Hash Tables

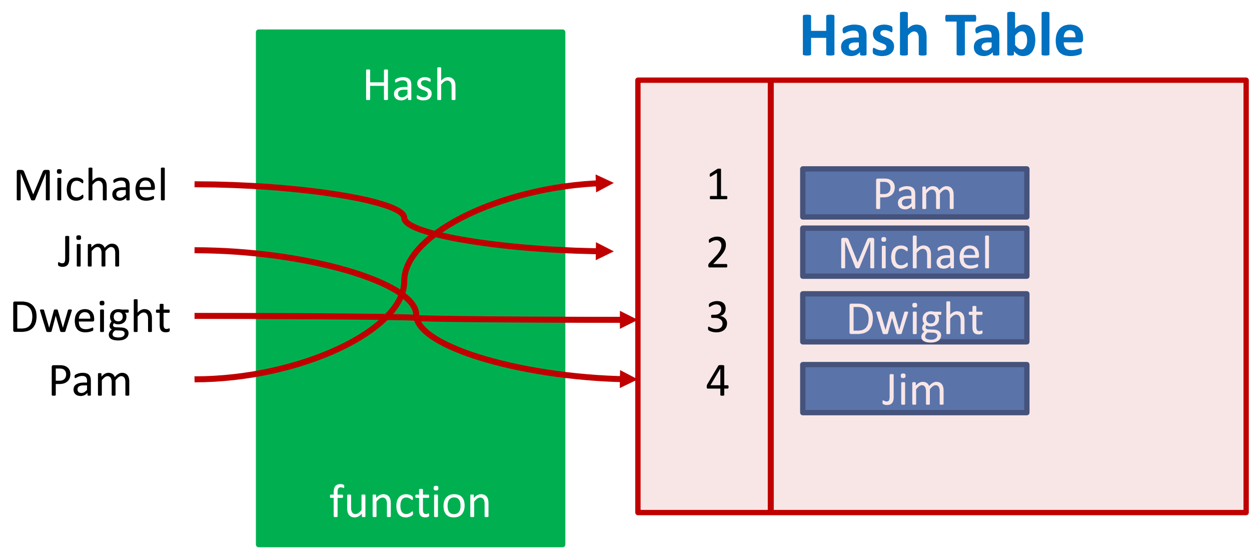
CS223: Data Structures

Why do we need hashing?

- Many applications deal with lots of data, e.g., Search engines and web pages
- There are countless look ups.
- The look ups are time critical.
- Typical data structures like arrays and lists, may not be sufficient to handle efficient lookups
- In general: When look-ups need to occur in near constant time. O(1)
- Used in: Web searches, Spell checkers, Databases, Compilers, passwords, and Many others

