Hash Tables

Seema Gondhalekar

Outcomes

- **Students will be able to**
  - **Know the concept of hashing as search technique**
  - **Understand terms related to hash table**
  - **Hashing functions**
  - **Collision handling**

Hash table fundamentals by Seema Gondhalekar
https://www.youtube.com/watch?v=JDpN0OlsnwE

Contents

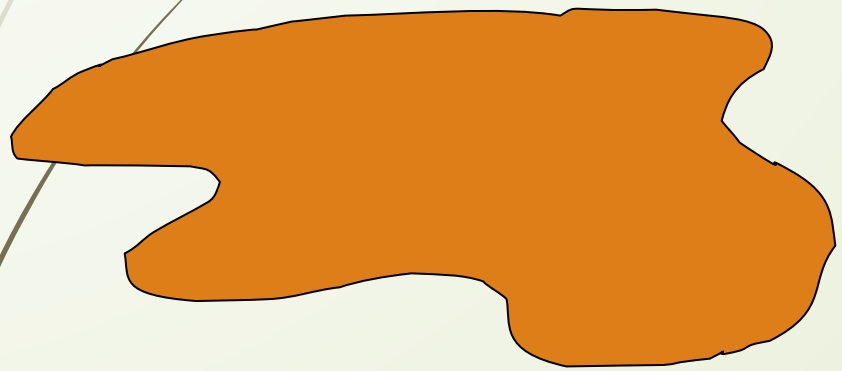
- **Searching and Hashing**
- **Hashing function**
- **Collision**

Search vs. Hashing

- Search tree methods: key comparisons
  - Time complexity:
  - Linear search : O(n)
  - Binary search : O(log n)
- Hashing methods: hash functions
  - Expected time: O(1)
- Types
  - Static hashing
  - Dynamic hashing

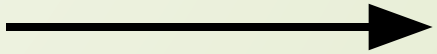
Hash Tables

- Constant time accesses!
- A **hash table** is an array of some fixed size, usually a prime number.
- General idea:

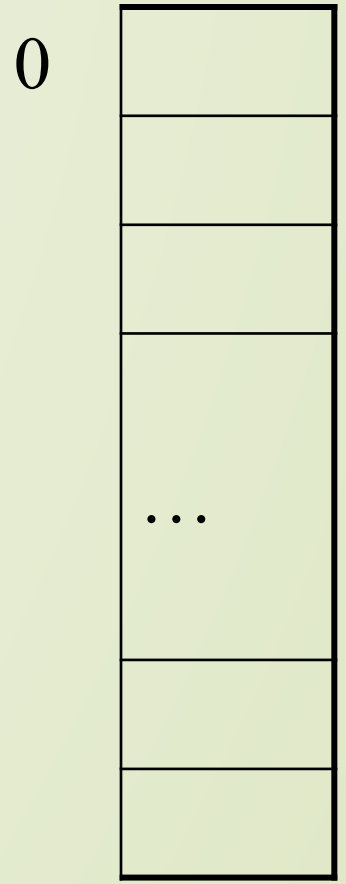


key space (e.g., integers, strings)

hash function:
h(K)



hash table



Concept of Hashing

- In CS, a **hash table**, or a **hash map**, is a data structure that associates keys (names) with values (attributes).
  - Look-Up Table
  - Dictionary
  - Cache

Dictionaries

- Collection of pairs.
  - (key, value)
  - Each pair has a unique key.
- Operations.
  - Get(Key)/search(key) -> get value
  - Delete(Key)
  - Insert(Key, Value)

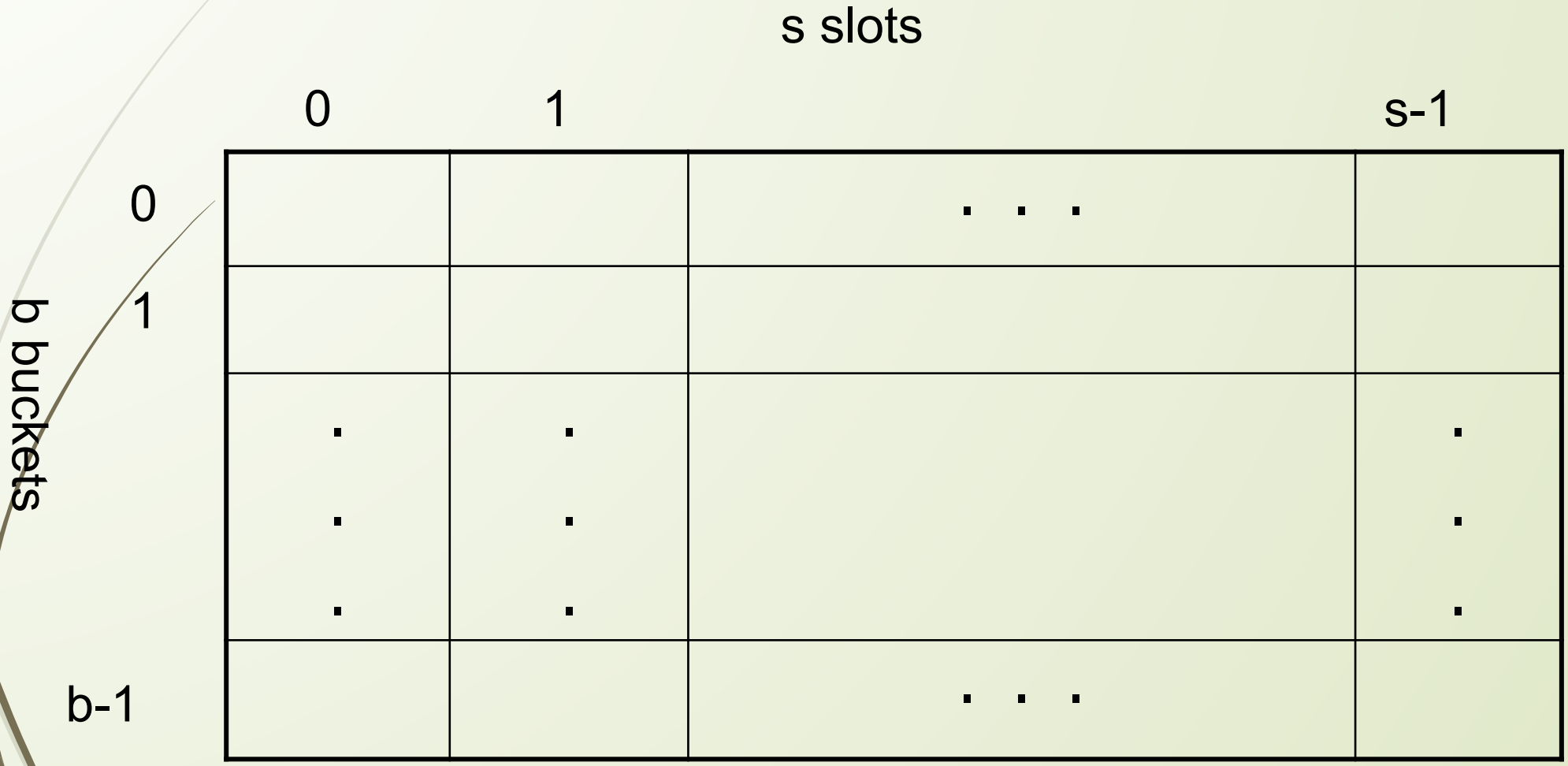
Idea

- Hash table :
  - Collection of pairs,
  - Lookup function (Hash function)
- Hash tables are often used to implement associative arrays,
  - Worst-case time for Get, Insert, and Delete is O(size).
  - Expected time is O(1).

Static Hashing

- Key-value pairs are stored in a fixed size table called a *hash table*.
  - A hash table is partitioned into many ***buckets*** .
  - Each bucket has many ***slots*** .
  - Each slot holds one record.
  - A hash function f(x) transforms the identifier (key) into an address in the hash table

Hash table



Data Structure for Hash Table

Hash table containing **integers**

```
#define MAX  10
int hash_table[MAX];
```

OR

Hash table containing **phonebook**

```
#define MAX  10
struct phonebook{
      string name;
      long phone;
};

phonebook hash_table[MAX];
```

0	20
1	11

0	anil	999999
1	bon	896778

Ideal Hashing

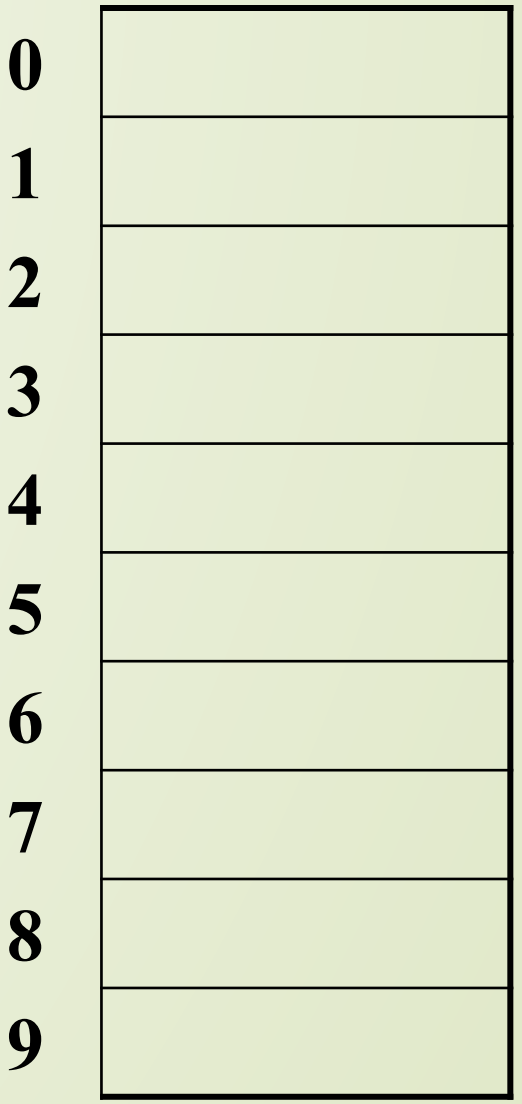
- Uses an array table[0:b-1].
  - Each position of this array is a bucket.
  - A bucket can normally hold only one dictionary pair.
- Uses a hash function f that converts each key k into an index in the range [0, b-1].
- Every dictionary pair (key, element) is stored in its home bucket table[f[key]].

