CS2040C Data Structures and Algorithms

Hashing

For efficient look-up in a table

Outline

- Direct Addressing
- What is hashing?
- Hash Table/Hash Function
- What is collision?
- How to resolve collision?
- Primary clustering and secondary clustering
- STL hash table

Lookup Table

- Most applications require a data structure to:
  - Store a number of items
  - Delete a particular item
  - Search for one particular item using a special piece of information (**key**)

- **Lookup Table** is an abstraction that captures the requirements above
  - Many different implementations possible!

Lookup Table: Example Implementations

Operations	Unsorted Array / Linked List	Sorted Array	Sorted Linked List
`Insert`	O(1)	O(N)	O(N)
`Delete`	O(N)	O(N)	O(N)
`Search`	O(N)	O($\log_2$ N)	O(N)

- Confirm your understanding by verifying the complexity required for each of the operations

Direct Addressing Table

A simplified version of hash table

Example: The SBS Bus Problem

- Consider a system to manage information about **bus services** for the bus company SBS
- The main operations are:
    - We assume bus service number is an integer

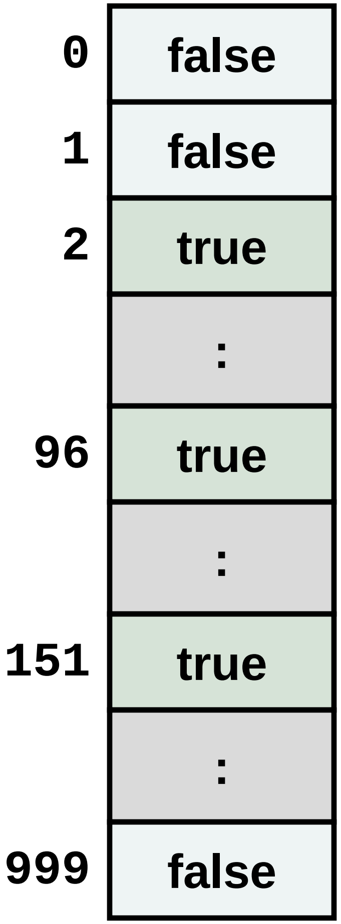
Operations	Functionality
***Find*(N)**	**Does bus service N exists?**
***Insert*(N)**	**Add bus service N**
***Delete*(N)**	**Remove bus service N**

Observations: The SBS Bus Problem

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

- Efficient Solution:
  - Use a **boolean array** of 1000 elements
  - Element at index ***N*** represents the bus service ***N***
    - **True** == exists, **False** == not exist

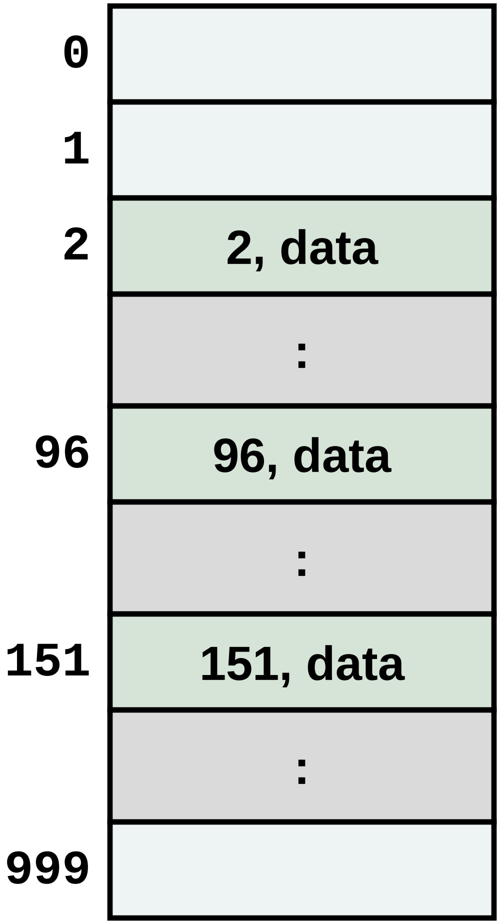
- Known as **direct addressing table**



0	false
1	false
2	true
	:
96	true
	:
151	true
	:
999	false

Direct Addressing Table

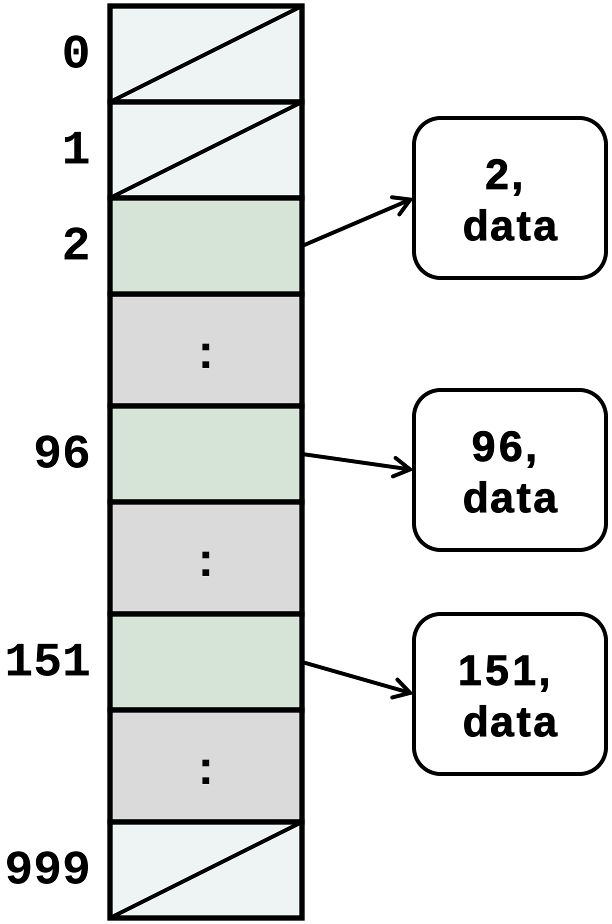
- Additional information can be stored
    - E.g. route, interval, number of buses serving for a particular bus service
- Instead of a single boolean value:
    - We can store a **structure/object** at each location



0	
1	
2	2, data
	:
96	96, data
	:
151	151, data
	:
999	

Direct Addressing Table

- Alternatively, we can store a **reference to object in each location**
  - Invalid bus service stores a **NULL**



Generalized Direct Addressing Table

- The generalized set of operations for direct addressing table are:

Operations	Basic Steps	Big-O
***Find*(N)**	**`return a[N]`**	O(1)
***Insert*(N,data)**	**`a[N] = data`**	O(1)
***Delete*(N)**	**`a[N] = NULL`**	O(1)

Direct Addressing Table: Summary

- Direct addressing table is very **efficient**
- However, there are many **restrictions**
  - Key must be **integer** *(what about bus no 95A or NR30?)*
  - Range of keys must be **small**
    - E.g. what if keys are telephone numbers?
  - Keys must be **dense**
    - Most keys in the range are valid
    - Not many "gaps" in the key values

What is Hashing?

