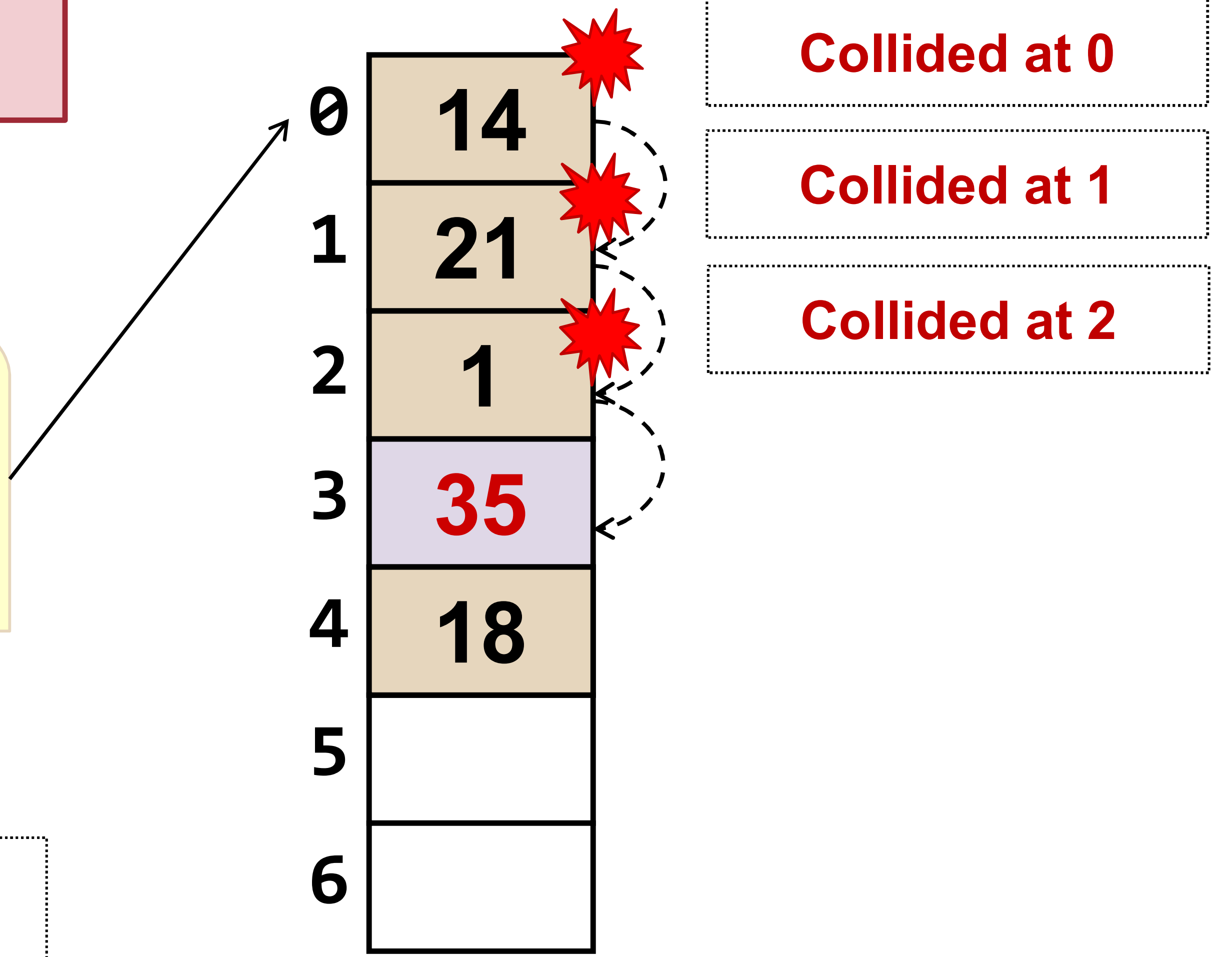
```
Hash( 1 )
= 1 % 7
= 1
```



# Example: Insert 35

```
Hash( key )
= key % 7
```

```
Hash( 35 )
= 35 % 7
= 0
```

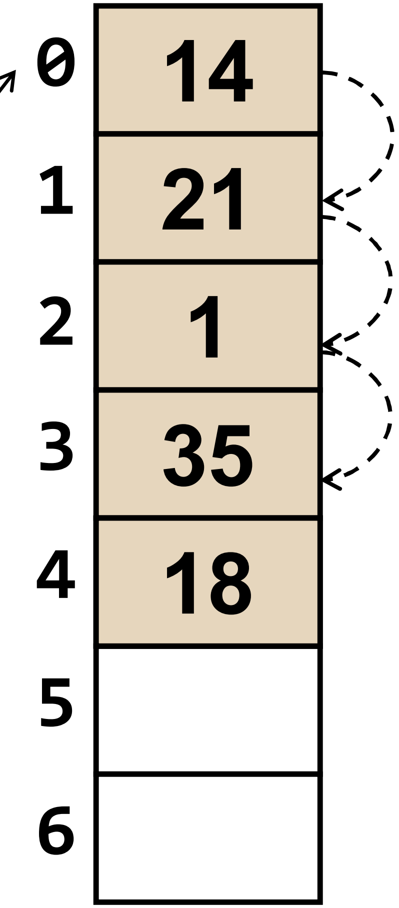



**Probe Sequence**
{ 0, 1, 2, 3 }

# Example: Find 35

```
Hash( key )
= key % 7
```

```
Hash( 35 )
= 35 % 7
= 0
```



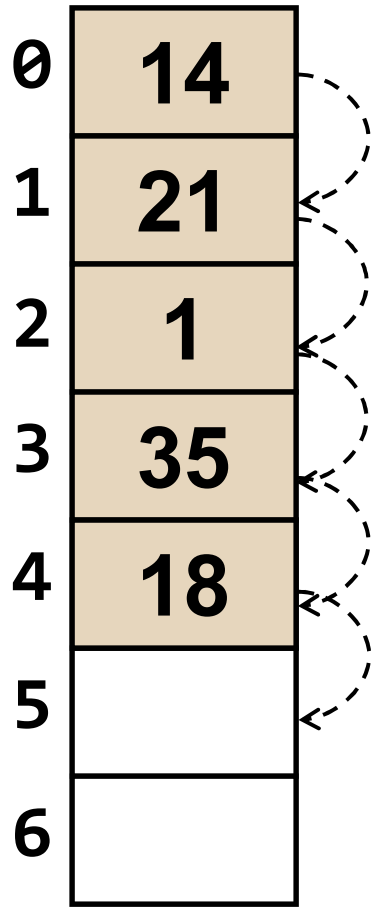

**Search**
we have to do
linear probe too!

**Probe Sequence**
{ 0, 1, 2, 3 }

# Example: Find 8

```
Hash( key )
= key % 7
```

```
Hash( 8 )
= 8 % 7
= 1
```




**Search**
Linear probe until the item is found or reach an empty slot

**Probe Sequence**
{ 1, 2, 3, 4, 5 }

# Linear Probing: **Probe Sequence**

- The **find** and **delete** must be replicate the same probe sequence used in **insert**
  - If the probe sequence is broken ➔ leads to incorrect operation!

- **Example:**
  - The probe sequence of **find(35)** is the same as **insert(35)**
  - The probe sequence of **find(8)** is the same as **insert(8)** if we were to perform the insertion