Example

- key space = integers
- TableSize = 10

- **h**(K) = K mod 10

- **Insert**: 7, 18, 41, 94

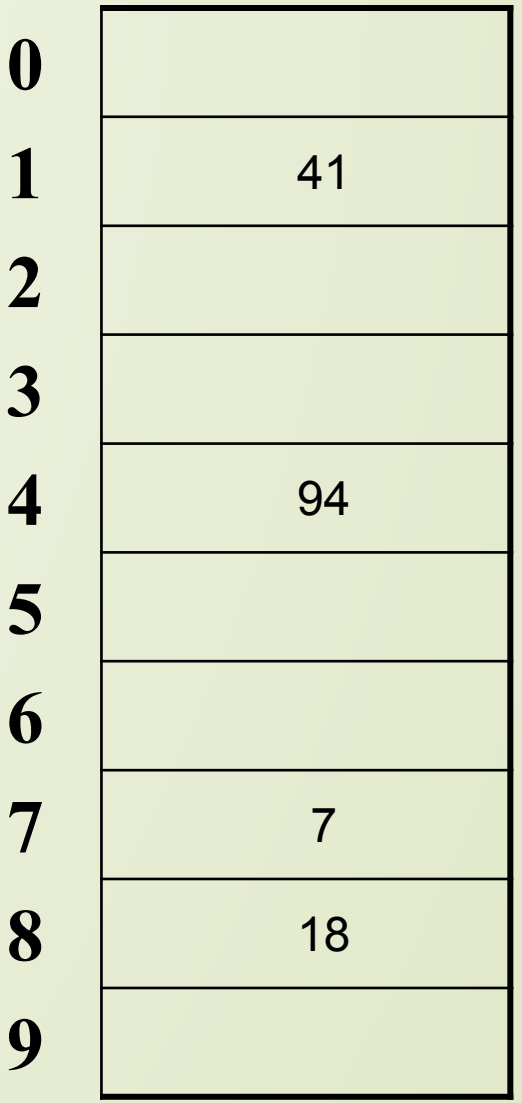


Example

- key space = integers
- TableSize = 10

- **h**(K) = K mod 10

- **Insert**: 7, 18, 41, 94

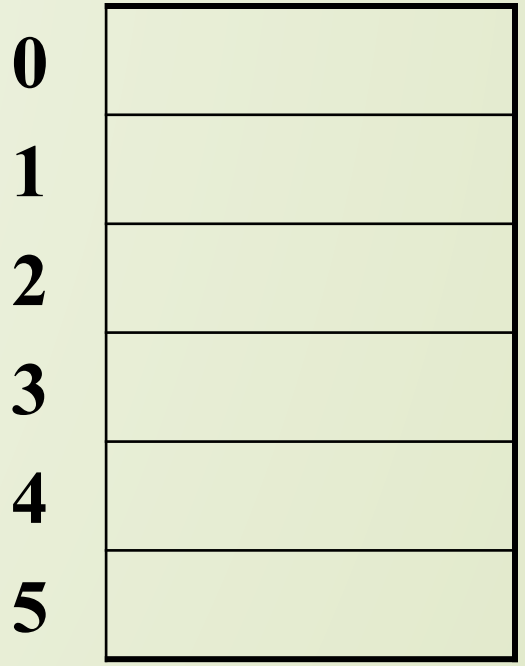


Another Example

- key space = integers
- TableSize = 6

- **h**(K) = K mod 6

- **Insert**: 7, 18, 41, 34



Another Example

- key space = integers
- TableSize = 6

- **h**(K) = K mod 6

- **Insert**: 7, 18, 41, 34

0	18
1	7
2	
3	
4	34
5	41

Concept

- Key-value pairs are stored in a fixed size table called a *hash table*.
  - **A hash table** is partitioned into many ***buckets***. HT[0], ...,HT[b − 1]. Each bucket is capable of holding s records. Each bucket has many ***slots***.
  - Each slot holds one record.
  - Thus, a bucket is said to consist of s slots, each slot being large enough to hold 1 record. Usually s = 1 and each bucket can hold exactly 1 record
  - A **hash function f(x)** transforms the identifier (key) into an **address** in the hash table
  - **Hashing** : The address or location of an identifier, X, is obtained by computing some arithmetic function, f, of X. f(X) gives the address of X in the table. This address will be referred to as the hash or home address of X.

Hash table concepts

- **Identifier density** :The ratio **n / T** is the identifier density, (the number of identifiers **n** and the total number of possible identifiers **T**)
- **loading factor** : n /(sb) is the loading density or loading factor. Since the number of identifiers, n, in use is usually several orders of magnitude less than the total number of possible identifiers T, the number of buckets b, in the hash table is also much less than T. Therefore, the hash function f must map several different identifiers into the same bucket.
- **Synonyms** : Two identifiers key1, key2 are said to be synonyms with respect to f if f(key1) = f(key2). Distinct synonyms are entered into the same bucket so long as all the s slots in that bucket have not been used.
- **An overflow** is said to occur when a new identifier keyn is mapped or hashed by f into a full bucket.
- **A collision** occurs when two non identical identifiers are hashed into the same bucket. **When the bucket size s is 1, collisions and overflows occur simultaneously.**

Important Points

- Widely useful technique for implementing dictionaries
- Constant time per operation (on the average) : O(1)
- Worst case time proportional to the size of the set for each operation

Applications of Hashing

- Hashing has two main applications.
  - Hashed values can be used to speed data retrieval,
  - Hashed values can be used to check the validity of data.

To speed data retrieval

- When we want to retrieve one record from many in a file, searching the file for the required record takes time that varies with the number of records.
- If we can generate a hash for the record's key, we can use that hash value as the "address" of the record and move directly to it; this takes the same time regardless of the number of records in the file.

To check the validity of data

- The validity of data can be checked with a hash. This can be used to check both that a file transferred correctly, and that a file has not been deliberately manipulated by someone between me uploading it somewhere and you downloading it.
- If I post both the file and the hash value I generated from it, you can generate a hash value from the file you received and compare the hash values.
- If the hashing algorithm is a good cryptographic hash, it's extremely unlikely that accident or malice would have modified the file even a little yet it would still yield the same hash value.

Revision of terms

Hash Table concepts

- The key density (or identifier density) of a hash table is the ratio n/T
  - n is the number of keys in the table
  - T is the number of distinct possible keys
- The loading density or loading factor of a hash table is $\alpha = n/(sb)$
  - s is the number of slots
  - b is the number of buckets

Synonyms

- Since the number of buckets b is usually several orders of magnitude lower than T, the hash function f must map several different identifiers into the same bucket
- Two identifiers, i and j are synonyms with respect to f if $f(i) = f(j)$

Overflow and Collision

- An overflow occurs when we hash a new identifier into a full bucket
- A collision occurs when we hash two non-identical identifiers into the same bucket

Example

synonyms:
char, ceil,
clock, ctime

overflow

	Slot 0	Slot 1
0	acos	atan
1		
2	char	ceil
3	define	
4	exp	
5	float	floor
6		
…		
25		

synonyms

synonyms

b=26, s=2, n=10, α=10/52=0.19, f(x)=the first char of x

x: acos, define, float, exp, char, atan, ceil, floor, clock, ctime

f(x):0, 3, 5, 4, 2, 0, 2, 5, 2, 2

Loading factor

- As an example, let us consider the hash table HT with b = 26 buckets, each bucket having exactly two slots, i.e., s = 2.
- Assume that there are n = 10 distinct identifiers in the program and that each identifier begins with a letter. **The loading factor, for this table is 10/52 = 0.19**
- Assume that there are **n = 10 distinct identifiers and slot is 1** in the program and that each identifier begins with a letter. **The loading factor, for this table is 10/26 = 0.38**.

Hashing Functions

- Requirements for good hashing function
  - easy computation
  - minimal number of collisions
  - Uniform distribution
- mid-square (middle of square)

$$f_m(x) = middle(x^2)$$

- division

$$f_D(x) = x\%M \qquad (0 \sim (\text{M-1}))$$

Avoid the choice of M that leads to many collisions

Requirements for good hashing function

To achieve a good hashing mechanism, It is important to have a good hash function with the following basic requirements:

- Easy to compute: It should be easy to compute and must not become an algorithm in itself.
- Uniform distribution: It should provide a uniform distribution across the hash table and should not result in clustering.
- Less collisions: Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.

Need for a good hash function

- Let us understand the need for a good hash function. Assume that you have to store strings in the hash table by using the hashing technique {“abcdef”, “bcdefa”, “cdefab” , “defabc” }.
- To compute the index for storing the strings, use a hash function that states the following:
- The index for a specific string will be equal to the sum of the ASCII values of the characters modulo 599.
- The hash function will compute the same index for all the strings and the strings will be stored in the hash table in the following format

Hashing Functions

- mid-square (middle of square)
- division
- Folding
- Extraction
- Multiplication
- Rotation

- Universal Hashing

Hashing Functions (Division)

- Popular
- One simple choice for a hash function is to use the modulus operator, % or MOD.
- Here, Hash(I) = I % M
- The identifier is divided by some number M, and the remainder is used as the hash address for I. This function gives bucket addresses in the range of O through (M – 1),
- so the hash table is atleast of size n = M. The choice of M is critical. When using this method,

Hashing Functions (Multiplication)

- Another hash function which has been widely used in many applications is the multiplication method. The multiplication method works in the following steps.