- Hashing (system) uses a hash function, a hash table, and a conflict (collision) resolution scheme to implement a table ADT
- A conflict resolution scheme is the action taken to resolve the conflict between two keys that have been assigned the same address by the hash function

Generalized Idea:

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

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

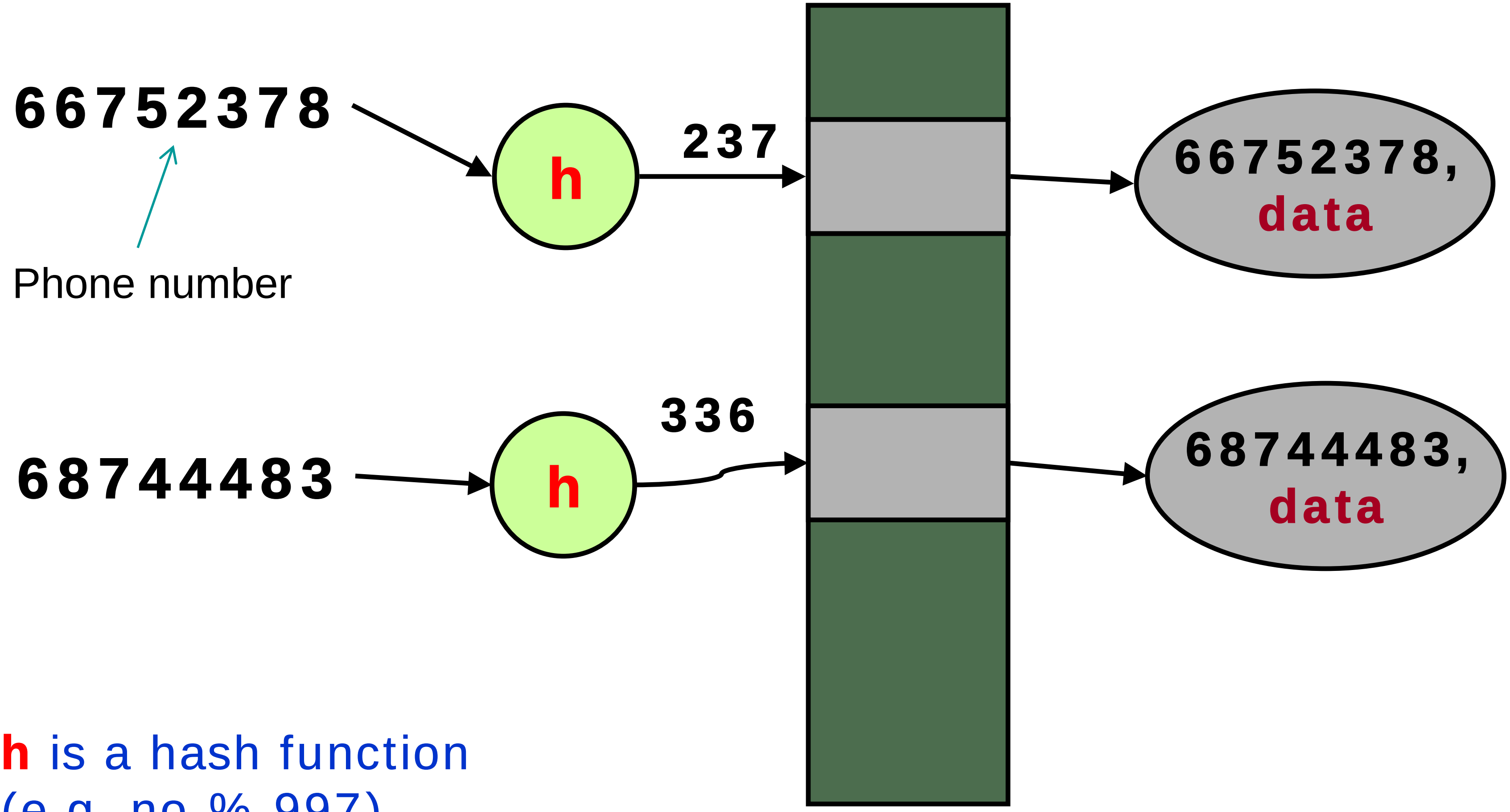
    = **Direct Addressing Table** + **Hash Function**

Hash Table

A generalization of direct addressing table, to remove its restrictions

Hash table

- Map large integers to smaller integers
- Map non-integer keys to integers



h is a hash function
(e.g. no % 997)

Hash Table: Operations

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

Operations	Basic Steps
`Find(N)`	`return a[ h(N) ]`
`Insert(N,data)`	`a[ h(N) ] = data`
`Delete(N)`	`a[ h(N) ] = NULL`

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

Hash Tables: Problems

- If the result of the hash function is **unique,** each key is mapped to a different home address (array index)
  - known as **perfect hash function**
- This is **not always the case**
  - 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**
  - Need to find ways to resolve them

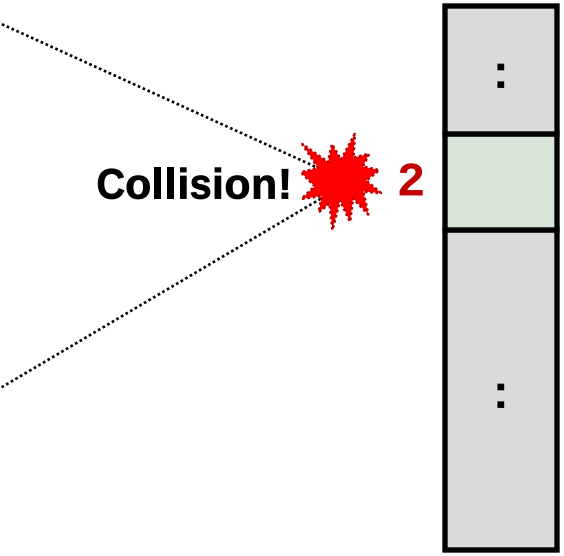
Hashing Collision

- Given the hash function:

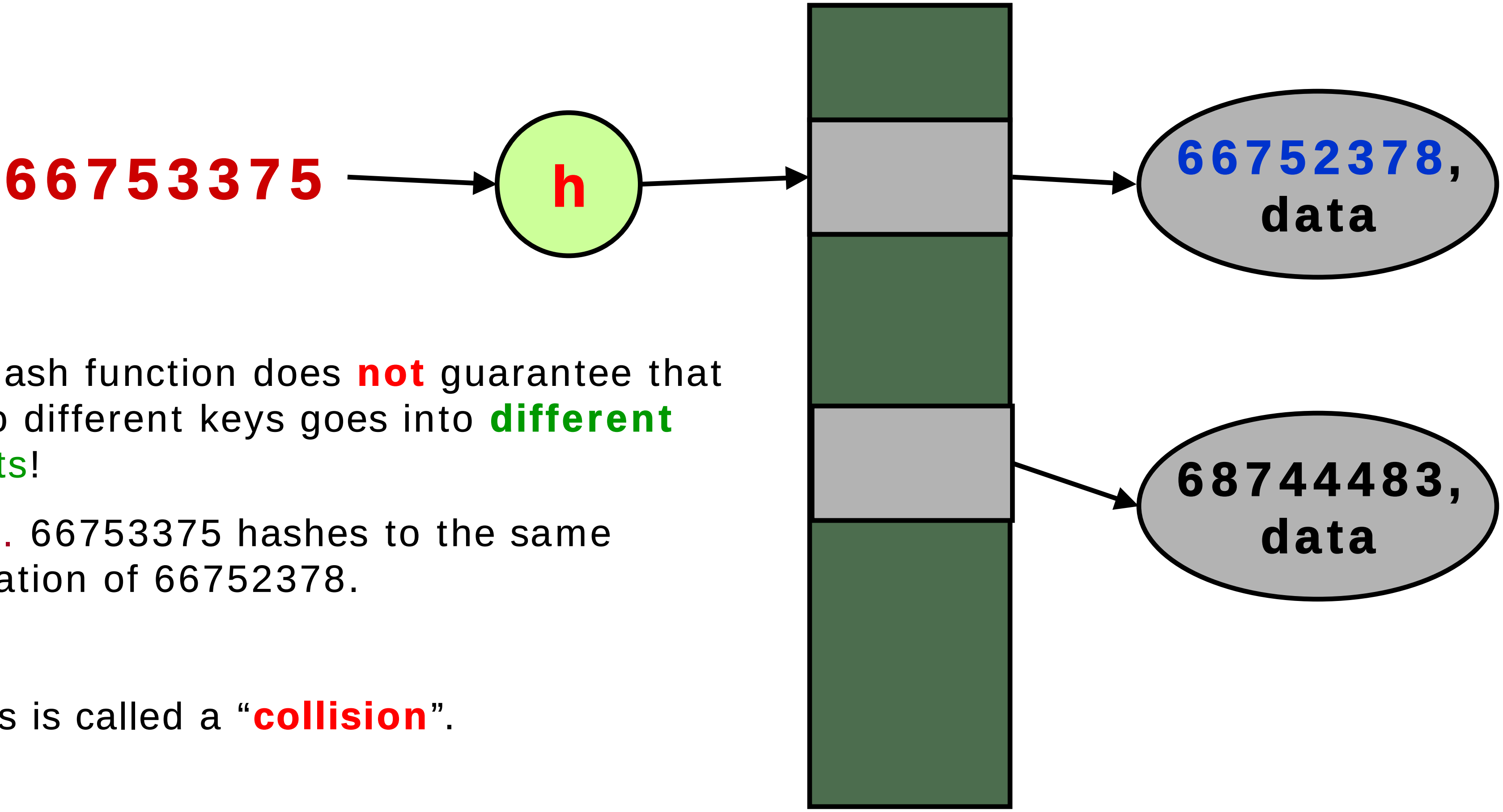
```
h( key ) = key % 17
```

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

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



Hash table



A hash function does **not** guarantee that two different keys goes into **different** slots!

E.g. 66753375 hashes to the same location of 66752378.

This is called a “**collision**”.

Hash Table: Important Issues

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

- How to **resolve collision**?

Hash Functions

Good Hash Function: Properties

- A good hash function should:
  - Be fast to compute (should be O(1))
  - Scatter keys evenly throughout the hash table
  - Result in few or none collisions
  - Allow the hash table to be small
- These properties should be evaluated in the context of the potential key range

Counter Example

- Selecting digits from several positions usually make poor hashing function
  - E.g. Hash($\mathbf{d_0 d_1 d_2 \ldots d_7}$) = $\mathbf{d_3 d_7}$
  - ➔ Hash(667**5**437**8**) = **58**
  - ➔ Hash(634**5**909**8**) = **58**

- What if we select the first three digits from Singapore phone numbers as the hash value?

Perfect Hash Function

- Perfect Hash Function:
  - One-to-one mapping between the keys and array indices
  - ➔ **NO collision**
- It is possible if we know all keys in advance
- **Example:**
  - When a compiler searches for keywords or reserved words

How to Define a Hash Function?

- Uniform hash function
- Division method
- Multiplication method
- 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 a hash table
  - ***H*(K)** is uniform if each location receives no more than $\left\lceil \frac{K}{M} \right\rceil$ keys

Uniform Hash Function

- 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:

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

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

Modulo Method

- Given a hash table of **m slots**
  - We can use the modulo operator "%" to map an integer to a value between **0** and **m-1**:

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

- One of the most popular methods
- Behaviour 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)

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**:

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

Hashing of Strings

- For non-integer keys:
  - We first convert the key into an integer, then apply hash function on the result
- Let us use string as illustration

```
int HashString( string str )
{
    int sum = 0;

    for ( i = 0; i < str.size(); i++ )
        sum += str[i];

    return sum % tableSizeM;
}
```

Convert **str** into an integer **sum**

Perform the actual hashing. Modulo method is used here as example.

Hashing of Strings: Problems

- The method used is not very good:
  - ❑ Many strings converted to the **same sum**
  - ➔ Results in large number of collisions
- Example:
  - ❑ HashString("abc") == HashString("bac") == HashString("cba")
- Problem:
  - ❑ The conversion fails to take the **position of each character into account**

  ➔ Permutation of a string gives the same sum!

Hashing of Strings: Problems

HashString (“Tan Ah Teck”)

= (“T” + “a” + “n” + “ ” +
“A” + “h” + “ ” +
“T” + “e” + “c” + “k”) % 11 // hash table size is 11

= (84 + 97 + 110 + 32 +
65 + 104 + 32 +
84 + 101 + 99 + 107) % 11

= 825 % 11

= 0

Hashing of Strings: Problems

- Lee Chin Tan
- Chen Le Tian
- Chan Tin Lee

All 3 strings above have the same hash value! Why?

- Problem: The hash code produced does not depend on positions of characters! – Bad

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:**
  - Let's take X = 17
  - Hash("abc") = $97*17^2 + 98*17^1 + 99*17^0 = 29798$
  - Check whether "bac", "cba" etc gives different sum?

