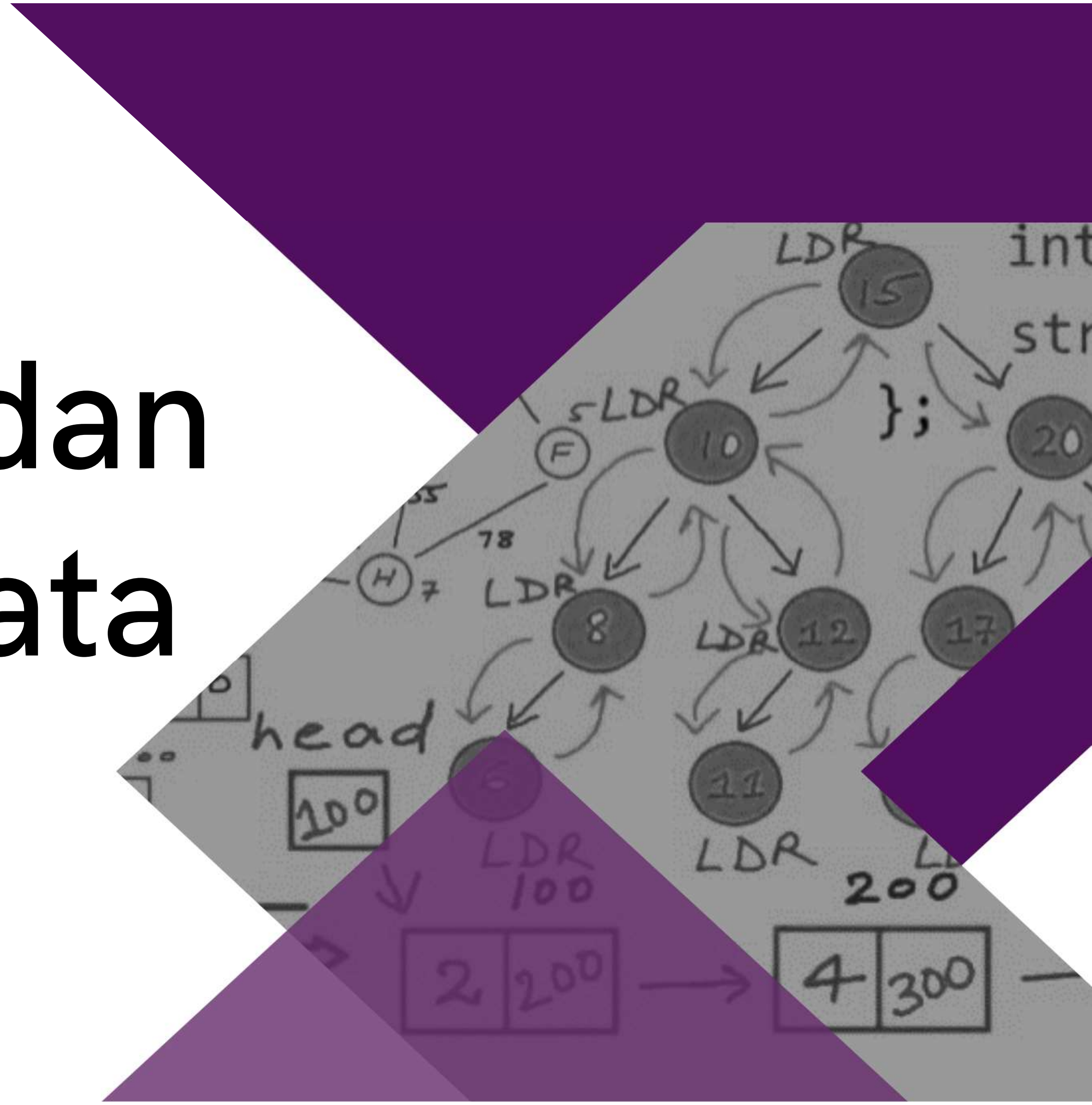
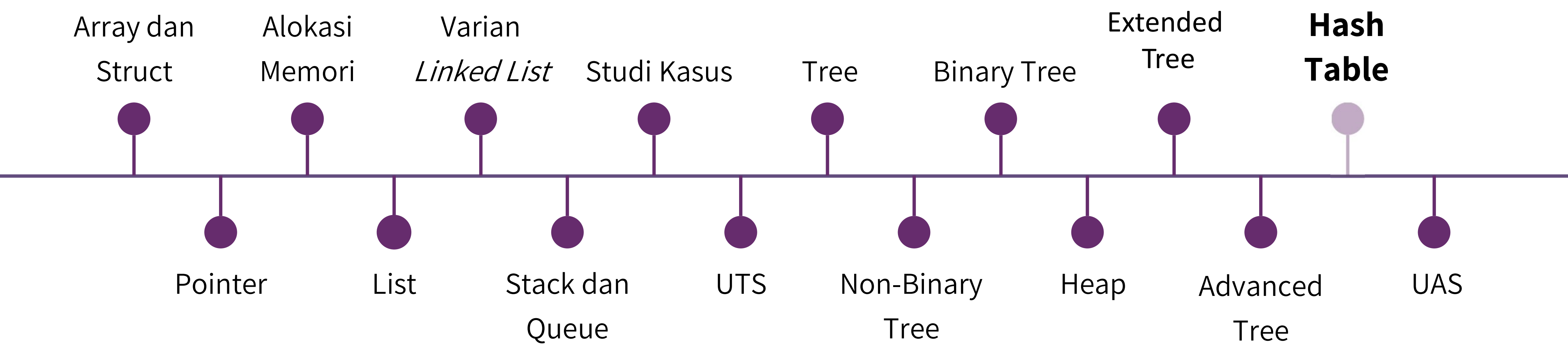
Algoritma dan Struktur Data