  1. Multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of KA.
  2. Then multiply this value by M and take the floor of the result.

- $$\text{Hash}(K) = m\,(KA \bmod 1),$$

- where, 'KA Mod 1' means the fractional part of KA, that is, KA – KA.

Hashing Functions (Mid square)

- In this method key is squared and the address is selected from the middle of the squared key. If the key is large then it is very difficult to store square of it. So mid square is used when key size is less than 4 digits..

Key	Square	Address
2341	5480281	802
1671	2792241	922

- We can select portion of key if key is larger in size and then squre the portion of it. For example

Key	Square	Address
234137	234*234=027889	788
567187	567*567= 321489	148

Hashing Functions

- mid-square (middle of square)

$$f_m(x) = middle(x^2)$$

2341 -> 5480281 -> 802

Limitation storage limit (4 digit)

- division

$$f_D(x) = x\%M \qquad (0 \sim (M\text{-}1))$$

Avoid the choice of M that leads to many collisions

- M should be prime number > 20

Hashing Functions(Folding)

There are two types of folding method

1) **fold Shift :** Key value is divided into parts of address size. Left and right parts are shifted and added with middle part.
2) **Fold boundary :** Key value is divided into parts of address size. Left and right parts are folded on fixed boundary between them and the center part.

For example

If key is 987654321

Left 987 center 654 right 321

For fold shift Addition is 987 + 654 + 321 =1962 Discard digit 1 and address is 962

For fold boundary Addition of reverse part is 789 + 456 + 123 =1368 Discard digit 1 and address is 368

Hashing Functions

- Folding
  - Partition the identifier x into several parts
  - All parts except for the last one have the same length
  - Add the parts together to obtain the hash address
  - Two possibilities
    - Shift folding Key is 12320324111220
      - x1=123, x2=203, x3=241, x4=112, x5=20, address=699
    - Folding at the boundaries
      - x1=123, x2=203 (reverse 302), x3=241, x4=112(reverse 211), x5=20, address=897

Hashing Functions(Rotation)

- This method is used along with other method. Here the key is rotated as 120605 then it is rotated as 512060

Extraction method

- Portion of the key is used for address calculation
- In digit extraction method, selected digits are extracted from the key and used as the address.
- For example book accession number is of six digits and we require address of 3 digits then we can select odd number digits as first, third and fifth that can be used as address for hash table.

Example

Key	Address
345678	357
234137	243
952671	927

Digital Analysis

- Useful for static files where all keys are known in advance
- All the identifiers are known in advance

  M=1~999

  X_1 d_{11} d_{12} … d_{1n}

  X_2 d_{21} d_{22} … d_{2n}

  …

  X_m d_{m1} d_{m2} … d_{mn}

- Select 3 digits from n

  Criterion:

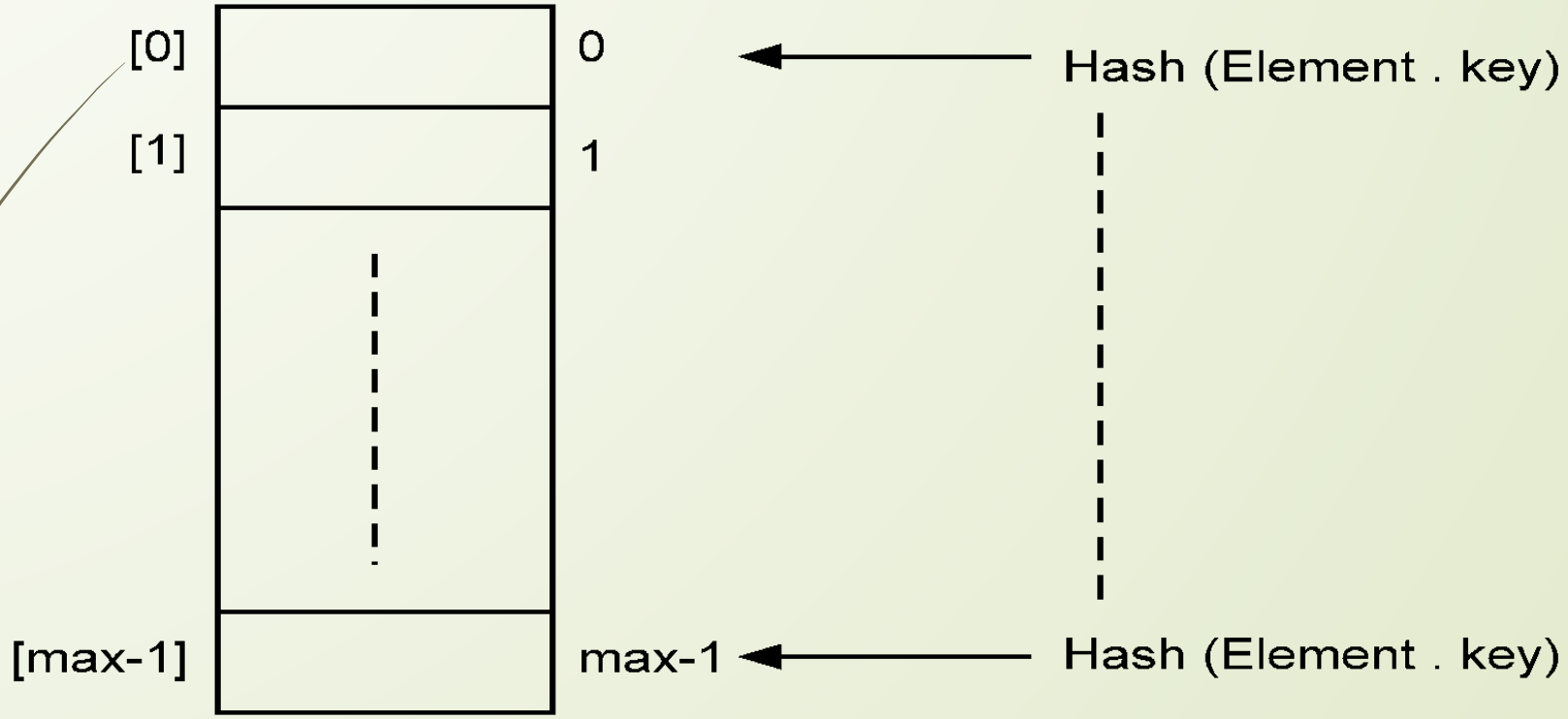
  Delete the digits having the most skewed distributions

Hashing Functions(Universal Hashing)

- If a malicious adversary chooses the keys to be hashed, then we can choose n keys that all hash to the same slot, yielding an average retrieval time of O(n).
- Any fixed hash function is vulnerable to this sort of worst case behaviour;
- the only effective way to improve the situation is to choose the hash function randomly in a way that is independent of the keys in a way that is independent of the keys that are actually going to be stored.
- This approach is called **universal hashing**, yields good performance on the average, no matter what keys are choosen by the adversary.
- The main idea behind universal hashing is to select the hash function at random at run time from a carefully designed class of functions.
- Because of randomization, the algorithm can behave differently on each execution; even for the same input. This approach guarantees good average case performance, no matter what keys are provided as input.

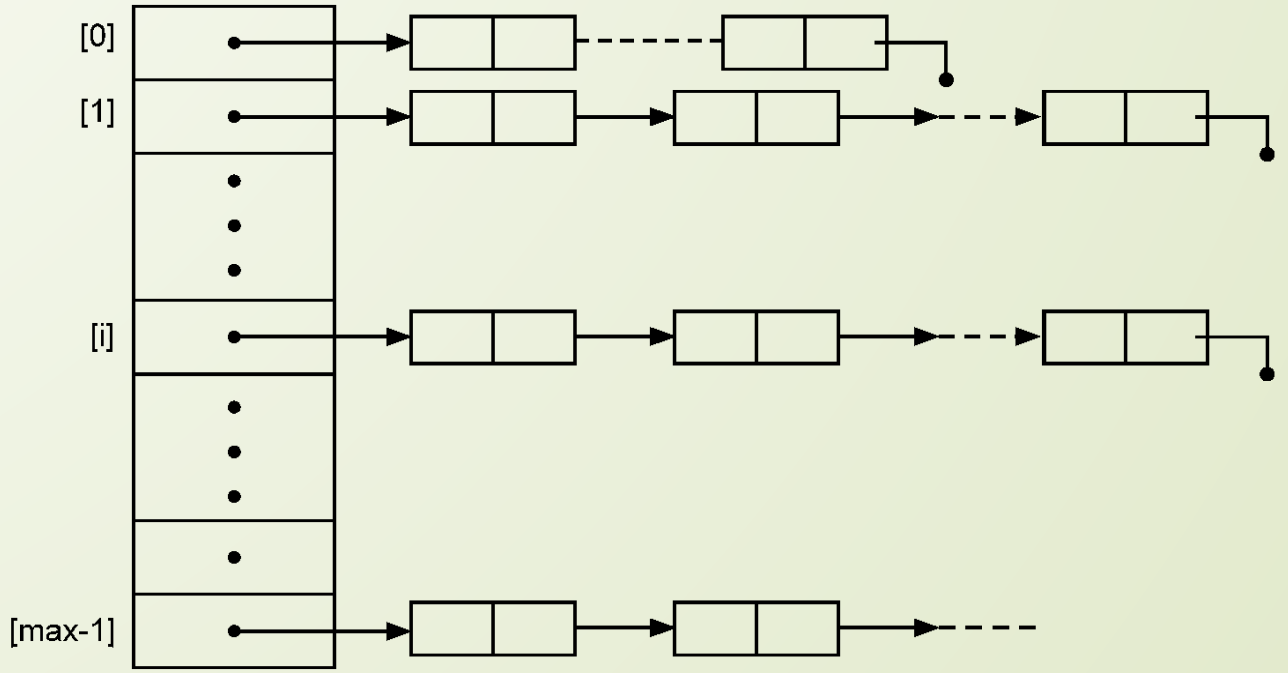
Closed Hashing

- A closed hash table is one where the buckets contain items. When an overflow occurs in a closed hash table, the table is searched for a free position in which to store the synonym.
- A close hash table keeps the items in the bucket table itself, rather than using that table to store the pointers to list headers



Open Hashing

- An open hash table is one in which a bucket does not contain an item, but rather a pointer to a list or chain of items that are synonyms.
- It is simply an array indexed by a function of the key where the component type of the array is a pointer to list, that is an array of list of items.
- The bucket size is 1.
- Open hashing which is also called as external hashing allows the set of keys to be stored in a potentially unlimited space, and there places no limit on the size of set of keys.