Hashing Strings: Better Conversion

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

```
int HashString( string str )
{
    int sum = 0;

    for ( i = 0; i < str.size(); i++ )
        sum = (17*sum) + str[i];

    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

Collision Resolution

Probability of Collision (1/2)

- **von Mises Paradox (The Birthday Paradox)**: "How many people must be in a room before the probability that some share a birthday, ignoring the year and leap days, becomes at least 50 percent?"

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

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

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

= 1 – Q(n)

P(**23**) = 0.507

Probability of Collision (2/2)

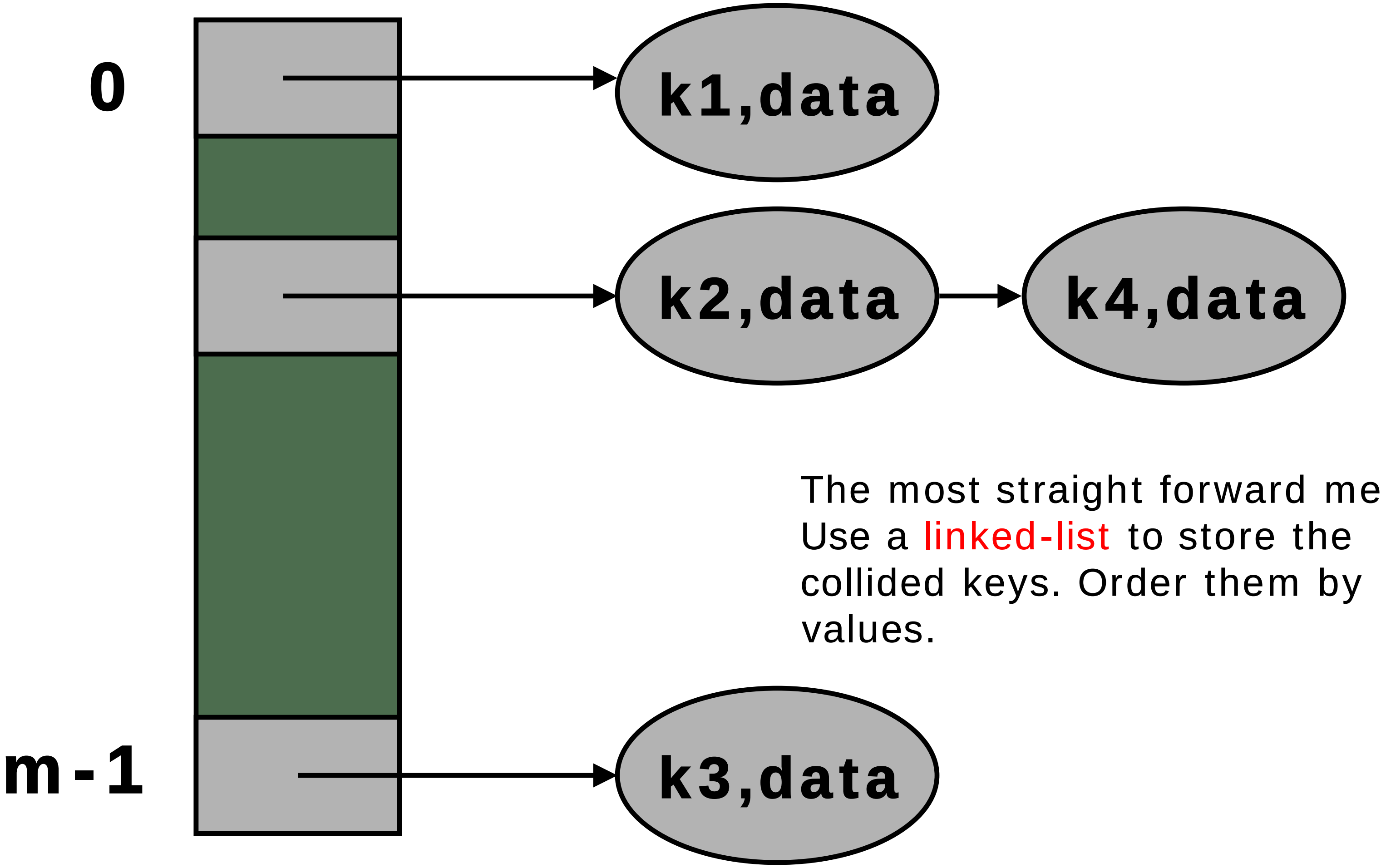
- This means that if there are 23 people in a room, the probability that some people share a birthday is 50.7%.
- So, if we insert 23 keys in to a table with 365 slots, more than half of the time we get collisions. Such a result is counter-intuitive to many.
- So, collision is very likely!

How to resolve collisions?

Conflict resolution schemes commonly used:

- Separate Chaining
- Linear Probing
- Quadratic Probing
- Double Hashing

Separate chaining



The most straight forward method. Use a linked-list to store the collided keys. Order them by key values.

Hash table (separate chaining)

insert (key, data)

insert data into the list a[h(key)]

delete (key)

delete data from the list a[h(key)]

find (key)

find key from the list a[h(key)]

Load Factor

- n: number of keys in the hash table
- m: size of the hash table – number of slots
- Define the load factor α

$$\alpha = n/m$$

a measure of how full the hash table is.

(Note: α can be >= 1)

If table size is the number of linked lists, then α is the average length of the linked lists.

Average Running Time

- Find $O(1 + \alpha)$
- Insert $O(1)$
- Delete $O(1 + \alpha)$

- Note that α affects the performance of find and delete operations.
- If α is bounded by some constant, then all three operations are $O(1)$.

Reconstructing hash table