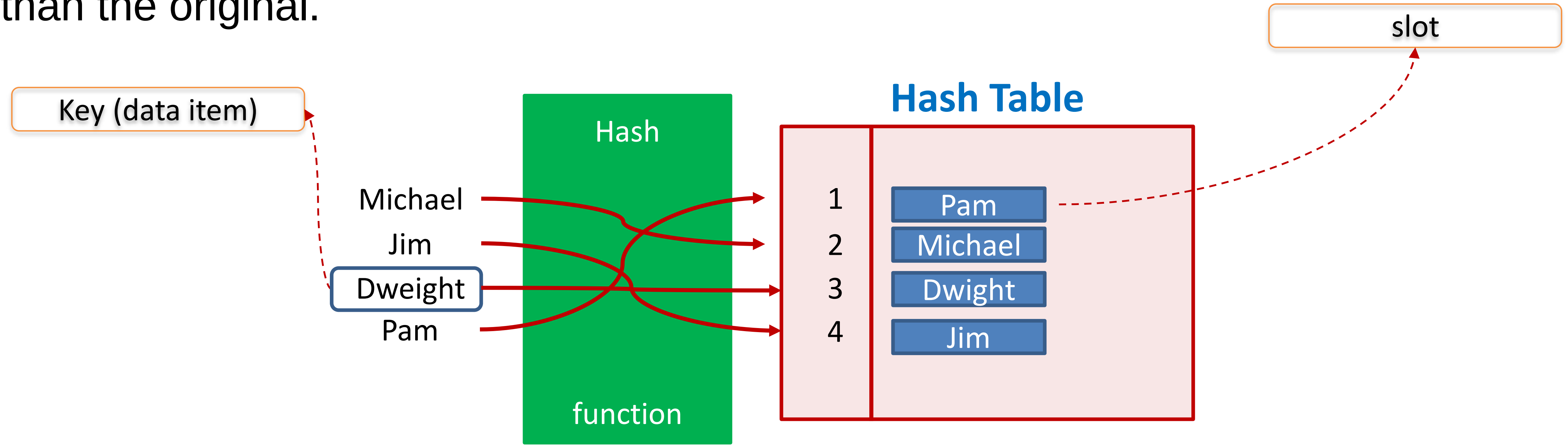
Hash Table

- **Hash table** (or hash map) is a data structure used to store **key-value pairs**.
- It is a collection of items stored to make it easy to find them later.
- It uses a **hash function** to compute an index into an array of slots from which the desired value can be found.
- The **load factor** is the number of keys stored in the hash table divided by the capacity



Hash Functions

- Mapping between an item and the slot where item belongs in the hash table is called the **hash function**.
- This function takes a key and maps it to a value of a certain length which is called a **Hash value** or **Hash.**
- Hash value represents the original string of characters, but it is normally smaller than the original.



Hash Functions

A good hash function

1. Avoid collisions
2. Have keys distributed evenly among cells
3. Simple/fast to compute

Hash Functions

Different methods to choose a good hashing function

➢ Division Method (Mod Operator)

- Map into a hash table of m slots
- Use the modulo operator (%) to map an integer to a value between 0 and m − 1
- n mod m = remainder of n divided by m, where n and m are positive integers (n is the key and m is the table size)
- The most popular method
- Chose the table size as prime number

Hash Functions

- Folding Methods

Fold-shifting: Here actual values of each parts of key are added.

- Suppose, the key is : 12345678, and the required address is of two digits,
- Then break the key into: 12, 34, 56, 78.
- Add these, we get 12 + 34 + 56 + 78 : 180, ignore first 1 we get 80 as location

Fold-boundary: Here the reversed values of outer parts of key are added.

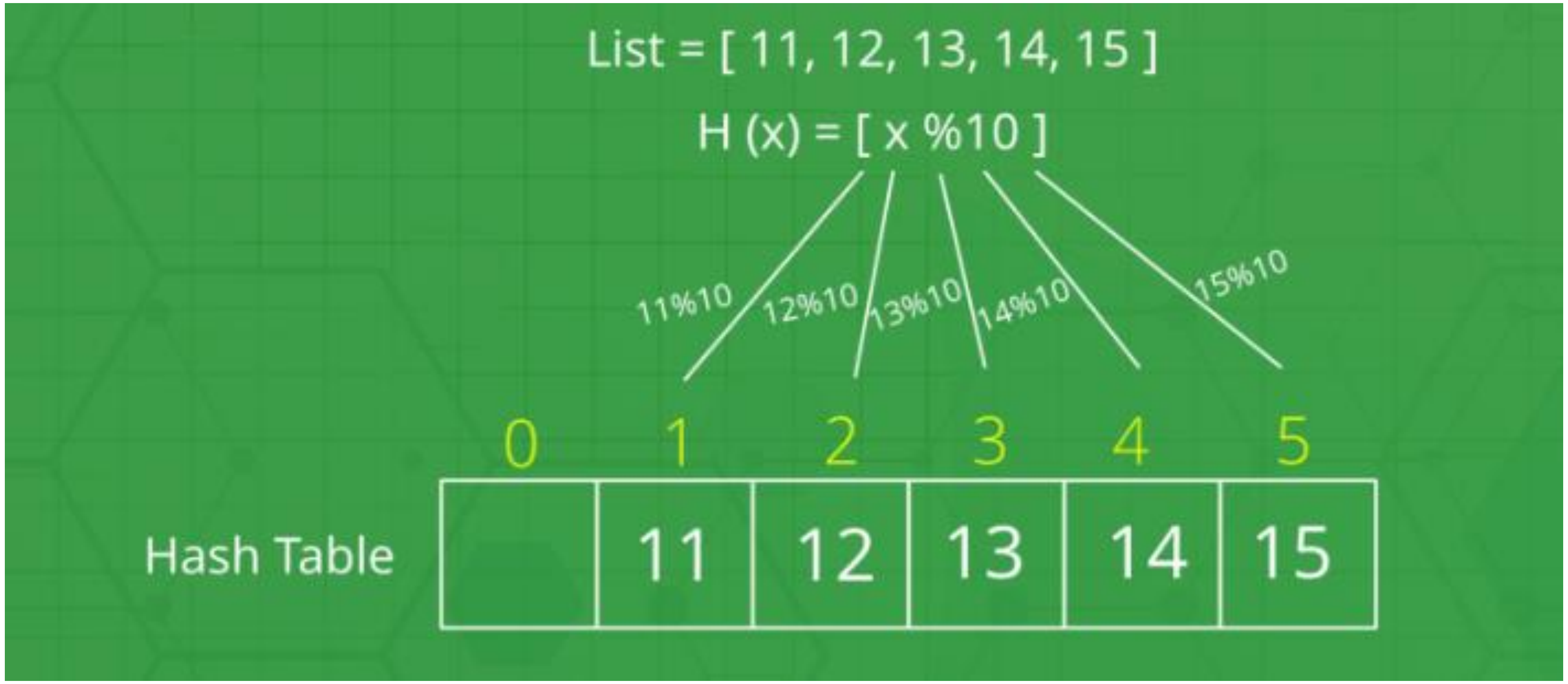
- Suppose, the key is : 12345678, and the required address is of two digits,
- Then break the key into: 21, 34, 56, 87.
- Add these, we get 21 + 34 + 56 + 87 : 198, ignore first 1 we get 98 as location

