Hashing

- Introduction to hashing
- Hash functions
- Collision resolution strategies- Open Addressing and Chaining
- Hash Table Overflow

Introduction to Hashing

Suppose that we want to store 10,000 students records (each with a 5-digit ID) in a given container.

- ❑ A linked list implementation would take **O(n)** time.
- ❑ A height balanced tree would give **O(log n)** access time.*(will see in Unit-4)*
- ❑ Using an array of size 100,000 would give **O(1)** access time but will lead to a lot of space wastage.

Is there some way that we could get **O(1)** access without wasting a lot of space?

The answer is hashing.

Introduction to Hashing

❑Why Hashing?

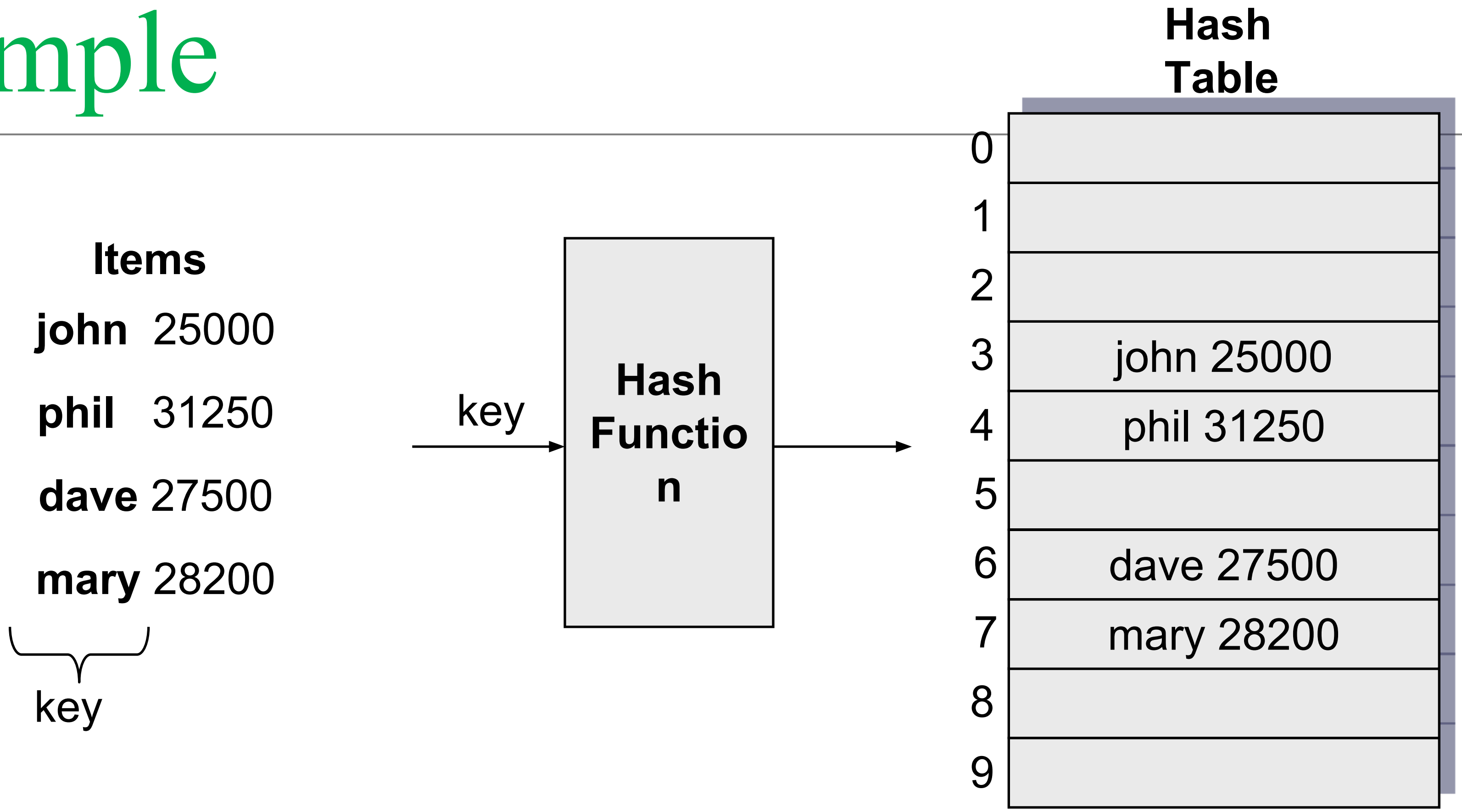
- Sequential Search requires O(n) Comparisons.
- It is not useful for large database.
- Binary Search requires less comparisons O(nlogn)
- But it requires data to be sorted.

Introduction to Hashing

❑What is Hashing?

- Record for the key value is directly referred by calculating address from key value
- Address of element x is obtained by computing arithmetic function f(x).
- Function f(x) is called as hash function
- Table used for storing records is known as hash table
- Best Case Time Complexity of hashing=O(1)
- Worst Case Time Complexity =O(n)

Example



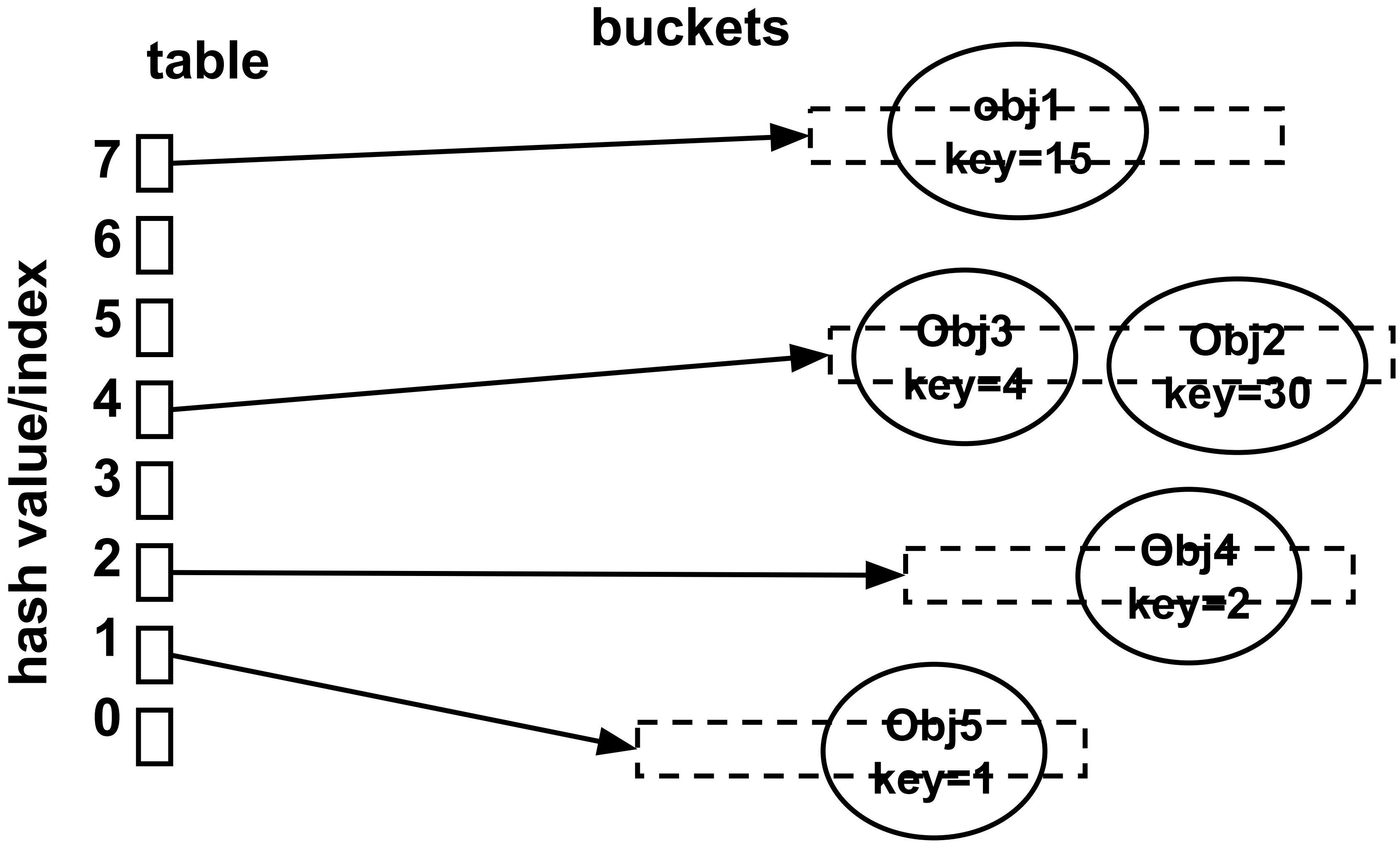
Items

john	25000
phil	31250
dave	27500
mary	28200

Hash Table

0	
1	
2	
3	john 25000
4	phil 31250
5	
6	dave 27500
7	mary 28200
8	
9	

Hash Tables – Conceptual View



Key terms

- **Hash Table**: is an array of size MAX[0 to MAX-1]
- **Hash Function**: that transforms a key into an address.
- **Bucket-** Bucket is an index position in hash table that can store more than one record
- When the same index is mapped with two keys, then both the records are stored in the same bucket
- **Probe**-Each action of address calculation and check for success is called probe
- **Collision-**The result of two keys hashing into the same address is called collision
- **Synonym-**Keys those hash to the same address are called synonyms

Key terms

- **Overflow-**The result of more keys hashing to the same address and if there is no room in the bucket, then it is said that overflow has occurred
- Collision and overflow are synonymous when the bucket is of size 1
- **Open or external hashing-**When we allow records to be stored in potentially unlimited space, it is called as open or external hashing
- **Closed or internal hashing**-When we use fixed space for storage eventually limiting the number of records to be stored, it is called as closed or internal hashing

Key Terms

Load density- The maximum storage capacity that is maximum number of records that can be accommodated is called as loading density

Full Table- Full table is the one in which all locations are occupied

Owing to the characteristics of hash functions, there are always empty locations

Load factor- the number of records stored in table divided by maximum capacity of table, expressed in terms of percentage

Hashing

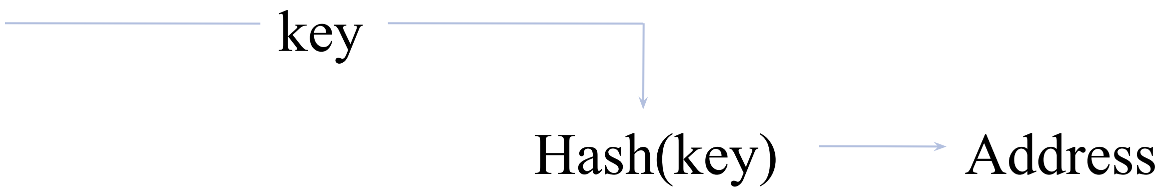
- Example

31,42,35,67,24,19

F(x)=x mod 10

Location	X
1	31
2	42
3	
4	24
5	35
6	
7	67
8	
9	19

X	F(X)
31	1
42	2
35	5
67	7
24	4
19	9



- The resulting address is used as the basis for storing and retrieving records and this address is called as *home address of the record*

- *For array to store a record in a hash table, hash function is* applied to the key of the record being stored, returning an index within the range of the hash table

- The item is then stored in the table of that index position

Basic Idea

Use *hash function* to map keys into positions in a *hash table*

Ideally

If element *e* has key *k* and *h* is hash function, then *e* is stored in position *h(k)* of table

To search for *e*, compute *h(k)* to locate position. If no element, dictionary does not contain *e*.

Hash Functions

A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table.

The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

Example:

H(x)=x mod 10

Here, *H(x) is called as hash function while value returned by function is called as hash value.*

Characteristics of a good hashing function

- The average performance of hashing depends on how the hash function distributes the set of keys among the slots

- Assumption is that any given record is equally likely to hash into any of the slots, independently of whether any other record has been already hashed to it or not

- This assumption is called as simple uniform hashing

- A good hash function is the one which satisfies the assumption of simple uniform hashing

Characteristics of a good hashing function

- Addresses generated from the key are uniformly and randomly distributed

- Small variations in the value of key will cause large variations in the record addresses to distribute records (with similar keys) evenly

- The hashing function must minimize the collision

Division Method

One of the required features of the hash function is that the resultant index must be within the table index range

One simple choice for a hash function is to use the modulus division indicated as MOD

The function returns an integer

If any parameter is NULL, the result is NULL

Hash(Key) = Key % M

Hash Tables: Insert Example

For example, if we hash keys 0…1000 into a hash table with 5 entries and use **h(key) = key mod 5** , we get the following sequence of events:

Insert 2

	key	data
0		
1		
2	2	…
3		
4		

Insert 21

	key	data
0		
1	21	…
2	2	…
3		
4		

Insert 34

	key	data
0		
1	21	…
2	2	…
3		
4	34	…

Insert 54

There is a ***collision*** at array entry #4

???

Multiplication Method

The multiplication method works as:

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

$$Hash(k) = M (kA),$$

Donald Knuth suggested to use A=0.61803398987

Ex: key=107, assume M=50 key=123 m=1000

$$h(k)= M*107*0.61803398987$$
$$= 66.12$$

fractional part=0.12

$$h(k)=50*0.12$$
$$=6$$
$$h(k)=6$$

That means 107 will be placed at index 6 in hash table.

Multiplication Method

Disadvantage: Slower than the division method.

Advantage: Value of *m* is not critical.

- Typically chosen as a power of 2, i.e., $m = 2^p$, which makes implementation easy.