- To keep α bounded, we may need to reconstruct the whole table when the load factor exceeds the bound.
- Whenever the load factor exceeds the bound, we need to rehash all keys into a bigger table (increase m to reduce α), say double the table size.

Open Addressing

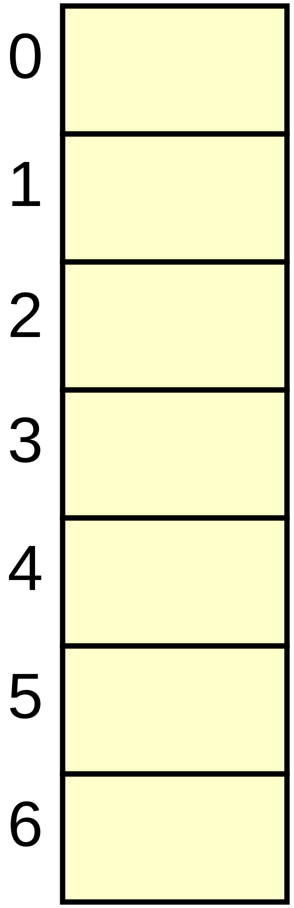
- Separate chaining is a close addressing system as the address given to a key is fixed
- When the hash address given to a key is open (not fixed), the hashing is an open addressing system
- Open Addressing:
  - Hashed items are in a single array
  - Hash code gives the home address
  - Collision is resolved by checking multiple positions
  - Each check is called a probe into the table

Linear Probing

hash(k) = k mod 7

Here the table size m=7

Note: 7 is a prime number



In linear probing, when there is a collision, we scan forward for the the next empty slot (wrapping around when we reach the last slot)

Linear Probing: Insert 18

hash(k) = k mod 7

hash(18)
= 18 mod 7
= 4

0	
1	
2	
3	
4	**18**
5	
6	

Linear Probing: Insert 14

hash(k) = k mod 7

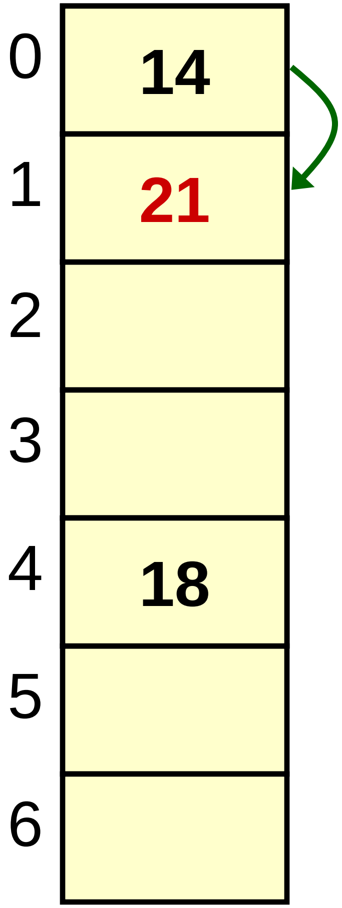
hash(14)
= 14 mod 7
= 0

0	14
1	
2	
3	
4	18
5	
6	

Linear Probing: Insert 21

hash(k) = k mod 7

hash(21)
= 21 mod 7
= 0



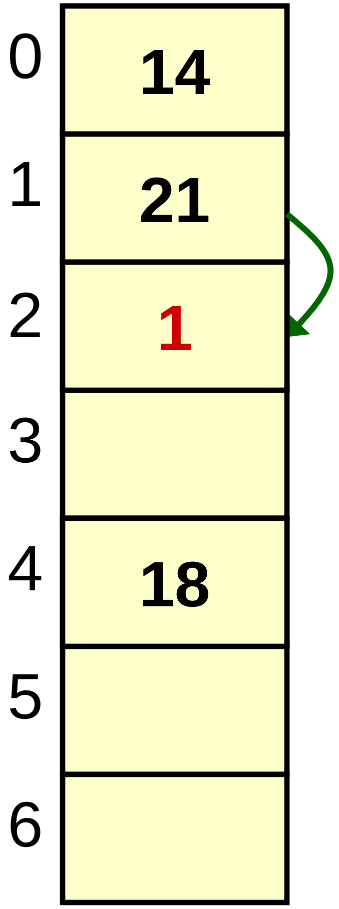
0	**14**
1	**21**
2	
3	
4	**18**
5	
6	

Collision occurs!
Look for next empty slot

Linear Probing: Insert 1

hash(k) = k mod 7

hash(1)
= 1 mod 7
= 1



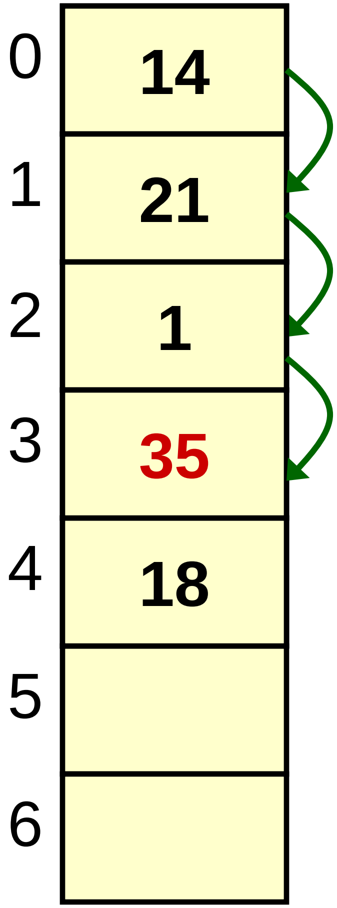
0	14
1	21
2	1
3	
4	18
5	
6	

Collides with 21 (hash value 0). Look for next empty slot

Linear Probing: Insert 35

hash(k) = k mod 7

hash(35)
= 35 mod 7
= 0



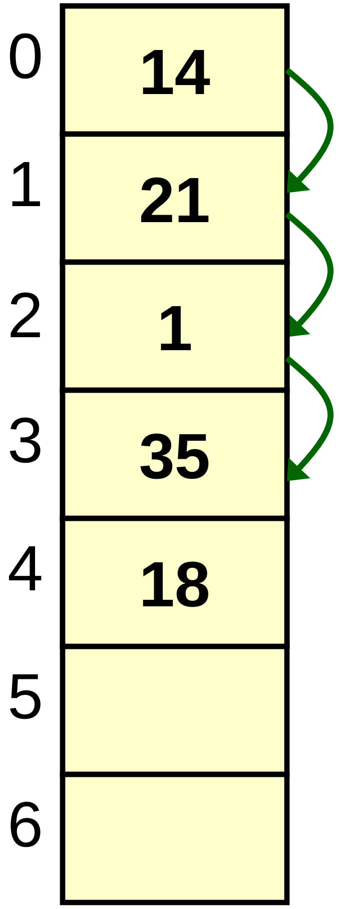
0	**14**
1	**21**
2	**1**
3	**35**
4	**18**
5	
6	

Collision, need to check next 3 slots

Linear Probing: Find 35

hash(k) = k mod 7

hash(35) = 0



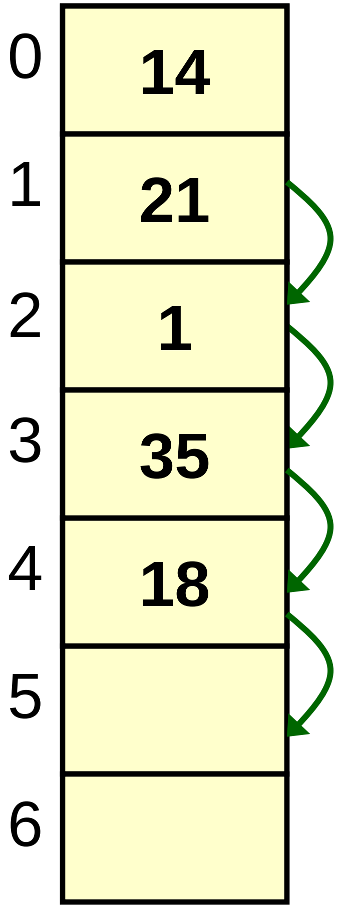
0	14
1	21
2	1
3	35
4	18
5	
6	

Found 35, after 4 probes

Linear Probing: Find 8

hash(k) = k mod 7

hash(8) = 1



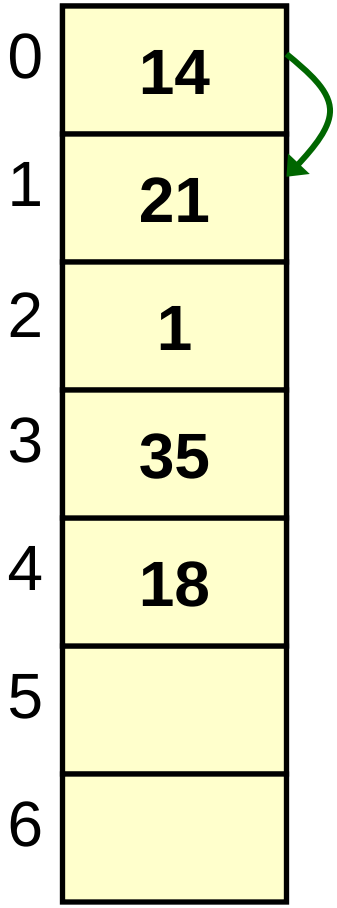
0	14
1	21
2	1
3	35
4	18
5	
6	

8 NOT found.
Need 5 probes!

Linear Probing: Delete 21

hash(k) = k mod 7

hash(21) = 0

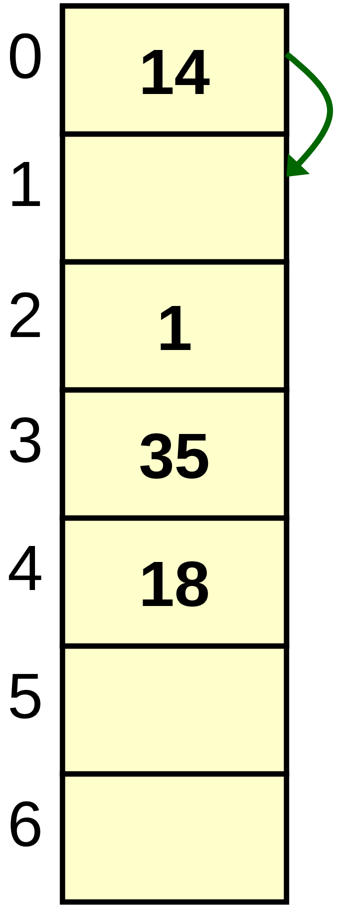


