Data Structures

Hashing

Hashing

- Balanced binary search trees support operations such as insert, delete and search in ***O(logn)*** time

- But, if we need these operations in ***O(1)***, then ***Hashing*** provides a way

Hashing

- ***Hashing*** is a technique used for storing and retrieving information as quickly as possible

- ***Hash Tables*** are the data structure we use for implementing *hashing*

Scenario

- **Problem:** Find no. of occurrences of each character in a given string.
  (for simplicity, assume that only letters are considered)

C	L	U	E	A	T	F	I	V	E

Scenario

- **Problem:** Find no. of occurrences of each character in a given string.

- ***Brute Force approach:***
  - Given a string, for each character check whether that character is repeated or not
  - **Inefficient approach**, with $O(n \,^\wedge\, 2)$ complexity

Scenario

- ***Better approach:***
  - Consider that the no. of possible characters is fixed *i.e. 26*
  - Create an array of *size 26* and initialize it with all *zeros*
  - For each of the input characters go to the corresponding index and increment its count
  - Since it is an array, it takes constant time [*O(1)*] for reaching any position

Better Approach

C	L	U	E	A	T	F	I	V	E

It takes just O(1) step to find no. of occurrences' for any given letter

This is the idea behind ***Hashing***

0	1	{ A }
1		{ B }
2	1	{ C }
3		{ D }
4	2	{ E }
5	1	{ F }
6		{ G }
7		{ H }
8	1	{ I }
9		{ J }
10		{ K }
11	1	{ L }
12		{ M }
13		{ N }
14		{ O }
15		{ P }
16		{ Q }
17		{ R }
18		{ S }
19	1	{ T }
20	1	{ U }
21	1	{ V }
22		{ W }
23		{ X }
24		{ Y }
25		{ Z }

Issue

- Now let's assume we had integers instead of letters
- How do we solve this problem using the previous approach since possible values can be infinite *(or very large)*
- Storing counters in arrays is not possible in this case

Hashing

- The technique just proposed is the idea behind ***Hashing***

- ***Hash Tables*** are generalization of arrays in this technique

Components of Hashing

- Hashing has four components:

1. **Hash Table**
2. **Hash Functions**
3. **Collisions**
4. **Collision resolution techniques**

Using Simple Arrays

- Given a key *k, we find the element whose key is k by just* looking in the *kth position of the array. This is called* ***Direct Addressing***

- *Direct addressing* is practical only when we can afford to allocate an array with one position for every possible key