Example: $m = 1000$, $k = 123$, $A \approx 0.6180339887\ldots$

$$h(k) = \lfloor 1000(123 * 0.6180339887\ ) \rfloor$$

$$= \lfloor 1000\ (\ 76.018\ldots\ \rfloor\ .$$

$$= \lfloor 1000\ (\ .018\ldots\ ) \rfloor = 18$$

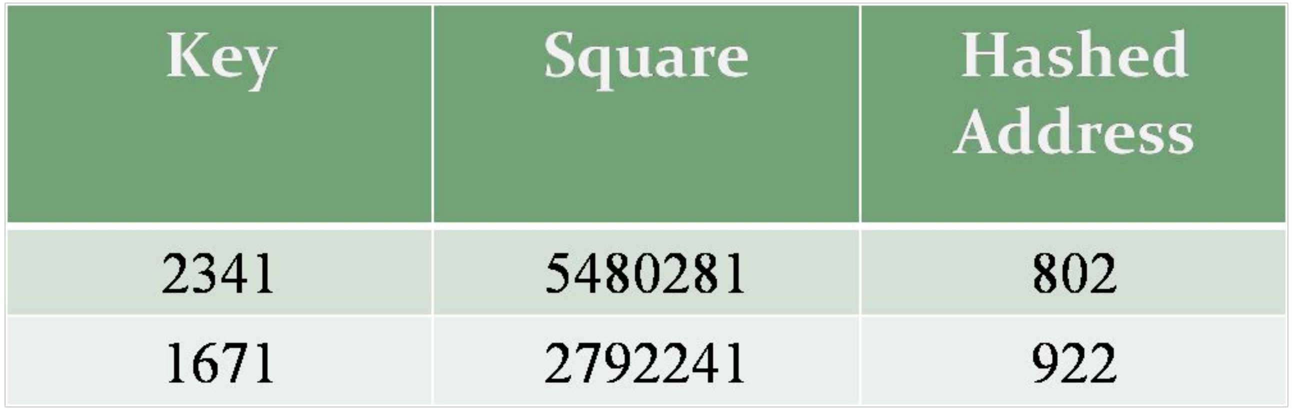
Mid-Square Hashing

- The mid-square hashing suggests to take square of the key and extract the middle digits of the squared key as address

- The difficulty is when the key is large. As the entire key participates in the address calculation, if the key is large, then it is very difficult to store the square of it as the square of key should not exceed the storage limit

- So mid-square is used when the key size is less than or equal to 4 digits

Keys and addresses using mid-square



Key	Square	Hashed Address
2341	5480281	802
1671	2792241	922

The difficulty of storing larger numbers square can be overcome if for squaring we use few of digits of key instead of the whole key

Keys and addresses using mid-square

We can select a portion of key if key is larger in size and then square the portion of it

Key	Square	Hashed Address
234137	234*234=54756	475
567187	567*567=321489	148

Folding Technique

- In folding technique, the key is subdivided into subparts that are combined or folded and then combined to form the address

- For the key with digits, we can subdivide the digits in three parts, add them up, and use the result as an address.

- Here the size of subparts of key could be as that of the address

Folding Technique (contd…)

- There are two types of folding methods:
- **Fold shift** — Key value is divided into several parts of that of the size of the address. Left, right, and middle parts are added
- **Fold boundary** — Key value is divided into parts of that of the size of the address. Left and right parts are folded on fixed boundary between them and the centre part

Folding Technique

- For example, if the key is 987654321, it is understood as Left 987 Centre 654 Right 321
- For fold shift, addition is
- 987 + 654 + 321 = 1962

- Now discard digit 1 and the address is 962
- For fold boundary, addition of reverse part is
- 789 + 456 + 123 = 1368

- Discard digit 1 and the address is 368

Extraction Method

- When a portion of the key is used for the address calculation, the technique is called as the extraction method

- In digit extraction, few digits are selected and extracted from the key which are used as the address

Keys and addresses using digit extraction

Key	Hashed Address
345678	357
234137	243
952671	927

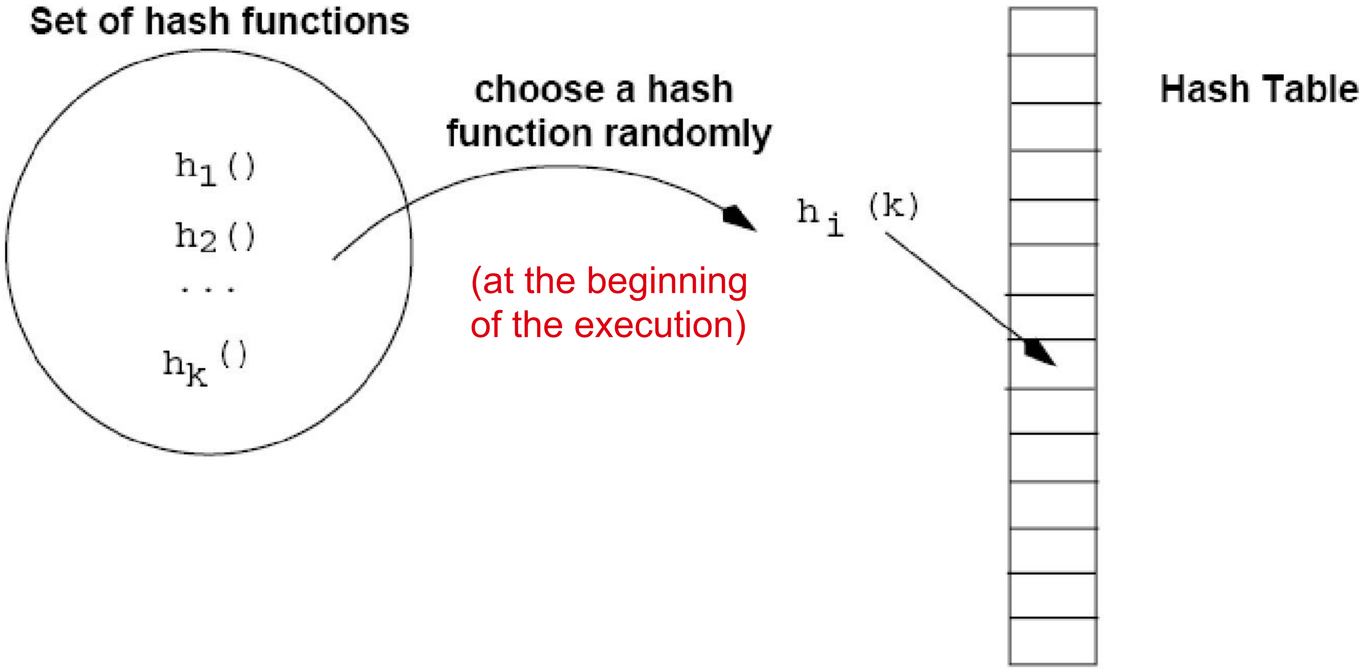
Universal Hashing

- The main idea behind universal hashing is to select the hash function at random at run time from a carefully designed set 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

Universal Hashing



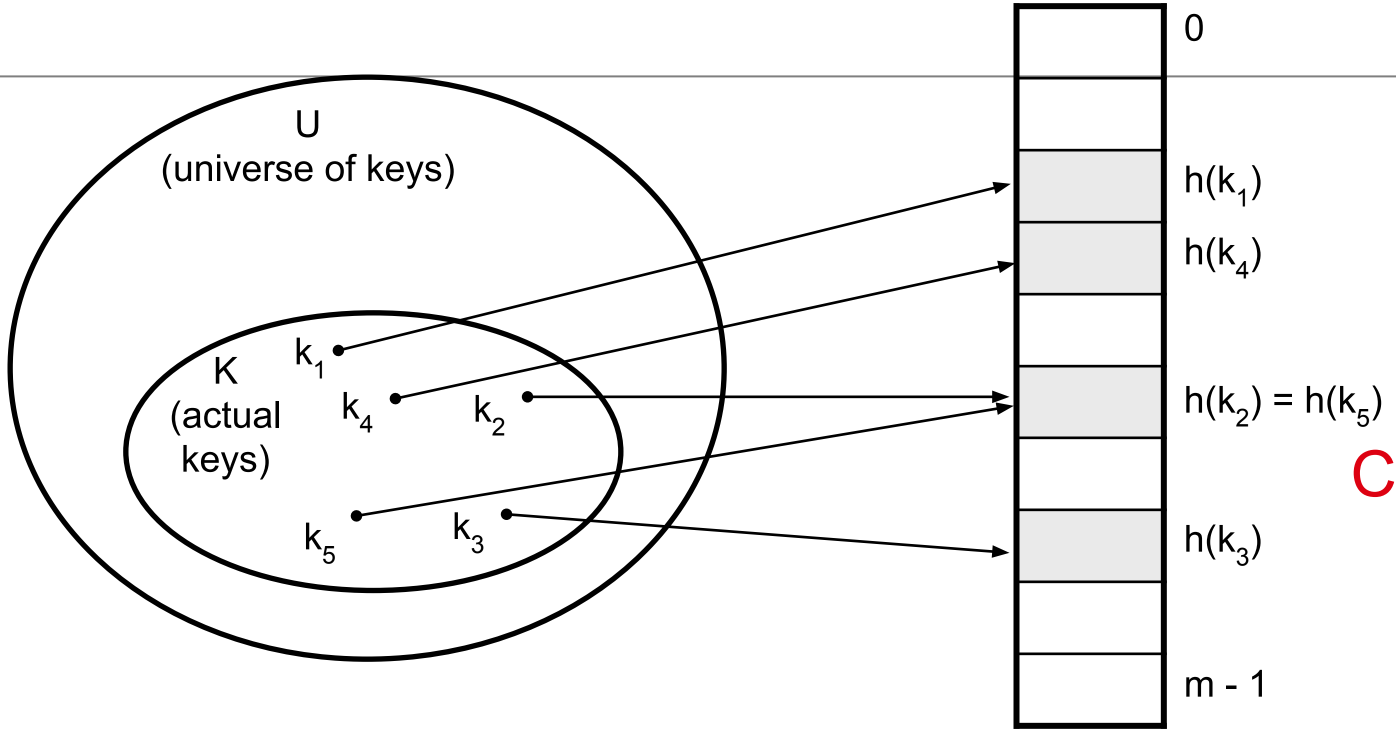
(at the beginning of the execution)

Issues

- A problem arises, however, when the hash function returns the same value when applied to two different keys

- To handle the situation, where two records need to be hashed to the same address we can implement a table structure, so as to have a room for two or more members at the same index positions

Do you see any problems with this approach?



Collisions!

Collision Resolution Strategies

Collision-Element to be inserted is mapped to same location

Example:

31,42,35,67,24,19,22

F(x)=x mod 10

Where to store 22?

Location	X
1	31
2	42
3	
4	24
5	35
6	
7	67
8	
9	19

Collision Resolution Strategies

Collision-Element to be inserted is mapped to same location

Example:

31,42,35,67,24,19,22

F(x)=x mod 10

Where to store 22?

Location	X
1	31
2	42
3	22
4	24
5	35
6	
7	67
8	
9	19

Collision Resolution Strategies

1. Open Addressing
    1. Linear probing
    2. Quadratic probing
    3. Double hashing, and
    4. Key offset
2. Separate chaining (or linked list)
3. Bucket hashing (defers collision but does not prevent it)

Open Addressing

When collision occurs, it is resolved by finding an available empty location other than the home address.

If Hash(key) is not empty, the positions are probed in the following sequence until an empty location is found.

When we reach the end of the table ,the search is wrapped around to start and search continues till the current collision location.

Closed hash tables use open addressing.

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 it is found
- The function that we can use for probing linearly from the next location is as follows:
- (Hash(x) + p(i)) MOD Max
- As p(i) = i for linear probing, the function becomes
- (Hash(x)+ i)) MOD Max
- Initially i = 1, if the location is not empty then it becomes 2, 3, 4, …, and so on till empty location is found.

Example Linear (Closed) Hashing

D=8, keys *a,b,c,d* have hash values h(a)=3, h(b)=0, h(c)=4, h(d)=3

- Where do we insert *d*? 3 already filled
- Probe sequence using linear hashing:

  $h_1(d) = (h(d)+1)\%8 = 4\%8 = 4$

  $h_2(d) = (h(d)+2)\%8 = 5\%8 =$ **5***

  $h_3(d) = (h(d)+3)\%8 = 6\%8 = 6$

  etc.

  7, 0, 1, 2
- Wraps around the beginning of the table!