Terminology Alert!

"**Open** Hashing"
equals
"Separate Chaining"

"Closed Hashing"
equals
"**Open** Addressing"

Collision Resolution

Collision: when two keys map to the same location in the hash table.

Two ways to resolve collisions:

1. Open Addressing (linear probing, quadratic probing, double hashing)
2. Separate Chaining

Linear Probing

A hash table in which a collision is resolved by putting the item in the next empty place in following the occupied place is called linear probing.

This strategy looks for the next free location until one is found.

Here,

(Hash(x)+i)ModMax

Let i=1.

Linear Probing

A hash table in which a collision is resolved by putting the item in the next empty place in following the occupied place is called linear probing.

This strategy looks for the next free location until one is found.

Here,

(Hash(x)+i) Mod Max

Let i=1.

- Deletion is difficult because finding collisions again relies or not creating empty slot.

Linear probing can be done using

- With replacement - If the slot is already occupied by the key with other hash address then the new key having address at that slot is replaced at that position and the key with other address is placed in next empty position
- Without replacement - If the slot is already occupied by the new key is placed in next empty position.

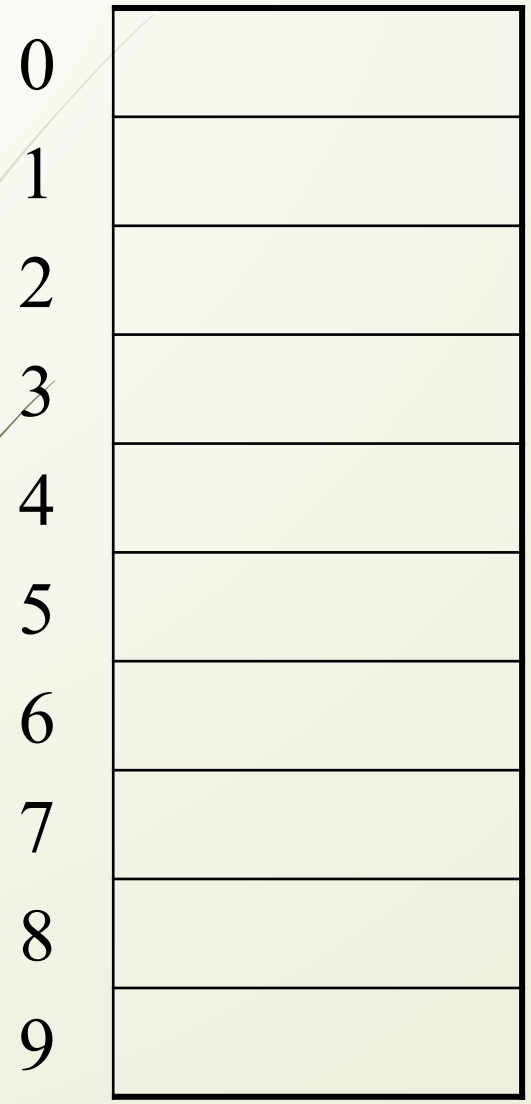
tableSize: Why Prime?

- Suppose
  - data stored in hash table: 7160, 493, 60, 55, 321, 900, 810
  - tableSize = 10
    data hashes to 0, 3, 0, 5, 1, 0, 0
  - tableSize = 11
    data hashes to 10, 9, 5, 0, 2, 9, 7

Real-life data tends to have a pattern

Being a multiple of 11 is usually *not* the pattern ☺

Open Addressing



Insert:
38
19
8
109
10

Linear Probing: after checking spot h(k), try spot h(k)+1, if that is full, try h(k)+2, then h(k)+3, etc.

Linear Probing

$$f(i) = i$$

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + 1) mod TableSize

  2^{th} probe = (h(k) + 2) mod TableSize

  . . .

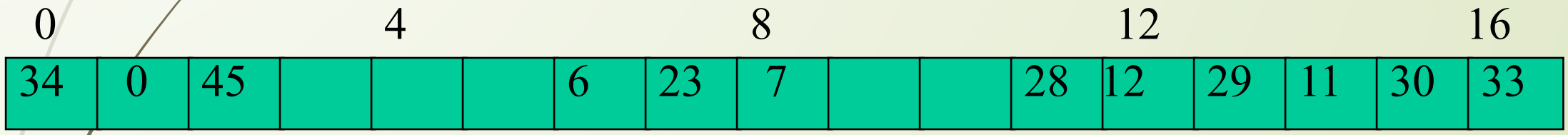
  i^{th} probe = (h(k) + i) mod TableSize

Example of Linear Probing

- Explain linear probing with and without replacement.
  Using following data
  12, 01, 04, 03, 07, 08, 10, 02, 05,14, 06, 28
  Assume buckets from 0 to 9 and each bucket has one slot Calculate average cost/number of comparisons for the both.
  Linear probing without replacement using hashing .function x%10

Linear Probing – Get And Insert

- divisor = b (number of buckets) = 17.
- Home bucket = key % 17.



- Insert pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

Example of Linear Probing(Linear probing without replacement)

bucket	Initially
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

Example of Linear Probing(Linear probing without replacement)

	Initially	Insert 12	Insert 01	Insert 04	Insert 03	Insert 07	Insert 08	Insert 10	Insert 02	05	14	06	28
No of comparisons													
bucket													
0													
1													
2													
3													
4													
5													
6													
7													
8													
9													
T													

Example of Linear Probing(Linear probing without replacement)

	Initially	Insert 12	Insert 01	Insert 04	Insert 03	Insert 07	Insert 08	Insert 10	Insert 02	05	14	06	28
No of comparisons		1											
bucket													
0													
1													
2		12											
3													
4													
5													
6													
7													
8													
9													
T													

Example of Linear Probing(Linear probing without replacement)

	Initially	Insert 12	Insert 01	Insert 04	Insert 03	Insert 07	Insert 08	Insert 10	Insert 02	05	14	06	28
No of comparisons		1											
bucket													
0													
1			01										
2		12	12										
3													
4													
5													
6													
7													
8													
9													
T													

Example of Linear Probing(Linear probing without replacement)

	Initially	Insert 12	Insert 01	Insert 04	Insert 03	Insert 07	Insert 08	Insert 10	Insert 02	05	14	06	28
No of comparisons		1	1	1	1	1	1	1	4	2	6	10	10
bucket													
0								10	10				
1			01	01	01	01	01	01	01				
2		12	12	12	12	12	12	12	12				
3					03	03	03	03	03				
4				04	04	04	04	04	04				
5													
6													
7						07	07	07	07				
8							08	08	08				
9													

Table is full after inserting 14, so 06 and 28 can not be inserted.

Total Number of comparisons 1+ 1+ 1+ 1 + 1 + 1 + 1 + 4 + 2 + 6 + 10+10 = 39

Example of Linear Probing(Linear probing without replacement)

	Initially	Insert 12	Insert 01	Insert 04	Insert 03	Insert 07	Insert 08	Insert 10	Insert 02	05	14	06	28
No of comparisons		1	1	1	1	1	1	1	4	2	6	10	10
bucket													
0								10	10	10			
1			01	01	01	01	01	01	01	01			
2		12	12	12	12	12	12	12	12	12			
3					03	03	03	03	03	03			
4				04	04	04	04	04	04	04			
5									02	02			
6													
7						07	07	07	07	07			
8							08	08	08	08			
9													

Table is full after inserting 14, so 06 and 28 can not be inserted.

Total Number of comparisons 1+ 1+ 1+ 1 + 1 + 1 + 1 + 4 + 2 + 6 + 10+10 = 39

Example of Linear Probing(Linear probing without replacement)

	Initially	Insert 12	Insert 01	Insert 04	Insert 03	Insert 07	Insert 08	Insert 10	Insert 02	05	14	06	28
No of comparisons		1	1	1	1	1	1	1	4	2	6	10	10
bucket													
0								10	10	10	10		
1			01	01	01	01	01	01	01	01	01		
2		12	12	12	12	12	12	12	12	12	12		
3					03	03	03	03	03	03	03		
4				04	04	04	04	04	04	04	04		
5									02	02	02		
6										05	05		
7						07	07	07	07	07	07		
8							08	08	08	08	08		
9													

Table is full after inserting 14, so 06 and 28 can not be inserted.

Total Number of comparisons 1+ 1+ 1+ 1 + 1 + 1 + 1 + 4 + 2 + 6 + 10+10 = 39

Example of Linear Probing(Linear probing without replacement)

	Initially	Insert 12	Insert 01	Insert 04	Insert 03	Insert 07	Insert 08	Insert 10	Insert 02	05	14	06	28
No of comparisons		1	1	1	1	1	1	1	4	2	6	10	10
bucket													
0								10	10	10	10		
1			01	01	01	01	01	01	01	01	01		
2		12	12	12	12	12	12	12	12	12	12		
3					03	03	03	03	03	03	03		
4				04	04	04	04	04	04	04	04		
5									02	02	02		
6										05	05		
7						07	07	07	07	07	07		
8							08	08	08	08	08		
9											14		

Table is full after inserting 14, so 06 and 28 can not be inserted.