Hash Table

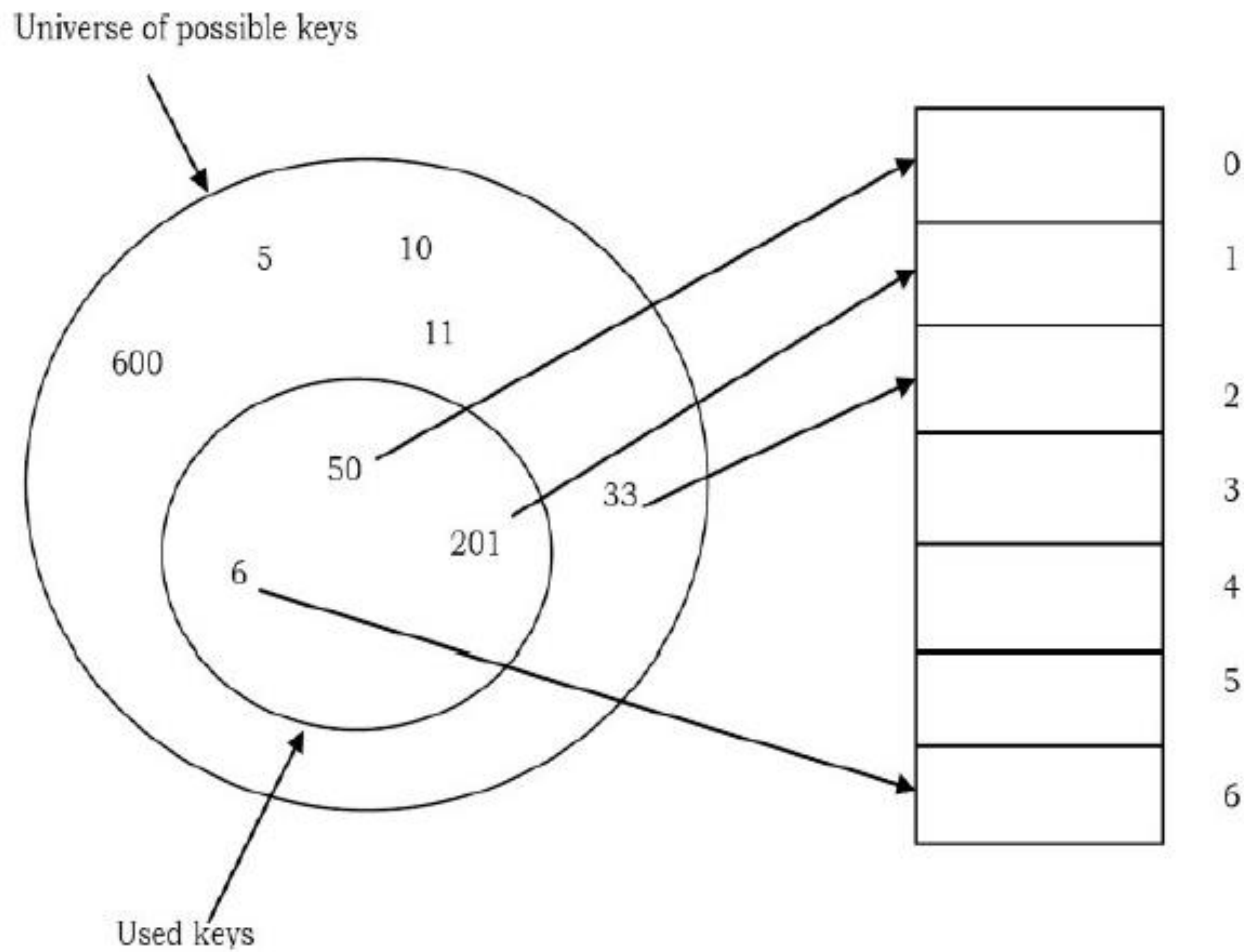
- Normally, we do not have enough space to allocate a location for each possible key, so we need a mechanism to handle this case

- In other words, if we have less locations and more possible keys, then simple array implementation is not enough