0	b
1	
2	
3	a
4	c
5	**d**
6	
7	

Linear Probing

- 1.With replacement 2.Without replacement

- **With replacement :**
- If the slot is already occupied by the key there are two possibilities, that is, either it is home address (collision) or not key’s home address

- If the key’s actual address is different, then the new key having the address at that slot is placed at that position and the key with other address is placed in the next empty position

Linear Probing

Example :

Given the input {4371, 1323, 6173, 4199, 4344, 9699, 1889} and hash function as Key % 10, show the results for the following:

1. Open addressing using linear probing
2. Open addressing using quadratic probing
3. Open addressing using double hashing h2 (x) = 7−(x MOD 7)

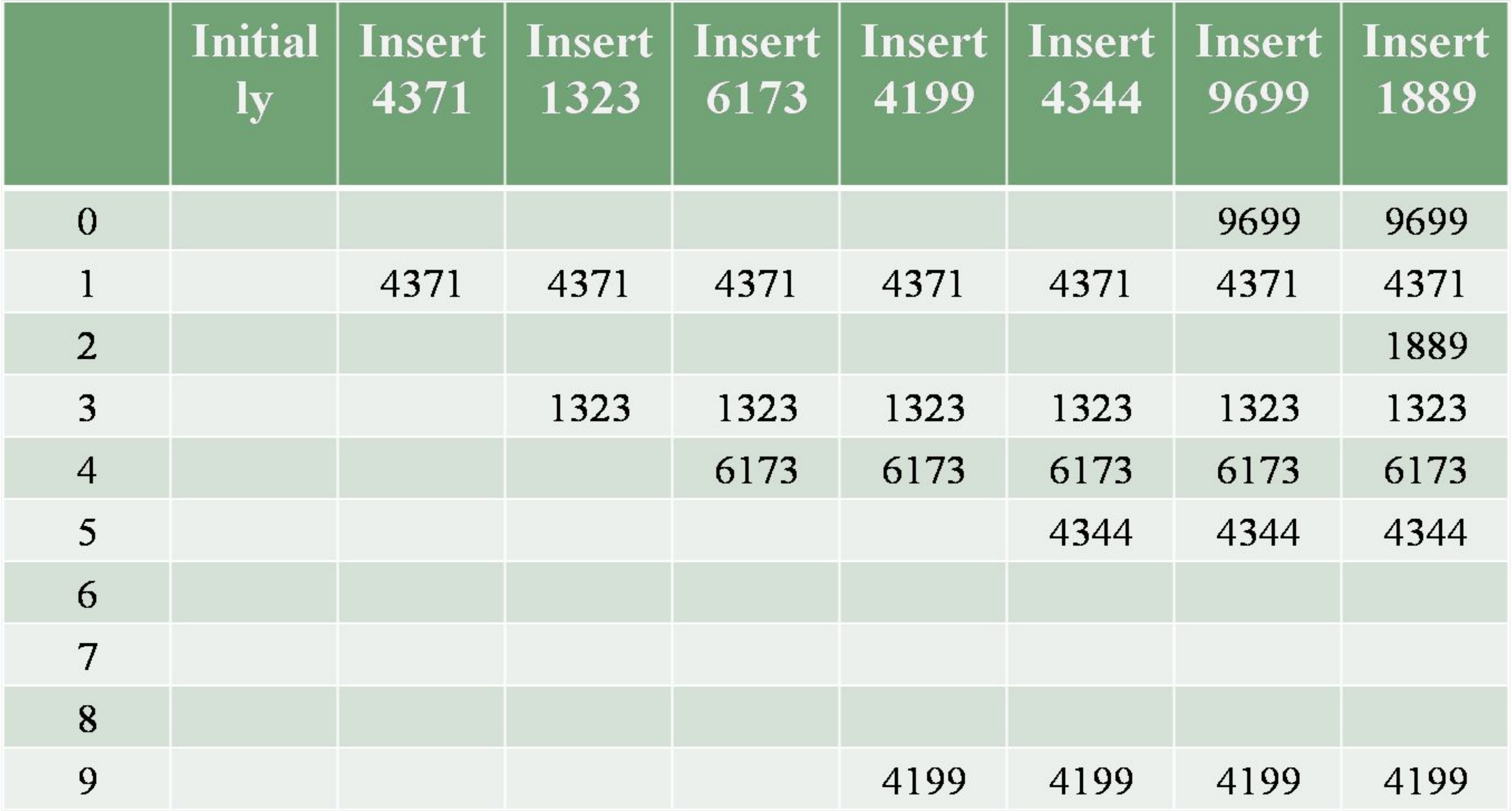
Open addressing using linear probing with replacement

	Initially	Insert 4371	Insert 1323	Insert 6173	Insert 4199	Insert 4344	Insert 9699	Insert 1889
0							9699	9699
1		4371	4371	4371	4371	4371	4371	4371
2								1889
3			1323	1323	1323	1323	1323	1323
4				6173	6173	4344	4344	4344
5						6173	6173	6173
6								
7								
8								
9					4199	4199	4199	4199

Linear Probing

- **Without replacement :**
- When some data is to be stored in hash table, and if the slot is already occupied by the key then another empty location is searched for a new record
- There are two possibilities when location is occupied—it is its home address or not key’s home address.
- In both the cases, the without replacement strategy empty position is searched for the key that is to be stored

Open addressing using linear probing without replacement

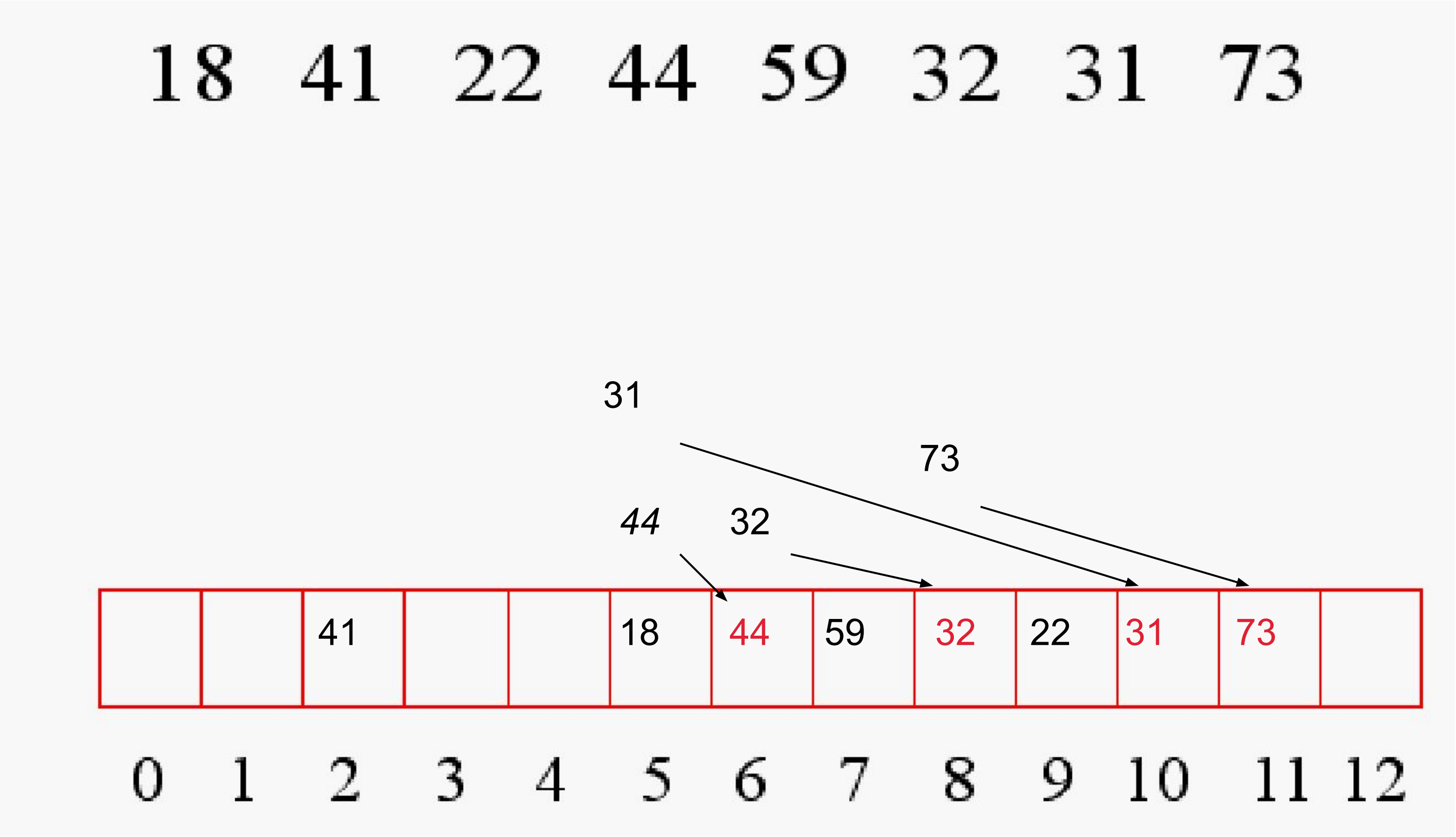


	Initially	Insert 4371	Insert 1323	Insert 6173	Insert 4199	Insert 4344	Insert 9699	Insert 1889
0							9699	9699
1		4371	4371	4371	4371	4371	4371	4371
2								1889
3			1323	1323	1323	1323	1323	1323
4				6173	6173	6173	6173	6173
5						4344	4344	4344
6								
7								
8								
9					4199	4199	4199	4199

Linear Probing without replacement Example

h(k) = k mod 13

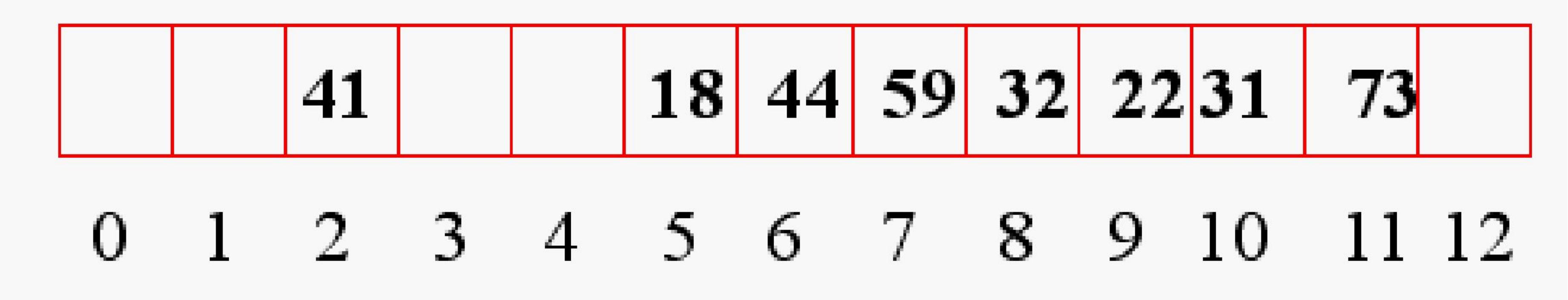
Insert keys:



18 41 22 44 59 32 31 73

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

Linear Probing without replacement Example(cont.)



		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

Linear probing (chaining)with and without replacement

12,01,04,03,07,08,10,02,05,14

Assume buckets from 0 to 9 and each bucket has 1 slot.

Open addressing using linear probing with replacement

	Initially	12	01	04	03	07	08	10	02	05	14
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	05	05
6										02	02
7						07	07	07	07	07	07
8							08	08	08	08	08
9											14

Open addressing using linear probing without replacement

	Initially	12	01	04	03	07	08	10	02	05	14
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

Linear Probing without replacement Example

In Linear Probing, cells are searched sequentially for an empty cell.

31,42,35,67,24,19,22

F(x)=x mod 10

Location	X
1	31
2	42
3	22
4	24
5	35
6	
7	67
8	
9	19

Functions for Linear Probing without replacement insert function

```
Algorithm int insert_linear_prob(int hashtable[],int key)
{
      loc=key % MAX;
      If(hashTable[loc]==-1)
        {
              hashtable[loc]=key;
              return(loc);
        }
      else{i=(loc+1)%MAX;
while(i!=loc)
{
if(hashTable[i]==-1)
        {
              hashtable[i]=key;
              return(i);    }
i=(i+1)%MAX;
}
if(i==loc) print("Hash is full")
}
```

Functions for Linear Probing without replacement- search function

```
Search_linear_probe(hashtable,key)
{
    int i,j;
    loc=key % MAX;
    if(hashtable[loc]==key) print “key found”
    else{ i=(loc+1)%MAX;
   while(i!=loc){
      if(hashtable[i]==key) {print “key found” break;}
       i=(i+1)%MAX;}
       }
     if(i==loc) print(“key not found”)
}
```

Functions for Linear Probing with replacement insert function

```
Algorithm int insert_linear_prob(int hashtable[],int key)
{
    loc=key % MAX;
    If(hashTable[loc]==-1)
      {
           hashtable[loc]=key;
           return(loc);
      }
    else{
temp=key
if(loc!=(hashtable[loc]%MAX){
temp=hashtable[loc];
Hashtable[loc]=key;}
i=(loc+1)%MAX;
while(i!=loc)
{
if(hashTable[i]==-1)
      {
           hashtable[i]=temp;
           return(i);      }
i=(i+1)%MAX;
}

if(i==loc) print("Hash is full")
}
```

Functions for Linear Probing with replacement- search function

```
Search_linear_probe(hashtable,key)
{
       int i,loc;
       loc=key % MAX;
       if(hashtable[loc]==key) print “key found”
       else{ i=(loc+1)%MAX;
      while(i!=loc){
          if(hashtable[i]==key) {print “key found” break;}
           i=(i+1)%MAX;}
           }
         if(i==loc) print(“key not found”)
}
```

Clustering

One problem with the above technique is the tendency to form “clusters”

A cluster is a group of items not containing any open slots

The bigger a cluster gets, the more likely it is that new values will hash into the cluster, and make it ever bigger

Clusters cause efficiency to degrade

Quadratic Probing

- In quadratic probing, we add offset by amount square of collision probe number
- In quadratic probing, the empty location is searched using the following formula
- (Hash(Key) + i^2) MOD Max where i lies between 1 to (Max − 1)/2
- Quadratic probing probes the array at location Hash(key)+1, Hash(key)+4, ______,Hash(key)+9 etc.
- 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 so that the address lies in table boundaries*

Open addressing using quadratic probing

- Let us insert these keys using quadratic probing
- For 6173, the hashed address 6173 % 10 gives 3 and it is not empty, hence using quadratic probing we get the address as follows:

  Hash(6173) = (6173 + 1^2) % 10 = 4 and as it is empty, the key 6173 is stored there
- Now while inserting 4344, the location 4 is not empty and hence quadratic probing generates the address as (4344 + 1^2) % 10 = 5 and as is empty 4344 is stored
- For key 9699, the address is (9699 + 1^2) % 10 = 0 and is empty so store.
- While inserting 1889, the address (1889 + 1^2) % 10 = 0 is not empty so probe again
- The address (1889 + 2^2) % 10 = 3 is not empty so probe again.
- The address(1889 + 3^2) % 10 = 8 is empty so store 1889 at location 8

Open addressing using quadratic probing

	Initially	Insert 4371	Insert 1323	Insert 6173	Insert 4199	Insert 4344	Insert 9699	Insert 1889
0							**9699**	**9699**
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								**1889**
9					**4199**	**4199**	**4199**	**4199**

Quadratic Probing

If cell j is already occupied then location j+1,j+4,j+9 are checked.

