ĐẠI HỌC ĐÀ NẴNG

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG VIỆT - HÀN

VIETNAM - KOREA UNIVERSITY OF INFORMATION AND COMMUNICATION TECHNOLOGY

한-베정보통신기술대학교

Nhân bản – Phụng sự – Khai phóng

Hash Table

Data Structures & Algorithms

CONTENT

- Introduction
- Static hashing
- Dynamic hashing

CONTENT

- **Introduction**
- Static hashing
- Dynamic hashing

- A *table* has several *fields* (types of information)
  - A telephone book may have fields **name**, **address**, **phone number**
  - A user account table may have fields **user id**, **password**, **home folder**

⇨ To find an *entry* in the table, you only need know the contents of one of the fields (not all of them).

- The *key* field
  - In a telephone book, the key is usually **name**
  - In a user account table, the key is usually **user id**

⇨ Key *uniquely identifies* an entry

  - If the key is **name** and no two entries in the telephone book have the same name, the key uniquely identifies the entries

- **Element: a key and its entry**

For searching purposes, it is best to store the key and the entry separately (even though the key’s value may be inside the entry)

	key	**entry**
Element	“Smith”	“Smith”, “124 Hawkers Lane”, “9675846”
	“Yeo”	“Yeo”, “1 Apple Crescent”, “0044 1970 622455”

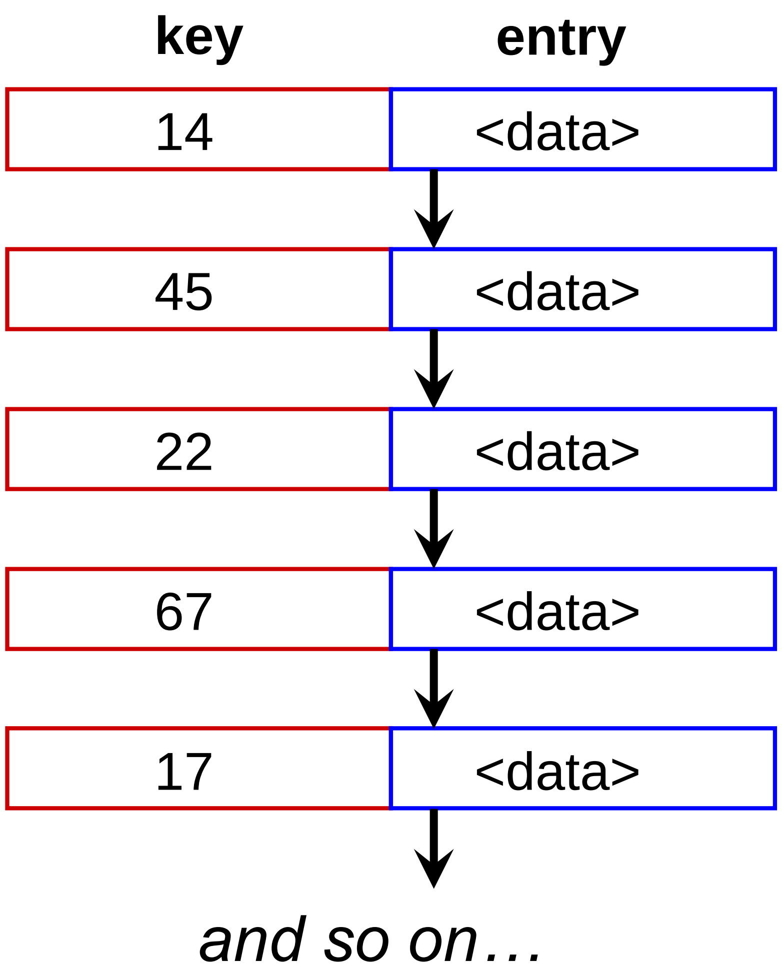
- **Implementation 1- Unsorted Sequential Array**
  - An array in which elements are stored consecutively in *any* order
  - **insert**: add to back of array; O(1)
  - **find**: search through the keys one at a time, potentially all of the keys; O(*n*)
  - **remove**: find + replace removed node with last node; O(*n*)

	key	entry
0	14	<data>
1	45	<data>
2	22	<data>
3	67	<data>
4	17	<data>
⋮	*and so on…*	

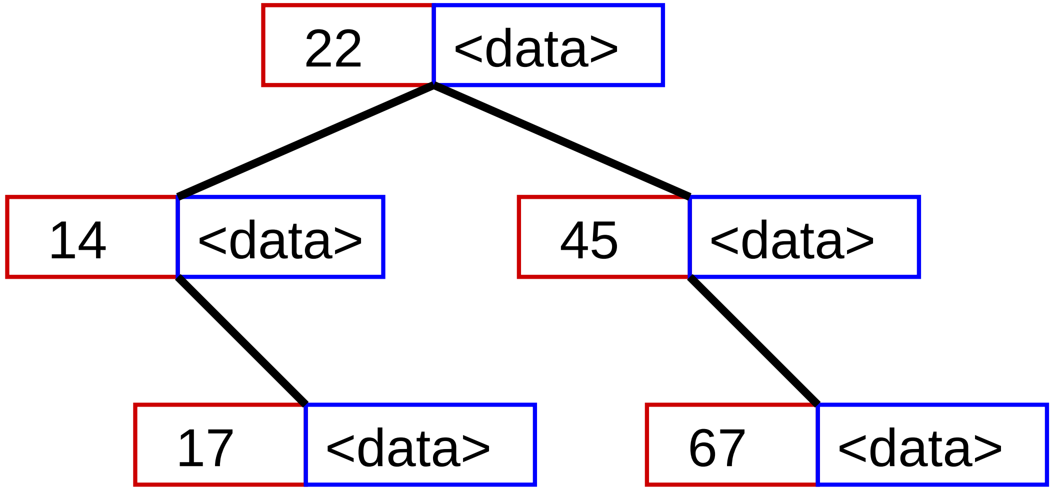
- **Implementation 2 - Sorted Sequential Array**
  - An array in which elements are stored consecutively, *sorted* by key
  - **insert**: add in sorted order; O(*n*)
  - **find**: binary search; O(log *n*)
  - **remove**: find, remove element; O(log *n*)

	key	entry
0	15	<data>
1	17	<data>
2	22	<data>
3	45	<data>
4	67	<data>
⋮	*and so on…*	

- **Implementation 3 - Linked List (Unsorted or Sorted)**
  - Elements are again stored consecutively
  - **insert**: add to front; O(1) or *O(n)* for a sorted list
  - **find**: search through potentially all the keys, one at a time; O(*n*) still *O(n)* for a sorted list
  - **remove**: find, remove using pointer alterations; O(*n*)

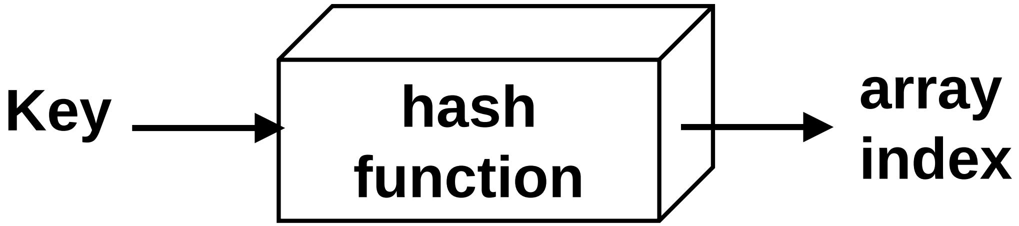


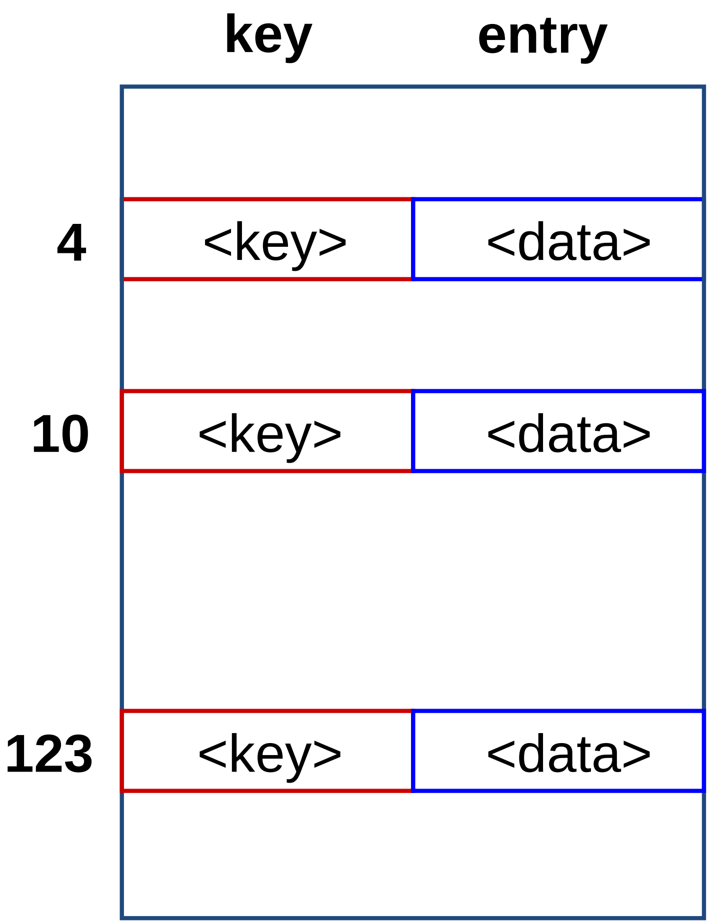
- **Implementation 4 - BST**
  - A BST, ordered by key
  - **insert**: a standard insert; O(log *n*)
  - **find**: a standard find (without removing, of course); O(log *n*)
  - **remove**: a standard remove; O(log *n*)



and so on…

- **Implementation 5 - Hash Table**
  - An array in which elements are ***not*** stored consecutively - their place of storage is calculated using the key and a *hash function*