Hash Functions

➢ Multiplication Method

- A hash function that uses the first p bits of the key times an irrational number.
  Definition: $h(k) = \lfloor m(k A \pmod 1)\rfloor$, where m is usually an integer 2^p
- kA mod 1 gives the fractional part kA,
- $\lfloor\ \rfloor$ gives the floor value
- A is any constant. The value of A lies between 0 and 1. But, an optimal choice will be $\approx (\sqrt{5}-1)/2$ suggested by Knuth.

Hash Functions - Example

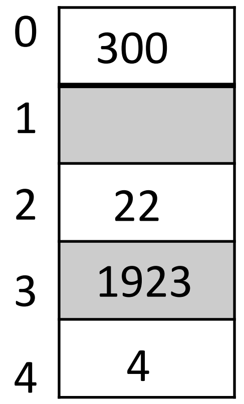


Example of hash table

Num%10

1923
4
300
22

What happens if we try to add 12 now??



Collision

Hash function example

Store data based on first character

A→0, B→1, ….

Apple
Car
Boy
Dog

0	Apple
1	Boy
2	Car
3	
4	

Is there a “dog” (or any word that starts with D) in the list??

You only need to check only position 4 and not the whole list

What happens if you try to add Amman??

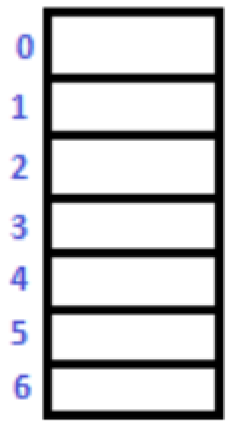
Collision

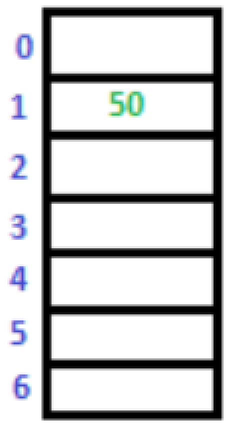
Handle Collisions (1) – Separate Chaining

Separate Chaining:

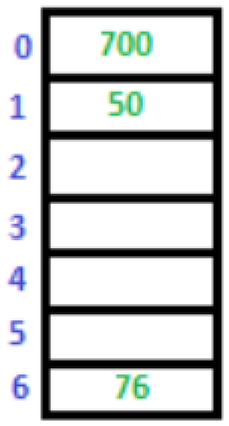
The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