This reduces primary clustering of data.

Main Idea: Spread out the search for an empty slot – **Increment by i^2 instead of i**

hi(X) = (Hash(X) + i2) % TableSize

h0(X) = Hash(X) % TableSize

h1(X) = Hash(X) + 1 % TableSize

h2(X) = Hash(X) + 4 % TableSize

h3(X) = Hash(X) + 9 % TableSize

Let the sequence of **keys** = 9 ,19 ,29 ,39 ,49 ,59,71 These keys are to be inserted into the hash table.

The hash function for indexing,=$Kmod$10, where k = key value.

index	keys
0	19
1	71
2	
3	29
4	59
5	49
6	
7	
8	39
9	9

Quadratic Probing Operation

```
Algorithm Insert(int hashtable[],int key)
{
    start=key % MAX;
    j=start
    for(i=0;i<MAX,i++)
    {
        If(hashtable[j]==-1)add key
        j=(start+i*i)%MAX;
    }
 }
```

Double Hashing

- Double hashing uses two hash functions, one for accessing the home address of a Key and the other for resolving the conflict. The sequence for probing is generated in the following sequence:

- The formula to be used for double hashing is

H1(key)= key % tablesize

H2(key)= M-(key % M)

where M is the prime no. smaller than the size of the table.

The formula is : H1(key)+ i*H2(key)

i=0,1,2….

- The resultant address is divided by modulo Max

Open addressing using double hash function

- While inserting 6173, the address is Hash1(6173) = 6173 % 10 = 3 and 3 is not empty
- Let us use double hashing. Hence the address is as follows:
- Hash(6173) = [Hash1(6173) + Hash2(6173)] % 10= 3 + (R − 6173 % R) (let R be 7)= 3+ (7 − 6) = 4 Since 4 is empty, we store 6173 at location 4
- Now let us store 4344. The address 4344 % 10 = 4 and as location 4 is not empty, we use double hashing and we get Hash(4344) = 7
- Now for 9699 double hashing generates address 2 and as it is empty, we store it there.
- For key 1889, double hashing generates address 0 and as it is empty, we store 1889 at location 0

Open addressing using double hash

	Initially	Insert 4371	Insert 1323	Insert 6173	Insert 4199	Insert 4344	Insert 9699	Insert 1889
0								**1889**
1		**4371**	**4371**	**4371**	**4371**	**4371**	**4371**	**4371**
2							**9699**	**9699**
3			**1323**	**1323**	**1323**	**1323**	**1323**	**1323**
4				**6173**	**6173**	**6173**	**6173**	**6173**
5								
6								
7						**4344**	**4344**	**4344**
8								
9					**4199**	**4199**	**4199**	**4199**

Double Hashing Example

	insert(14) 14%7 = 0	insert(8) 8%7 = 1	insert(21) 21%7 =0 5-(21%5)=4	insert(2) 2%7 = 2	insert(7) 7%7 = 0 5-(21%5)=4
0	14	14	14	14	14
1		8	8	8	8
2				2	2
3					
4			21	21	21
5					
6					
probes:	1	1	2	1	??

Double Hashing Example

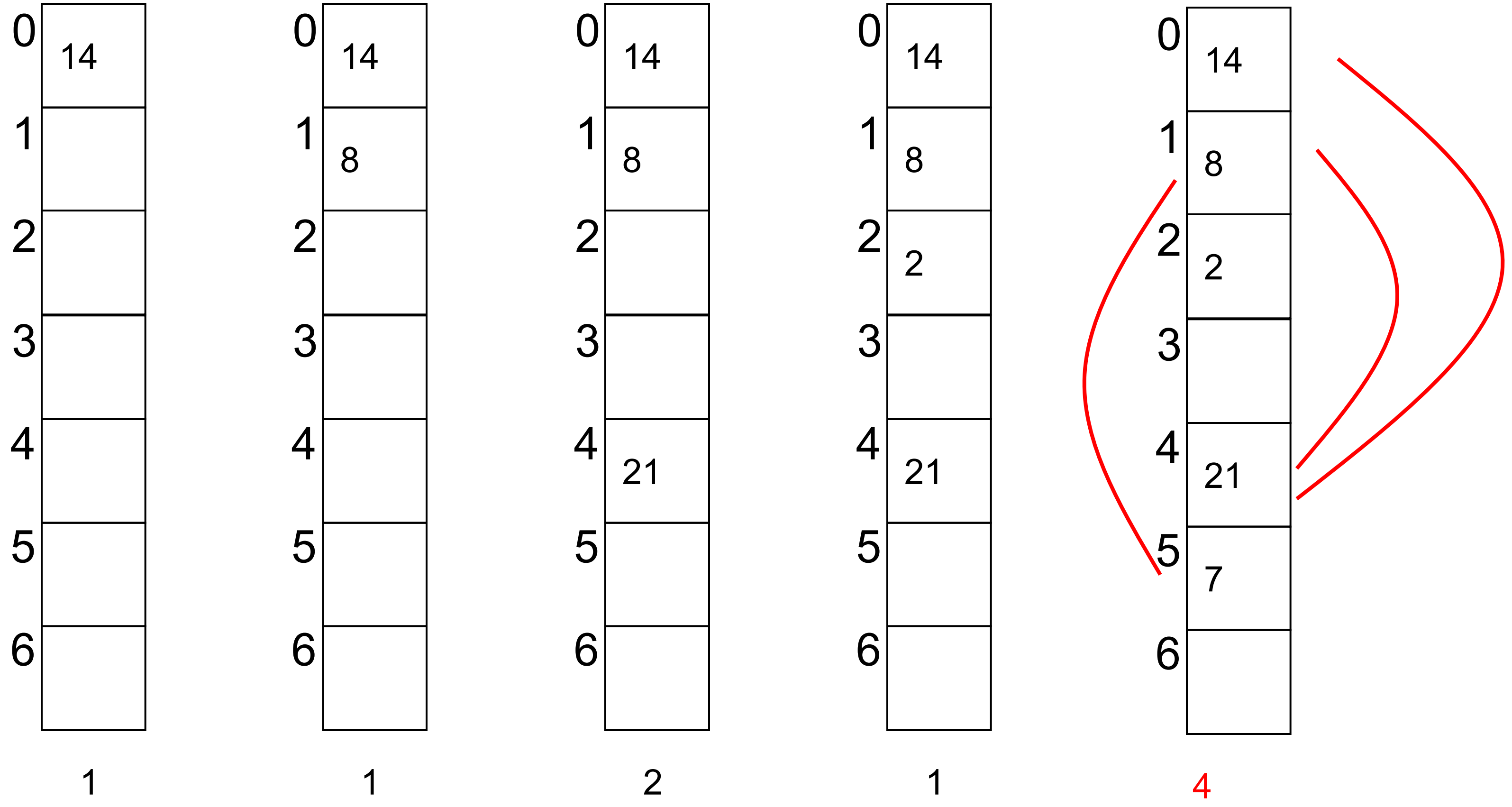
insert(14)
14%7 = 0

insert(8)
8%7 = 1

insert(21)
21%7 =0
5-(21%5)=4

insert(2)
2%7 = 2

insert(7)
7%7 = 0
5-(21%5)=4



	insert(14)	insert(8)	insert(21)	insert(2)	insert(7)
0	14	14	14	14	14
1		8	8	8	8
2				2	2
3					
4			21	21	21
5					7
6					
probes:	1	1	2	1	4

89,18,49,58,69,23

0	69
1	
2	
3	58
4	
5	
6	49
7	
8	18
9	89

Rehashing

- If table gets full, insertion using open addressing with quadratic probing might fail or it might take too much time.

- Rehashing tells us what to do when the hash table gets full.

- Instead of waiting for the hash table to get completely full,it is more efficient to rehash when the table is about 70% or 80 % full.

- To find the solution for this is to build another table that is about twice as big and scan down the entire original hash table, compute the new hash value for each record, and Insert them in a new table.

Rehashing (contd…)

For example, if table is of size 7 and hash function is key % 7 then,

Insert 7,15,13,74,73

0	**7**
1	**15**
2	
3	**73**
4	**74**
5	
6	**13**

Rehashing (contd…)

0	
1	
2	
3	
4	
5	73
6	74
7	7
8	
9	
10	
11	
12	
13	13
14	
15	15
16	

As the table is more than 70 % full, new table is created and the values are inserted in the new table

The size of the new table is 17, that is next prime of double of 7 that is 14
Rehashing is very expensive, as its running time is O(N).

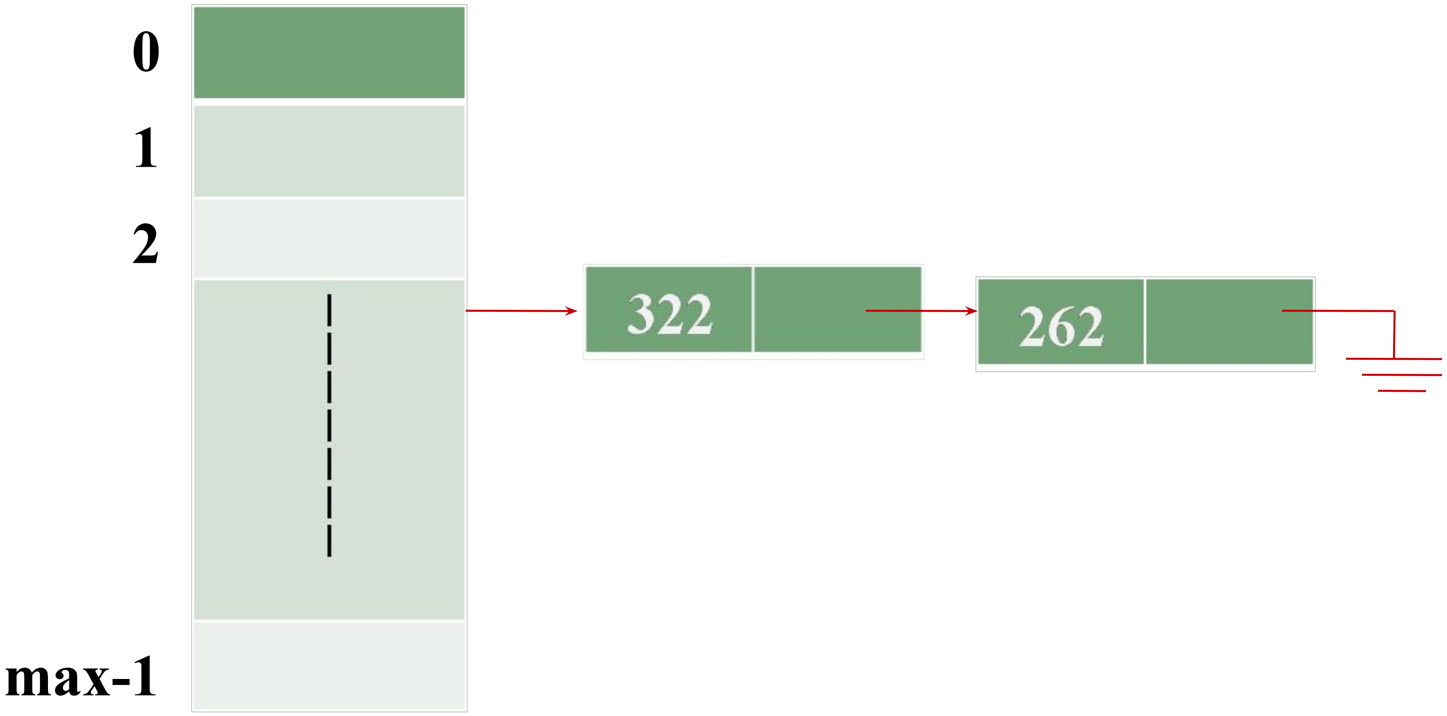
Chaining

- This technique used to handle synonym is chaining that chains together all the records that hash to the same address. Instead of relocating synonyms, a linked list of synonyms is created whose head is home address of synonyms

- However, we need to handle pointers to form a chain of synonyms

- The extra memory is needed for storing pointers

Chaining



Comparison : Rehashing and chaining

Chaining

- Unlimited number of synonyms can be handled in chaining
- Additional cost to be paid is overhead of multiple linked lists
- Sequential search through chain takes more time

Rehashing

- Limited but still a good number of synonyms are taken care of
- The table size is doubled but no additional field of link is to be maintained
- Searching is faster when compared to chaining

Chaining without replacement

-	Key	Chain
0	10	-1
1	01	-1
2	12	-1
3	03	-1
4	04	-1
5		-1
6		-1
7	07	-1
8	08	-1
9		-1

	Key	Chain
0	10	-1
1	01	-1
2	12	5
3	03	-1
4	04	-1
5	02	-1
6	05	-1
7	07	-1
8	08	-1
9		-1

After insertion of 02 and 05

	Key	Chain
0	10	-1
1	01	-1
2	12	5
3	03	-1
4	04	9
5	02	-1
6	05	-1
7	07	-1
8	08	-1
9	14	-1

After insertion of 14

6 and 28 can not be inserted as the table is full

Chaining with replacement

	Key	Chain
0	10	-1
1	01	-1
2	12	-1
3	03	-1
4	04	-1
5		-1
6		-1
7	07	-1
8	08	-1
9		-1

	Key	Chain
0	10	-1
1	01	-1
2	12	5
3	03	-1
4	04	-1
5	02	-1
6		-1
7	07	-1
8	08	-1
9		-1

After insertion of 02

	Key	Chain
0	10	-1
1	01	-1
2	12	6
3	03	-1
4	04	9
5	05	-1
6	02	-1
7	07	-1
8	08	-1
9	14	-1

After insertion of 05 and 14

6 and 28 can not be inserted as the table is full

Hash Table Overflow

- An overflow is said to occur when a new identifier is mapped or hashed into a full bucket

- When the bucket size is one, collision and overflow occur simultaneously

Open Addressing for Overflow Handling

- When a new identifier is hashed into a full bucket, we need to find another bucket for this identifier

- The simplest solution is to find the closest unfilled bucket.