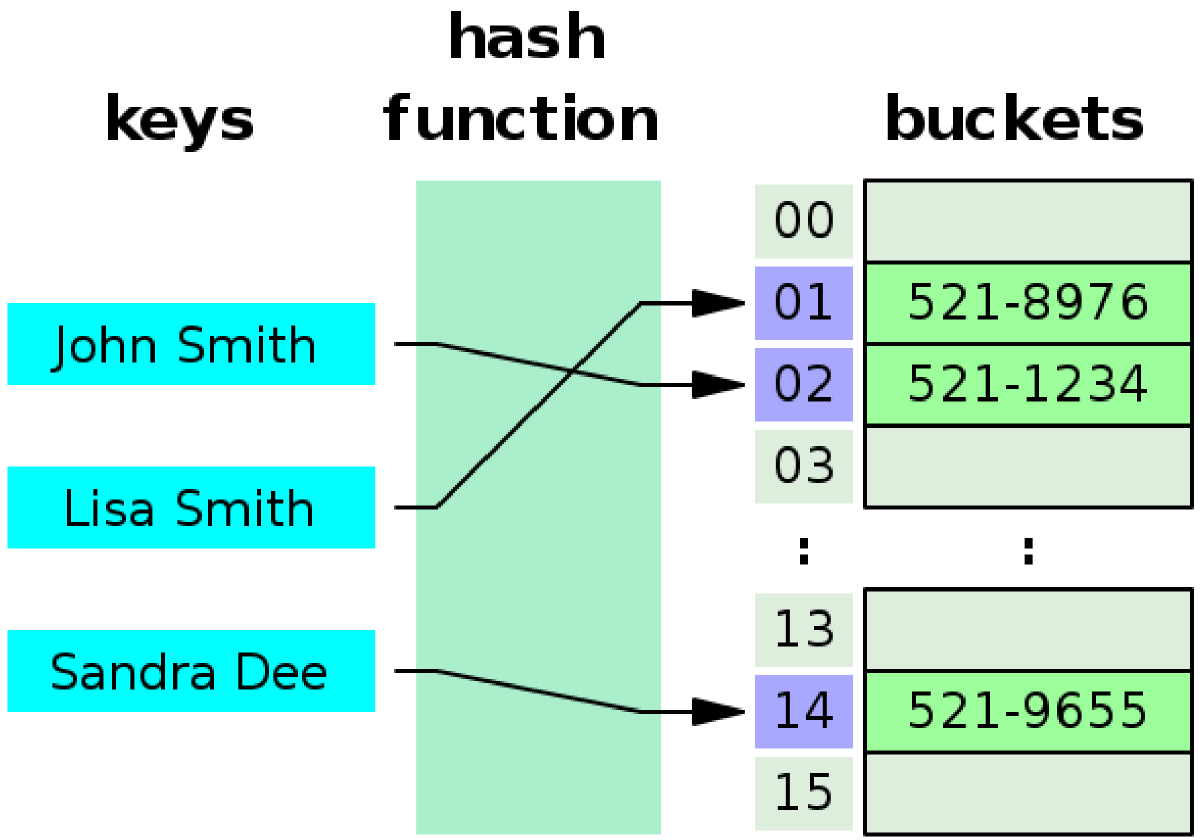


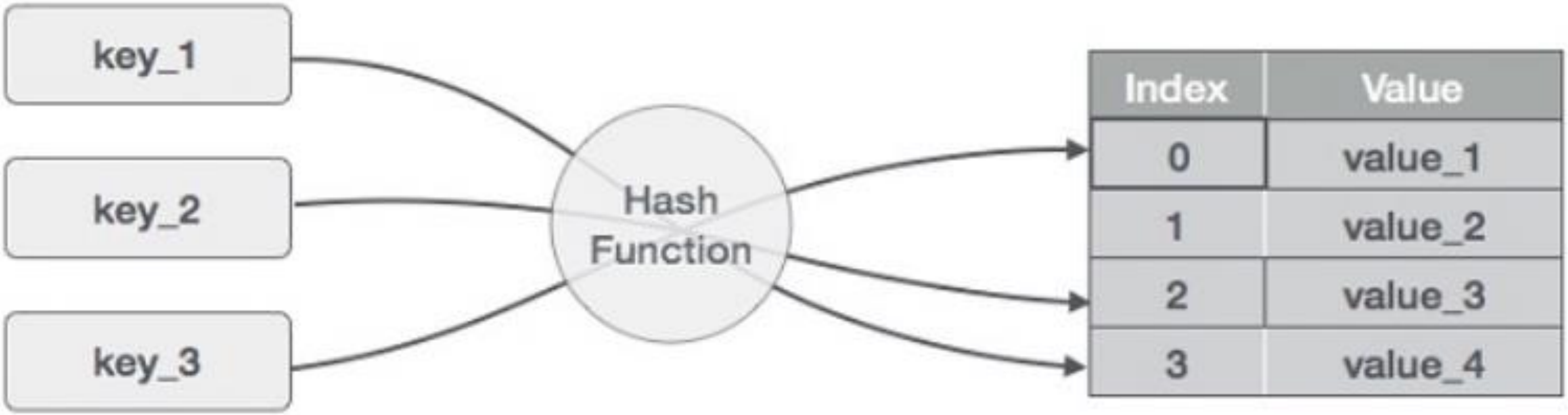


Hash values → mappings of the keys as the indexes in the hash table

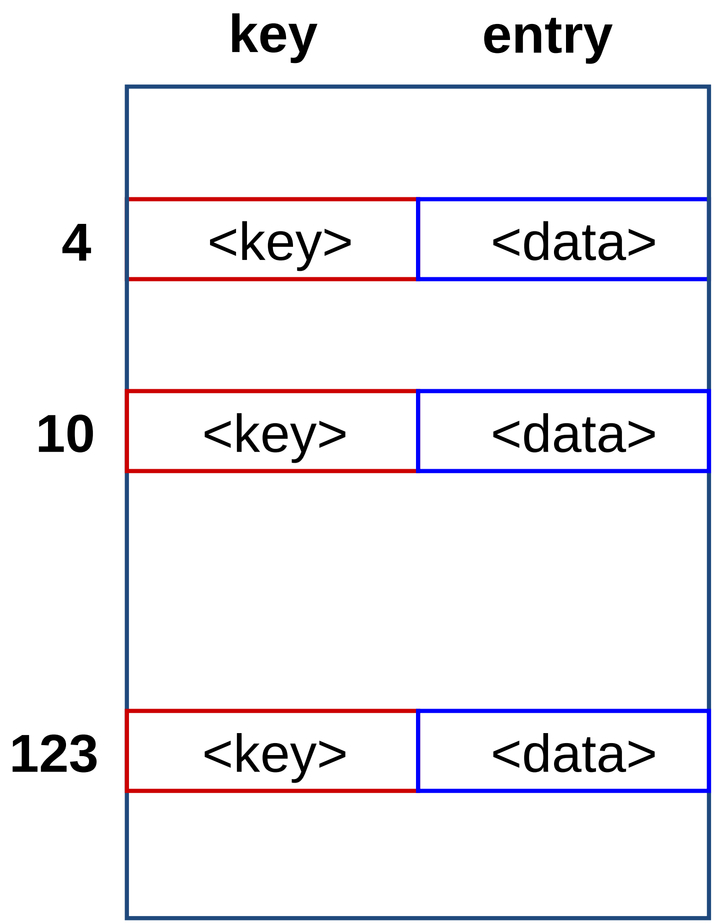
We need to save the phone numbers of 3 people:

John Smith: 521-1234
Lisa Smith: 521-8976
Sandra Dee: 521-9655




key_1
key_2
key_3
Hash Function
Index
Value
0
value_1
1
value_2
2
value_3
3
value_4

- **Implementation 5 - Hash Table**
  - **Hashed key**: result of applying a hash function to a key
  - Keys and entries are scattered throughout the array
  - Array elements are not stored consecutively, their place of storage is calculated using the key & a hash function
  - **insert**: calculate place of storage, insert TableNode; O(1)
  - **find**: calculate place of storage, retrieve entry; O(1)
  - **remove**: calculate place of storage, set it to null; O(1)



All are O(1) !

- **Applications of Hashing**
  - Compilers use hash tables to keep track of declared variables
  - A hash table can be used for on-line spelling checkers — if misspelling detection (rather than correction) is important, an entire dictionary can be hashed and words checked in constant time
  - Game playing programs use hash tables to store seen positions, thereby saving computation time if the position is encountered again
  - Hash functions can be used to quickly check for inequality — if two elements hash to different values they must be different
  - Storing sparse data

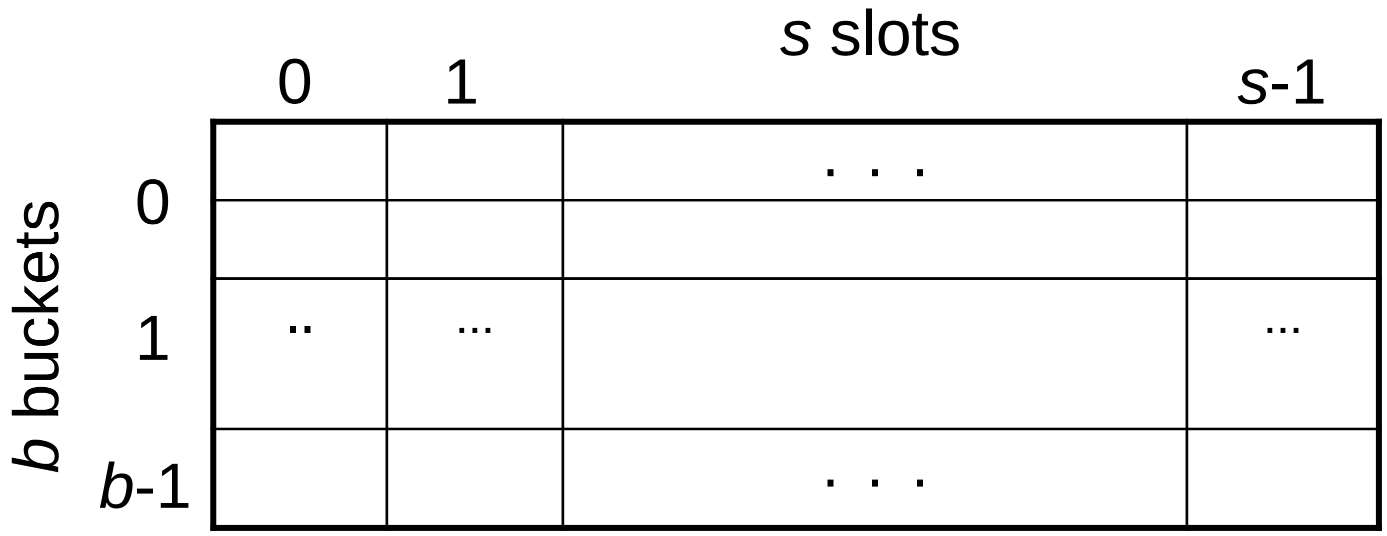
- **When are other representations more suitable than hashing?**
  - Hash tables are very good if there is a need for many searches in a reasonably stable table
  - Hash tables are not so good if there are many insertions and deletions, or if table traversals are needed

- **Types of hashing**
  - Static hashing
    - Tables with a fixed size
  - Dynamic hashing
    - Table sizes may vary

CONTENT

- Introduction
- **Static hashing**
  - Hash table
  - Hash methods
  - Collision resolution
- Dynamic 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 (i.e. key) into an address in the hash table



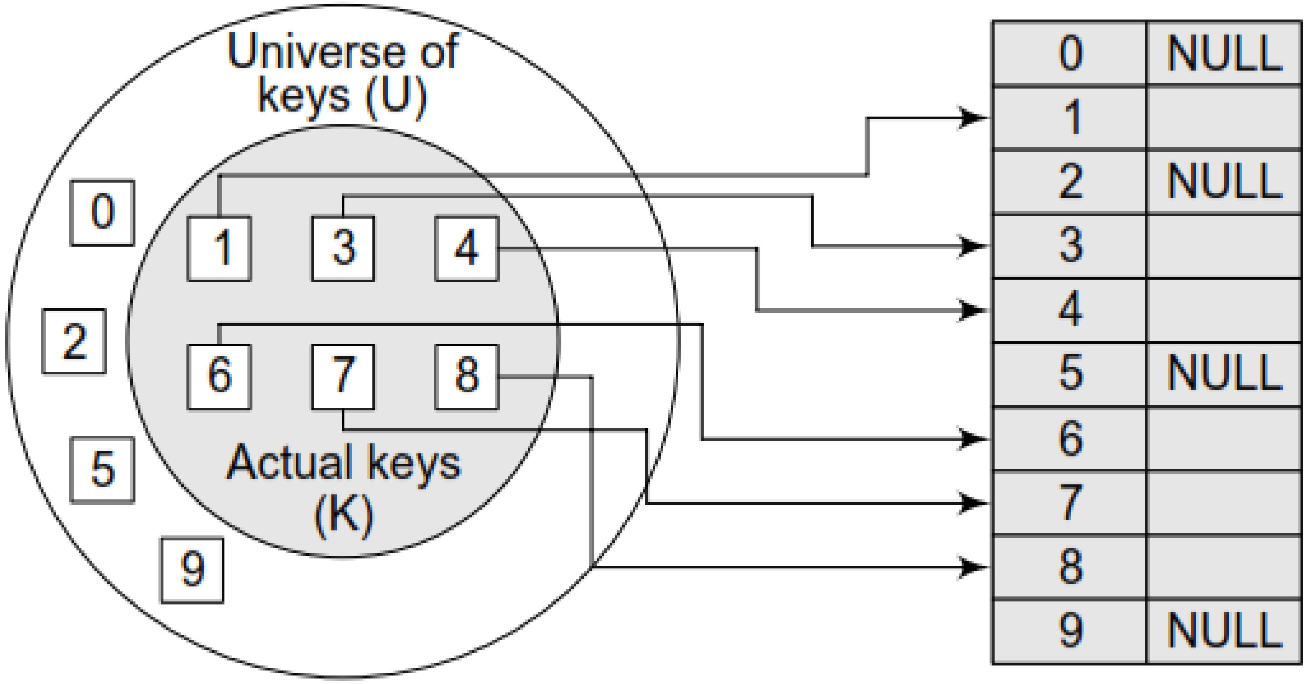
- **Uses an array hash_table[0..b-1].**
  - Each position of this array is a **bucket**
  - A bucket can normally hold only one dictionary pair (key, element)
- **Uses a hash function *f***
  - that converts each key *k* into an index in the range [0, *b*-1].