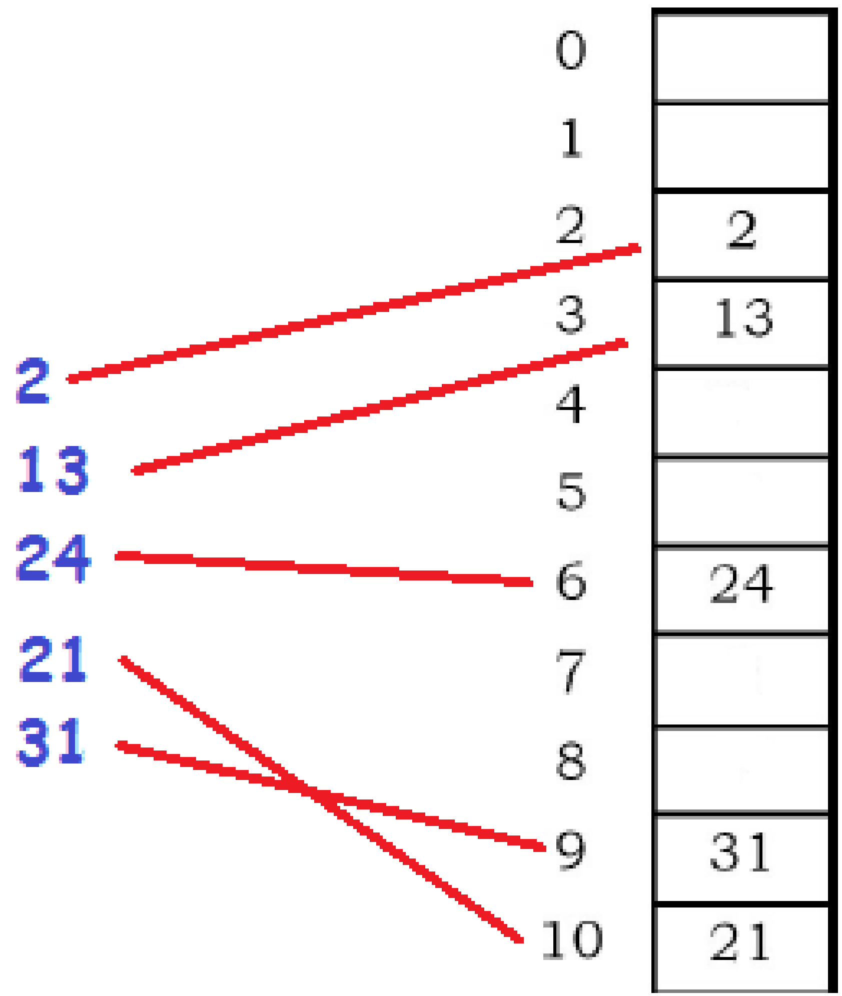
Hash Table

- We use ***Hash Tables*** to overcome this issue
- *Hash table (or hash map)* is a data structure that stores the keys and their associated values, and hash table uses a hash function to map keys to their associated values
- In general, we use *hash table* when the no. of keys actually stored is small relative to the no. of possible keys

Hash Table: Example



Hash Table: Another example



Point to ponder

- ***How exactly are these keys mapped to an actual index?***

Point to ponder

- ***How exactly are these keys mapped to an actual index?***
  - **Answer:** *By using Hash Function*

Hash Function

- The ***Hash Function*** is used to transform the key into the index
- Ideally, the ***hash function*** should map each possible key to a unique slot index, but it is difficult to achieve in practice

Hash Function

- Hash functions can map keys to index but what happens if two keys map to the same index?
- This situation is called **Collision**
- There are ways to deal with collisions

Perfect Hash Function

- Given a collection of elements, a hash function that maps each item into a unique slot is referred to as a ***Perfect Hash Function***

- If we know the elements and the collection will never change, then it is possible to construct a ***perfect hash function***

Perfect Hash Function

- One way to always have a *perfect hash function* is to increase the size of the hash table so that each possible value in the element range can be accommodated

- *Not always possible*

- **<u>Example:</u>** Storing 11-digit telephone numbers

Perfect Hash Function

- Unfortunately, given an arbitrary collection of elements, there is no systematic way to construct a perfect hash function

- Fortunately, we do not need hash functions to be perfect to gain performance efficiency

Characteristics of Good Hash Function

- A good hash function should have the following characteristics:

1. Minimizes collision
2. Is computed quickly & easily
3. Distributes keys evenly in the hash table
4. Uses as much information from key as possible
5. Have a high ***load factor***

Load Factor

- The ***Load Factor*** of a non-empty hash table is the number of items stored in the table divided by the size of the table

$$\text{Load Factor} = \frac{\text{no. of elements in hash table}}{\text{size of hash table}}$$

- It is an important characteristic when we are expanding (*rehashing*) the hash table

Common Hash Functions

Some methods for key to index mapping are:

1. **Division method (remainder method)**
2. **Division method with folding**
3. **Knuth division method**
4. **Multiplication method**