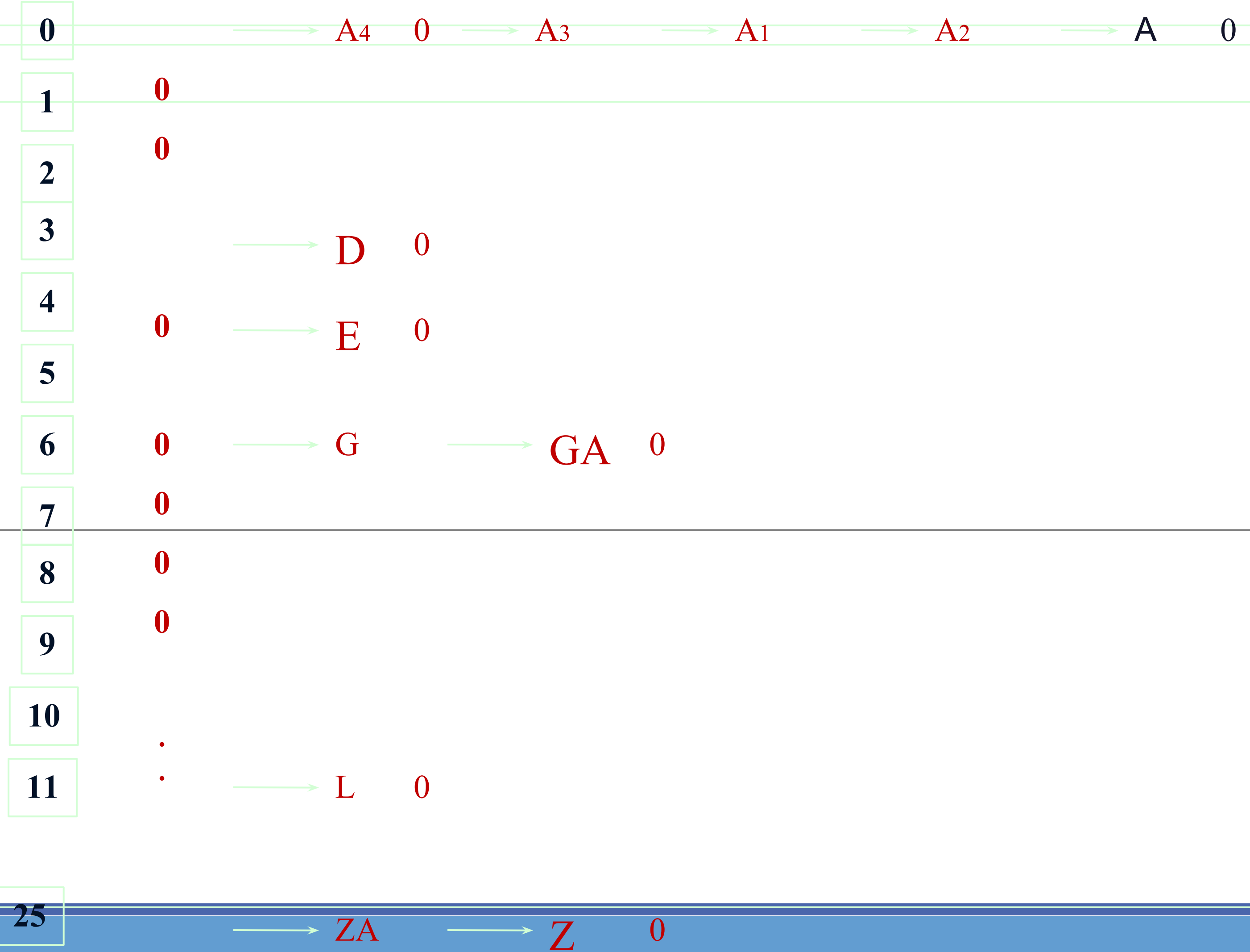
- This is called as linear probing or linear open addressing

Overflow Handling by Chaining

- Since the sizes of these lists are not known in advance, the best way to maintain them is as linked chains
- In each slot, additional space is required for a link
- Each chain has a head node.
- The head node, however, usually is much smaller than the other nodes, since it has to retain only a link
- As the list is accessed at random, the head nodes should be sequential

0	1	2	3	4	5	6	7	8	9	10	11		25
A	A2	A1	D	A3	A4	GA	G	ZA	E		L	….	Z

Chaining



0 → A4 0 → A3 → A1 → A2 → A 0

1 0

2 0

3 → D 0

4 0 → E 0

5

6 0 → G → GA 0

7 0

8 0

9 0

10 .

11 . → L 0

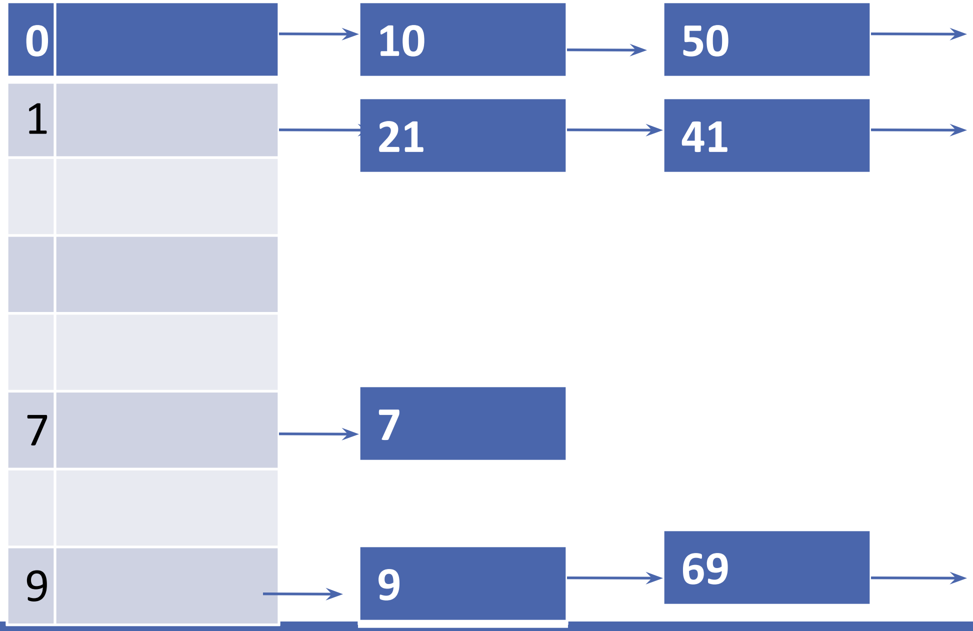
25 → ZA → Z 0

Separate Chaining-Used with Open Hashing

Open Hashing allows records to be stored in unlimited space.

It places no limitation on size of the tables.

Additional Memory space is required for storing links.



Separate Chaining-Used with Open Hashing

Data Structure:

```
#define MAX 1
class node
{
    int data;
    node * next;
};
node *hashtable[MAX];
```

```
Algorithm Initialize(node *hashtable[])
{
        for (i=0;i<MAX;i++)
  {
     allocate memory for head[i];
     Hashtable[i]->data=i
     Hashtable[i]->next=NULL;
  }
}
```

Separate Chaining-Used with Open Hashing

Operations on Hash Table:

```
Algorithm Insert(node *hashtable[],int x)
{
    int loc;
    node *p,*q;
    loc=x % MAX;
    q=new node;
    q->data=x;
    q->next=NULL;
    If(hashtable[loc]->next==NULL)
      Hashtable[loc]->next=q;
else
    {
for(p=hashtable[loc];p->next!=NULL;p=p->next);
    p->next=q;
}
}
```

Separate Chaining-Used with Open Hashing

Operations on Hash Table:

```
Algorithm node *find(node *hashtable[],int x)
{
    int loc,
    node *p;
    loc=x % MAX;
    p=hashtable[loc];
    While(p!=NULL && x!=p->data)
     p=p->next;
     return (p);
}
```

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 is full, then expensive rehash should be used

- For fast searching and less disk access, extendible 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.

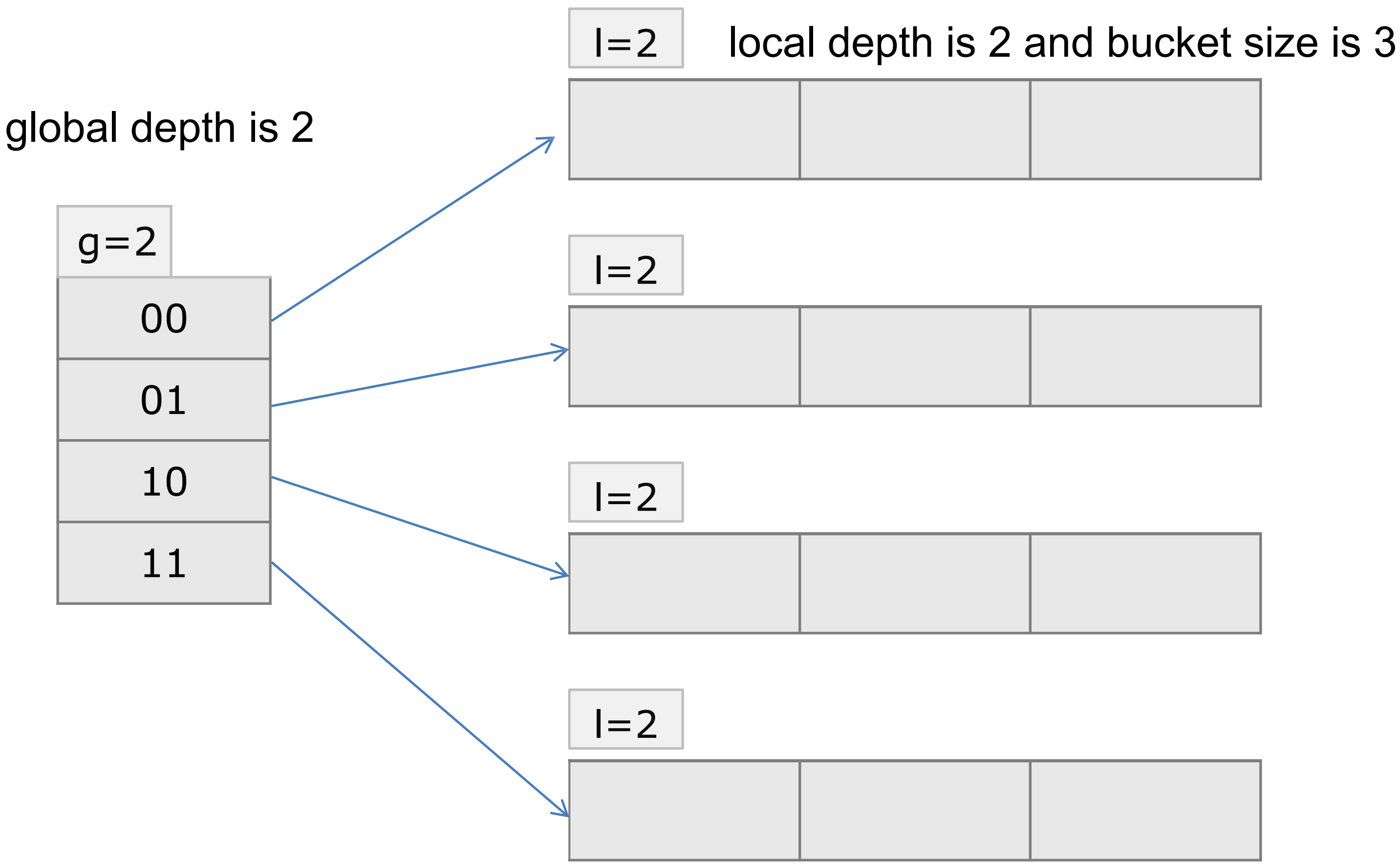
- The first bit of each string will be used as indices to figure out whether they will go in the hash table.