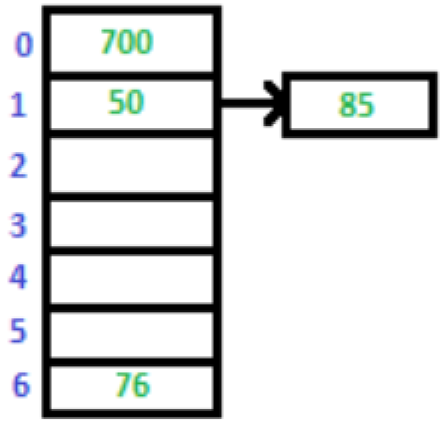




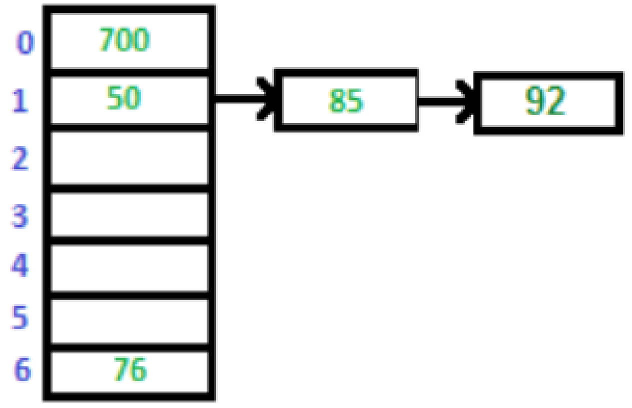
Insert 50



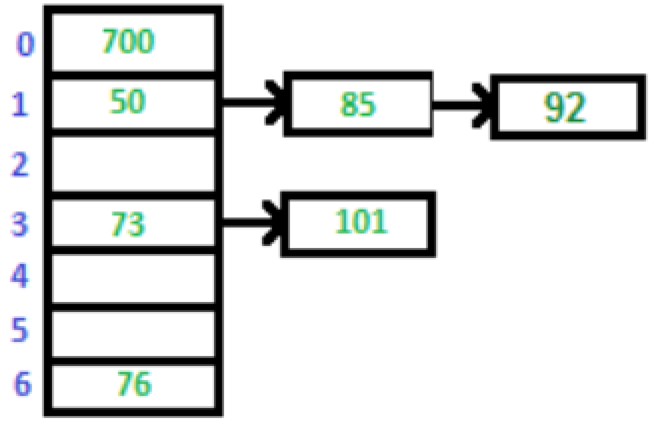
Insert 700 and 76



Insert 85: Collision Occurs, add to chain



Inser 92 Collision Occurs, add to chain



Insert 73 and 101

Handle Collisions (1) – Separate Chaining

Separate Chaining:

The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Advantages

- ✓ Simple to implement.
- ✓ Hash table never fills up, we can always add more elements to the chain.
- ✓ Less sensitive to the hash function or load factors.
- ✓ It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages

- ✗ Performance of chaining is not good as keys are stored using a linked list.
- ✗ Wastage of Space (Some Parts of hash table are never used)
- ✗ If the chain becomes long, then search time can become O(n) in the worst case.
- ✗ Uses extra space for links.

Handle Collisions (2) – Open Addressing

- Open Addressing: Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, size of the table must be greater than or equal to the total number of keys.

- Open Addressing is done following ways:
  - Linear Probing
  - Quadratic Probing
  - Double Probing