Total Number of comparisons 1+ 1+ 1+ 1 + 1 + 1 + 1 + 4 + 2 + 6 + 10+10 = 39

Example of Linear Probing(Linear probing without replacement)

	Initially	Insert 12	Insert 01	Insert 04	Insert 03	Insert 07	Insert 08	Insert 10	Insert 02	05	14	06	28
No of comparisons		1	1	1	1	1	1	1	4	2	6	10	10
bucket													
0								10	10	10	10	10	10
1			01	01	01	01	01	01	01	01	01	01	01
2		12	12	12	12	12	12	12	12	12	12	12	12
3					03	03	03	03	03	03	03	03	03
4				04	04	04	04	04	04	04	04	04	04
5									02	02	02	02	02
6										05	05	05	05
7						07	07	07	07	07	07	07	07
8							08	08	08	08	08	08	08
9											14	14	14

Table is full after inserting 14, so 06 and 28 can not be inserted.

Total Number of comparisons 1+ 1+ 1+ 1 + 1 + 1 + 1 + 4 + 2 + 6 + 10+10 = 39

Linear probing with replacement

Example of Linear Probing(Linear probing with replacement)

	Initially	Insert 12	Insert 01	Insert 04	Insert 03	Insert 07	Insert 08	Insert 10	Insert 02	05	14	06	28
No of comparisons		**1**	**1**	**1**	**1**	**1**	**1**	**1**	**4**	**3**	**6**	**10**	**10**
bucket													
0								10	10	10	10	10	10
1			01	01	01	01	01	01	01	01	01	01	01
2		12	12	12	12	12	12	12	**12**	12	12	12	12
3					03	03	03	03	03	03	03	03	03
4				04	04	04	04	04	04	04	**04**	04	04
5									**02**	**05**	05	02	02
6										**02**	02	05	05
7						07	07	07	07	07	07	07	07
8							08	08	08	08	08	08	08
9											**14**	14	14

When inserting 05 at position 5 which is occupied by other key is replaced by 05 and 02 is placed in next empty position.

Table is full after inserting 14, so 06 and 28 can not be inserted.

Total Number of comparisons 1+ 1+ 1+ 1 + 1 + 1 + 1 + 4 + 3 + 6 + 10+10 = 40

```
#define MAX 10

//hash function to get position
int hash(int key)
{
      return ( key % MAX);
}
```

```
/ //function for inserting element using linear probe
        int insert_linear_prob( int key) {
            int pos, i ;
            pos = hash(key);

            if (Hashtable[pos] == 0)    {  // empty slot
                Hashtable[pos] = key;
                return pos;
            }
            else    {   // slot is not empty
                for( i=(pos+1)%MAX;(i % MAX)!=pos;i=(i % MAX)+1){
                    if ( Hashtable[i] == 0)
                    {
                        Hashtable[i] = key;
                        return i ;
                    }
                }
            }
            // Table overflow
            return -1;
    } // end of insert
```

Problem of Linear Probing

- Identifiers tend to cluster together
- Adjacent cluster tend to join together
- Increase the search time

Quadratic Probing

- Although linear probing is easy to implement, it tends to for clusters of synonyms, resulting in secondary clustering (merging of clusters). In quadratic probing the empty location is searched using following formula.
  - (hash(Key1)±i^2) mod Max
  - where, i goes from 1 to (Max – 1)/2. If Max is a prime number of the form (4 * integer + 3) quadratic probing covers all of the buckets in the table. Quadratic probing slows down the growth of secondary clusters.
  - Quadratic probing works much better than linear probing, but to make full use of hash table, there are constraints on the values of i and Max. Also, if two keys have the same
  - initial probe position, then their sequences are same, since Hash(1,1)=Hash(Key2,1) implies Hash(Key1,i)=Hash(Key2,i). This leads to a milder form of clustering, called as **secondary clustering.**
  - As a linear probing, the initial probes determines the entire sequence, so only max district probe sequences are used.

Quadratic Probing

Less likely to encounter Primary Clustering

$f(i) = i^2$

- (hash(Key)±i^2) mod Max

- Probe sequence:

  0^{th} probe = (h(k) + 0) mod TableSize

  1^{th} probe = (h(k) + 1) mod TableSize

  2^{th} probe = (h(k) + 4) mod TableSize

  3^{th} probe = (h(k) + 9) mod TableSize

  . . .

  i^{th} probe = (h(k) + i^2) mod TableSize

Open addressing using quadratic probing

Given the input { 4371, 1323, 6173, 4199, 4344, 9679, 1989 } and hash function h(x)=x mod 10, show the results for the following

	Initially	Insert 4371	Insert 1323	Insert 6173	Insert 4199	Insert 4344	Insert 9679	Insert 1989
0								
1		4371	4371	4371				
2								
3			1323	1323				
4								
5								
6								
7								
8								
9								

Open addressing using quadratic probing

Given the input { 4371, 1323, 6173, 4199, 4344, 9679, 1989 } and hash function h(x)=x mod 10, show the results for the following

We will consider collision conditions

When inserting 6173 address is (6173 %10 + 1^2)%10= 4

	Initially	Insert 4371	Insert 1323	Insert 6173	Insert 4199	Insert 4344	Insert 9679	Insert 1989
0								
1		4371	4371	4371				
2								
3			1323	1323				
4				**6173**				
5								
6								
7								
8								
9								

Open addressing using quadratic probing

Given the input { 4371, 1323, 6173, 4199, 4344, 9679, 1989 } and hash function h(x)=x mod 10, show the results for the following

When inserting 4344 address is (4344 % 10 + 1^2)%10= 5

	Initially	Insert 4371	Insert 1323	Insert 6173	Insert 4199	Insert 4344	Insert 9679	Insert 1989
0								
1		4371	4371	4371	4371	4371		
2								
3			1323	1323	1323	1323		
4				6173	6173	6173		
5						**4344**		
6								
7								
8								
9					**4199**	4199		

Open addressing using quadratic probing

Given the input { 4371, 1323, 6173, 4199, 4344, 9679, 1989 } and hash function h(x)=x mod 10, show the results for the following

When inserting 9699 address is $(9699 \% 10 + 1^2) \% 10 = 10 \% 10 = 0$

	Initially	Insert 4371	Insert 1323	Insert 6173	Insert 4199	Insert 4344	Insert 9679	Insert 1989
0							9699	
1		4371	4371	4371	4371	4371	4371	
2								
3			1323	1323	1323	1323	1323	
4				6173	6173	6173	6173	
5						4344	4344	
6								
7								
8								
9					4199	4199	4199	

Open addressing using quadratic probing

Given the input { 4371, 1323, 6173, 4199, 4344, 9679, 1989 } and hash function h(x)=x mod 10, show the results for the following

When inserting 1889 address is 1989 % 10 =9 occupied

(1989%10+1^2)%10=0 occupied (1989%10+2^2)%10=3 occupied (1989%10+3^2)%10= (9 + 9)%10=18 % 10 = 8

	Initially	Insert 4371	Insert 1323	Insert 6173	Insert 4199	Insert 4344	Insert 9679	Insert 1989
0							9679	9679
1		4371	4371	4371	4371	4371	4371	4371
2								
3			1323	1323	1323	1323	1323	1323
4				6173	6173	6173	6173	6173
5						4344	4344	4344
6								
7								
8								**1989**
9					4199	4199	4199	4199

Double hashing

This strategy makes use of formula f(i) = i * hash(key). Using formula we apply second hash function to key and probe at distance hash(key), 2*hash(key) ….. so on.
For example we take hash function as key % 10 then Open addressing using second hash function h2 (x) = 7- (x mod 7)

	Initially	Insert 4371	Insert 1323	Insert 6173	Insert 4199	Insert 4344	Insert 9679	Insert 1989
0							9699	
1		4371	4371	4371	4371	4371	4371	4371
2								
3			1323	1323	1323	1323	1323	1323
4				6173	6173	6173	6173	6173
5							9679	9679
6								
7						4344	4344	4344
8								1989
9					4199	4199	4199	4199

Double hashing

We will consider collision conditions

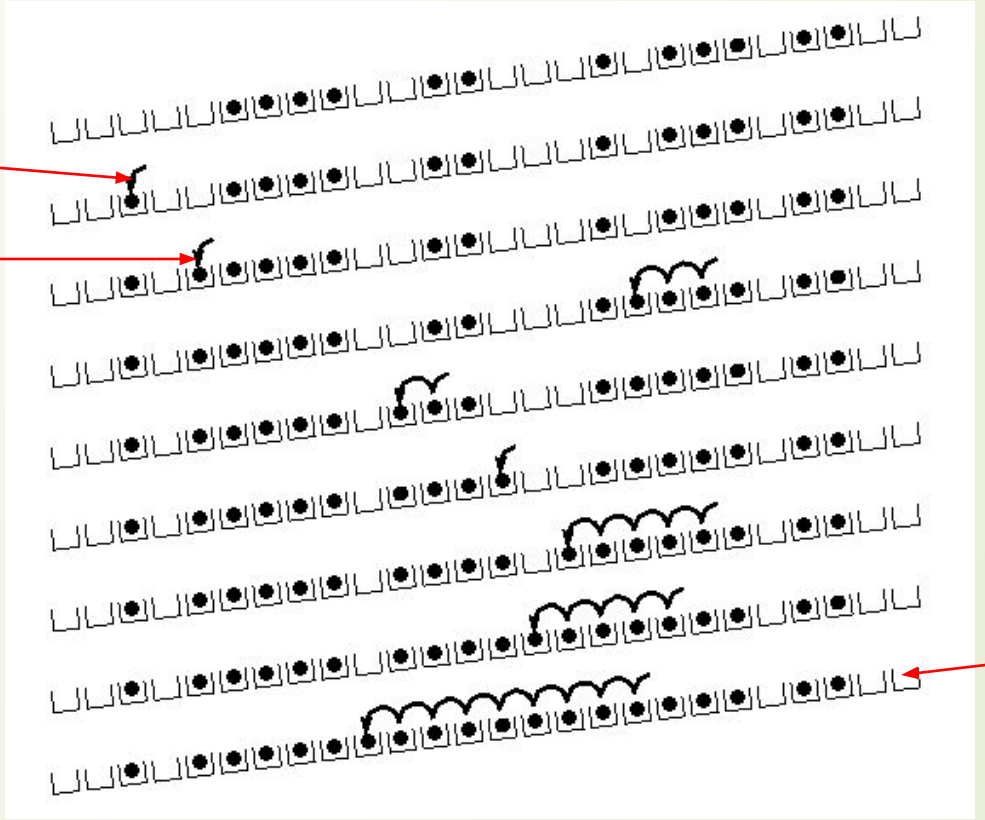
1) When inserting 6173 address is 7-6173 %7 = 7-6= 1
2) When inserting 4344 address is 7 - 4344 % 7 = 7 – 4= 3
3) When inserting 9699 address is 7-9699 % 7 = 7-5=2
4)When inserting 1889 address is 7- 1889 % 7 = 7- 1 = 6 ocuupied