Extendible Hashing

- Suppose that g=2 and bucket size = 3.
- Suppose that we have records with these keys and hash function h(key) = key mod 64:

key	h(key) = key mod 64	bit pattern
1111	23	010111
3333	5	000101
1235	19	010011
2378	10	001010
1212	60	111100
1456	48	110000
2134	22	010110
2345	41	101001
1111	23	010111
8231	39	100111
2222	46	101110
9999	15	001111

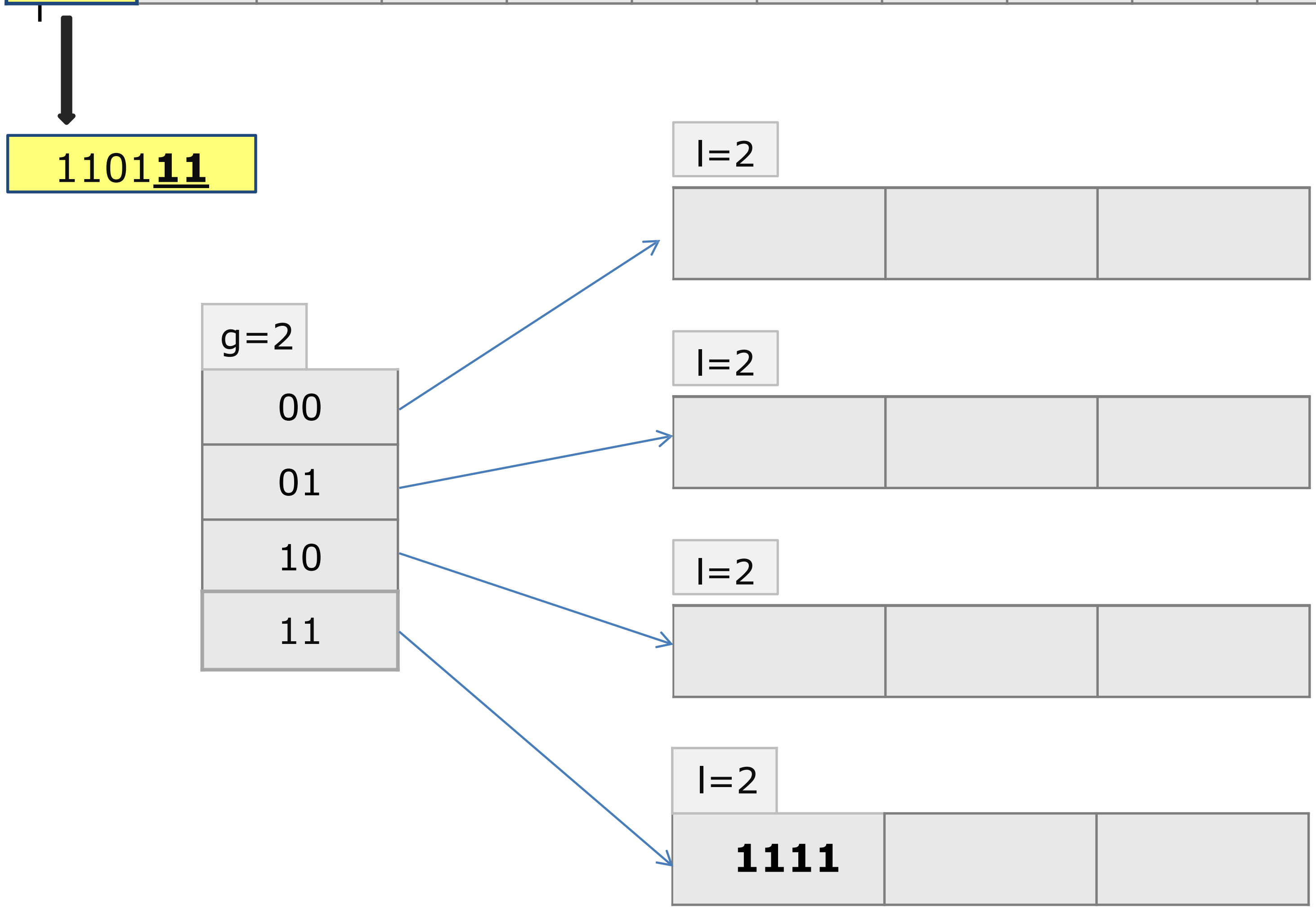
1111	3333	1235	2378	1212	1456	2134	2345	1111	8231	2222	9999