Handle Collisions (2) – Open Addressing

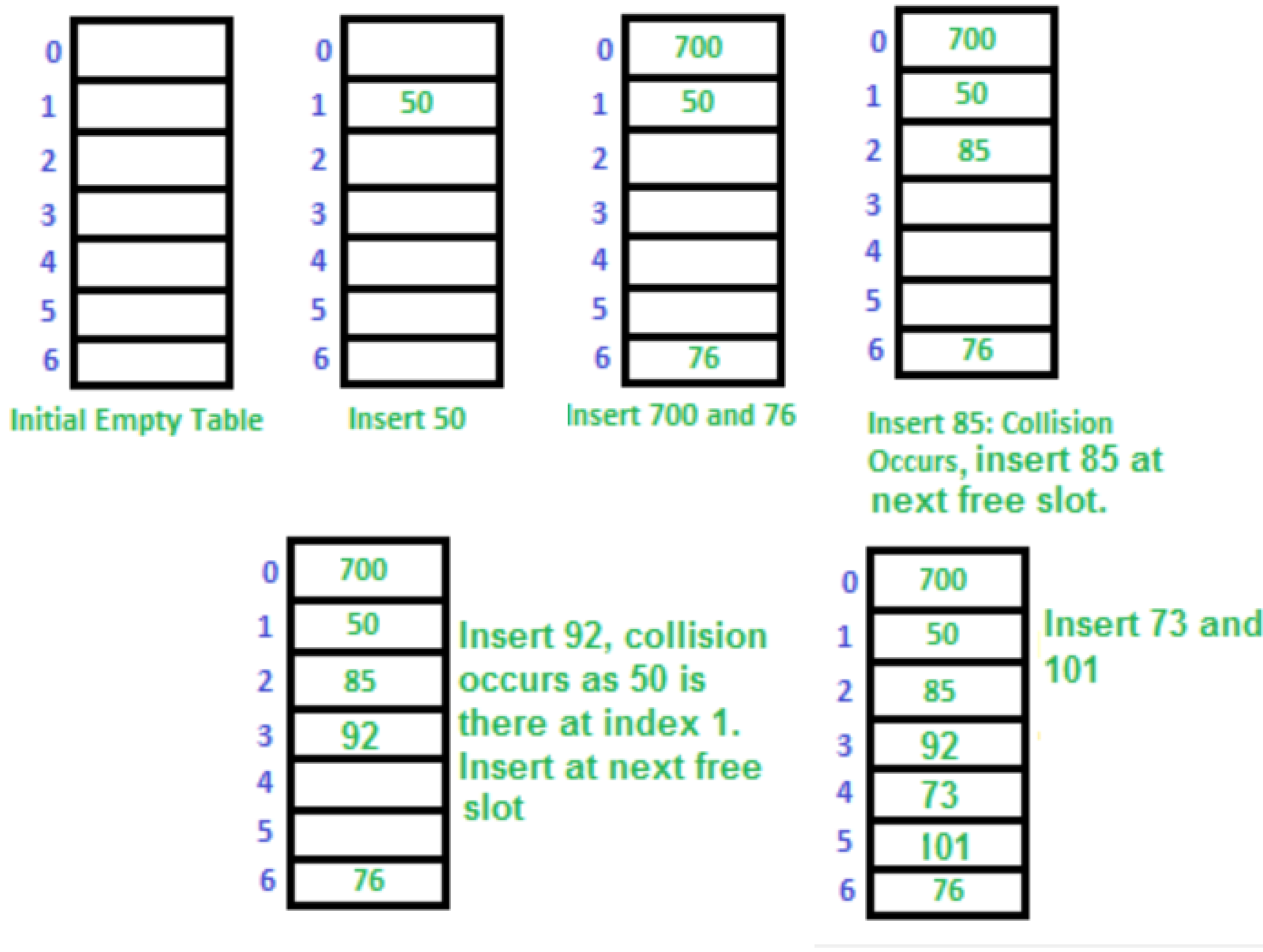
a) Linear Probing: In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also.

let hash(x) be the slot index computed using hash function and S be the table size.

1. If slot hash(x) % S is full, then we try (hash(x) + 1) % S
2. If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S
3. If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S
4. Etc.