- This requirement complicates the deletion operation

# Example: Delete 21

```
Hash( key )
= key % 7
```

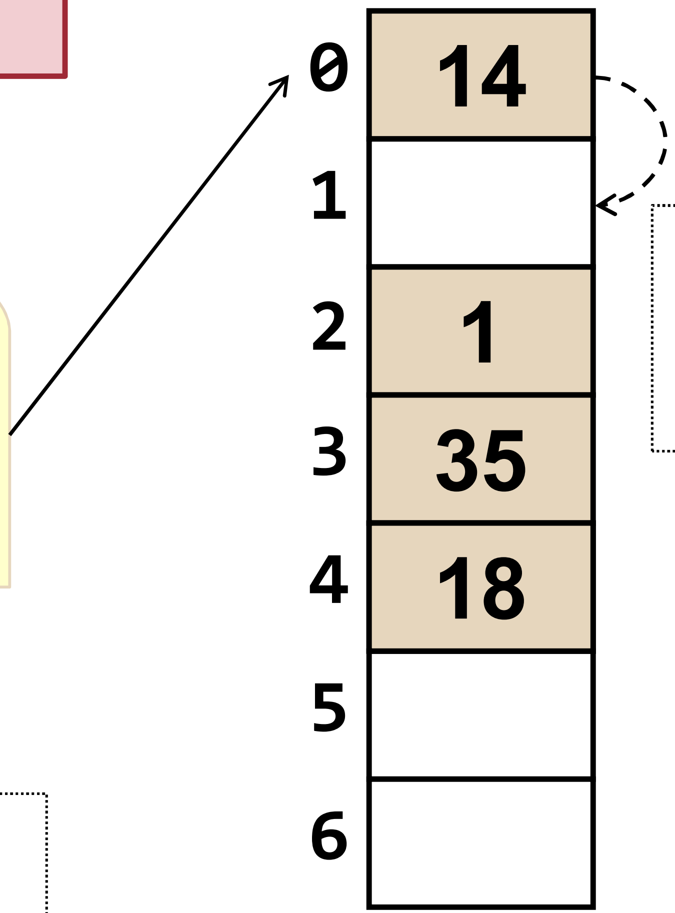
```
Hash( 21 )
= 21 % 7
= 0
```

| | |
|---|---|
| 0 | 14 |
| 1 | ~~21~~ |
| 2 | 1 |
| 3 | 35 |
| 4 | 18 |
| 5 | |
| 6 | |

# Example: Find 35

```
Hash( key )
= key % 7
```

```
Hash( 35 )
= 35 % 7
= 0
```




**Search**
Reached an empty slot
➔ **35 not found!**

**Probe Sequence**
{ 0, 1 }

# Linear Probing: Deletion

- **Problem:**
  - Cannot break off the probe sequence in any location

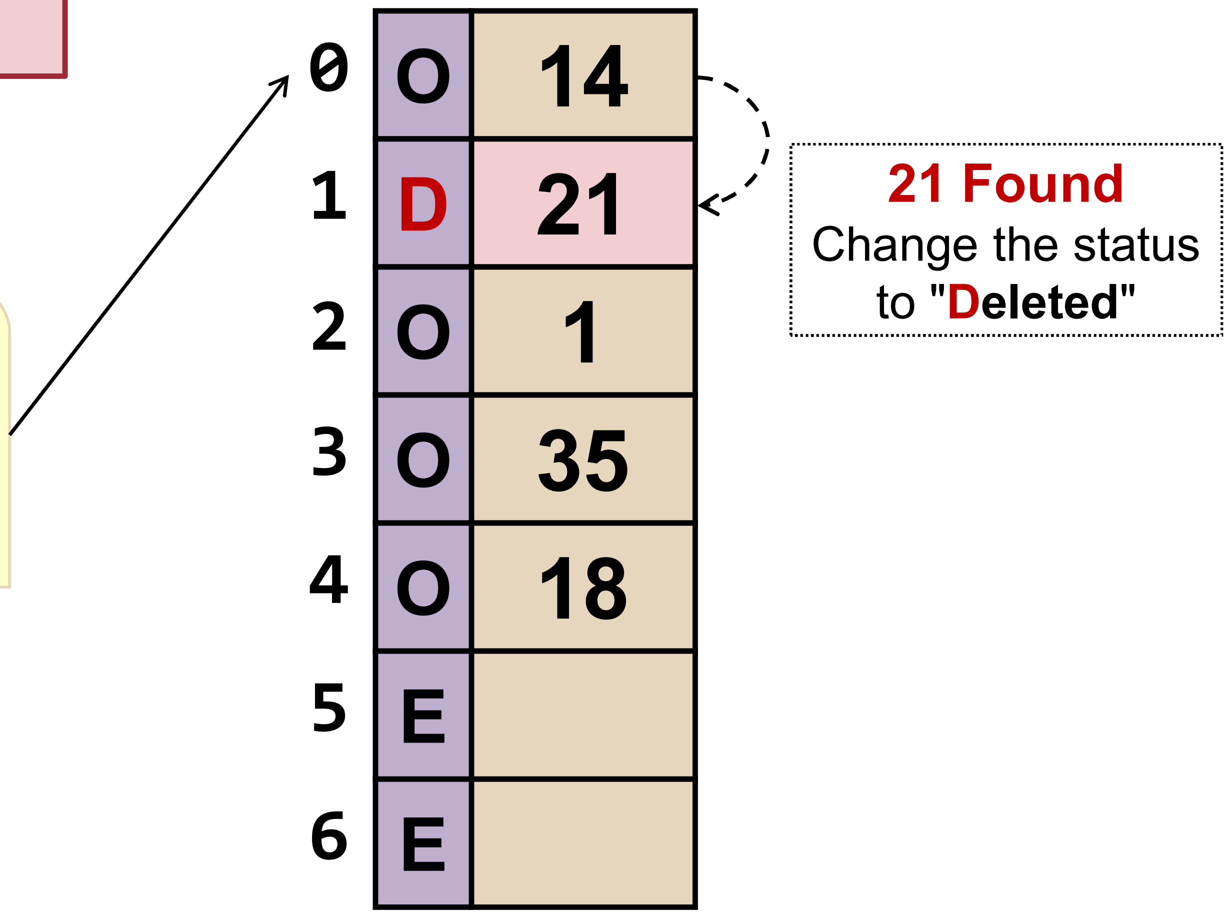
  ➔ Cannot remove any item!