0	14
1	21
2	1
3	35
4	18
5	
6	

Linear Probing: Find 35

hash(k) = k mod 7

hash(35) = 0



0	**14**
1	
2	**1**
3	**35**
4	**18**
5	
6	

35 NOT found!

Incorrect!

We cannot simply remove a value, because it can affect find()!

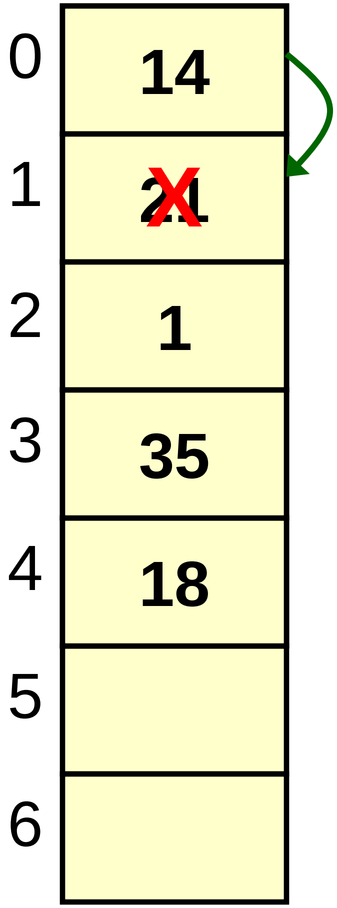
How to delete?

- **Lazy** Deletion
- Use three different states of a slot
  - Occupied
  - Deleted
  - Empty
- When a value is removed from linear probed hash table, we just mark the status of the slot as “deleted”, Instead of emptying the slot
- Need to use a state array the same size as the hash table

Linear Probing: Delete 21

hash(k) = k mod 7

hash(21) = 0



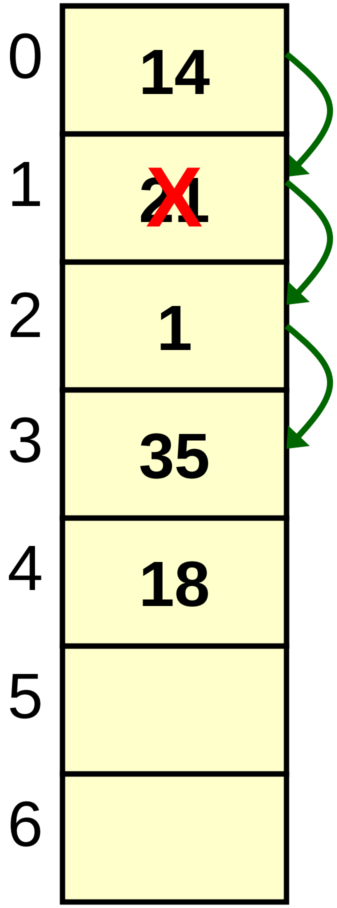
0	**14**
1	~~**21**~~ X
2	**1**
3	**35**
4	**18**
5	
6	

Slot 1 is occupied but now marked as deleted.

Linear Probing: Find 35

hash(k) = k mod 7

hash(35) = 0



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

Found 35

Now we can find 35

Linear Probing: Insert 15

hash(k) = k mod 7

hash(15) = 1

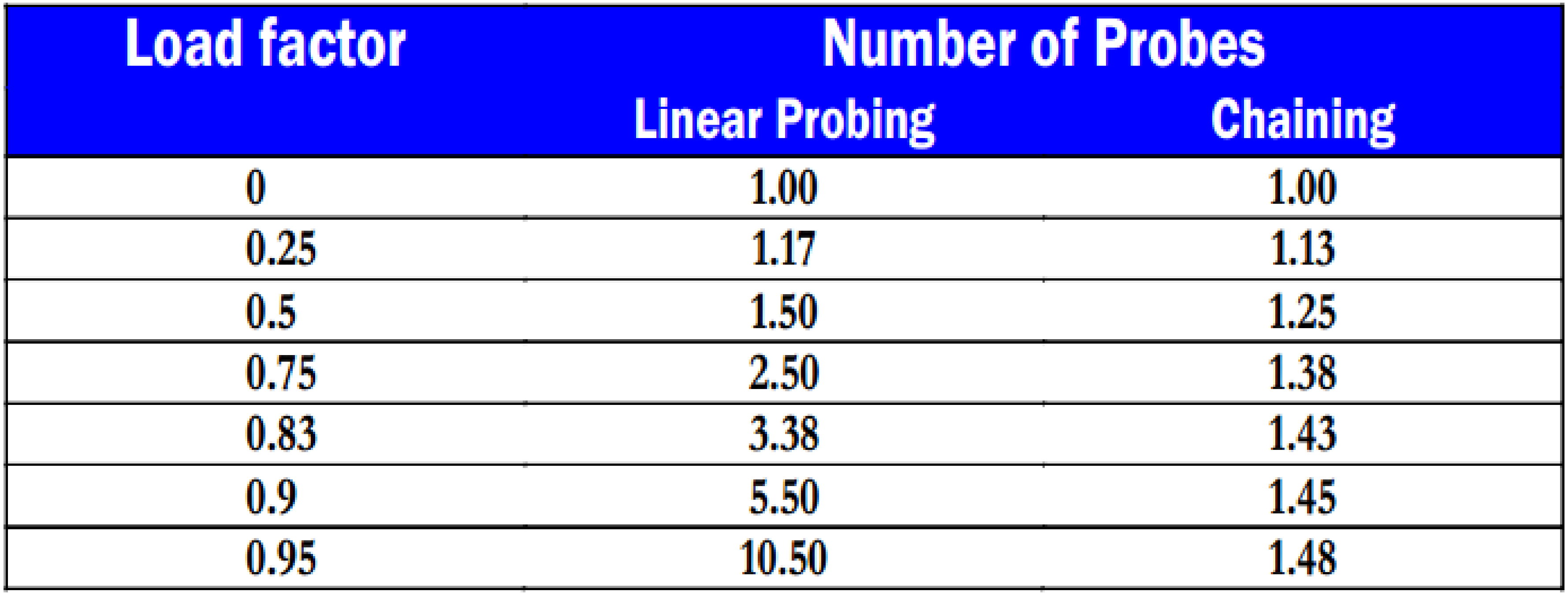
Note: We continue to search for 15, and found that 15 is not in the hash table (total 5 probes).

0	**14**
1	**15**
2	**1**
3	**35**
4	**18**
5	
6	

15 is inserted into the slot 1 which was marked as deleted

We can insert a new value into a slot that has been marked as deleted.

Performance of Hash Table



Load factor	Number of Probes	
	Linear Probing	Chaining
0	1.00	1.00
0.25	1.17	1.13
0.5	1.50	1.25
0.75	2.50	1.38
0.83	3.38	1.43
0.9	5.50	1.45
0.95	10.50	1.48

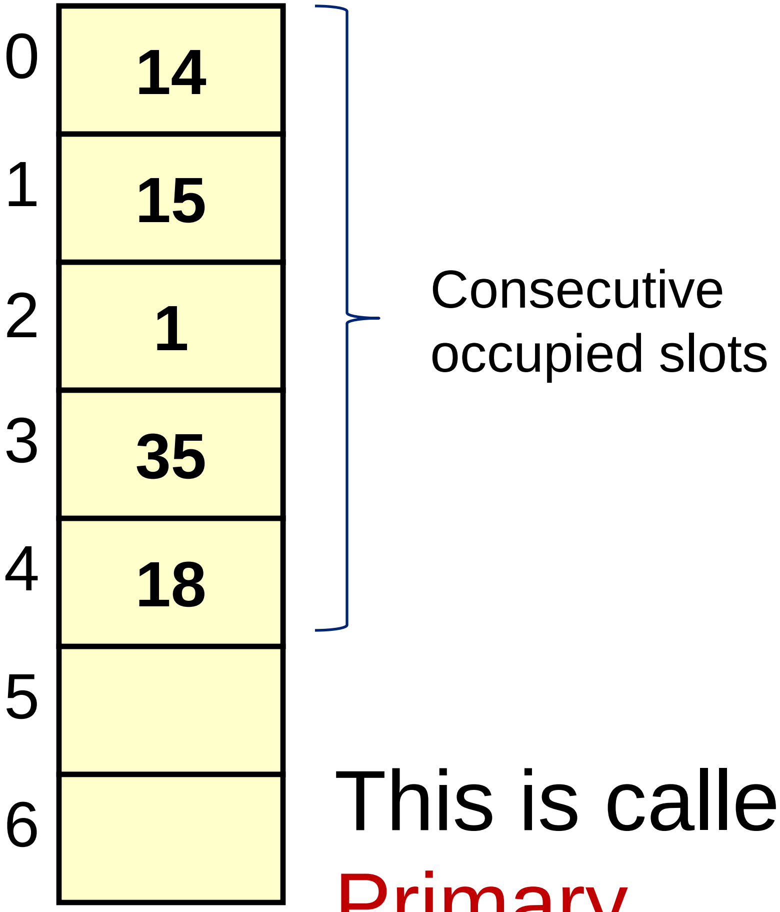
For successful search, the number of probes is

Linear probing: $\frac{1}{2}(1 + 1/(1- \alpha))$

Separate chaining: $1 + \alpha/2$

Problem of Linear Probing

- A cluster is a collection of consecutive occupied slots
- A cluster that covers the home address of a key is called a primary cluster of the key
- Linear probing can create large primary clusters that will increase the running time of find/insert/delete operations



0	14
1	15
2	1
3	35
4	18
5	
6	

Consecutive occupied slots

This is called Primary Clustering

Problem of Linear Probing

The probe sequence of linear probing is:

```
hash (key)                   // first probe, the home
(hash(key) + 1) % m          // second probe
(hash(key) + 2) % m          // third probe
(hash(key) + 3) % m          // fourth probe
```

- If there is an empty slot, we are sure to find it.
- When an empty slot is found, conflict is resolved, but the primary cluster of the key is expanded as a result
- The size of the resulting primary cluster may be very big due to the annexation of the neighbouring cluster

Modified Linear Probing

Q: How to modify linear probing to avoid primary clustering?

We can modify the probe sequence as follows:

$$
\begin{gathered}
\text{hash(key)} \\
(\ \text{hash(key)} + \mathbf{1} * \mathbf{d}\ )\ \%\ m \\
(\ \text{hash(key)} + \mathbf{2} * \mathbf{d}\ )\ \%\ m \\
(\ \text{hash(key)} + \mathbf{3} * \mathbf{d}\ )\ \%\ m \\
\vdots
\end{gathered}
$$

where d is some constant integer >1 and is co-prime to m.
Note: Since d and m are co-primes, the probe sequence covers all the slots in the hash table.

Quadratic Probing

To escape from the primary cluster quickly, use quadratic probing to look for an empty slot.

The probe sequence is

hash(key)

(hash(key) + **1**) % m   jump 1^2

(hash(key) + **4**) % m   jump 2^2

(hash(key) + **9**) % m   jump 3^2

:

Distance from previous probe

+1

+3

+5

+7

…

+2j-1 for j^{th} probe from home

Quadratic Probing: Insert 3, 18

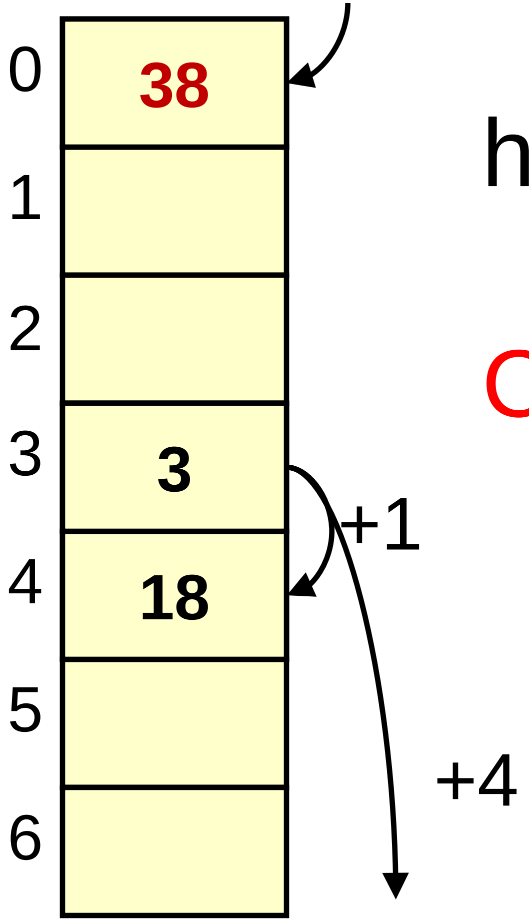
hash(k) = k mod 7

hash(3) = 3
hash(18)= 4

0	
1	
2	
3	**3**
4	**18**
5	
6	

Quadratic Probing: Insert 38

hash(k) = k mod 7



0 **38**
1
2
3 **3**
4 **18**
5
6

+1

+4

hash(38) = 3

Collision

Can quadratic probing always find a free slot?

Insert 12 into the previous example, followed by 10.

what happens?

Theorem

If $\alpha < 0.5$, and m is prime, then we can always find an empty slot.

(m is the table size and α is the load factor)

When quadratic probing is used, in the worst case, 50% of the hash table is wasted.

Secondary Clustering

- In quadratic probing, clusters are formed along the path of probing, instead of around the home location
- These clusters are called secondary clusters
- Secondary clusters are formed as a result of using the same pattern in probing by all keys

Double Hashing

- To resolve the secondary clustering problem, we have to break the probing pattern of quadratic hashing
- We may use another hash function $hash_2$ to generate different probe sequences for different keys

$$hash(key)$$

$$(hash(key) + 1*hash_2(key)) \% m$$

$$(hash(key) + 2*hash_2(key)) \% m$$

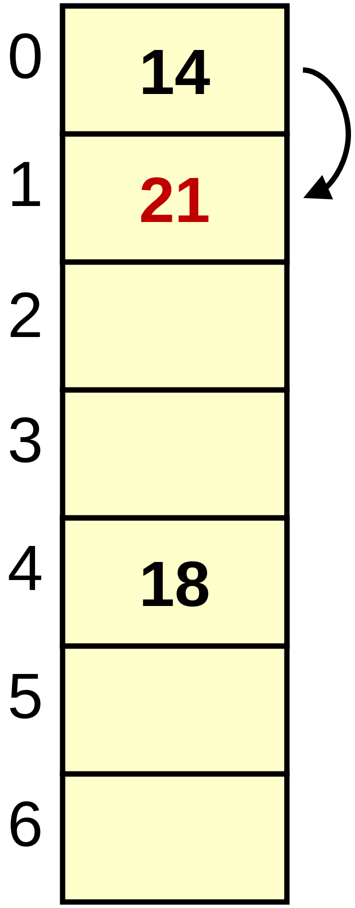
$$(hash(key) + 3*hash_2(key)) \% m$$

:

$hash_2$ is called the secondary hash function, the no of slots to jump each time a collision occurs

After Inserting 14 and 18, Insert 21

hash(k) = k mod 7
hash_2(k) = k mod 5



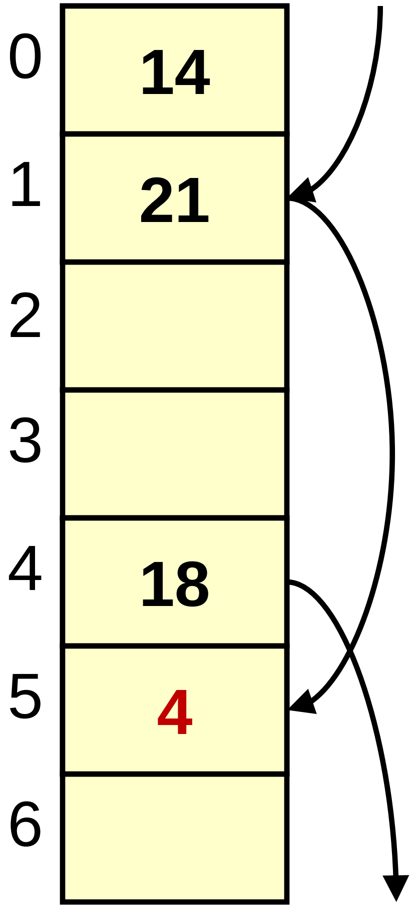
hash(21)
=21 mod 7
= 0

hash_2(21)
= 21 mod 5
= 1

Double Hashing: Insert 4

hash(k) = k mod 7

$hash_2$(k) = k mod 5



If we insert 4, the probe sequence is 4 (home), 8, 12, …

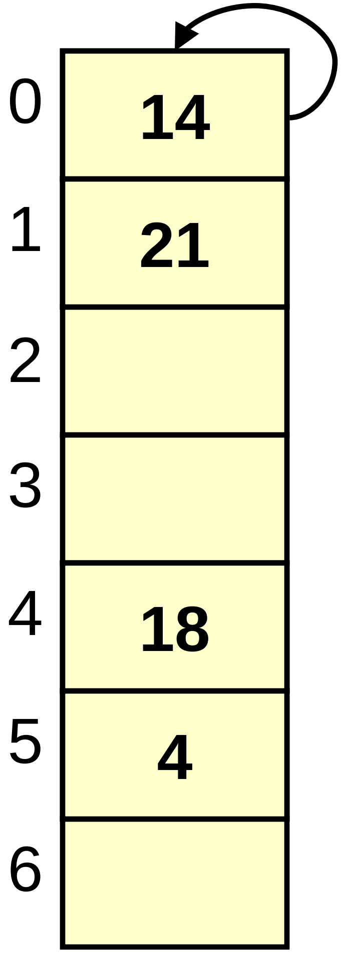
hash(4) = 4

$hash_2$(4) = 4

Double Hashing: Insert 35

hash(k) = k mod 7
hash$_2$(k) = k mod 5

hash(35) = 0
hash$_2$(35) = 0



0	**14**
1	**21**
2	
3	
4	**18**
5	**4**
6	

But if we insert 35, the probe sequence is **0, 0, 0,** …

What is wrong?
Since hash$_2$(35)=**0**.
Not acceptable!

Warning

- **Secondary hash function must not evaluate to 0 !**
- To solve this problem, simply change $\text{hash}_2(\text{key})$ in the above example to:

$$\text{hash}_2(\text{key}) = 5 – (\text{key} \% 5)$$

Note: If $\text{hash}_2(k) = 1$, then it is the same as linear probing.

If $\text{hash}_2(k) = d$, where d is a constant and $d > 1$, then it is the same as modified linear probing.

Good Collision Resolution Method

- Small cluster size
- Always find an empty slot if it exists
- Give different probe sequences when 2 keys collide (i.e. no secondary clustering)
- Fast

Rehash

- Time to rehash:
  - When table is getting full, the operations are getting slow
  - For quadratic probing, inserts might fail when the table is more than half full
- Rehash operation:
  - Build another table about twice as big with a new hash function
  - Scan the original table, for each key, compute the new hash value and insert the data into the new table
  - Delete the original table
- The load factor used to decide the time to rehash:
  - For open addressing: 0.5
  - For closed addressing: 1

STL unordered_map

- STL unordered_map implements a Hash Table with separate chaining:
- Associate containers that store elements formed by the combination of a key value and a mapped value, and which allows for fast retrieval of individual elements based on their keys
- https://en.cppreference.com/w/cpp/container/unordered_map

unordered_map example