   among many more…

Division Method

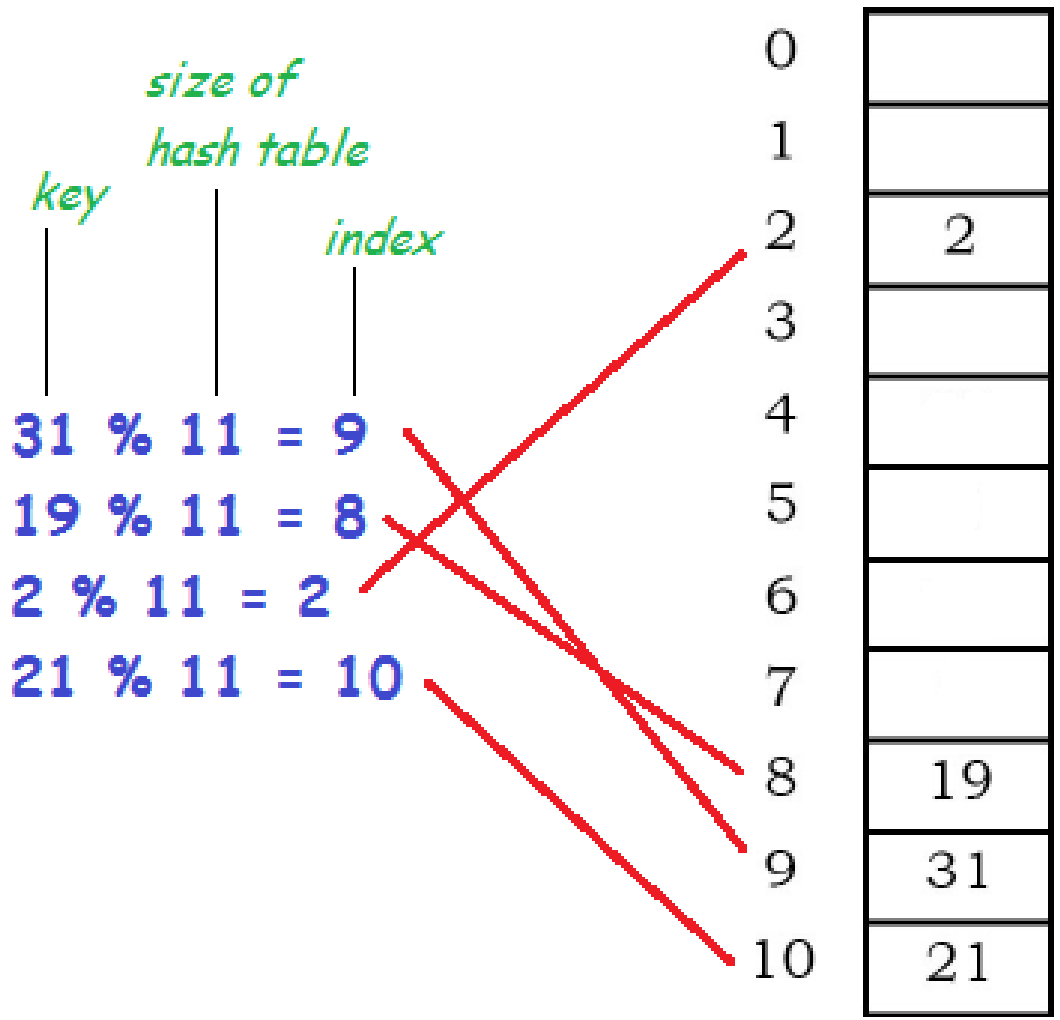
- To map a single integer key k to an index in a hash table of size m, the hash function h(k) is given as:

$$\mathbf{h(k) = k \% m}$$

- **<u>Example:</u>** Map key **3** to an index in a hash table of size **9**:

$$\boldsymbol{h(3) = 3 \% 9 = 3}$$

Example: Division Method



Division Method with Folding

- In this method, we divide the elements into equal size pieces (the last piece may not be of equal size). Then add these pieces before finding the remainder

Division Method with Folding

- **<u>Example:</u>** Mapping for the given 10-digit telephone number **436-555-4601** and hash table of size **11**