⇨**Every dictionary pair (key, element) is stored in its home bucket hash_table[f(key)]**

- **Data Structure for Hash Table**

```
#define MAX_CHAR  10
#define TABLE_SIZE  13
typedef struct {
    char key[MAX_CHAR];
    /* other fields */
} element;
element hash_table[TABLE_SIZE];
```

- **Hash table is a data structure in which keys are mapped to array positions by a hash function**



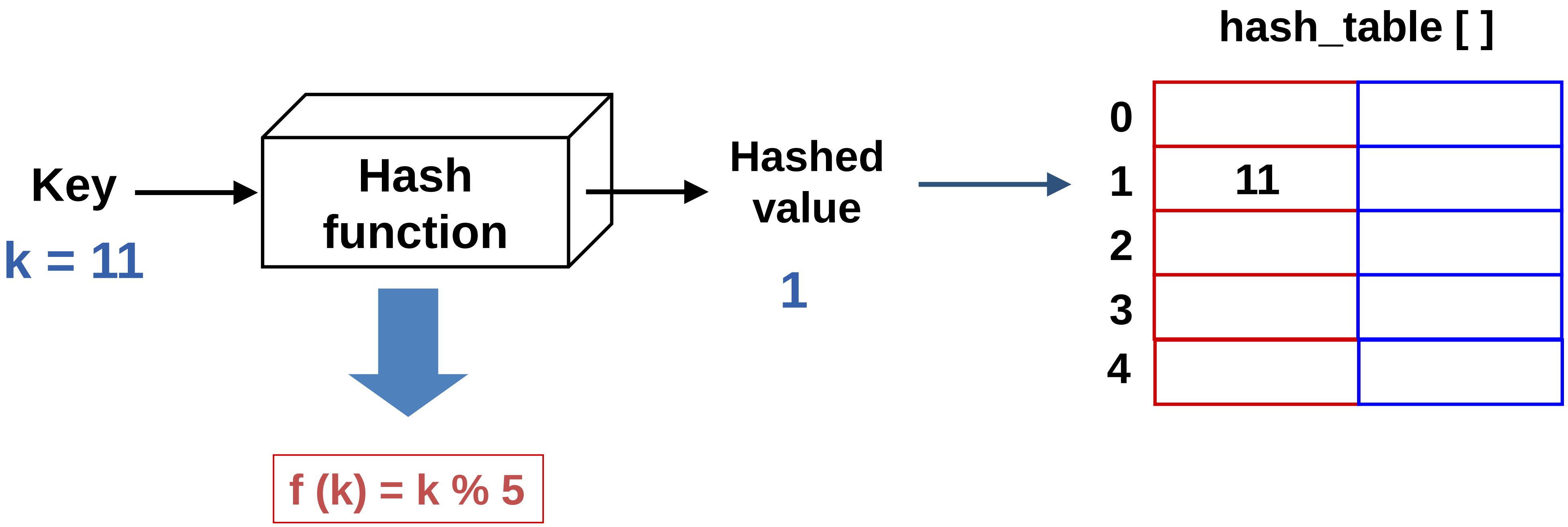
Direct relationship between key and index in the array
(hash function: f(k)=k)

- *f(k)=k* is useful when the total universe of keys is small and when most of the keys are actually used from the whole set of keys
- However, when the set K of keys that are actually used is smaller than the universe of keys (U), a **hash table** consumes less storage space. The storage requirement for a hash table is O(k), where k is the number of keys actually used

- **In a hash table, an element with key k is stored at index f(k) and not k.**
- **It means a hash function f is used to calculate the index at which the element with key k will be stored.**
- **This process of mapping the keys to appropriate locations (or indices) in a hash table is called hashing.**

⇨ Access of data becomes very fast if we know the index of the desired data.

- Example

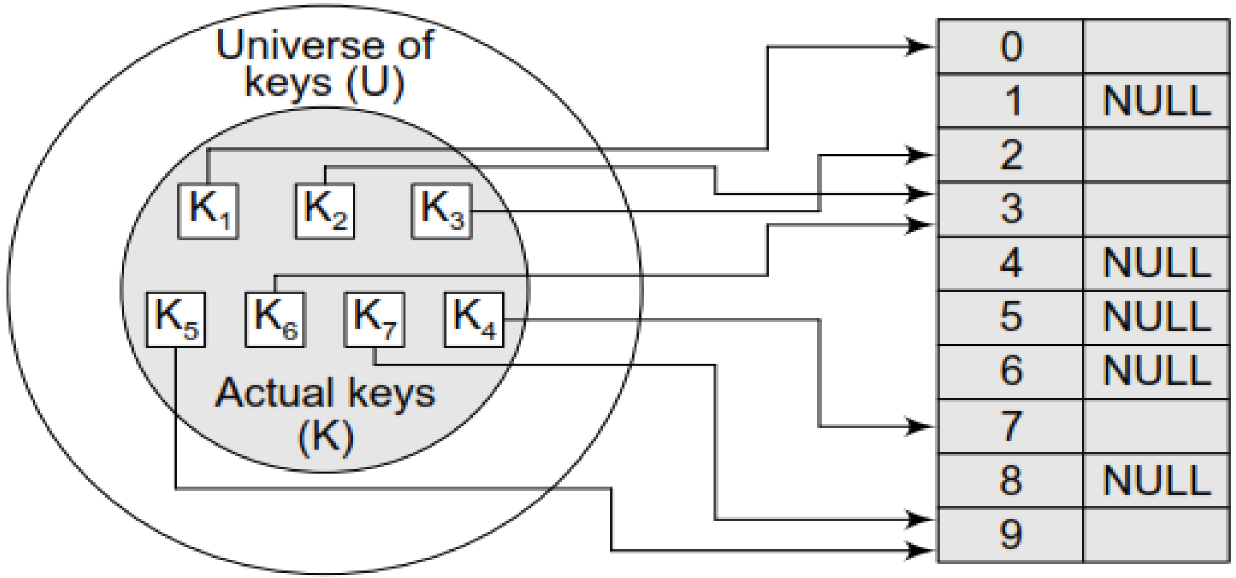

Key
k = 11
Hash function
f (k) = k % 5
Hashed value
1
hash_table [ ]
0
1
11
2
3
4

- **Hash Function - Example**

```
void init_table(element ht[]){
        int i;
        for (i=0; i<TABLE_SIZE; i++)    ht[i].key[0]=NULL;
}

int hash( char *key, int TABLE_SIZE ) {
        unsigned int hash_val = 0;
        while (*key != '\0' )   hash_val += *key++;
        return( hash_val % TABLE_SIZE );
}
```

- Each key from the set K is mapped to locations generated by using a hash function.



Relationship between keys and hash table index (an element with key k is stored at index k(k) and not k)

- Note that keys K_2 and K_6 point to the same memory location. This is known as collision. Similarly, keys K_5 and K_7 also collide.
- **Collision:** That is, when two or more keys map to the same memory location (a collision is said to occur).
- The main goal of using a hash function is to reduce the range of array indices that have to be handled. Thus, instead of having U values, we just need K values, there by reducing the amount of storage space required.

- **Collision**
  - In a hash table with a single array table (single slot bucket), two different keys may be hashed to **the same hash value**
    - Two different k1 and k2
    - Hash (k1) = Hash (k2)
    - Keys that have the same home bucket are synonyms
  - This is called **collision.** Example: k1 = 11, k2 = 21

    Hash (11) = 11 % 5 = 1

    Hash (21) = 21 % 5 = 1
- **Choice of hash method**
  - To avoid collision (two different pairs are in the same the same bucket)
  - Size (number of buckets) of hash table
- **Overflow handling method**
  - Overflow: there is no space in the bucket for the new pair

- **Overflow Example**

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

synonyms (row 0)

synonyms (row 2)

synonyms:
char, ceil,
clock, ctime

↑

overflow

- A **hash function** is a mathematical formula which, when applied to a key, produces an integer which can be used as an index for the key in the hash table.
- The main aim of a hash function is that elements should be relatively, randomly, and uniformly distributed. It produces a unique set of integers within some suitable range in order to reduce the number of collisions.
- In practice, there is no hash function that eliminates collisions completely. A good hash function can only minimize the number of collisions by spreading the elements uniformly throughout the array.

- Properties of a Good Hash Function
  - **Low cost**: The cost of executing a hash function must be small, so that using the hashing technique becomes preferable over other approaches.

    For example, if binary search algorithm can search an element from a sorted table of n items with $\log_2 n$ key comparisons, then the hash function must cost less than performing $\log_2 n$ key comparisons.
  - **Determinism**: A hash procedure must be deterministic. This means that the same hash value must be generated for a given input value.
  - **Uniformity**: A good hash function must map the keys as evenly as possible over its output range. This means that the probability of generating every hash value in the output range should roughly be the same. The property of uniformity also minimizes the number of collisions.

- **Choice of Hash Method**
  - Requirements
    - easy to compute
    - minimal number of collisions
  - If a hashing function groups key values together, this is called **clustering** of the keys
    - The larger the cluster, the longer the search
  - A good hashing function **distributes the key values uniformly** throughout the range
    - For a random variable X, $P(X = i) = 1/b$ (b is the number of bucket)

- **Choice of Hash Method**
  - The **worst hash function** maps all keys to the same bucket
  - The **best hash function** maps all keys to distinct addresses
  - **Ideally, distribution of keys to addresses is uniform and random**

- **Many hashing methods**
  - Truncation
  - Division
  - Mid-square
  - Folding
  - Digit analysis
  - and so on

- **Truncation method**
  - Ignore part of the key and use the rest as the array index (converting non-numeric parts)
  - Example
    - If students have an 9-digit identification number, take the last **3** digits as the table position
    - e.g. 925371**622** becomes 622

- **Division method**
  - Hash function ***f*(*k*) = *k* % *b***
    - Requires only a single division operation (quite fast)
  - Certain values of *b* should be avoided
    - if $b=2^p$, then *f*(k) is just the *p* lowest-order bits of *k*; the hash function does not depend on all the bits
  - It's a good practice to set the table size *b* to be a **prime number**

- **Division Method**

Example: calculate the hash values of keys 1234 and 5462.

Solution: Setting b = 97, hash values can be calculated as:

f(1234) = 1234 % 97 = 70

f(5642) = 5642 % 97 = 16

- **Mid-square method**
  - Middle of square method
  - This method squares the key value, and then takes out the number of bits from the middle of the square
  - The mid-square method is a good hash function which works in two steps:

  Step 1: Square the value of the key. That is, find k^2 .

  Step 2: Extract the middle r digits of the result obtained in Step 1.

  - The number of bits to be used to obtain the bucket address depends on the table size
    - If *r* bits are used, the range of values is 0 to 2^r-1

- **Mid-Square Method**
- The algorithm works well because most or all digits of the key value contribute to the result.
- In the mid-square method, the same r digits must be chosen from all the keys. Therefore, the hash function can be given as:

$$f(k) = s$$

where s is obtained by selecting r digits from k^2

- **Mid-square method**
  - Example
    - consider records whose keys are 4-digit numbers in base 10
    - The goal is to hash these key values to a table of size 100
    - This range is equivalent to two digits in base 10, that is, r = 2
    - If the input is the number 4567, squaring yields an 8-digit number, 20857489
    - The middle two digits of this result are 57

- **Mid-Square Method**

Example: Calculate the hash value for keys 1234 and 5642 using the mid-square method. The hash table has 100 memory locations.

Solution: Note that the hash table has 100 memory locations whose indices vary from 0 to 99. This means that only two digits are needed to map the key to a location in the hash table, so r = 2.

- When k = 1234, k^2 = 1522756, f (1234) = 27
- When k = 5642, k^2 = 31832164, f (5642) = 21

(Observe that the 3rd and 4th digits starting from the right are chosen.)

- **Mid-Square Method**

Example: Calculate the hash value for keys 1234 and 5642 using the mid-square method. The hash table has 1000 memory locations.

Solution: Note that the hash table has 1000 memory locations whose indices vary from 0 to 999. This means that only two digits are needed to map the key to a location in the hash table, so r = 3.

- When k = 1234, k^2 = 1522756, f (1234) = 227
- When k = 5642, k^2 = 31832164, f (5642) = 832

- **Folding method**
  - Partition the key into several parts of the same length except for the last
  - These parts are then added together to obtain the hash address for the key
  - Two ways of carrying out this addition
    - shift folding
    - folding and reverse

Shift-folding

123 203 241 112 020

```
 123
 203
 241
 112
 020
----
 699
```

Folding and reverse

123 203 241 112 020

```
 123
 302
 241
 211
 020
----
 897
```

Example

- **Folding Method**

  The folding method works in the following two steps:

  - Step 1: Divide the key value into a number of parts. That is, divide k into parts k_1, k_2,...,k_n where each part has the same number of digits except the last part which may have lesser digits than the other
  - Step 2: Add the individual parts. That is, obtain the sum of $k_1+k_2+\ldots+k_n$. The hash value is produced by ignoring the last carry, if any.