- **Solution:**
  - Keep a status information for each slot
    - { **Empty**, **Occupied**, **Deleted** }
  - Mark the slot as **Deleted** when item is removed
  - Known as **lazy deletion**

# Example: Delete 21

*Hash*( key )
= key % 7

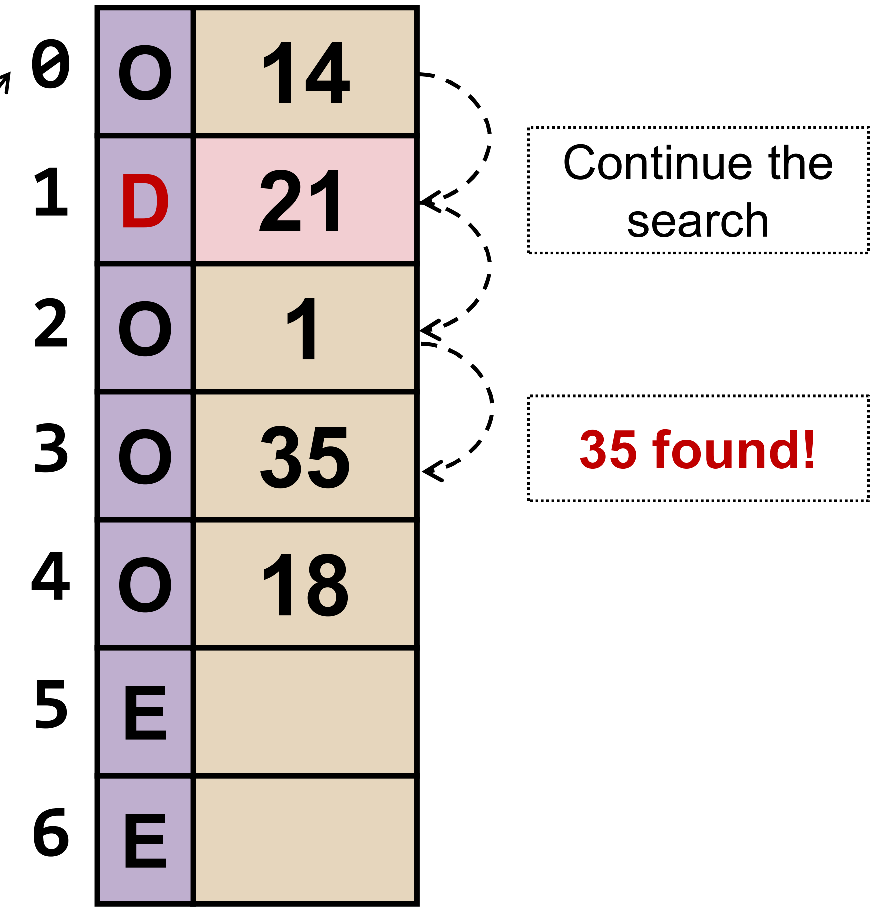
*Hash*( 21 )
= 21 % 7
= 0



# Example: Find 35

```
Hash( key )
= key % 7
```

```
Hash( 35 )
= 35 % 7
= 0
```




During search, "**deleted**" slots are treated as "**occupied**" to maintain the probe sequence