- **<u>Step 1:</u>**

  43+65+55+46+01 = 210

- **<u>Step 2:</u>**

  h(210) = 210 % 11 = 1

Other Methods

- **Knuth Variant on Division:**

$$h(k) = k(k+3) \bmod m$$

- **Multiplication Method:**

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

$$0 < A < 1$$

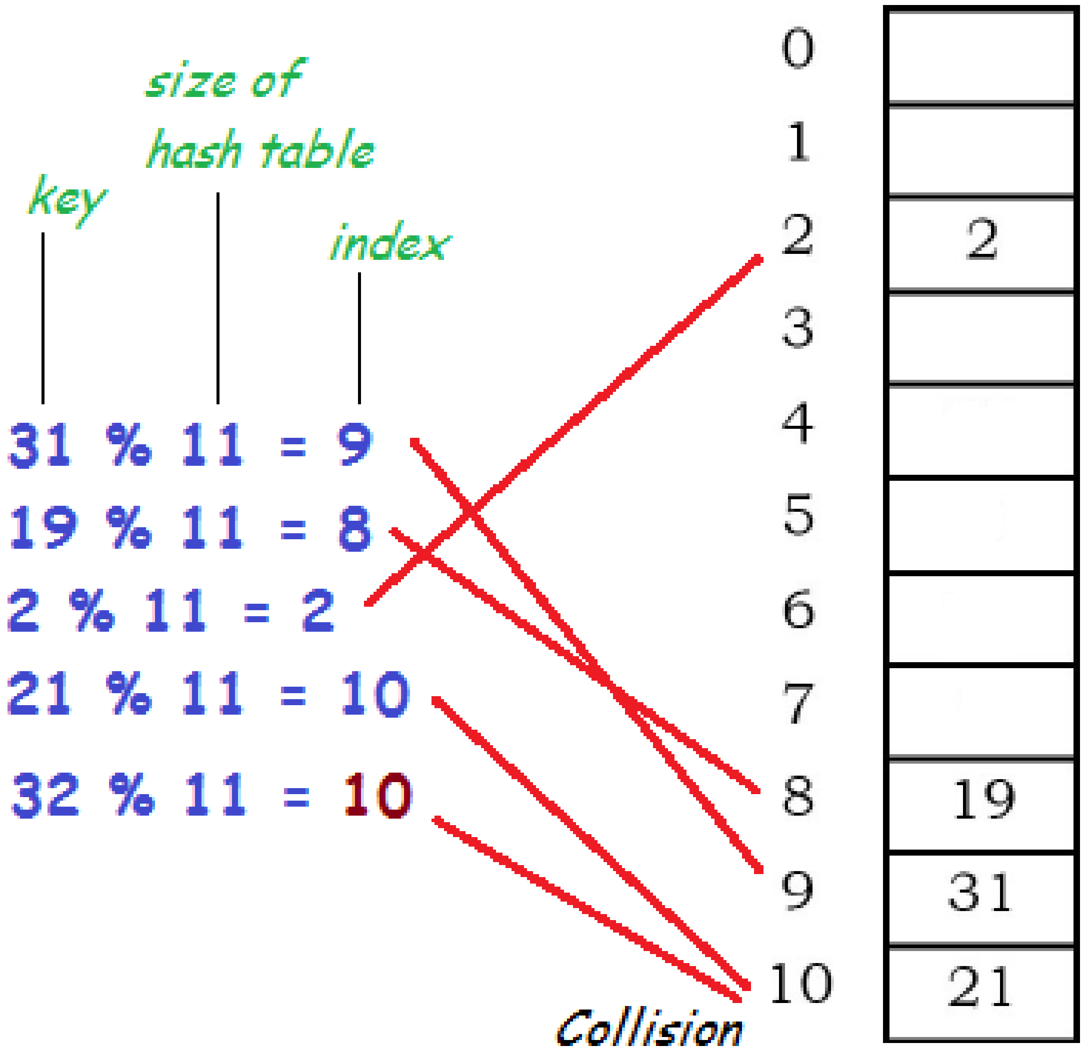
fractional part of kA

Collision

- **Collision** is the condition where two records are stored in the same location

- In hashing, *collision* occurs when hash function maps multiple keys map to the same index

Example



size of
hash table
key
index
31 % 11 = 9
19 % 11 = 8
2 % 11 = 2
21 % 11 = 10
32 % 11 = 10
Collision
0
1
2
3
4
5
6
7
8
9
10
2
19
31
21

Collision Resolution Techniques

- The process of finding an alternate location is called ***collision resolution***

- The most popular techniques are direct chaining and open addressing (closed hashing)

Collision Resolution Techniques

- **Open Addressing**
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

- **Direct Chaining**
  - Separate Chaining

Linear Probing

- In linear probing, we search the hash table sequentially, starting from the original hash location

- If a location is occupied, we check the next location. We wrap around from the last table location to the first table location if necessary