Handle Collisions (2) – Open Addressing

Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

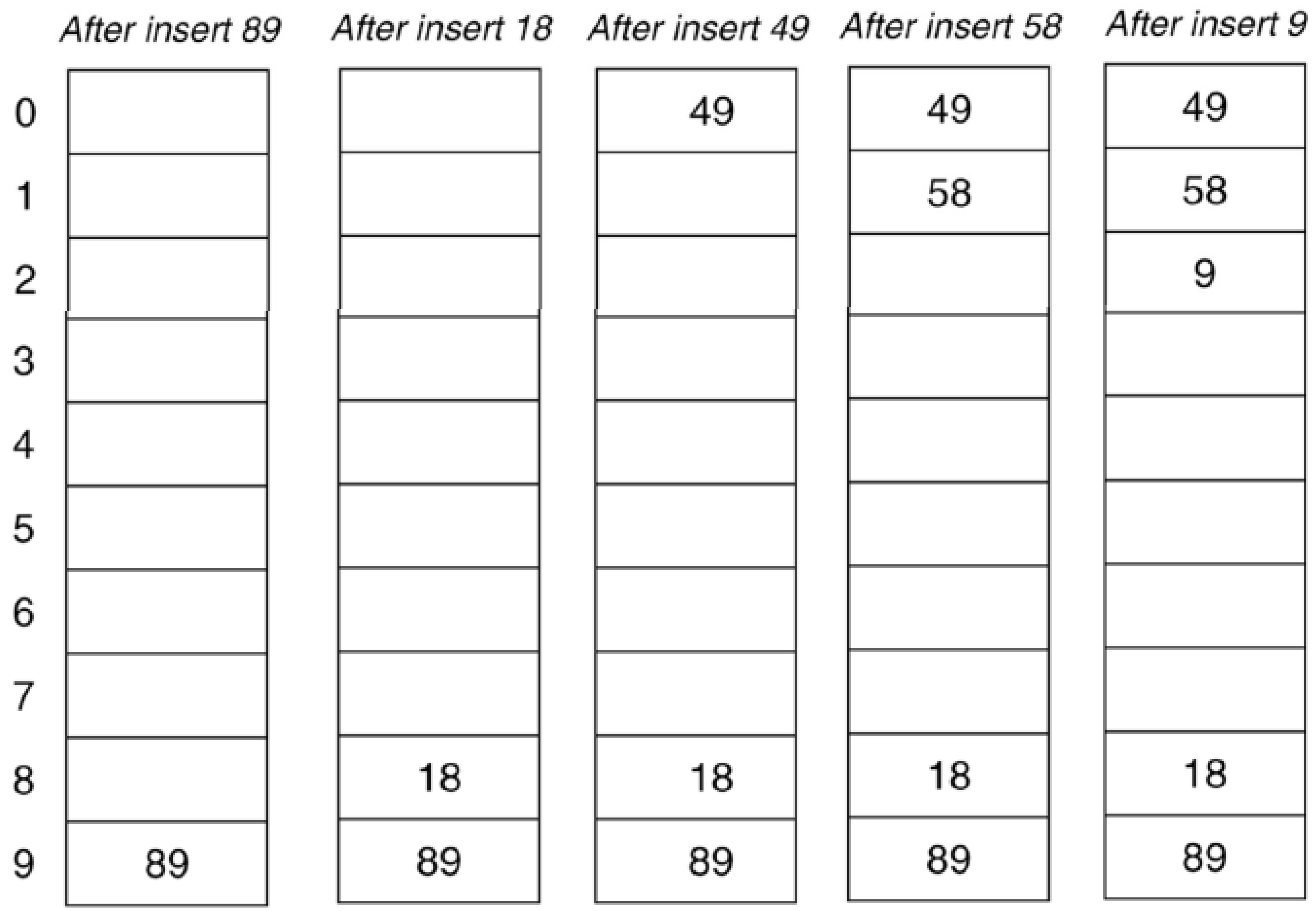


0 1 2 3 4 5 6
Initial Empty Table
50
Insert 50
700
50
76
Insert 700 and 76
700
50
85
76
Insert 85: Collision Occurs, insert 85 at next free slot.
700
50
85
92
76
Insert 92, collision occurs as 50 is there at index 1. Insert at next free slot
700
50
85
92
73
101
76
Insert 73 and 101

Handle Collisions (2) – Open Addressing

Let us consider a simple hash function as "key mod 10" and sequence of keys as 89, 18, 49, 58, 9.

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```



	After insert 89	After insert 18	After insert 49	After insert 58	After insert 9
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Clustering: The main problem with linear probing is primary clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element (i.e., creates a long sequence of filled slots).

Primary Clustering

- A cluster is a collection of consecutive occupied slots
- A cluster that covers the home location of a key is called the primary cluster of the key

The main problem with linear probing is primary clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element (i.e., creates a long sequence of filled slots).

Handle Collisions (2) – Open Addressing

b) Quadratic Probing: We look for i^2'th slot in i'th iteration..

let hash(x) be the slot index computed using hash function.