The number of digits in each part of the key will vary depending upon the size of the hash table.

For example, if the hash table has a size of 1000, then there are 1000 locations in the hash table. To address these 1000 locations, we need at least three digits; therefore, each part of the key must have three digits except the last part which may have lesser digits.

- **Folding Method**

Example: Given a hash table of 100 locations, calculate the hash value using folding method for keys 5678, 321, and 34567.

Solution: Since there are 100 memory locations to address, we will break the key into parts where each part (except the last) will contain two digits.

The hash values can be obtained as shown below:

key	5678	321	34567
Parts	56 and 78	32 and 1	34, 56 and 7
Sum	134	33	97
Hash value	34 (ignore the last carry)	33	97

- **Folding Method**

Example: Given a hash table of 1000 locations, calculate the hash value using folding method for keys 467824, 34041, and 5217704.

- **Folding Method**

Example: Given a hash table of 1000 locations, calculate the hash value using folding method for keys 467824, 34041, and 5217704.

Solution: Since there are 1000 memory locations to address, we will break the key into parts where each part (except the last) will contain three digits.

The hash values can be obtained as shown below:

key	467824	34041	5217704
Parts	467 and 824	340 and 41	521, 770 and 4
Sum	1291	381	1295
Hash value	291 (ignore the last carry)	381	295 (ignore the last carry)

Digit analysis method

- All the identifiers/keys in the table are known in advance
- The index is formed by extracting, and then manipulating specific digits from the key
- For example, the key is 925371622, we select the digits from 5 through 7 resulting 537
- The manipulation can then take many forms
  - Reversing the digits (735)
  - Performing a circular shift to the right (753)
  - Performing a circular shift to the left (375)
  - Swapping each pair of digits (357)

- **Multiplication Method**

The steps involved in the multiplication method are as follows:

Step 1: Choose a constant A such that 0 < A < 1.

Step 2: Multiply the key k by A.

Step 3: Extract the fractional part of kA.

Step 4: Multiply the result of Step 3 by the size of hash table (m).

The hash function can be given as:

$$\mathbf{f(k)} = \lfloor \boldsymbol{m}\mathbf{(kA\ mod\ 1)} \rfloor$$

where:

- (kA mod 1) gives the fractional part of kA
- m is the total number of indices in the hash table.

- **Multiplication Method**
- The greatest advantage of this method is that it works practically with any value of A.
- Although the algorithm works better with some values, the optimal choice depends on the characteristics of the data being hashed. Knuth has suggested that the best choice of A is " (sqrt5 – 1) /2 = 0.6180339887

Example: Given a hash table of size 1000, map the key 12345 to an appropriate location in the hash table.

Solution We will use $A = 0.618033$, $m = 1000$, and $k = 12345$

$$
\begin{aligned}
h(12345) &= \lfloor 1000\ (12345 \times 0.618033 \bmod 1) \rfloor \\
&= \lfloor 1000\ (7629.617385 \bmod 1) \rfloor \\
&= \lfloor 1000\ (0.617385) \rfloor \\
&= \lfloor 617.385 \rfloor \\
&= 617
\end{aligned}
$$

- **Multiplication Method**

Example: Given a hash table of size 1000, map the key 216734 to an appropriate location in the hash table.

- **Multiplication Method**

Example: Given a hash table of size 1000, map the key 216734 to an appropriate location in the hash table.

Solution: We will use A = 0.618033, m = 1000, and k = 216734

$$
\begin{aligned}
f(216734) &= \lfloor 1000 \text{ x } (216734 \text{ x } 0.618033 \bmod 1) \rfloor \\
&= \lfloor 1000 \text{ x } (133948.7642 \bmod 1) \rfloor \\
&= \lfloor 1000 \text{ x } 0.7642 \rfloor \\
&= \lfloor 764.2 \rfloor \\
&= 764
\end{aligned}
$$

- **Hash Function Implementations**
  - A generic hashing function does not exist
  - However, there are several forms of a hash function
  - Let's discuss some specific hash function implementations

```
typedef unsigned          HASH_VALUE
typedef unsigned short  KEY_TYPE

HASH_VALUE  ModulusHashFunc (KEY_TYPE key){
          return (key % HT_SIZE);
}
```

- **Folding hash function for integers**

```
typedef unsigned long int  KEY_TYPE

HASH_VALUE Fold_Integer_Hash (KEY_TYPE key){
        return ( (key / 10000 + key % 10000) % HT_SIZE);
}
```

High-order digits

Low-order digits

New integer

Source code above is for a 8-digit integer key

- **Folding hash function for integers**

```
typedef unsigned long int  KEY_TYPE

HASH_VALUE Fold_Integer_Hash (KEY_TYPE key){
        return ( (key / 10000 + key % 10000) % HT_SIZE);
}
```

Example: key = 87629426, HT_SIZE = 251

hash value = (87629426 / 10000 + 87629426 % 10000) % 251

hash value = (8762 + 9426) % 251 = 116

key = 87629426

High-order digits Low-order digits

- **Folding hash function for pointer-based character strings**

  Useful for applications involving symbols and names.

  Folding hash → adds ASCII values of each character and takes the modulus with respect to the hash table size.

```
typedef char  *KEY_TYPE

HASH_VALUE Fold_String_Hash (KEY_TYPE key){
        unsigned sum_ascii_value = 0;

        while (*key != '\0')
                sum_ascii_values += *key++;

        return (sum_ascii_values % HT_SIZE);
}
```

- **Folding hash function for pointer-based character strings**

Example:

key = “PRATIVA”, HT_SIZE = 31

hash value = (P + R + A + T + I + V + A) % 31

hash value = (80 + 82 + 65 + 84 + 73 + 86 + 65) % 31

hash value = 8

- **Digit Analysis-Based Folding**

```
static unsigned DigitFoldStringHash (char *key){
        unsigned long hash;

        hash = ( (key[0] ^ key[3]) ^ (key[1] ^ key[2]) ) % HT_SIZE;

        return (hash);
}
```

- **Collision Resolution/Overflow Handling**
  - An overflow occurs when the home bucket for a new pair **(key, element)** is full
  - Methods of solving collisions/overflows
    - **Open Addressing**
      - Insert the element into the next free position in the table
    - **Separate Chaining**
      - Each table position is a linked list

- **Open addressing**
  - relocate the key *k* to be inserted if it collides with an existing key. That is, we store *k* at an entry different from *hash_table*[*f*(*k*)].
- **Two issues arise**
  - what is the relocation scheme?
  - how to search for k later?
- **Common methods for resolving a collision in open addressing**
  - Linear probing
  - Quadratic probing
  - Double hashing
  - Rehashing

- **Open Addressing**
  - To insert a key k, compute $f_0(k)$. If *hash_table*[$f_0(k)$] is empty, insert it there.
  - If collision occurs, probe alternative cell $f_1(k)$, $f_2(k)$, ... until an empty cell is found
    - $f_i(k) = (f(k) + g(i)) \% b$, with $g(0) = 0$ (b is the size of the hash table)
    - g: **collision resolution strategy**

- **Linear Probing**
  - The simplest approach to resolve a collision is linear probing.
  - **g(i) =i**
    - cells are probed **sequentially** (with wraparound)
    - $f_i(k) = (f(k) + i) \% b$

  - **Insertion**
    - Let k be the new key to be inserted. We compute f(k)=k % b
    - For i = 0 to b-1
      - – compute L = (f(k) + i) % b
      - – hash_table[L] is empty, then we put k there and stop
    - If we cannot find an empty entry to put *k*, it means that the table is full and we should report an error

- **Linear Probing – Insert**
  - divisor = b (number of buckets) = 17
  - Home bucket = f(key) = key % 17
- Insert pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

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

Insertion

Let k be the new key to be inserted. We compute f(k)=k % b

For i = 0 to b-1

  compute L = (f(k) + i) % b

  hash_table[L] is empty, then we put k there and stop

If we cannot find an empty entry to put *k*, it means that the table is full and we should report an error

- **Linear Probing – Insert**
  - divisor = b (number of buckets) = 17
  - Home bucket = f(key) = key % 17
- Insert pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
34	**0**	**45**				**6**	**23**	**7**			**28**	**12**	**29**	**11**	**30**	**33**

- **Linear Probing – Insert**

```
void linear_insert(element item, element ht[]){
    int i, hash_value;
    i = hash_value = hash(item.key);
    while(strlen(ht[i].key)) {
        if (!strcmp(ht[i].key, item.key)) {
            fprintf(stderr, "Duplicate entry\n");
            exit(1);
        }
        i = (i+1)%TABLE_SIZE;
        if (i == hash_value)
            fprintf(stderr, "The table is full\n"); exit(1);
    }
    ht[i] = item;
}
```

- **Data Structure for Hash Table**

```
#define MAX_CHAR  10
#define TABLE_SIZE  13
typedef struct {
    char key[MAX_CHAR];
    /* other fields */
} element;
element hash_table[TABLE_SIZE];
```

- **Linear Probing – Delete**

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

- **Delete(0)**

0				4				8				12				16
34		45				6	23	7			28	12	29	11	30	33

- Search cluster for pair (if any) to fill vacated bucket.

0				4				8				12				16
34	45					6	23	7			28	12	29	11	30	33

- **Linear Probing – Delete**
  - **Delete(34)**

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

0				4				8				12				16
	0	45				6	23	7			28	12	29	11	30	33

- Search cluster for pair (if any) to fill vacated bucket.

0				4				8				12				16
0		45				6	23	7			28	12	29	11	30	33

0				4				8				12				16
0	45					6	23	7			28	12	29	11	30	33

- **Linear Probing – Delete**
  - **Delete(29)**

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

0				4				8				12				16
34	0	45				6	23	7			28	12		11	30	33

  - Search cluster for pair (if any) to fill vacated bucket

0				4				8				12				16
34	0	45				6	23	7			28	12	11		30	33

0				4				8				12				16
34	0	45				6	23	7			28	12	11	30		33

0				4				8				12				16
34	0					6	23	7			28	12	11	30	45	33

- **Performance Of Linear Probing**

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

- Worst-case find/insert/erase time is **Θ(n)**, where n is the number of pairs in the table
- This happens when all pairs are in the same cluster (the same index/bucket)

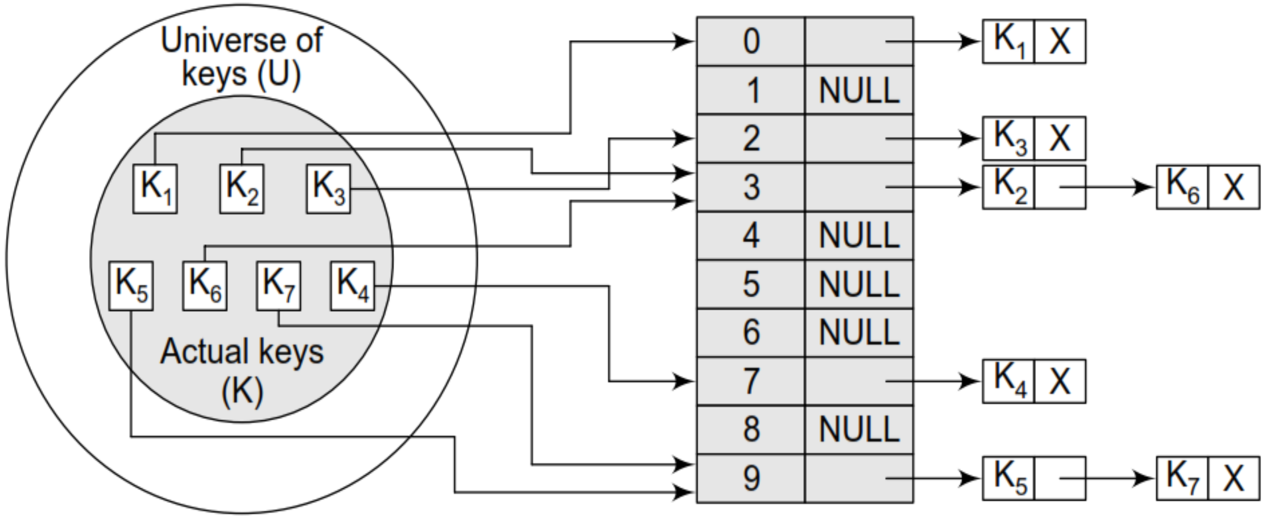
- **Quadratic Probing**
  - Linear probing searches buckets (f(x)+i)%b
  - Quadratic probing uses a quadratic function of *i* as the increment
  - Examine buckets f(x), $(f(x)+i^2)\%b$, $(f(x)-i^2)\%b$, for $1<=i<=(b-1)/2$
  - b is a prime number of the form 4j+3, j is an integer