Address is 6173 % 10 + 1x h2(6173) = 3 + 1= 4
Address is 4344 % 10 + 1x h2(4344) = 4 + 3= 7
Address is 9699 % 10 + 3x h2(9699) = 9 + 6 = 15 % 10 = 5
Address is 1889 % 10 + 6x h2(1889) = 15 % 10 = 5 which is already

	Initially	Insert 4371	Insert 1323	Insert 6173	Insert 4199	Insert 4344	Insert 9679	Insert 1989
0							9699	
1		4371	4371	4371	4371	4371	4371	4371
2								
3			1323	1323	1323	1323	1323	1323
4				6173	6173	6173	6173	6173
5							9679	9679
6								
7						4344	4344	4344
8								1989
9					4199	4199	4199	4199

```
/ //function for inserting element using Quadratic probing
    int insert_Quadratic_prob( int key)
    {
          int pos, i ;
          for( i=0; i< MAX ; i++)
          {
                pos = (hash(key) + i * i ) % MAX;
                if ( Hashtable[pos] == 0)
                {
                      Hashtable[i] = key;
                      break;
                }
          }
    }
 } // end of insert
```

Linear Probing – Clustering



[R. Sedgewick]

Load Factor in Linear Probing

- For *any* $\lambda < 1$, linear probing *will* find an empty slot
- Expected # of probes (for large table sizes)
  - successful search:

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$$

  - unsuccessful search:

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

- Linear probing suffers from ***primary clustering***
- Performance quickly degrades for $\lambda > 1/2$

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

Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	

Initial Empty Table

0	
1	50
2	
3	
4	
5	
6	

Insert 50

0	700
1	50
2	
3	
4	
5	
6	76

Insert 700 and 76

0	700
1	50
2	85
3	
4	
5	
6	76

Insert 85:
Collision occurs.
Insert at 1 + 1*1 position

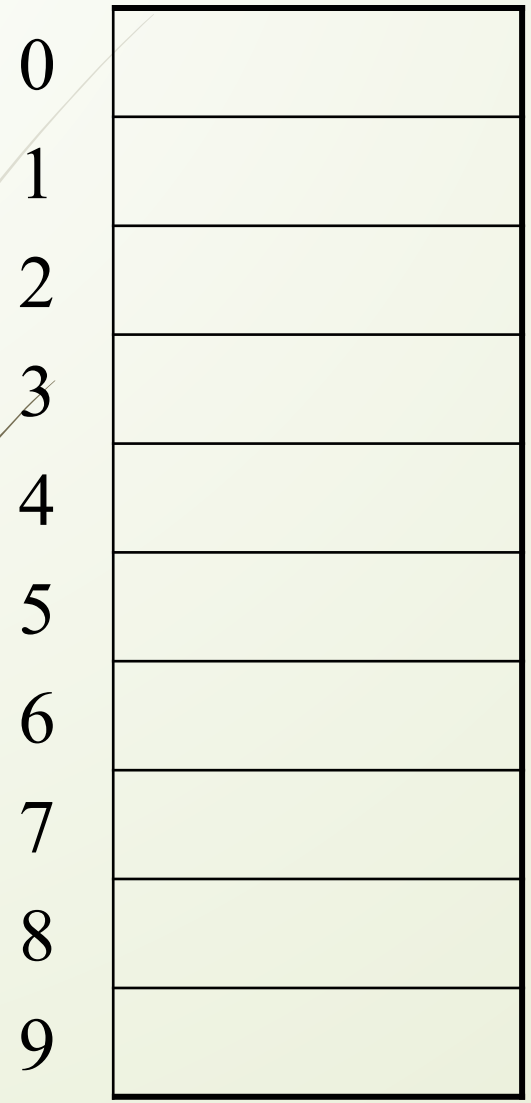
0	700
1	50
2	85
3	
4	
5	92
6	76

Insert 92:
Collision occurs at 1.
Collision occurs at 1 + 1*1 position
Insert at 1 + 2*2 position.

0	700
1	50
2	85
3	73
4	101
5	92
6	76

Insert 73 and 101

Quadratic Probing



Insert:
89
18
49
58
79

Quadratic Probing Example

insert(76)
76%7 = 6

insert(40)
40%7 = 5

insert(48)
48%7 = 6

insert(5)
5%7 = 5

insert(55)
55%7 = 6

0	48
1	
2	5
3	55
4	
5	40
6	76

But…

insert(47)
47%7 = 5

Quadratic Probing: Properties

- Quadratic probing does not suffer from *primary* clustering: keys hashing to the same *area* are not bad
- But what about keys that hash to the same *spot*?
  - ***Secondary Clustering!***

Double Hashing

$$f(i) = i * g(k)$$

where g is a second hash function

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + g(k)) mod TableSize

  2^{th} probe = (h(k) + 2*g(k)) mod TableSize

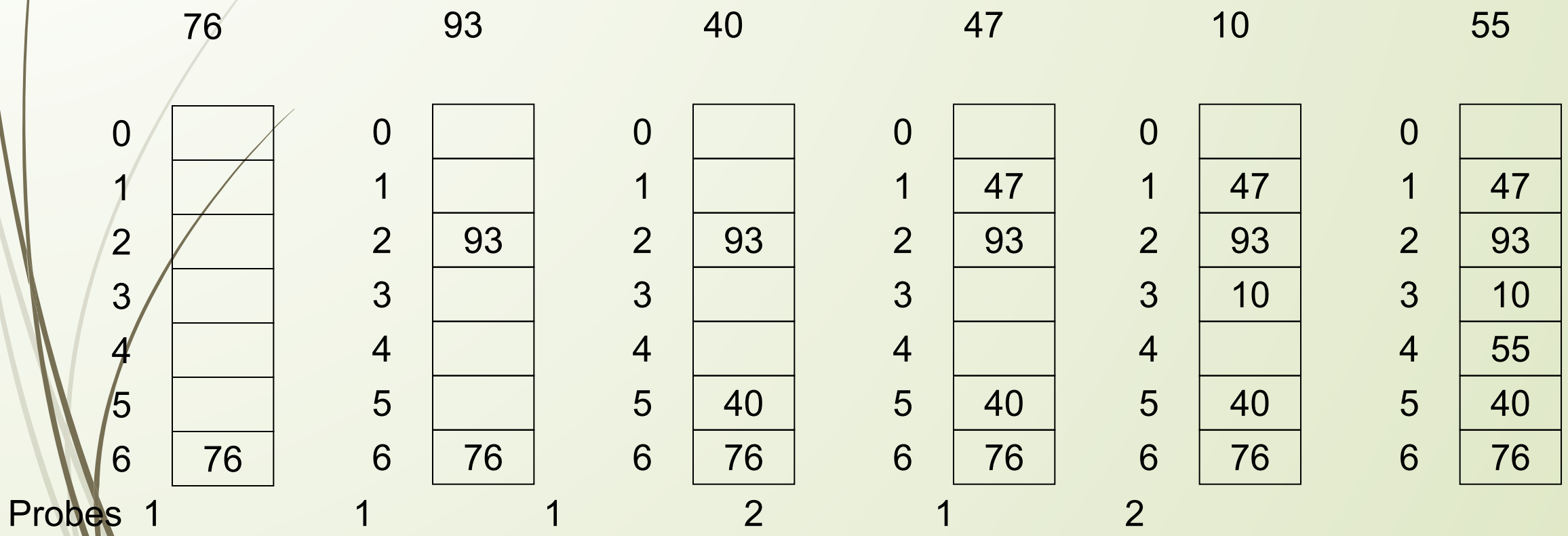
  3^{th} probe = (h(k) + 3*g(k)) mod TableSize

  . . .