# Example: Insert 15

```
Hash( key )
= key % 7
```

```
Hash( 15 )
= 15 % 7
= 1
```

| | | |
|---|---|---|
| 0 | O | 14 |
| 1 | O | 15 |
| 2 | O | 1 |
| 3 | O | 35 |
| 4 | O | 18 |
| 5 | E | |
| 6 | E | |

**15 inserted**
the previously deleted slot is changed to occupied

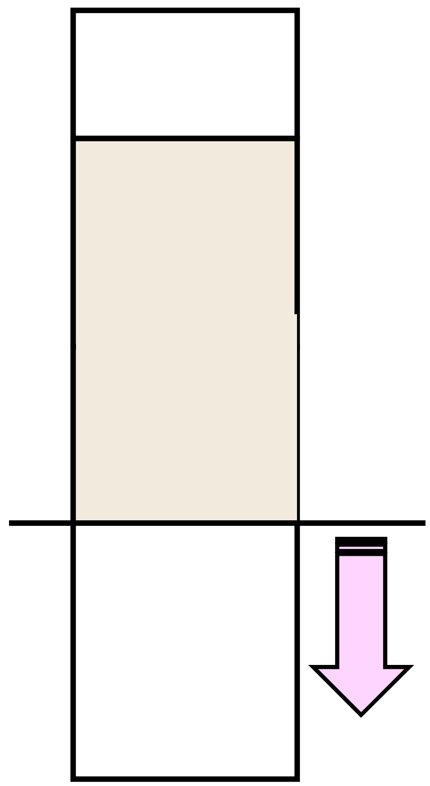
During insertion, "**deleted**" slots are treated as "**empty**"

# Linear Probing: **Problem**

- Linear probing can cause:
  - Many **consecutive occupied slots** in the hash table
  - Known as the **clustering effect**
- Clustering is undesirable:
  - Increase the running time for all hash table operations
- In **linear probing**:
  - **Clustering occurs around the home address** of a key
  - Known as **Primary Clustering**

# Linear Probing: **Problem**

- Probe sequence in linear probing:
  - Hash( key )
  - (Hash( key ) **+ 1**) % m
  - (Hash( key ) **+ 2**) % m
  - ……..



- Primary Cluster keeps expanding as a result

# QUADRATIC PROBING

# Quadratic Probing: **Overview**

- **Basic Idea:**
  - Essentially a modified linear probing
  - Instead of probing the next slot, we jump by $P^2$ slots, where **P** is the number of probing

- **Quadratic Probe Sequence:**

| Hash( key ) | P = 0 |
|---|---|
| (Hash( key ) + $1^2$) % m | P = 1 |
| (Hash( key ) + $2^2$) % m | P = 2 |
| (Hash( key ) + $3^2$) % m | P = 3 |
| …… | …… |

# Quadratic Probing: **Overview**

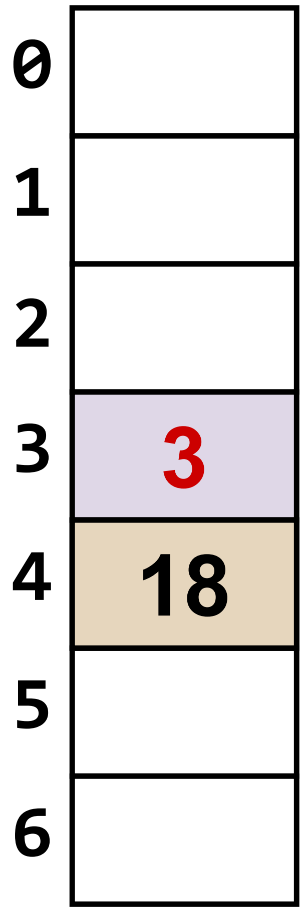
- The probe sequence can be calculated as a displacement from the home address
  - $P_i = ( \mathit{Hash}(key) + i^2) \% m$

- The sequence can also be calculated **as a displacement from the previous probe**
  - $P_0 = Hash(key)$
  - $P_1 = (P_0 + 1) \% m$
  - $P_2 = (P_1 + 3) \% m$
  - $P_3 = (P_2 + 5) \% m$
  - $P_i = (P_{i-1} + (2i-1)) \% m$