- **Random Probing**
  - Random Probing works incorporating with random numbers
    - f(x) = (f'(x) + S[i]) % b
      - S[] is a table with size b-1
      - S[i] is a random permuation of integers [1,b-1]

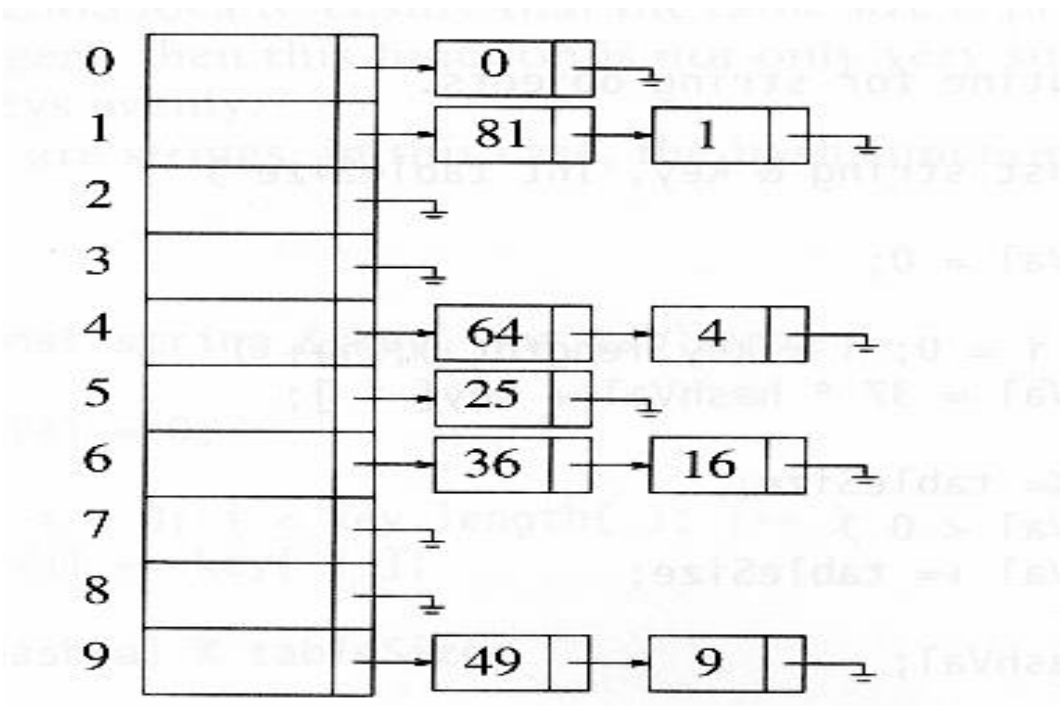
- **Double hashing**
  - Double hashing is one of the best method for dealing with collisions
  - If the slot is full, then a second hash function (which is different from the first one) is calculated and combined with the first hash function
    - $f(k, i) = (f_1(k) + i\, f_2(k))\ \%\ b$

- **Rehashing**
  - Enlarging the Table
  - To rehashing
    - Create a new table of double the size (adjusting until it is again prime)
    - Transfer the entries in the old table to the new table, by recomputing their positions (using the hash function)
  - Rehashing when the table is completely full

- **Separate Chaining**
  - Instead of a hash table, we use a table of linked list
  - keep a linked list of keys that hash to the same value



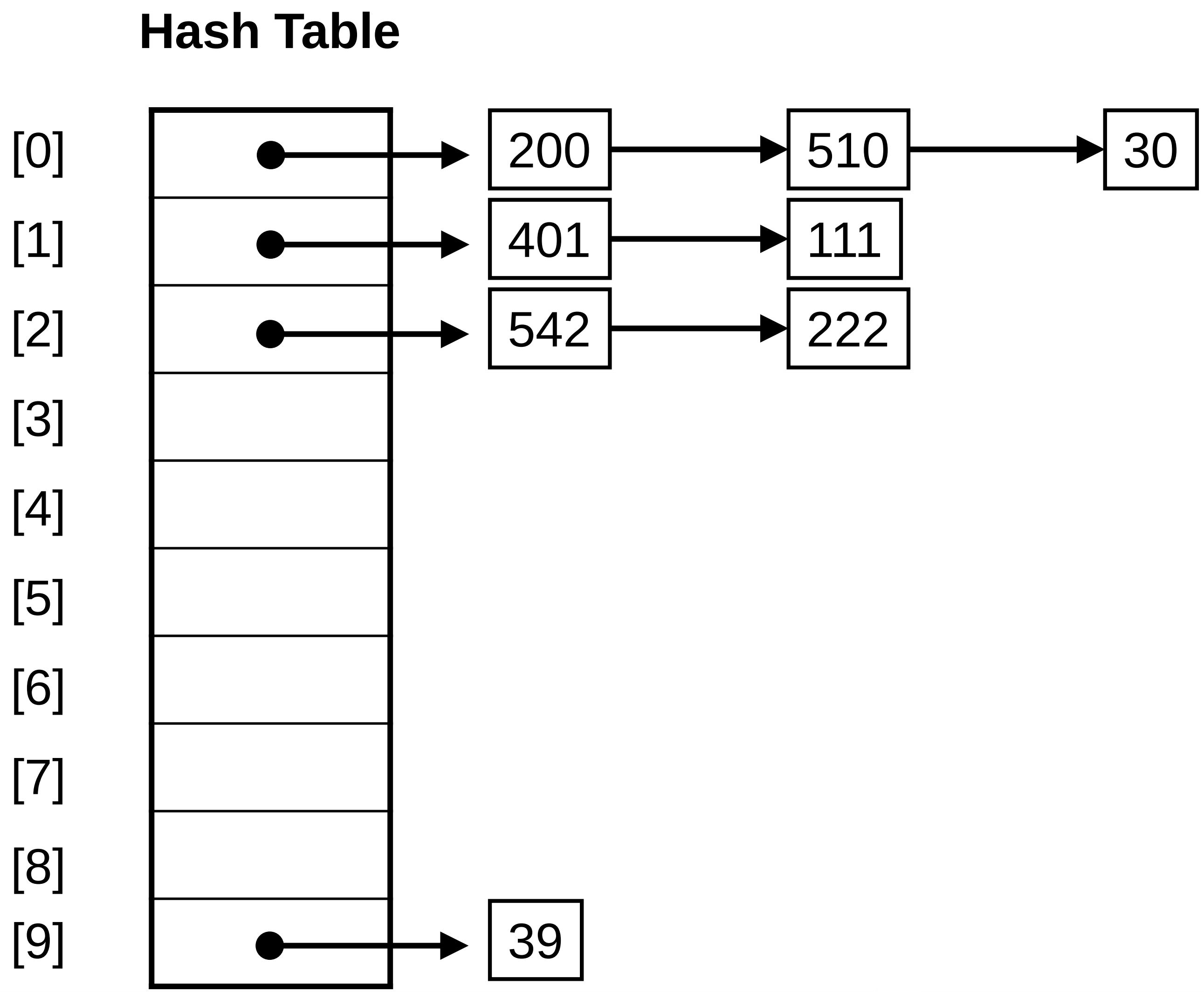
$f(k) = k \% 10$

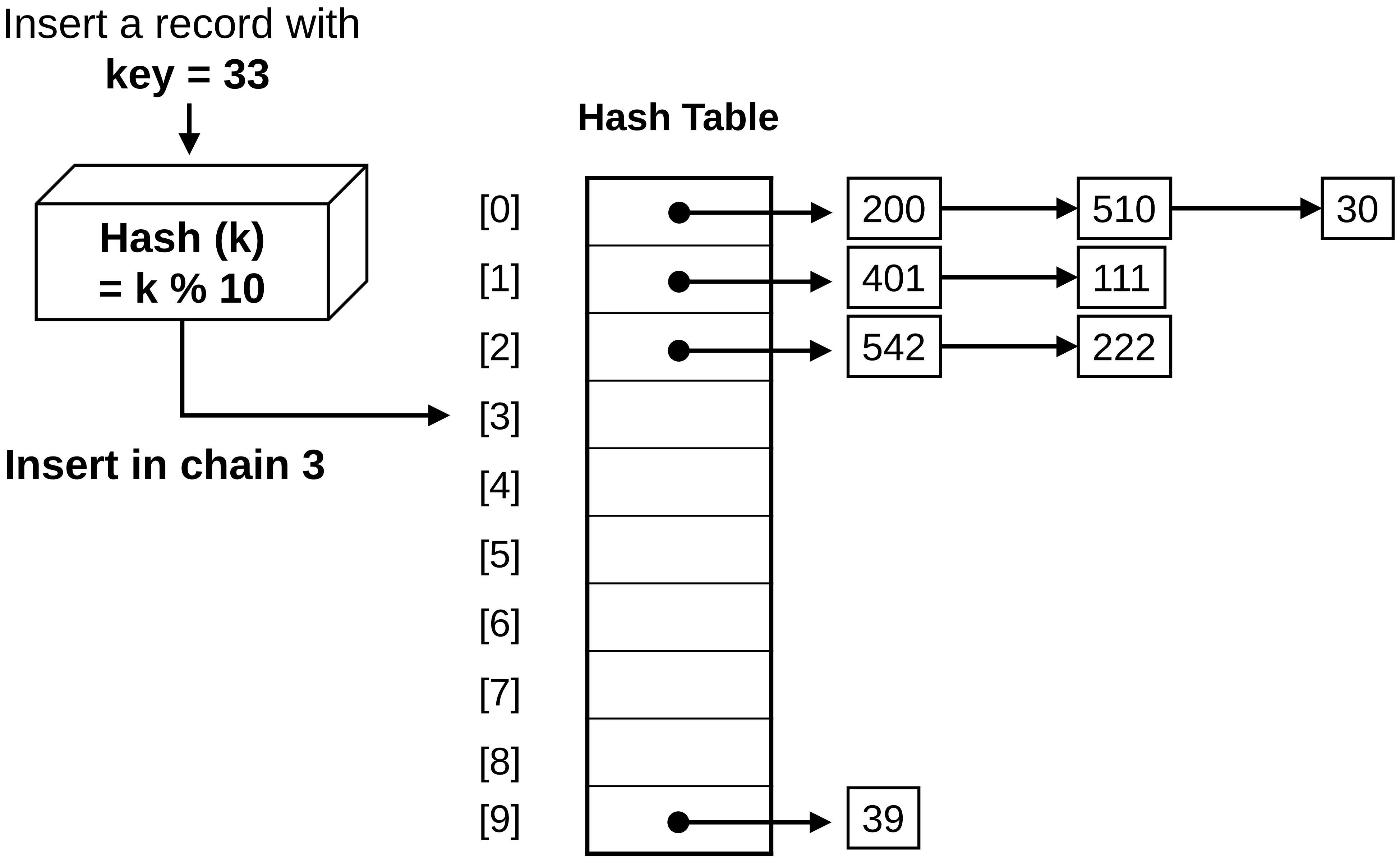


- **Separate Chaining**
  - To insert a key k
    - Compute f(k) to determine which list to traverse
    - If hash_table[f(k)] contains a null pointer, initiatize this entry to point to a linked list that contains k alone
    - If hash_table[f(k)] is a non-empty list, we add k at the beginning of this list
  - To delete a key k
    - compute f(k), then search for k within the list at hash_table[f(k)]. Delete k if it is found.