```
// std::unordered_map
#include <bits/stdc++.h>

int main()
{
    // Unordered map
    std::unordered_map<int, int> order;

    // Mapping values to keys
    order[5] = 10;
    order[3] = 5;
    order[20] = 100;
    order[1] = 1;

    // Iterating the map and printing unordered values
    for (auto i = order.begin(); i != order.end(); i++) {
        std::cout << i->first << " : " << i->second << '\n';
    }
}
```

```
Output:
1 : 1
3 : 5
20 : 100
5 : 10
```

STL unordered_set

- There is also an unordered_set if key->value pairs are not required
- Implemented using hash table where keys are stored in any order
- https://en.cppreference.com/w/cpp/container/unordered_set

unordered_set example

```
// C++ program to demonstrate various function of unordered_set
#include <bits/stdc++.h>
using namespace std;

int main()
{
    // declaring set for storing string data-type
    unordered_set<string> stringSet;

    // inserting various string, same string will be stored
    // once in set
    stringSet.insert("code");
    stringSet.insert("in");
    stringSet.insert("c++");
    stringSet.insert("is");
    stringSet.insert("fast");

    string key = "slow";
```

Example (cont’d)

```
//     find returns end iterator if key is not found,
//  else it returns iterator to that key
if (stringSet.find(key) == stringSet.end())
    cout << key << " not found\n\n";
else
    cout << "Found " << key << endl << endl;

key = "c++";
if (stringSet.find(key) == stringSet.end())
    cout << key << " not found\n";
else
    cout << "Found " << key << endl;

// now iterating over whole set and printing its
// content
cout << "\nAll elements : ";
unordered_set<string> :: iterator itr;
for (itr = stringSet.begin(); itr != stringSet.end(); itr++)
    cout << (*itr) << endl;
}
```

```
Output:
slow not found

Found c++

All elements :
is
fast
c++
in
code
```

Summary

- How to hash? Criteria for good hash functions?
- How to resolve collision?

  Collision resolution techniques:
  - separate chaining
  - linear probing
  - quadratic probing
  - double hashing
- Problem on deletions
- Primary clustering and secondary clustering
- STL unordered_map, unordered_set