1. If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S
2. If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S
3. If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S
4. etc

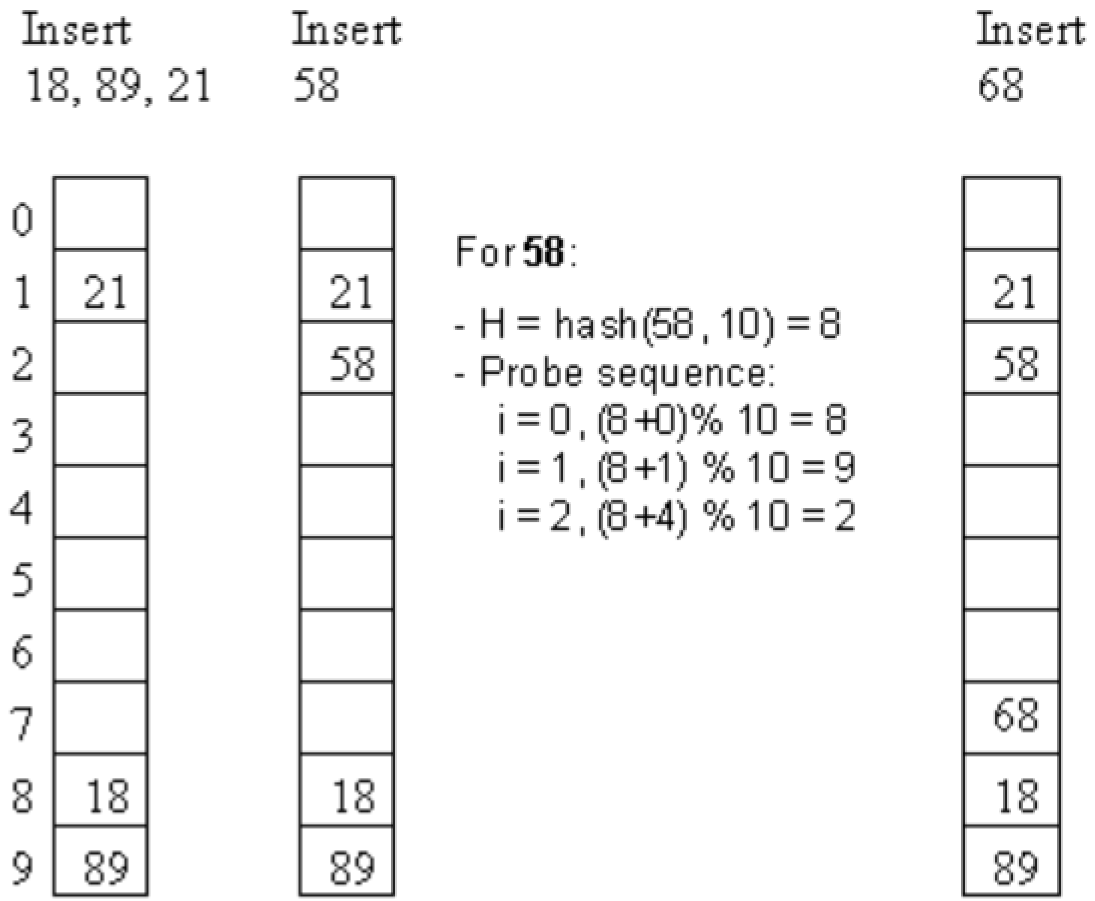
- It attempts to keep clusters from forming.
- The idea is to probe more widely separated cells, instead of those adjacent to the primary hash site

It might still encounter Clustering problem (Secondary clustering)

Secondary Clustering

- In quadratic probing, clusters are formed along the path of probing, instead of around the home location of a key
- These clusters are called secondary clusters
- Secondary clusters are formed as a result of using the same pattern in probing by all keys
  - If two keys have the same home location, their probe sequences are going to be the same
- But it is not as bad as primary clustering in linear probing

Example: Quadratic Probing



For **58**:

- H = hash(58, 10) = 8
- Probe sequence:
  i = 0, (8+0)% 10 = 8
  i = 1, (8+1) % 10 = 9
  i = 2, (8+4) % 10 = 2

For **68**:

- H = hash(68, 10) = 8
- Probe sequence:
  i = 0, (8+0)% 10 = 8
  i = 1, (8+1) % 10 = 9
  i = 2, (8+4) % 10 = 2
  i = 3, (8+9) % 10 = 7

Handle Collisions (2) – Open Addressing

b) Double Probing: We use another hash function hash2(x) and look for i*hash2(x) slot in i'th rotation.