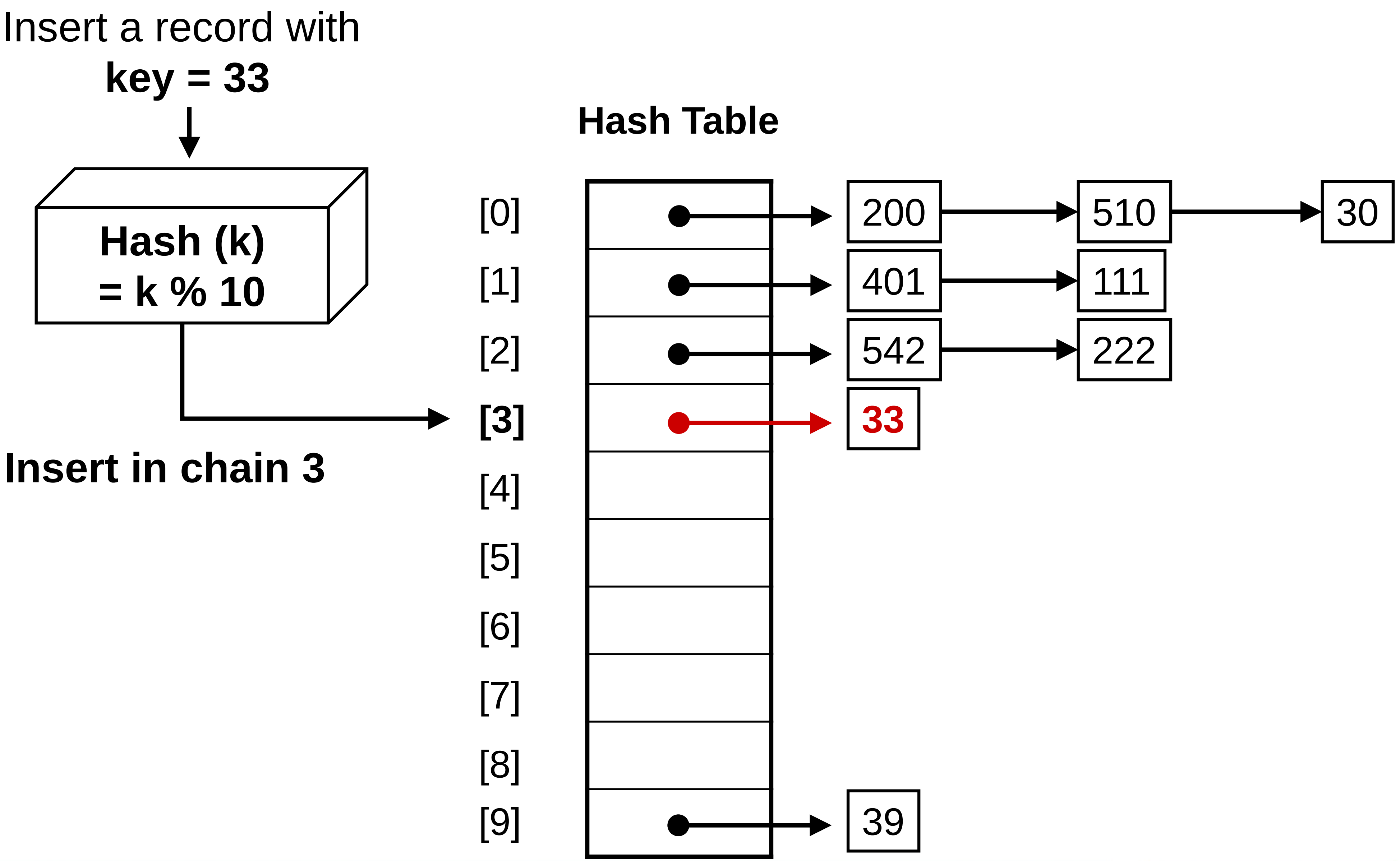
- **Separate Chaining**
  - If the hash function works well, the number of keys in each linked list will be a small constant
  - Therefore, we expect that each search, insertion, and deletion can be done in constant time
  - Disadvantage
    - Memory allocation in linked list manipulation will slow down the program
  - Advantage
    - Deletion is easy
    - Array size is not a limitation

- **Example**

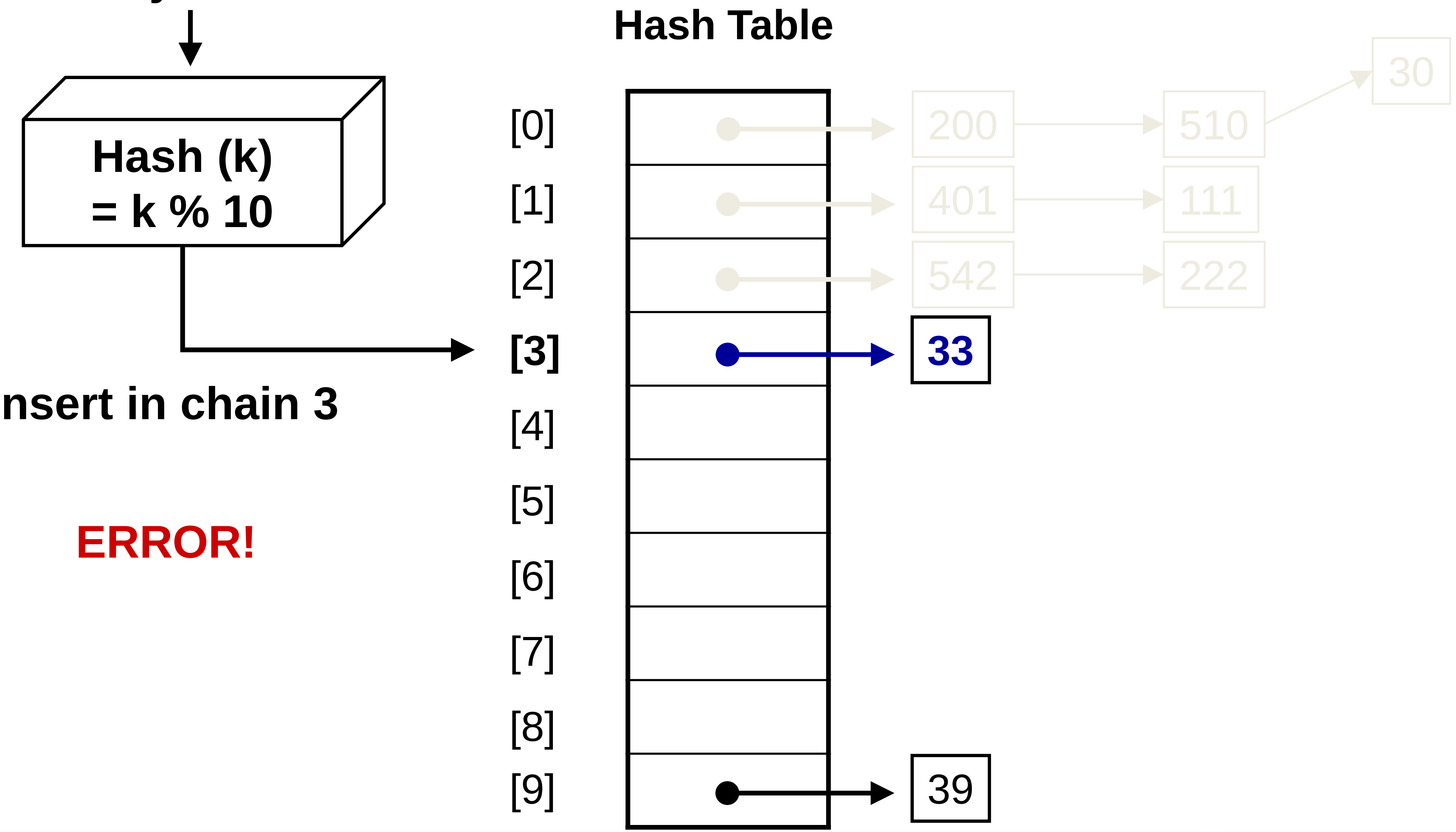




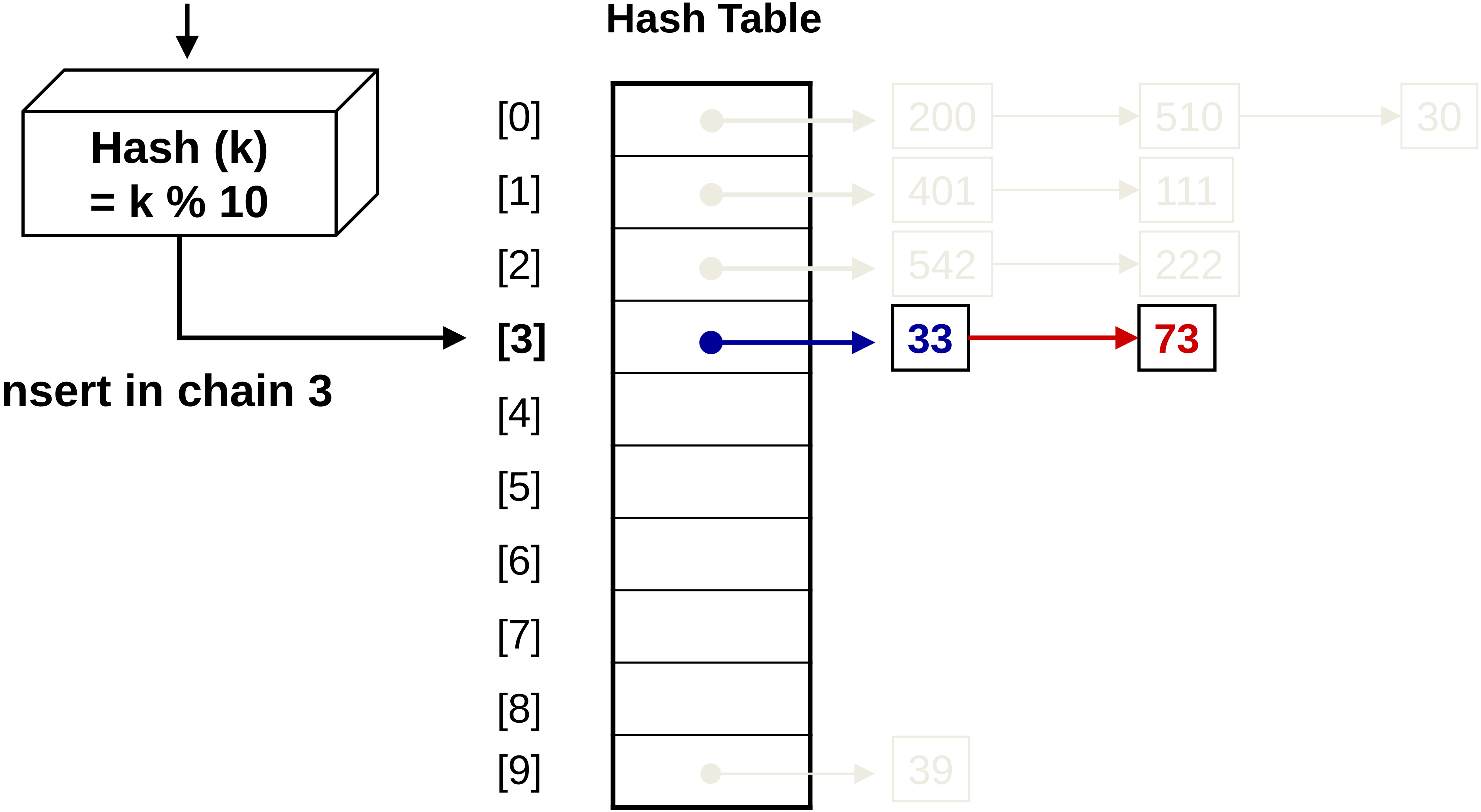
Insert a record with
key = 33
Hash (k)
= k % 10
Insert in chain 3
Hash Table
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
200
510
30
401
111
542
222
39


Insert a record with
key = 33
Hash (k)
= k % 10
Insert in chain 3
Hash Table
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
200
510
30
401
111
542
222
33
39