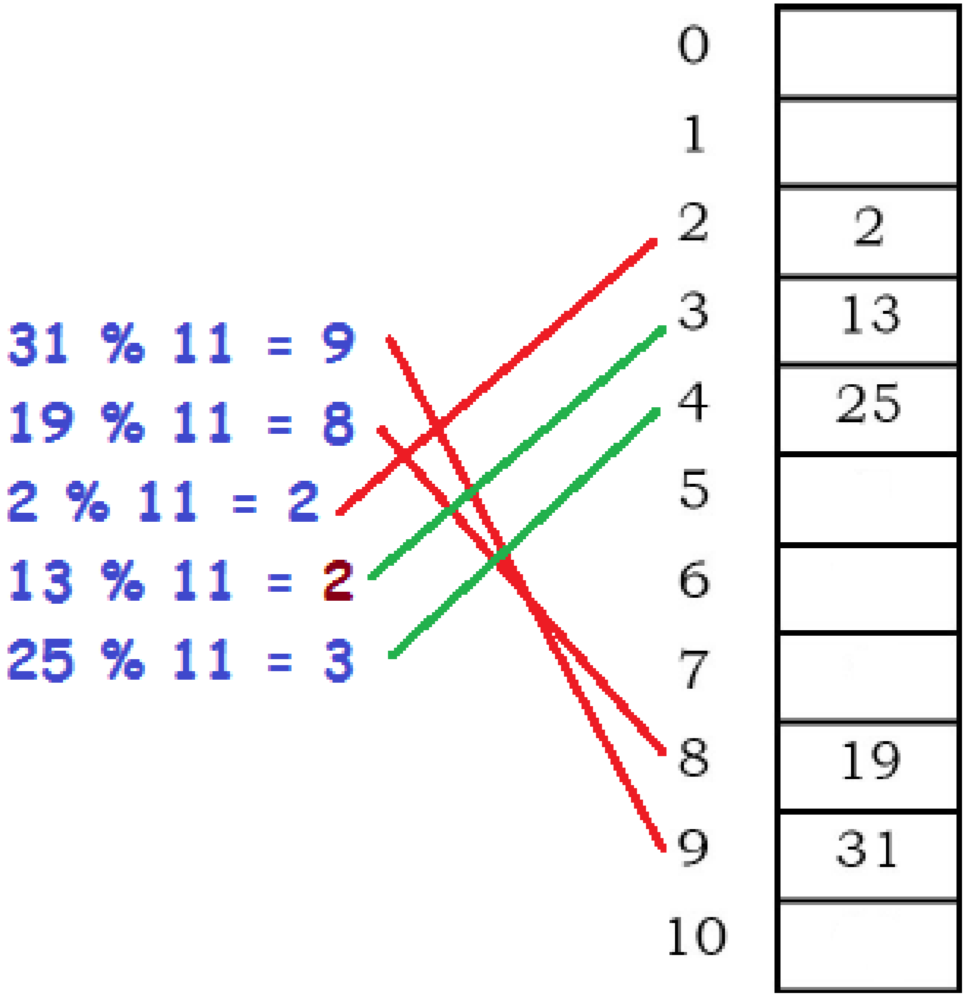
Linear Probing

Next Probe = (n + 1) % tablesize

where n is the originally generated index

- It is easier
- But suffers from the problem of clustering

Linear Probing



Quadratic Probing

- In **Quadratic Probing**, the interval between probes increases proportionally to the hash value (the interval thus increasing linearly, and the indices are described by a quadratic function)

- If a location is occupied, we check the locations $\boldsymbol{i + 1^2}$, $\boldsymbol{i + 2^2}$, $\boldsymbol{i + 3^2}$, $\boldsymbol{i + 4^2}$... *We wrap around from the last table location to* the first, if necessary

Quadratic Probing

Next Probe = (n + k²) % tablesize

where n is the originally generated index

- It reduces clustering
- However, still does not guarantee that no clustering occurs

Quadratic Probing

$31 \% 11 = 9$

$19 \% 11 = 8$

$2 \% 11 = 2$

$13 \% 11 = 2 \rightarrow 2 + 1^2 = 3$

$25 \% 11 = 3 \rightarrow 3 + 1^2 = 4$

$24 \% 11 = 2 \rightarrow 2 + 1^2, 2 + 2^2 = 6$

0	
1	
2	2
3	13
4	25
5	
6	24
7	
8	19
9	31
10	

Double Hashing

- In Double Hashing, the increments for the probing sequence are computed by using a second hash function h2(k) where:

  h2(key) ≠ 0   and   h2 ≠ h1

- We first probe the location *h1(key). If the location is occupied, we probe the location as:*
  *[h1(key) + 1 * h2(key)] % M*
  *[h1(key) + 2 * h2(key)] % M*
  [h1(key) + 3 h2(key)] % M*
  … and so on

- *Much less clustering*

Double Hashing

Table size 11

Hash Function: assume *h1*(*key*) = *key mod* 11 and *h2*(*key*) = 7- (*key mod* 7)