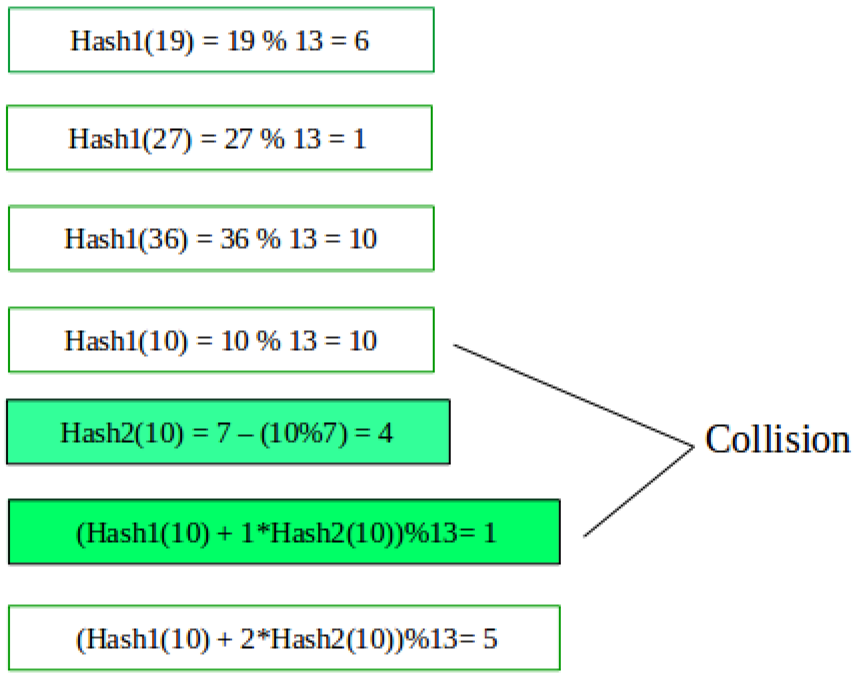
- It adds the hash codes of two hash functions

let hash(x) be the slot index computed using hash function.

1. If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S
2. If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S
3. If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S

Lets say, Hash1 (key) = key % 13

Hash2 (key) = 7 – (key % 7)



Separate Chaining vs. Open Addressing

Separate Chaining	Open Addressing
Chaining is Simpler to implement.	Open Addressing requires more computation.
In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care to avoid clustering and load factor.
Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.
Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn't map to it.
Chaining uses extra space for links.	No links in Open addressing

Hashing Method

Use the % in this example

```
int hashCode(int key){
   return key % SIZE;
}
```

Collision

Sr. No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Linear probing to handle collisions

In case of collision, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

Str. No.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

Search Operation

- Compute the hash code of the element
- Locate the element using that hash code as index in the array
- Use linear probing to get the element ahead if the element is not found at the computed hash code.

Delete Operation

- Compute the hash code of the key passed
- Locate the index using that hash code as an index in the array
- Use linear probing to get the element ahead if an element is not found at the computed hash code
- When found, set it back to dummy item (which has the key -1) which means empty in this question (**Lazy delete, why?**)

Insert Operation

- Compute the hash code of the key passed
- Locate the index using that hash code as an index in the array
- Use linear probing for next empty location, if an element is found at the computed hash code.

Rehashing

As with array-based stacks/queues/lists, if table gets too full, create a bigger table and copy everything

- With chaining, we should decide what “too full” means
  - Keep load factor (λ) reasonable (e.g., < 1)?
  - Consider average or max size of non-empty chains?
- For open addressing, half-full is good
- New table size
  - Twice-as-big is a good idea, except, that won’t be prime!
  - Can have a list of prime numbers in your code since you won’t grow more than 20-30 times

Rehashing

When the table gets too full, create a bigger table (usually 2x as large) and hash all the items from the original table into the new table.

- When to rehash?
  - Separate chaining: full ($\lambda = 1$)
  - Open addressing: half full ($\lambda = 0.5$)
  - When an insertion fails
  - Some other threshold