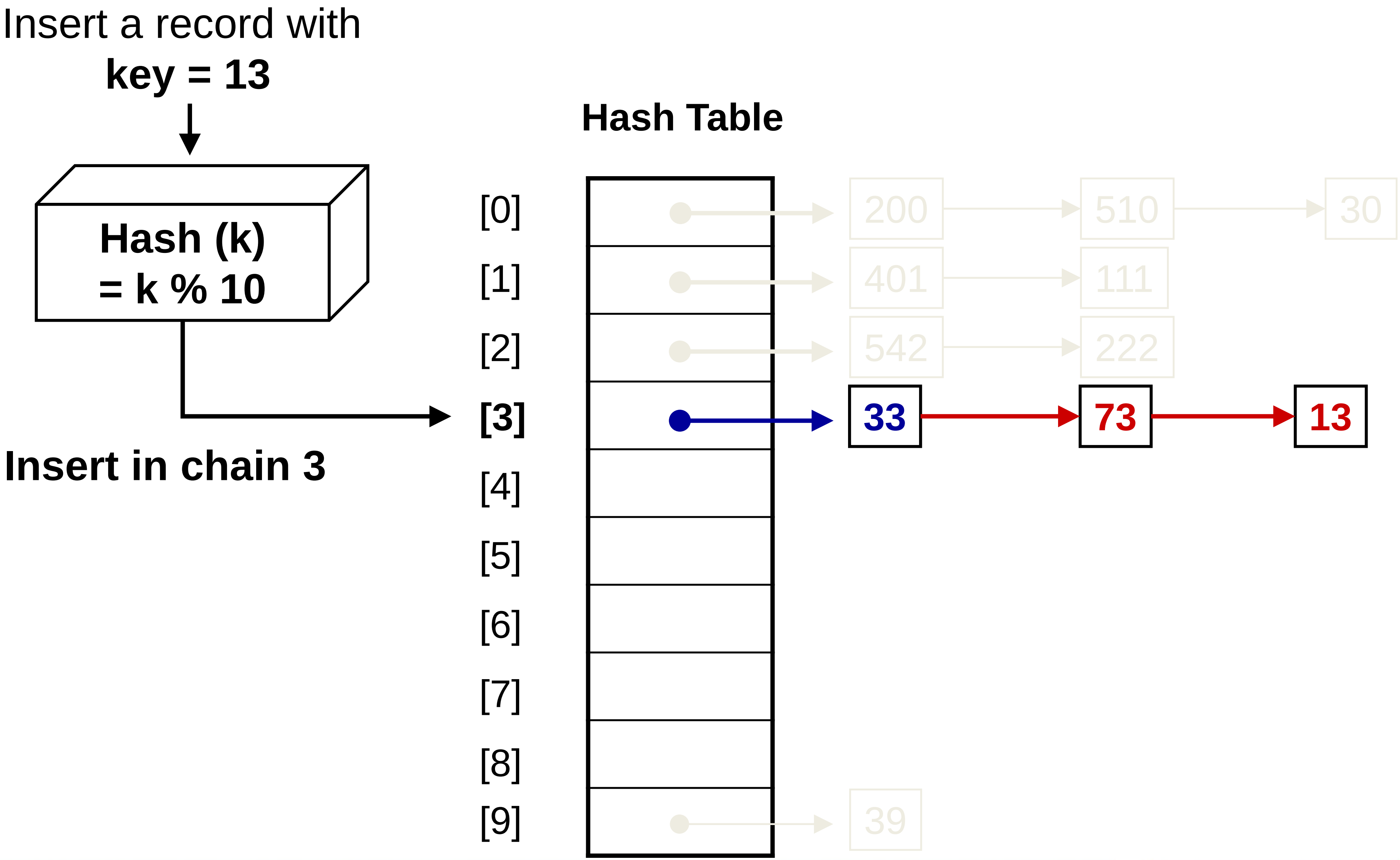
Insert a record with
key = 73

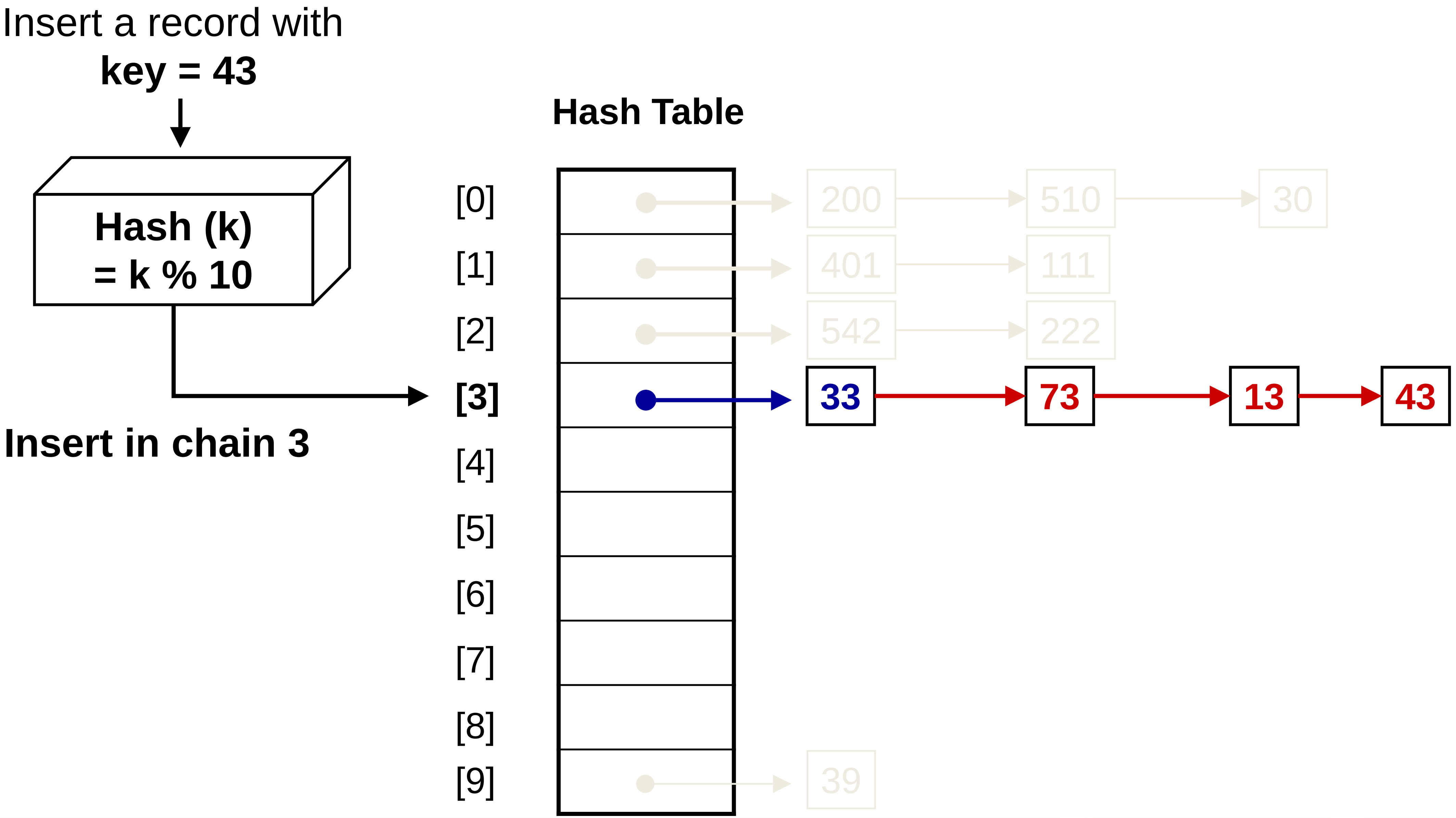


Insert a record with
key = 73



Insert in chain 3


Insert a record with
key = 13
Hash (k)
= k % 10
Insert in chain 3
Hash Table
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
200
510
30
401
111
542
222
33
73
13
39



Insert a record with
key = 43
Hash (k)
= k % 10
Insert in chain 3
Hash Table
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
200
510
30
401
111
542
222
33
73
13
43
39

- **Collision Resolution by Chaining**

Example: Insert the keys 7, 24, 18, 52, 36, 54, 11, and 23 in a chained hash table of 9 memory locations. Use f(k)= k%b (h(k) = k mod b).

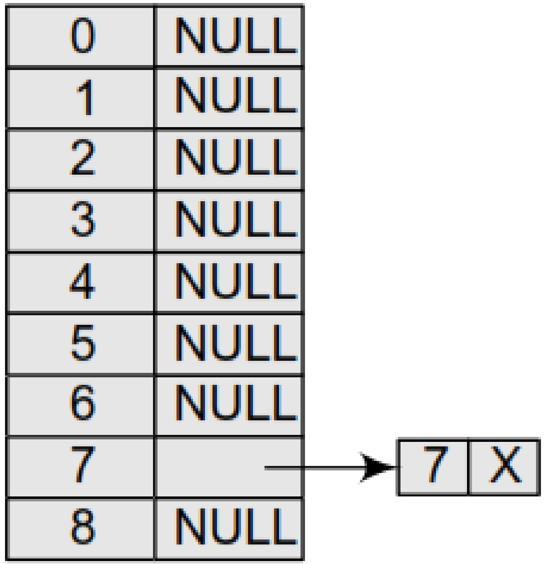
In this case, b=9. Initially, the hash table can be given as:

0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL

Step 1

```
 Key = 7
h(k) = 7 mod 9
     = 7
```

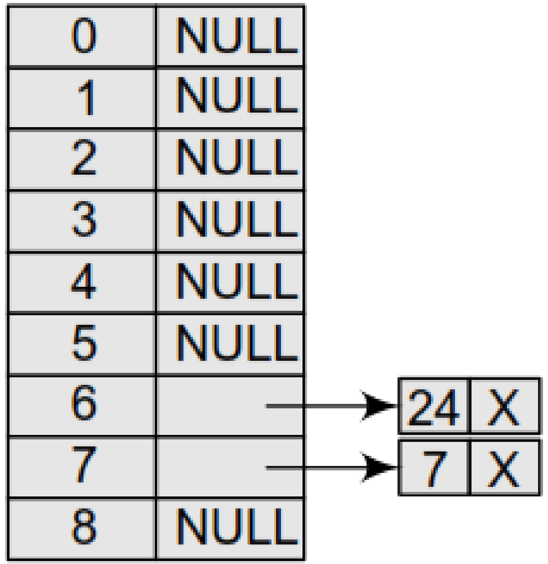
Create a linked list for location 7 and store the key value 7 in it as its only node.



Step 2

```
 Key = 24
h(k) = 24 mod 9
     = 6
```

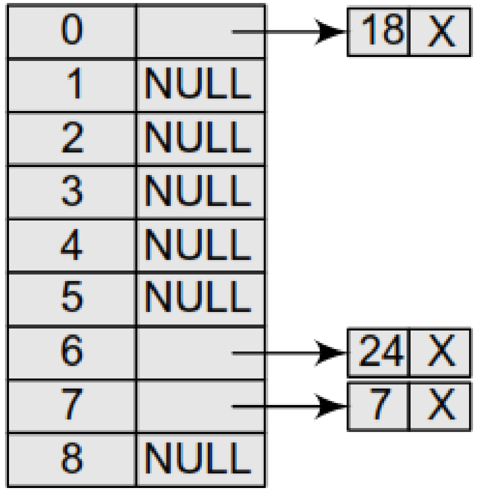
Create a linked list for location 6 and store the key value 24 in it as its only node.



Step 3

```
Key = 18
h(k)= 18 mod 9 = 0
```

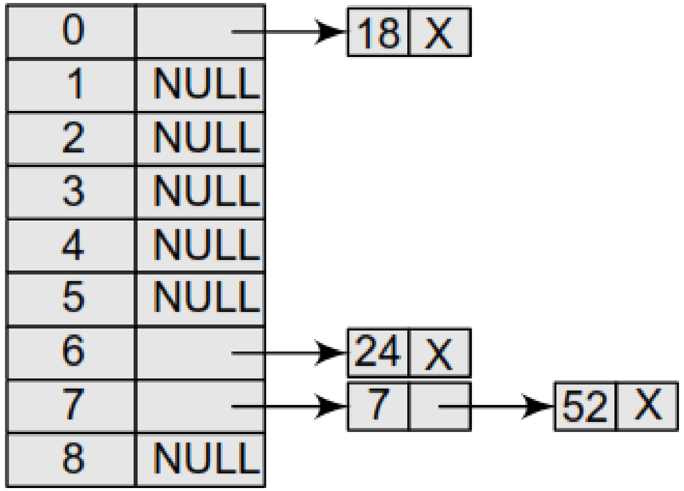
Create a linked list for location 0 and store the key value 18 in it as its only node.



Step 4

```
Key = 52
h(k) = 52 mod 9 = 7
```

Insert 52 at the end of the linked list of location 7.



Step 5:

```
Key = 36
h(k)= 36 mod 9 = 0
```

Insert 36 at the end of the linked list of location 0.