58 mod 11 = 3

14 mod 11 = 3

-> (3 + 1 * 7) mod 11 = 10

91 mod 11 = 3

-> (3 + 1 * 7) mod 11 = 10

-> (3 + 2 * 7) mod 11 = 6

25 mod 11 = 3

-> (3 + 1 * 3) mod 11 = 6

-> (3 + 2 * 3) mod 11 = 9

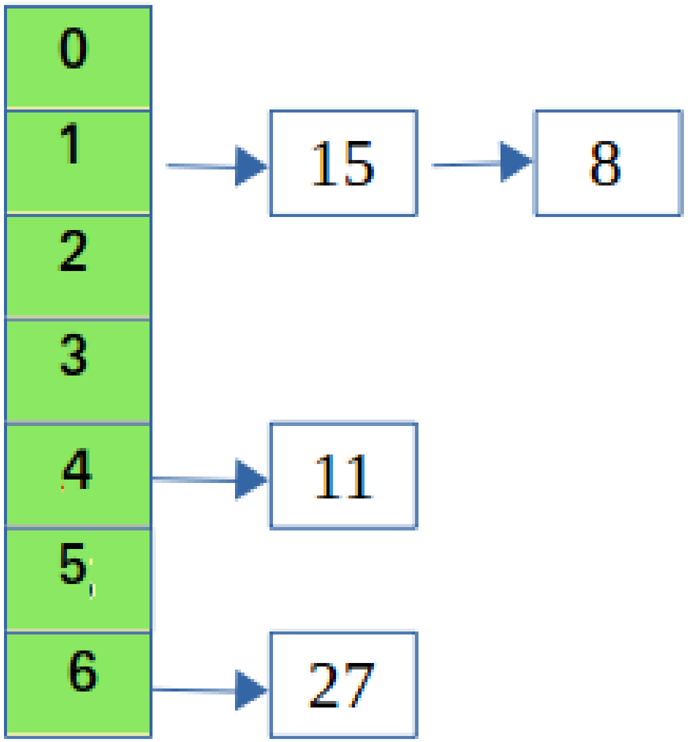
0	
1	
2	
3	58
4	
5	
6	91
7	
8	
9	25
10	14

Separate Chaining

- When two or more records hash to the same location, these records are constituted into a singly-linked list called a *chain*
- *Easy to implement*
- *Uses extra memory with long chains producing up to O(n) probing time*

Separate Chaining

15 % 7 = 1
11 % 7 = 4
27 % 7 = 6
8 % 7 = 1



Rehashing

- If the hash table becomes close to full or exceeds the threshold Load Factor, the search time grows and performance starts to deteriorate

- Rehashing: Building a second table twice as large as the original and rehash there all the keys of the original table

Rehashing

- Rehashing is done because whenever key value pairs are inserted into the map, the load factor increases, which implies that the time complexity also increases

- Rehashing itself is expensive operation but once done, the new hash table will have good performance

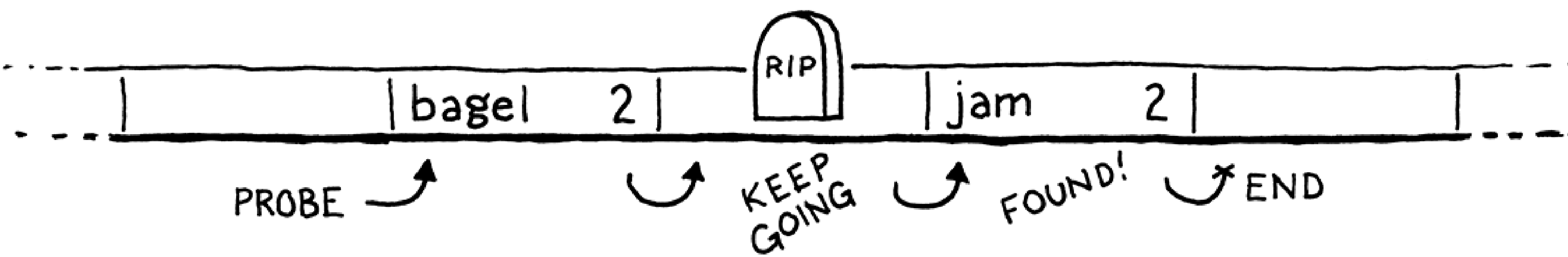
Tombstones: For Probing/Insertion

- The **tombstone** indicates that a record once occupied the slot but does so no longer

- If a **tombstone** is encountered when searching along a probe sequence, the search procedure continues with the search

- When a **tombstone** is encountered during insertion, that slot can be used to store the new record

Tombstones: For Probing/Insertion



Static vs Dynamic Hashing

- In static hashing, the data buckets are kept fixed and given in advance

- In dynamic hashing, the data buckets can be added or removed on-demand

Examples

- SHA1
- MD5

among many more…

When NOT to use Hashing

- Problems where ordering of data is important
- Problems having dynamic nature of data
- Problems in which there are several non-unique keys