# Example: Insert 3

```
Hash( key )
= key % 7
```

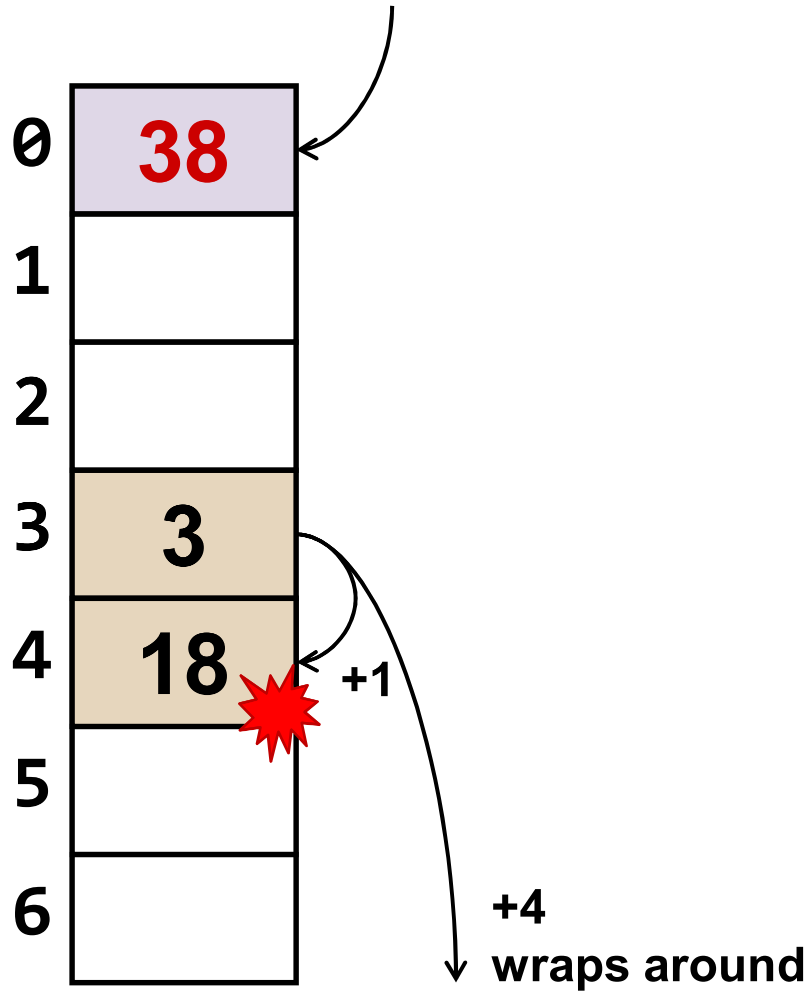
```
Hash( 3 )
= 3 % 7
= 3
```



# Example: Insert 38

```
Hash( key )
= key % 7
```

```
Hash( 38 )
= 38 % 7
= 3
```



# Quadratic Probing: **Theorem**

- **Question:**
  - How do we know when to stop in quadratic probing?

- **Theorem:**

> If load factor **α** < 0.5 and the table size `m` **is prime**, then we can **always find an empty slot** using quadratic probe

  - One of the advantages of prime table size

# Quadratic Probing: **Problem**

- If two keys share the **same home address**, then their probing sequences **are the same**

  ➔ **Clusters form** along the path of probing

  - Known as **Secondary Clustering**

- Secondary clustering is less severe compared to primary clustering

# DOUBLE HASHING

# Double Hashing: **Overview**

- **Idea:**
  - Give different probe sequence for keys **even when they have the same home address**
  - Use an additional hash function:
    - To determine the displacement used in each probe
    - Also known as **secondary hash** function
- **Double Hashing Probe Sequence:**