Step 6:

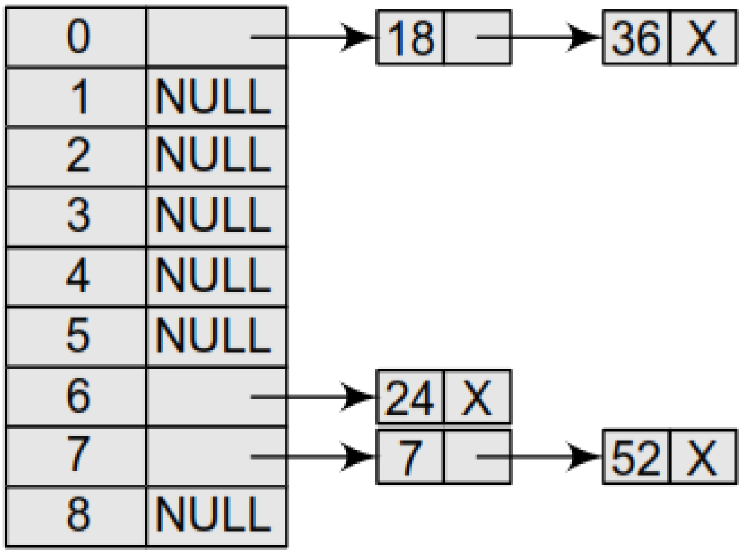
```
Key = 54
h(k) = 54 mod 9 = 0
```

Insert 54 at the end of the linked list of location 0.

Step 5: Key = 36

h(k) = 36 mod 9 = 0

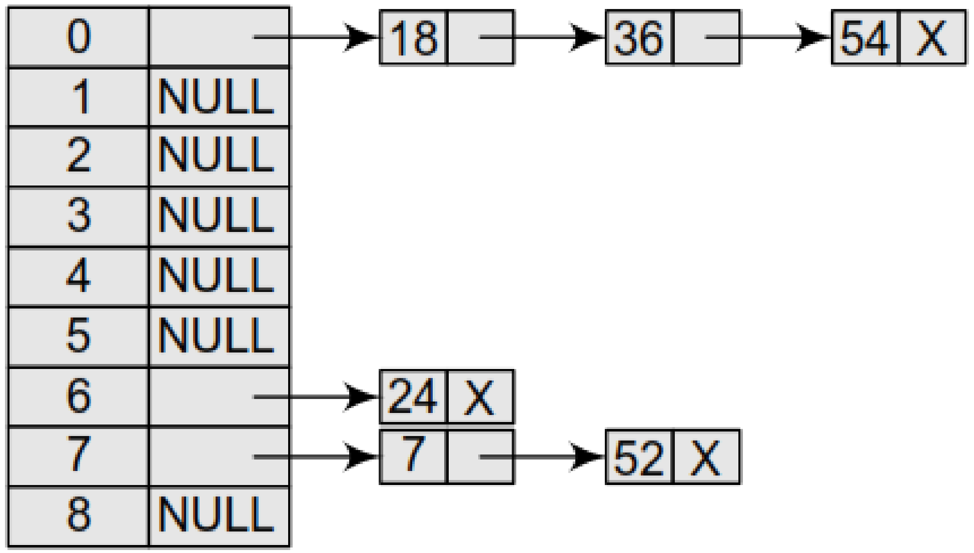
Insert 36 at the end of the linked list of location 0.



Step 6: Key = 54

h(k) = 54 mod 9 = 0

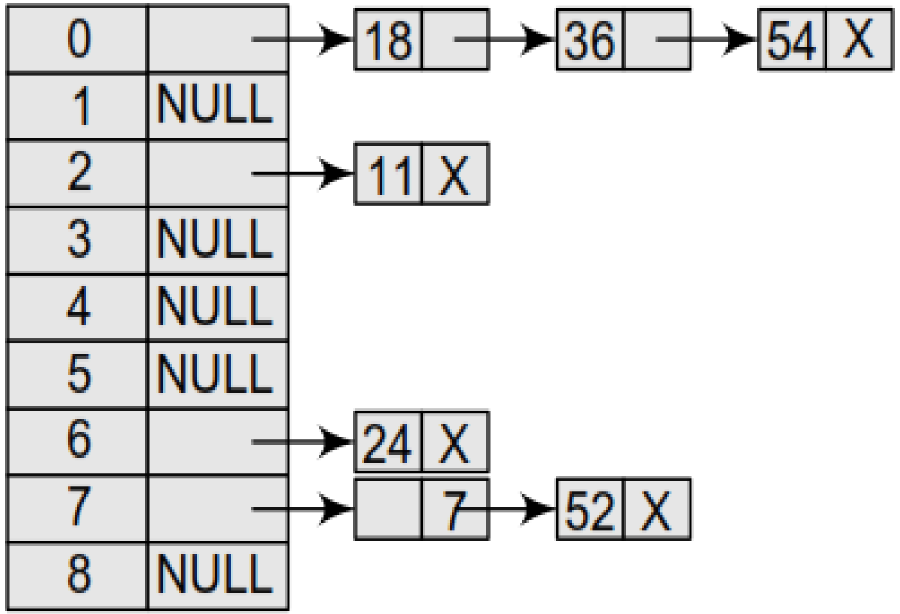
Insert 54 at the end of the linked list of location 0.



Step 7: Key = 11

h(k) = 11 mod 9 = 2

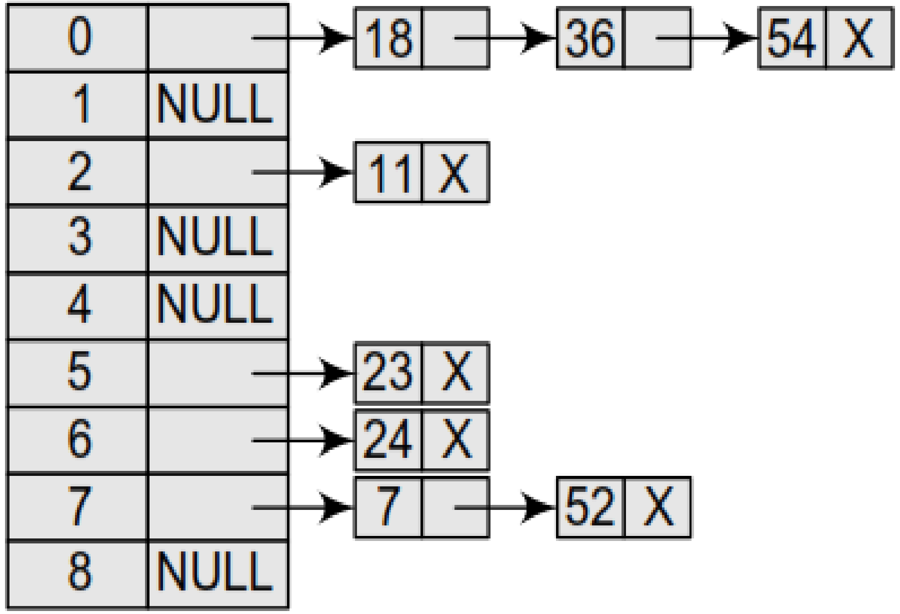
Create a linked list for location 2 and store the key value 11 in it as its only node.



Step 8: Key = 23

h(k) = 23 mod 9 = 5

Create a linked list for location 5 and store the key value 23 in it as its only node.



- **Data Structure for Chaining**

```
#define MAX_CHAR 10
#define TABLE_SIZE 13
#define IS_FULL(ptr) (!(ptr))
typedef struct {
        char key[MAX_CHAR];
        /* other fields */
} element;
typedef struct list *list_pointer;
typedef struct list {
        element item;
        list_pointer link;
};
list_pointer hash_table[TABLE_SIZE];
```

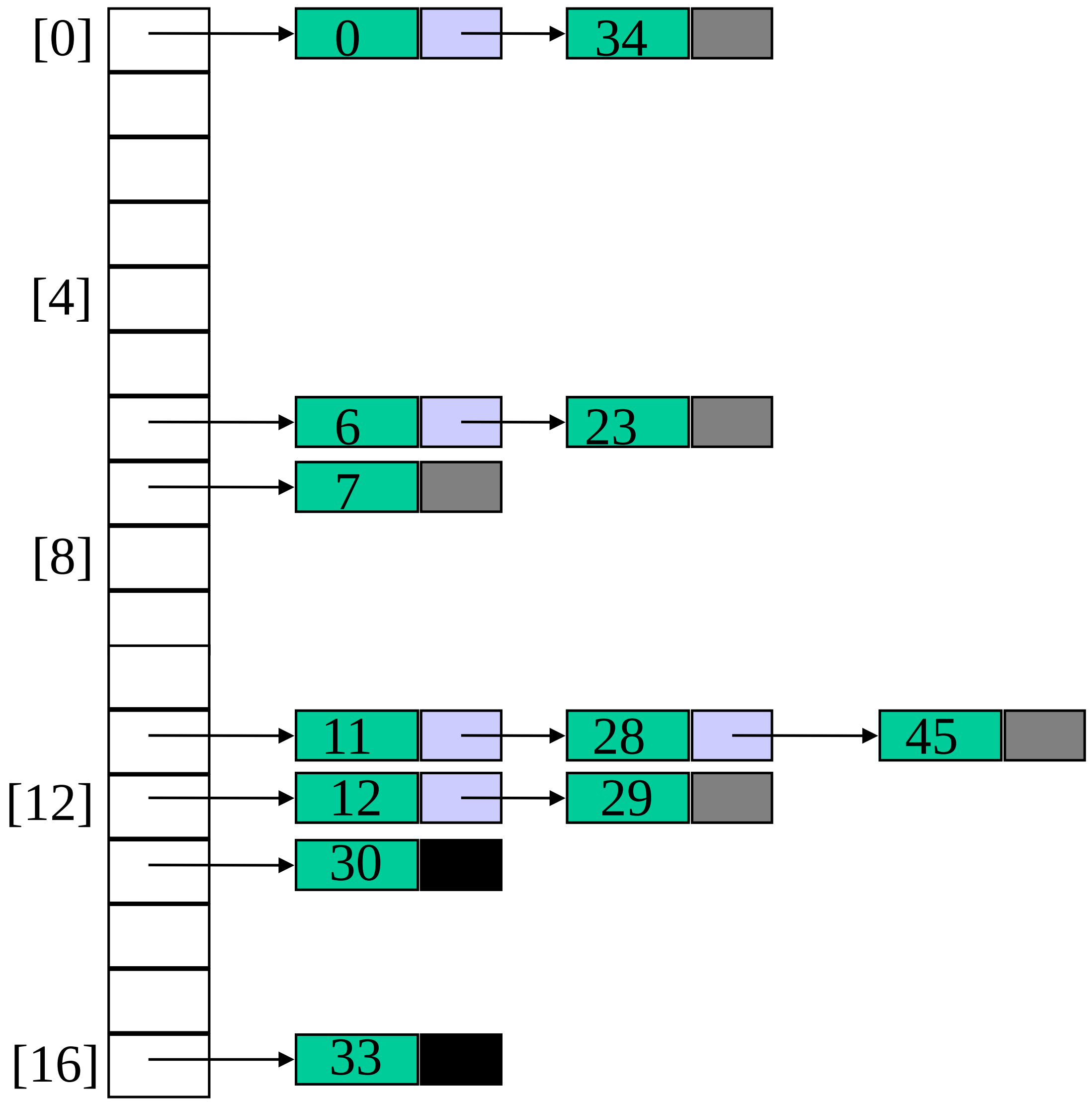
- **Separate Chaining – Insert**

```
void insert( element_type key, HASH_TABLE H ){
    position pos, new_cell; LIST L;
    pos = find( key, H );
    if( pos == NULL ) {
        new_cell = (position) malloc(sizeof(struct list_node));
        if( new_cell == NULL )   fatal_error("Out of space!!!");
        else {
            L = H->the_lists[ hash( key, H->table size ) ];
            new_cell->next = L->next;
            new_cell->element = key; /* Probably need strcpy!! */
            L->next = new_cell;
        }
    }
}
```

- **Other Implementation**

Sorted chains

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



CONTENT

- Introduction
- Static hashing
- **Dynamic hashing**

- **Dynamic hashing**
  - The number of identifiers in a hash table may vary
  - Use a small table initially; when a lot of identifiers are inserted into the table, we may increase the table size
  - When a lot of identifiers are deleted from the table, we may reduce the table size
  - This is called **dynamic hashing** or **extendible hashing**
  - Dynamic hashing usually involves databases and buckets may also be called **pages**

- We assume each page contains *p* records
- Each record is identified by a key (i.e., the identifiers in static hashing)
- Space utilization = *n*/*mp*
  - where *n* is the number of actual records
  - *m* is the number of pages reserved
  - *p* is the number of records per page

$$\frac{NumberOfRecord}{NumberOfPages * PageCapacity}$$

- **Objective: Find an extendible hashing function such that**
  - it minimizes the number of pages accessed
  - space utilization is as high as possible

- **There are two approaches**
  - Dynamic Hashing with Using Directories
  - Dynamic Hashing without Using Directories

SUMMARY

- Introduction
- Static hashing
- Dynamic hashing

Enjoy the Course…!