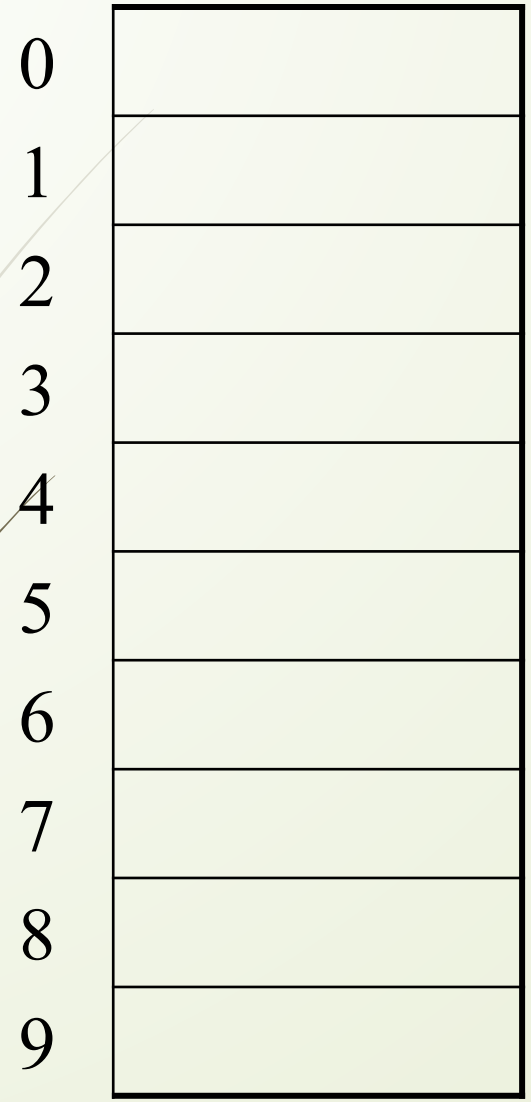
  i^{th} probe = (h(k) + i*g(k)) mod TableSize

Double Hashing Example

h(k) = k mod 7 and g(k) = 5 – (k mod 5)



Resolving Collisions with Double Hashing



Hash Functions:

$H(K) = K \bmod M$

$H_2(K) = 1 + ((K/M) \bmod (M-1))$

$M =$

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

13

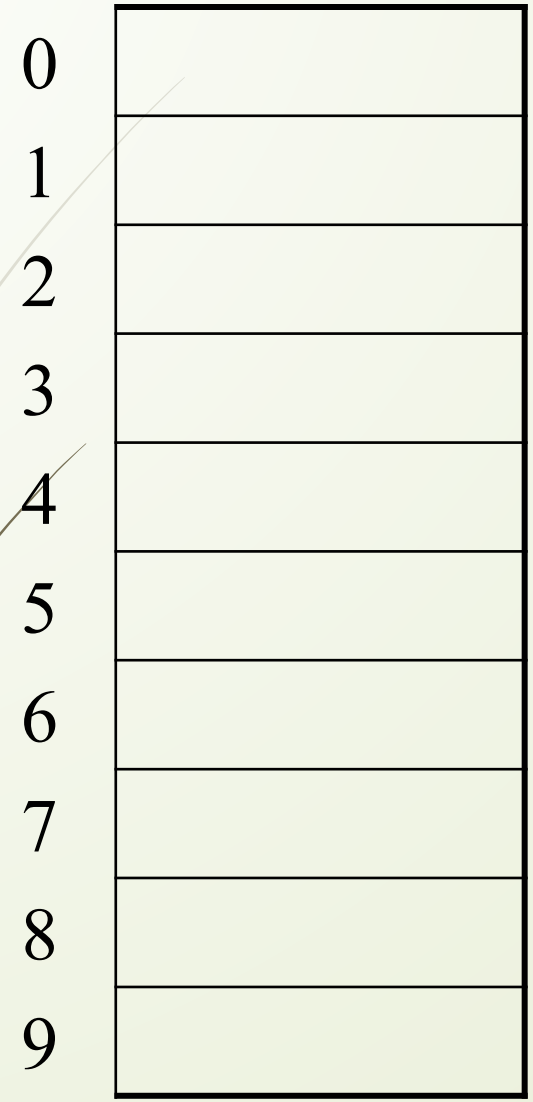
28

33

147

43

Separate Chaining



Insert:
10
22
107
12
42

- **Separate chaining**: All keys that map to the same hash value are kept in a list (or “bucket”).

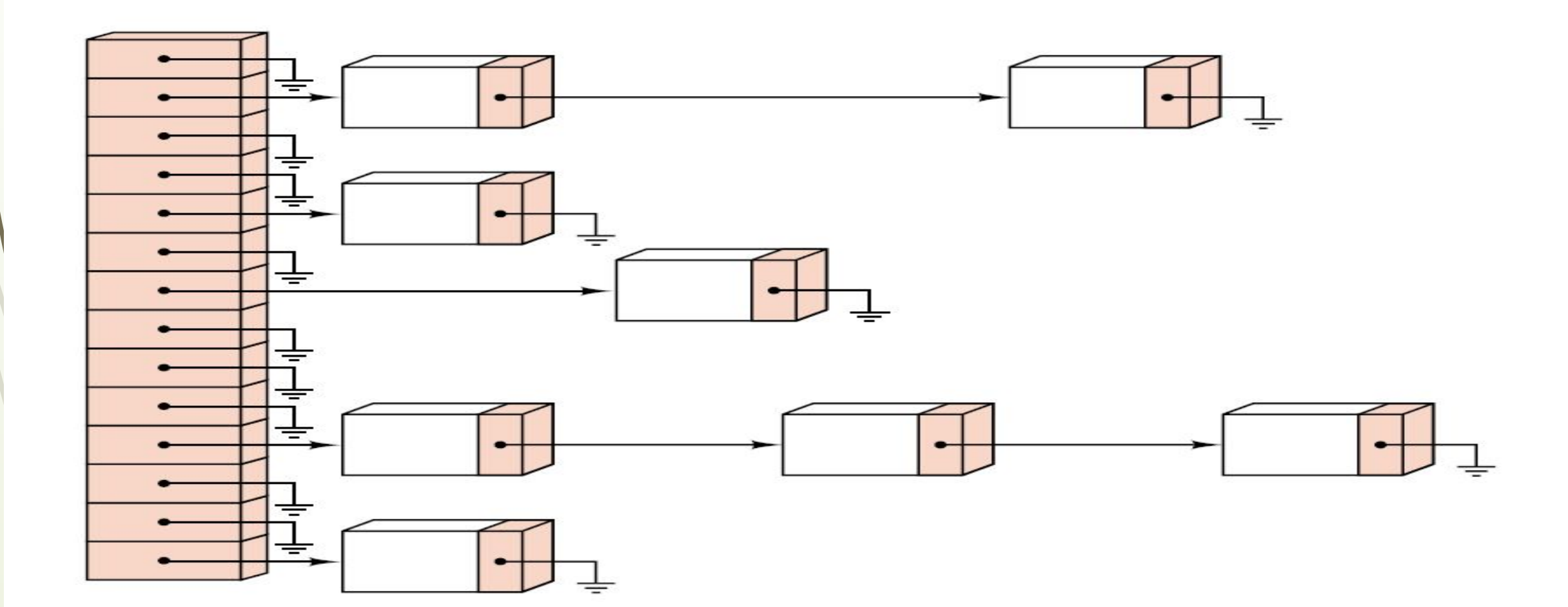
Data Structure for Chaining

```
#define TABLE_SIZE 13

class HT {
    int key;
    HT *link;
};
HT hash_table[TABLE_SIZE];
```

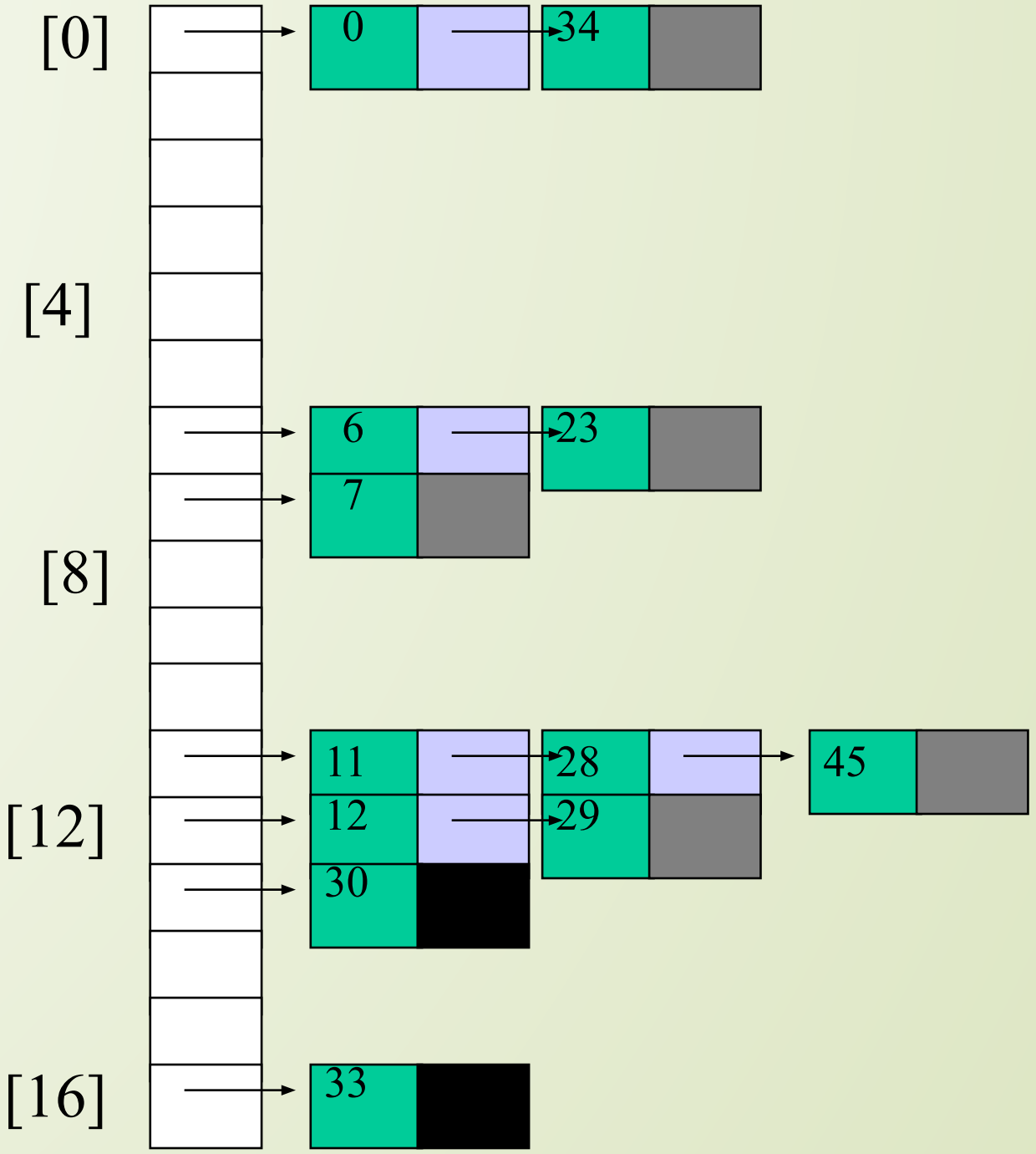
The idea of **Chaining** is to combine the linked list and hash table to solve the overflow problem.