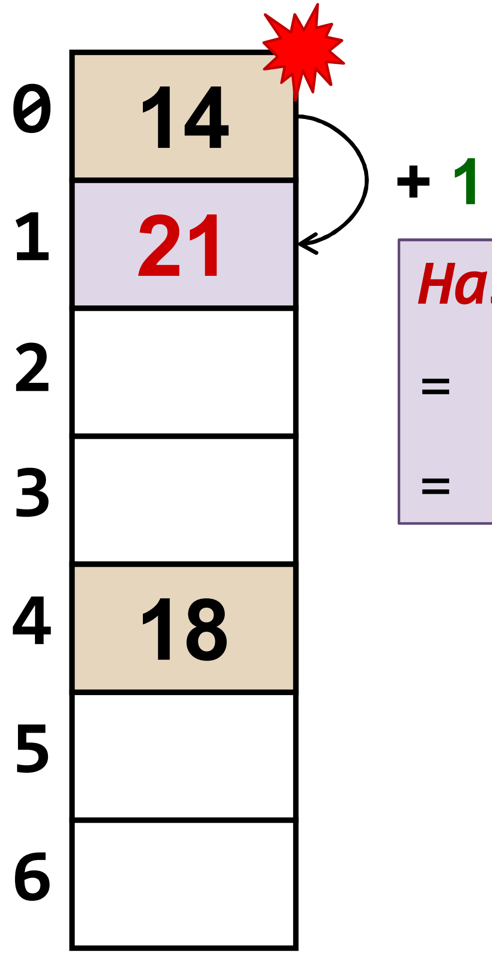
| | |
|---|---|
| Hash( key ) | P = 0 |
| (Hash( key ) + 1 * $Hash_2$(key)) % m | P = 1 |
| (Hash( key ) + 2 * $Hash_2$(key)) % m | P = 2 |
| …… | …… |
| (Hash( key ) + i * $Hash_2$(key)) % m | P = i |