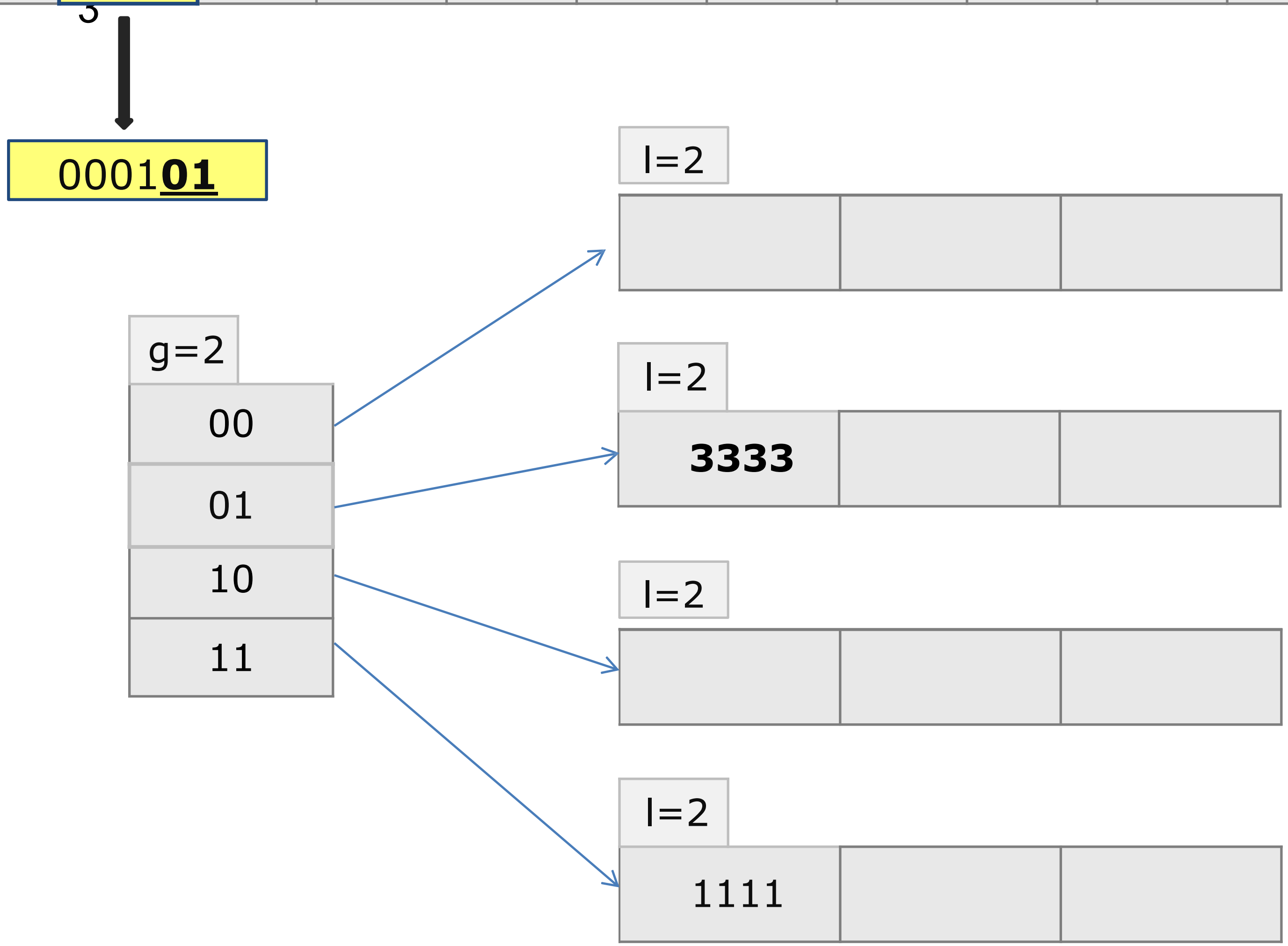
local depth is 2 and bucket size is 3

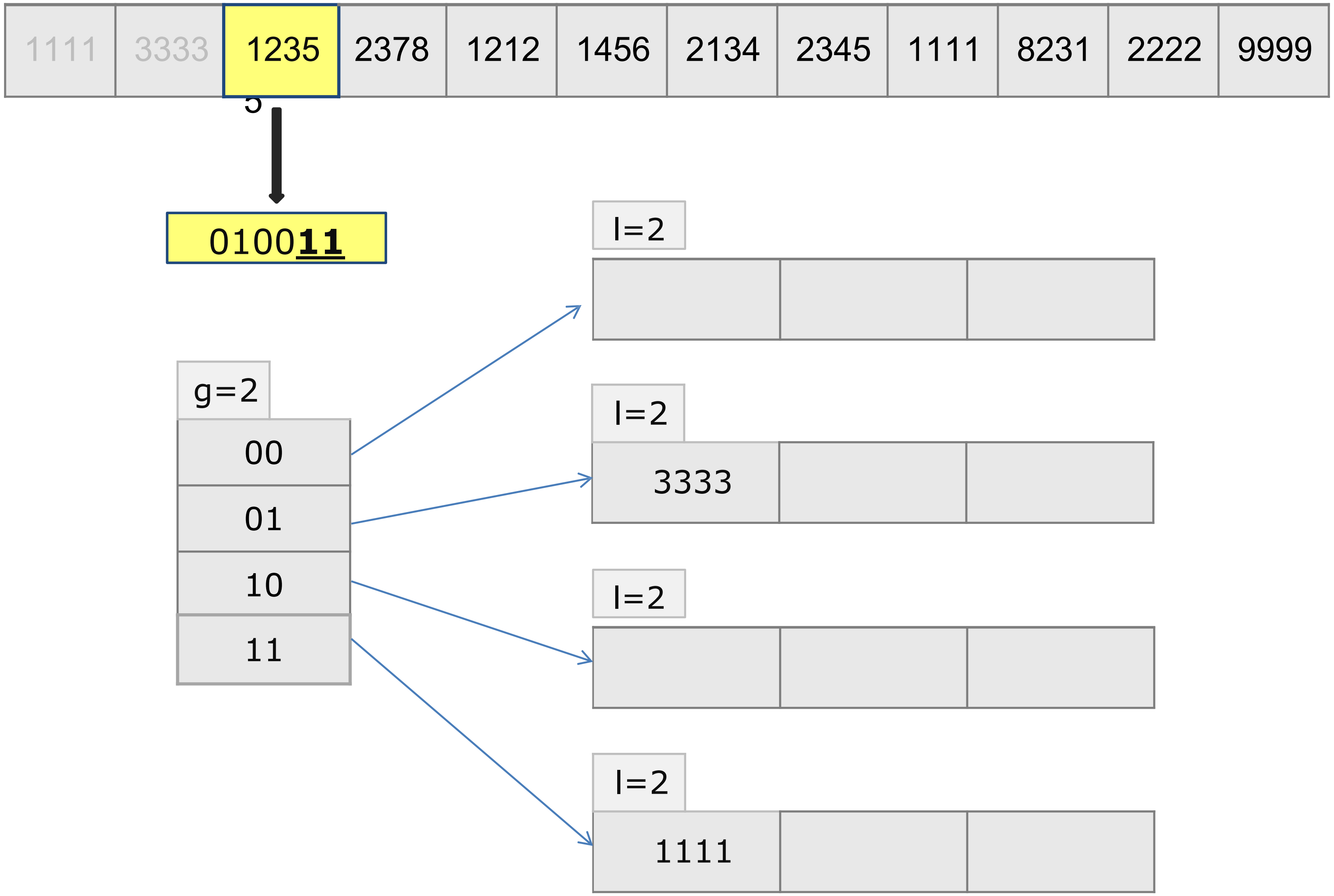
global depth is 2

1111	3333	1235	2378	1212	1456	2134	2345	1111	8231	2222	9999



1111	3333	1235	2378	1212	1456	2134	2345	1111	8231	2222	9999





1111	3333	1235	2378	1212	1456	2134	2345	1111	8231	2222	9999

5

0100**11**

g=2

00

01

10

11

l=2

l=2

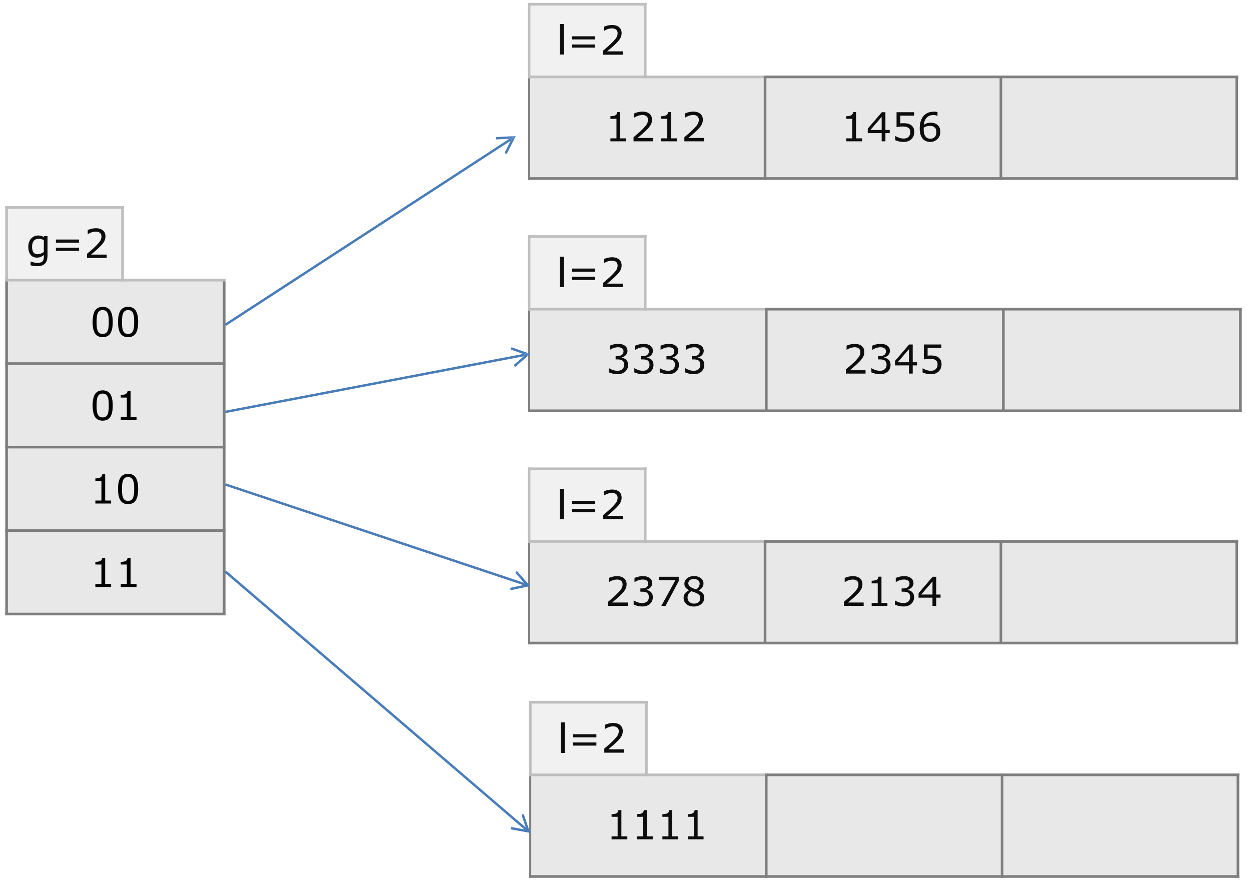
3333

l=2

l=2

1111

1111	3333	1235	2378	1212	1456	2134	2345	1111	8231	2222	9999



g=2

00

01

10

11

l=2

1212 | 1456

l=2

3333 | 2345

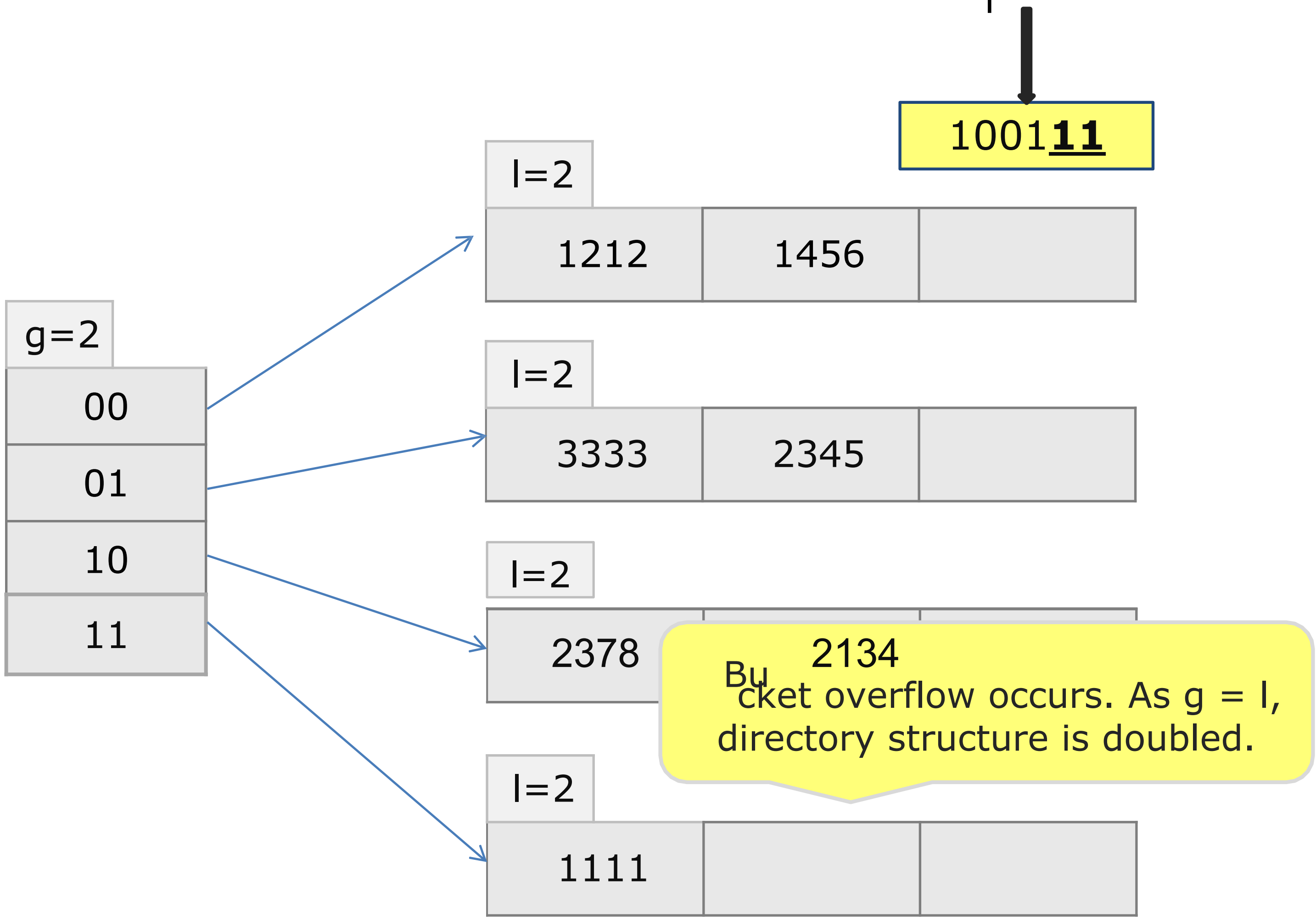
l=2

2378 | 2134

l=2

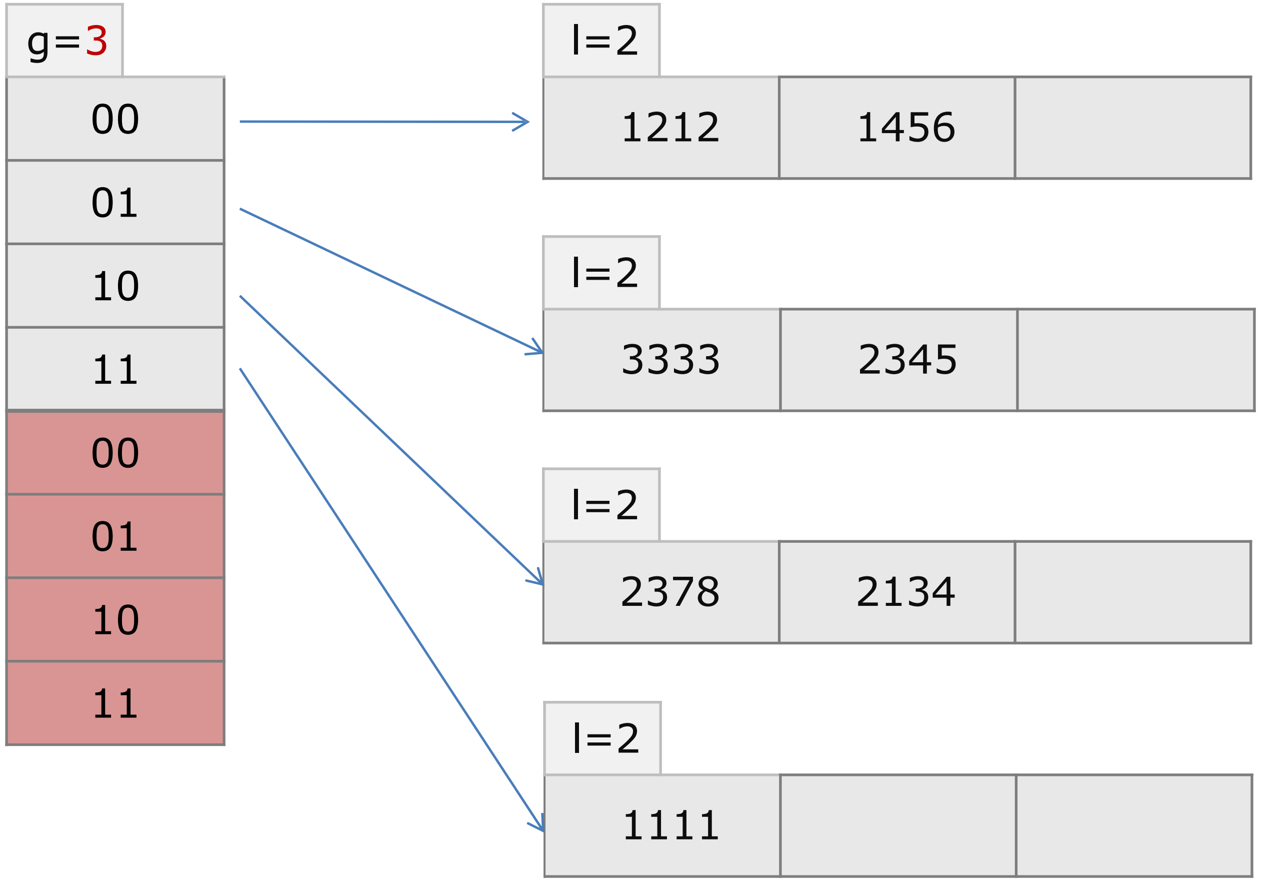
1111

1111	3333	1235	2378	1212	1456	2134	2345	1111	8231	2222	9999



Bucket overflow occurs. As g = l, directory structure is doubled.