Figure of Chaining



Sorted Chains

- Put in pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45
- Bucket = key % 17.



Analysis of find

- Defn: The load factor, λ, of a hash table is the ratio:
  ← no. of elements
  ← table size

$$\frac{N}{M}$$

For separate chaining, λ = average # of elements in a bucket

- Unsuccessful find:

- Successful find:

University Question

Q. 3

a. Construct hash table of size 10 using linear probing without replacement strategy for collision resolution. The hash function is h (x) = x % 10. Consider slot per bucket is 1.

31, 3, 4, 21, 61, 6, 71, 8, 9, 25 [6]

b. Explain about a skip list with an example. Give applications of skip list [6]

P.T.O.

Nov 2017

3. (a) Obtain AVL trees fro the following data : [6]

30, 50, 110, 80, 40, 10, 120, 60, 20, 70, 100, 90

(b) For the given set of values. [6]

11, 33, 20, 88, 79, 98, 44, 68, 66, 22

Create a hash table with size 10 and resolve collision using chaining with replacement and without replacement. Use the modulus Hash function. (key % size.)

MCQ

1.A technique for direct search is

a) Binary Search b) Linear Search c) Tree Search d) Hashing

2.The searching technique that takes O (1) time to find a data is

a) Linear Search b) Binary Search c) Hashing d) Tree Search

3.Which of the following is not a method of Hashing Function?

A. Modulo-Division method,
B. Digit-Extraction method,
C. Mid-Square method
D. Linear probing

MCQ

What is the best definition of a collision in a hash table?

A. Two entries are identical except for their keys.

B. Two entries with different data have the exact same key.

C. Two entries with different keys have the same exact hash value.

D. Two entries with the exact same key have different hash values.

MCQ

A hash table of length 10 uses open addressing with hash function h(k)=k mod 10, and linear probing. After inserting 6 values into an empty hash table, the table is as shown below.(GATE)

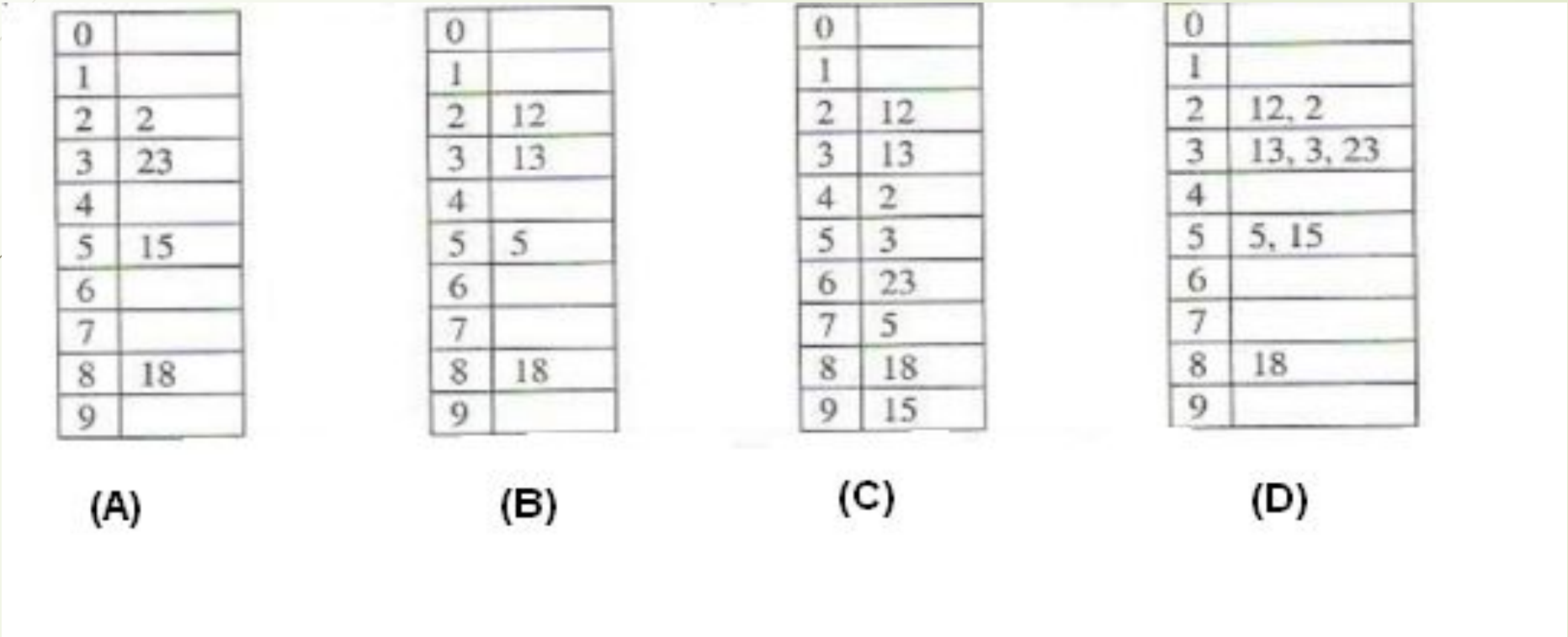
0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

Which one of the following choices gives a possible order in which the key values could have been inserted in the table?

(A) 46, 42, 34, 52, 23, 33
(B) 34, 42, 23, 52, 33, 46
(C) 46, 34, 42, 23, 52, 33
(D) 42, 46, 33, 23, 34, 52

MCQ

The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using open addressing with hash function h(k) = k mod 10 and linear probing. What is the resultant hash table?



0	
1	
2	2
3	23
4	
5	15
6	
7	
8	18
9	

(A)

0	
1	
2	12
3	13
4	
5	5
6	
7	
8	18
9	

(B)

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

(C)

0	
1	
2	12, 2
3	13, 3, 23
4	
5	5, 15
6	
7	
8	18
9	

(D)

(A) A
(B) B
(C) C
(D) D

MCQ

Consider a hash table of size seven, with starting index zero, and a hash function (3x + 4)mod7. Assuming the hash table is initially empty, which of the following is the contents of the table when the sequence 1, 3, 8, 10 is inserted into the table using closed hashing? Note that '_' denotes an empty location in the table.(GATE)

(A) 8, _, _, _, _, _, 10

(B) 1, 8, 10, _, _, _, 3

(C) 1,10, _, _, _, 8,3

(D) 1, 10, 8, _, _, _, 3

Rehashing

Idea: 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?
  - half full ($\lambda = 0.5$)
  - when an insertion fails
  - some other threshold
- Cost of rehashing?

Rehashing:

As the name suggests, rehashing means hashing again. Basically, when the load factor increases to more than its pre-defined value (default value of load factor is 0.75), the complexity increases.

So to overcome this, the size of the array is increased (doubled) and all the values are hashed again and stored in the new double sized array to maintain a low load factor and low complexity.

Why 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 as explained above. This might not give the required time complexity of O(1).

Hence, rehash must be done, increasing the size of the bucketArray so as to reduce the load factor and the time complexity.

How Rehashing is done?

Rehashing can be done as follows:

- For each addition of a new entry to the map, check the load factor.
- If it's greater than its pre-defined value (or default value of 0.75 if not given), then Rehash.
- For Rehash, make a new array of double the previous size and make it the new bucketarray.
- Then traverse to each element in the old bucketArray and call the insert() for each so as to insert it into the new larger bucket array.

Hashing Summary

- Hashing is one of the most important data structures.
- Hashing has many applications where operations are limited to find, insert, and delete.
- Dynamic hash tables have good amortized complexity.

Conclusion

- The main **tradeoffs** between these methods are that **linear probing** has the best cache performance but is most sensitive to clustering, while **double hashing** has poorer cache performance but exhibits virtually no clustering; **quadratic probing** falls in between the previous two methods.

Extendible hashing :

- If linear probing or separate chaining is used for collision handling then in case of collision several blocks are required to be examined to search a key and when table gets full then expensive rehash should be used.
- For fast searching and less disk access, extensible hashing is used.It is a type of hash system which treats a hash as a bit string, and uses a trie for bucket lookup.

  For example
- Assume that the hash function hash(key) returns a binary number.

  The first i bits of each string will be used as indices to figure out where they will go in the hash table. Additionally, i is the smallest number such that the first i bits of

  all keys are different.

Extendible hashing :

Keys to be used:
h(key1) = 100101
h(key2) = 011110
h(key3) = 110110
Let's assume that for this particular example, the bucket size is 1. The first two keys to be inserted, k1 and k2, can be distinguished by the most significant bit, and would be inserted into the table as follows:Directory

0	Bucket A for Key2
1	Bucket B for Key1

When key3 is hashed to the table, it wouldn't be enough to distinguish all three keys by one bit (because key3 and key1 have 1 as their leftmost bit). Also, because the bucket size is one, the table would overflow. Because comparing the first two most significant bits would give each key a unique location, the directory size is doubled as follows that is it is spited :
Directory

00	Bucket A for Key2
01	
10	Bucket B for Key1
11	Bucket C for Key3

Extendible Hashing

- And so now key1 and key3 have a unique location,being distinguished by the first two leftmost bits. Because k2 is in the top half of the table, both 00 and 01 point to it because there is no other key to compare to that begins with a 0.
- The root of the tree contains four pointers determined by the leading two bits of data. Each leaf has upto 4 elements. D will be represented by the number of bits used by the root, which is known as **Directory**.

Dictionary Implementations So Far

		Unsorted linked list	Sorted Array	Hash Table	BST	AVL
Insert	Best case Average case Worst case					
Find	Best case Average case Worst case					
Delete	Best case Average case Worst case					

Thank You