# Example: Insert 21

*Hash*( key )
= key % 7

$Hash_2$( key )
= key % 5

*Hash*( 21 )
= 21 % 7
= 0



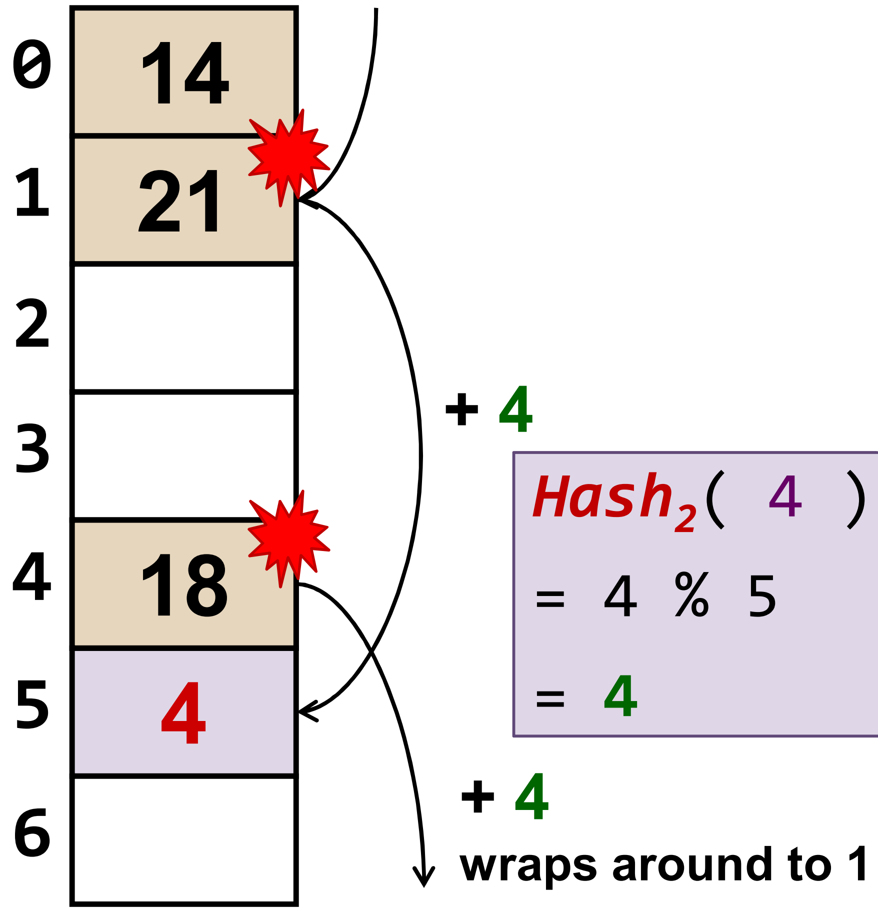

$Hash_2$( 21 )
= 21 % 5
= 1

# Example: Insert 4

*Hash*( key )
= key % 7

*Hash*$_2$( key )
= key % 5

*Hash*( 4 )
= 4 % 7
= 4



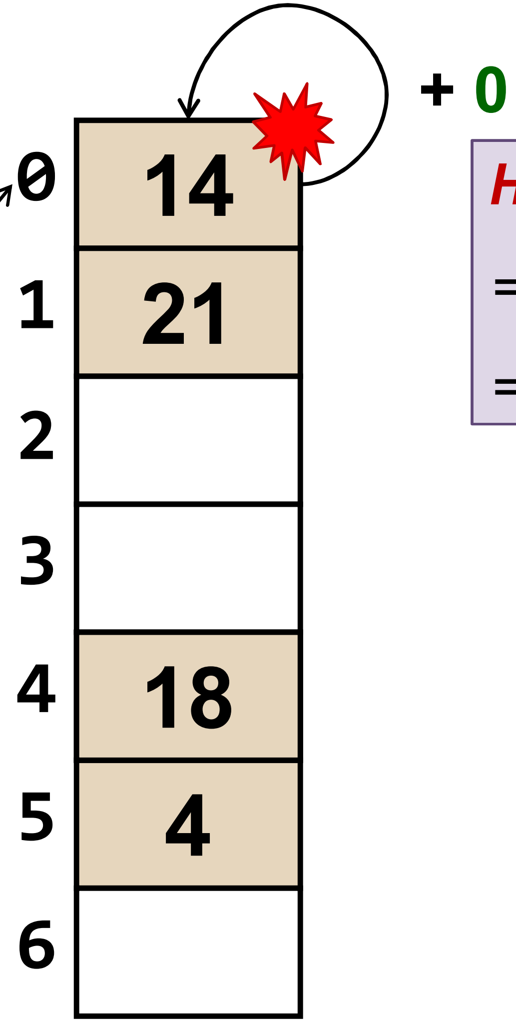

*Hash*$_2$( 4 )
= 4 % 5
= 4

# Example: Insert 35

*Hash*( key )
= key % 7

$Hash_2$( key )
= key % 5

*Hash*( 35 )
= 35 % 7
= 0




$Hash_2$( 35 )
= 35 % 5
= 0

## Double Hashing: **Secondary Hash Function**

- **Be careful:**
  - Secondary hash function should **never evaluates to zero**
    - to avoid infinite probe sequence

- **Easy solution:**
  - Secondary hash function usually takes the form of

    $Hash_2(k) = (k \% 5) + 1$ //Result = [1..5]

  OR

    $Hash_2(k) = 5 - (k \% 5)$ //Result = [1..5]

# Double Hashing: **Theorem**

If the secondary hash function evaluates to values that are **coprime** of the table size **`m`**,

Then, we **always take no more than `m` probes to find empty slot in table if it exists**

- Two integers **`i`** and **`j`** are **coprime if their greatest common divisor** is 1
- If the table size **`m`**, is **prime**, then any positive value lesser than **`m`** are coprime!

  ➔ Easy choice for secondary hash function

# Collision Resolution: Summary

- Good collision resolution method should:

1. Minimize clustering (Primary and Secondary)
2. Always find an empty slot if it exists
3. Give different probe sequences when 2 keys collide (i.e. no secondary clustering)
4. Fast

# Rehash (Enlarging Hash Table)

- When to rehash?
  - When the table is getting full, the operations are getting slow
  - For quadratic probing, insertions might fail when the table is more than half full

- Rehash operation:
  1. Build another table about twice as big **with a new hash function**
  2. Insert each key from the original table into the new table using the new hash function
  3. Delete the original table

# Table ADT: With Hash Table

| | Unsorted Array/List | Sorted Array | Sorted LinkedList |
|---|---|---|---|
| `insert` | O(1) | O(N) | O(N) |
| `delete` | O(N) | O(N) | O(N) |
| `search` | O(N) | $O(\log_2 N)$ | O(N) |

| | BST | Balanced BST | Hash Table |
|---|---|---|---|
| `insert` | O(h) | $O(\log_2 N)$ | **O(1) avg** |
| `delete` | O(h) | $O(\log_2 N)$ | **O(1) avg** |
| `search` | O(h) | $O(\log_2 N)$ | **O(1) avg** |

# Summary

- How to hash?
  - Criteria for good hash functions
- How to resolve collision?
  - Separate chaining
  - Linear probing
  - Quadratic probing
  - Double hashing
- Problem on deletions
- Primary clustering and secondary clustering

END