1111	3333	1235	2378	1212	1456	2134	2345	1111	8231	2222	9999



g=3

00
01
10
11
00
01
10
11

l=2

1212	1456	

l=2

3333	2345	

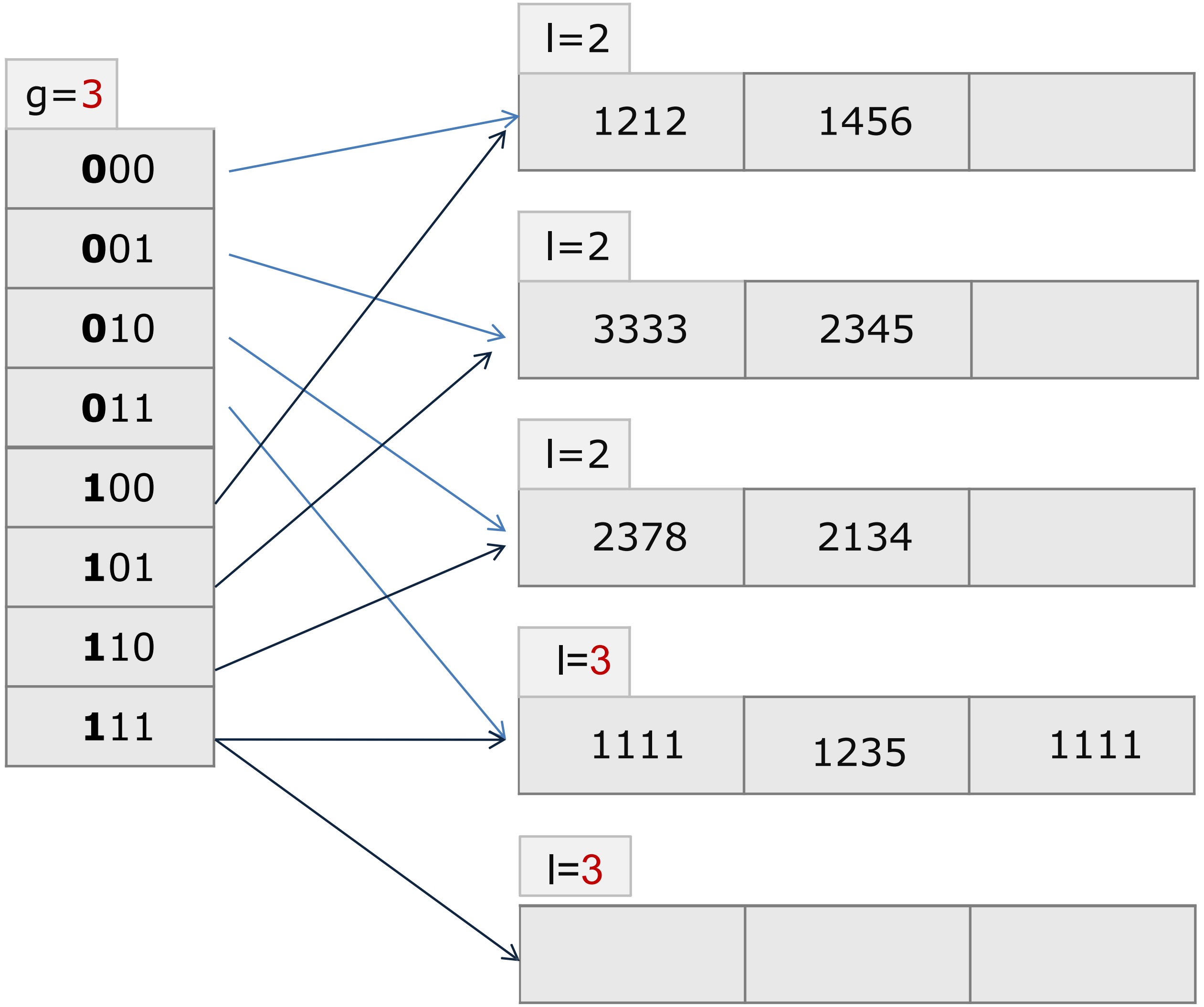
l=2

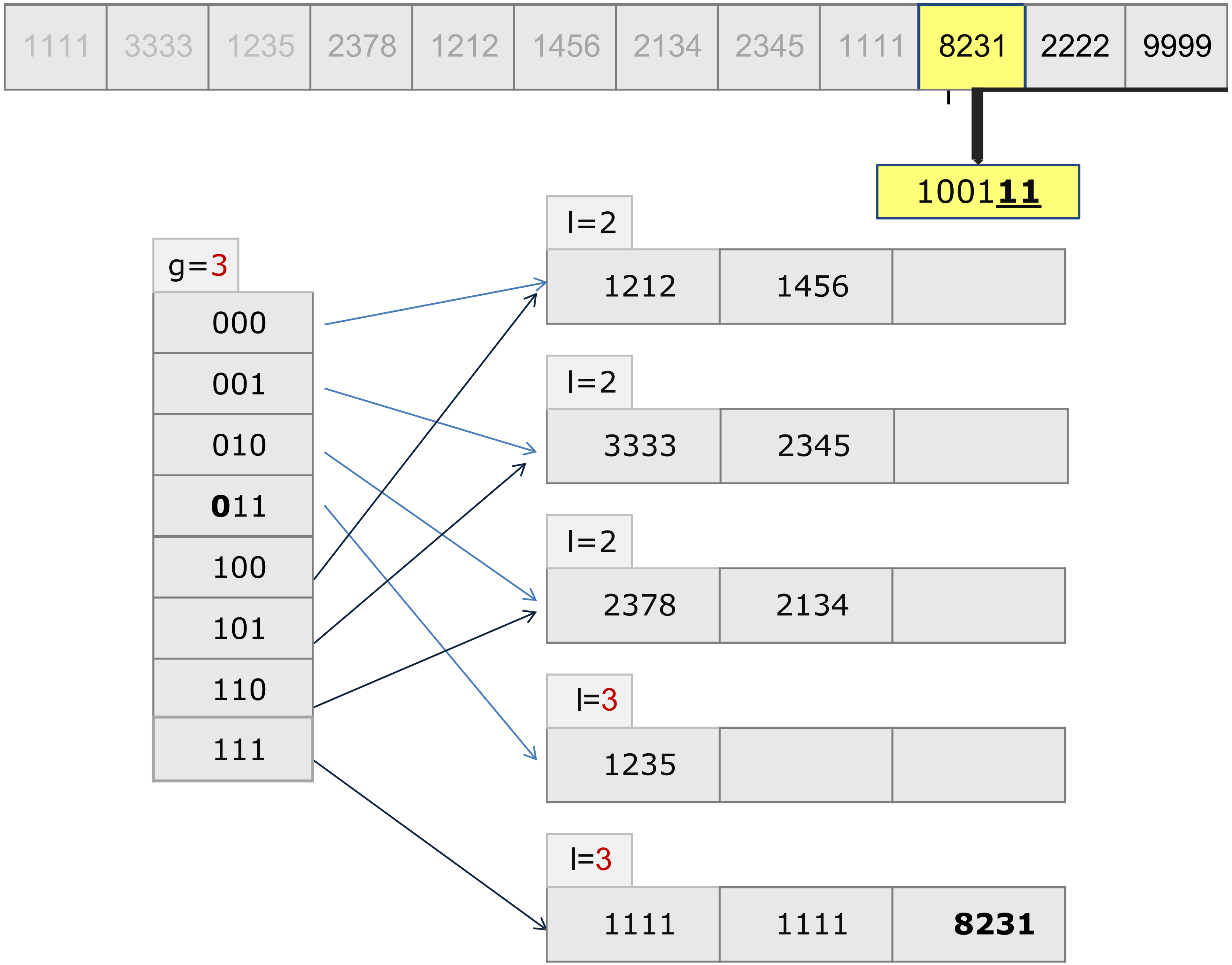
2378	2134	

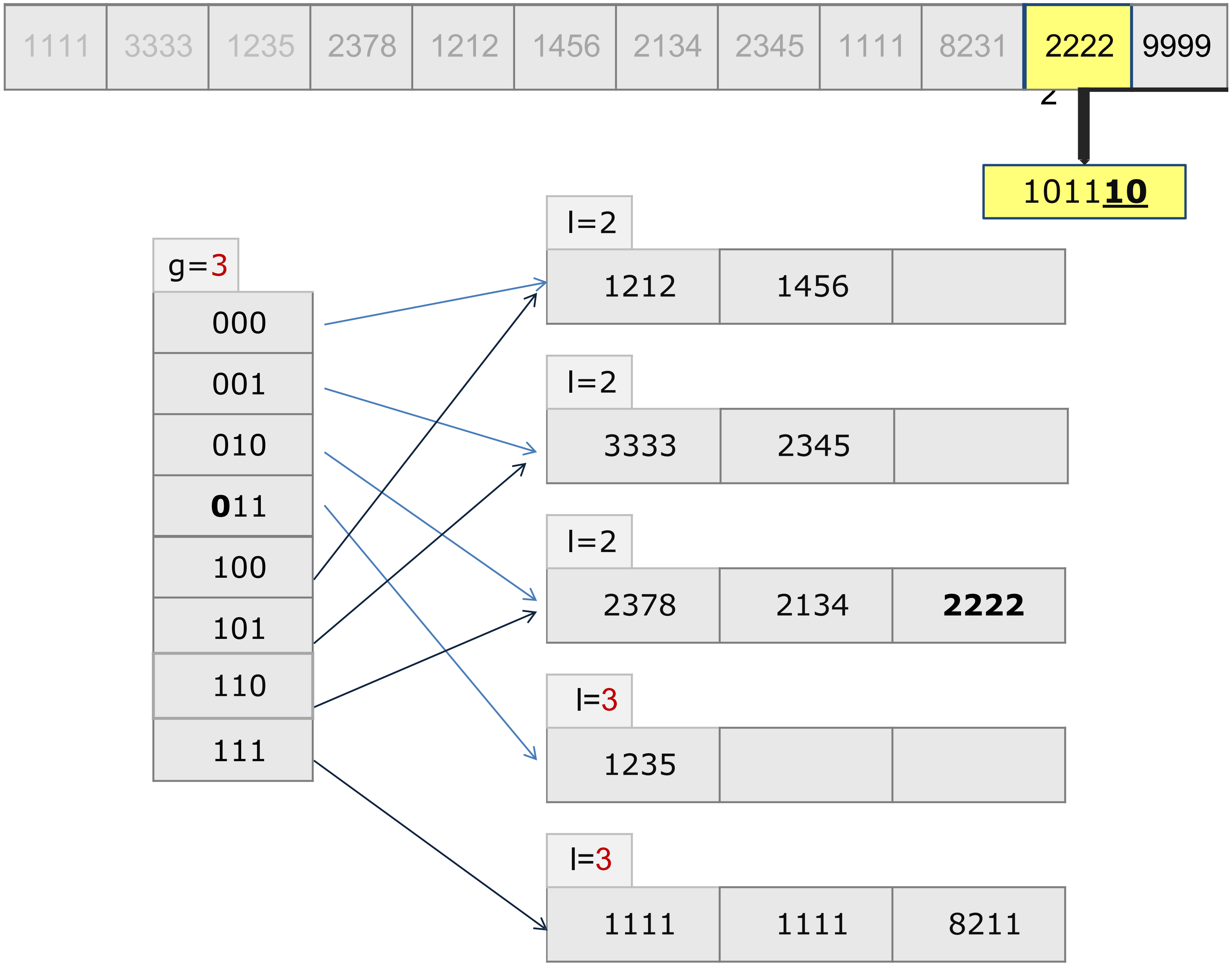
l=2

1111		

1111	3333	1235	2378	1212	1456	2134	2345	1111	8231	2222	9999




1111
3333
1235
2378
1212
1456
2134
2345
1111
8231
2222
9999
100111
g=3
000
001
010
011
100
101
110
111
l=2
1212
1456
l=2
3333
2345
l=2
2378
2134
l=3
1235
l=3
1111
1111
8231



1111	3333	1235	2378	1212	1456	2134	2345	1111	8231	2222	9999

2

1011**<u>10</u>**

g=3

000
001
010
011
100
101
110
111

l=2

1212	1456	

l=2

3333	2345	

l=2

2378	2134	**2222**

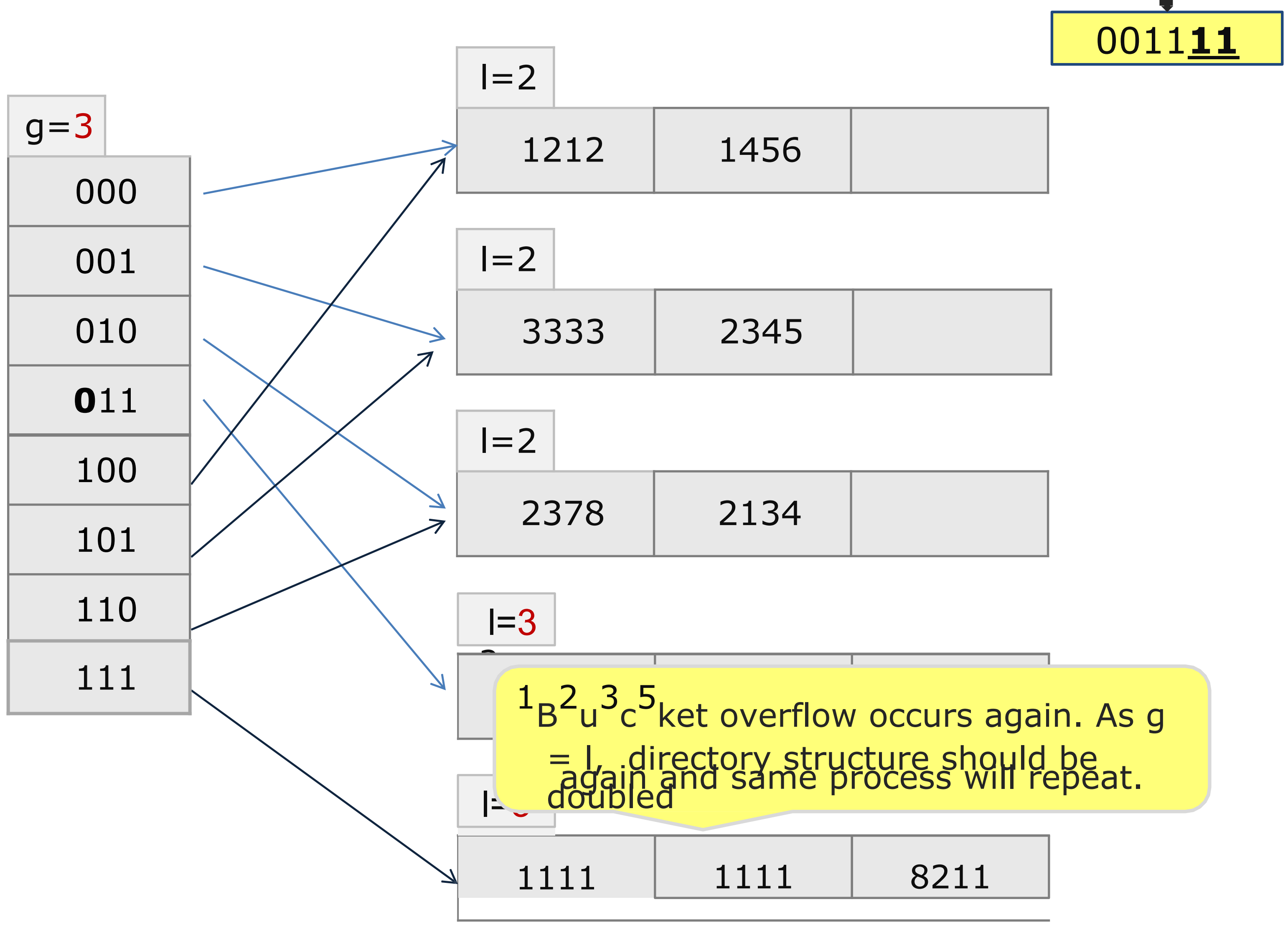
l=3

1235		

l=3

1111	1111	8211

1111	3333	1235	2378	1212	1456	2134	2345	1111	8231	2222	9999



^{1}B^{2}u^{3}c^{5}ket overflow occurs again. As g = l, directory structure should be doubled

again and same process will repeat.

FAQ

What is hashing?

Why there is need of hashing?

What is the time complexity of hashing?

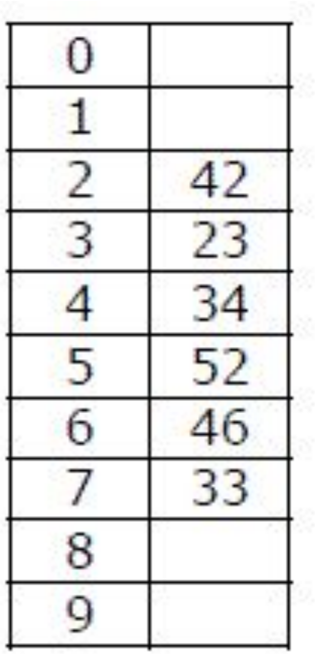
What is collision?

Define hash function, hash key

What are the collision resolution techniques?

MCQs

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



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

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

MCQs

How many different insertion sequences of the key values using the same hash function and linear probing will result in the hash table shown above?

A.10

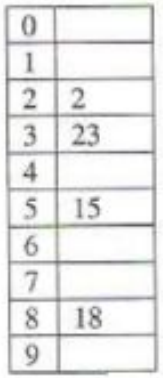
B.20

C.30

D.40

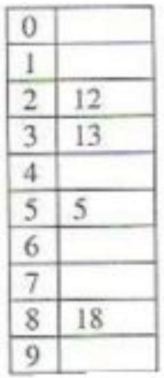
MCQs

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?



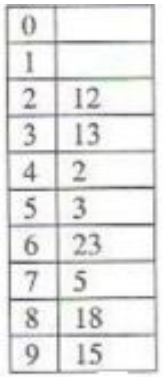
0	
1	
2	2
3	23
4	
5	15
6	
7	
8	18
9	

(A)



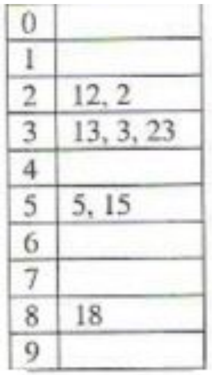
0	
1	
2	12
3	13
4	
5	5
6	
7	
8	18
9	

(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)

MCQs

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.

A 8, _, _, _, _, _, 10

B 1, 8, 10, _, _, _, 3

C 1, _, _, _, _, _,3

D 1, 10, 8, _, _, _, 3

MCQs

Given the following input (4322, 1334, 1471, 9679, 1989, 6171, 6173, 4199) and the hash function x mod 10, which of the following statements are true? i. 9679, 1989, 4199 hash to the same value ii. 1471, 6171 has to the same value iii. All elements hash to the same value iv. Each element hashes to a different value (GATE CS 2004)

A i only

B ii only

C i and ii only

D iii or iv

Practice Assignments

1. Write a pseudo code to implement quadratic probing
2. Store roll numbers of the students in a database and implement double hashing onto it.
3. Store roll numbers of the students in a database and implement linear probing with and without replacement onto it.
4. Implement chaining method by identifying one suitable application.

Takeaway

- Hashing is one of efficient searching technique.
- Hashing's best case time complexity is O(1)
- Hashing collision is resolved by using different collision resolution techniques.

References

1. Horowitz, Sahani, Dinesh Mehta, “Fundamentals of Data Structures in C++”, Galgotia Publisher, ISBN: 8175152788, 9788175152786.

2. Peter Brass, “Advanced Data Structures”, Cambridge University Press, ISBN: 978-1-107-43982