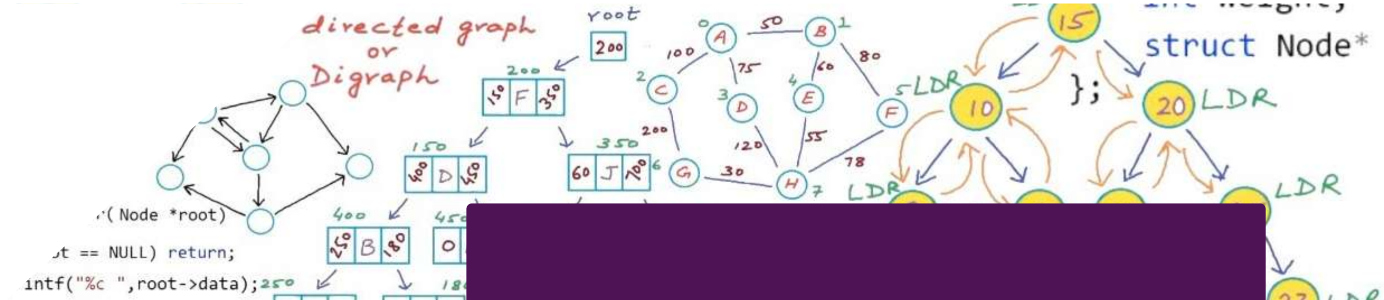


Pekan 15



Array dan Struct
Pointer
Alokasi Memori
List
Varian *Linked List*
Stack dan Queue
Studi Kasus
UTS
Tree
Non-Binary Tree
Binary Tree
Heap
Extended Tree
Advanced Tree
Hash Table
UAS

Tujuan



1	Mahasiswa memahami konsep dan implementasi struktur data dengan property tambahan berupa key.
2	Mahasiswa memahami perbedaan array dan hash table.
3	Mahasiswa mampu menerapkan hash table untuk berbagai kebutuhan komputasi dalam permasalahan dunia nyata.

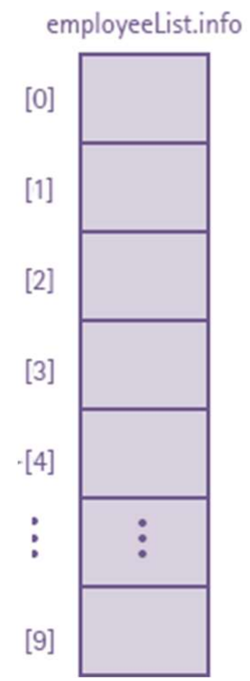
Rationale

- Unsorted list → $O(n)$
- Sorted list → $O(\log_2 n)$

Can we do better? → $O(1)$

Example

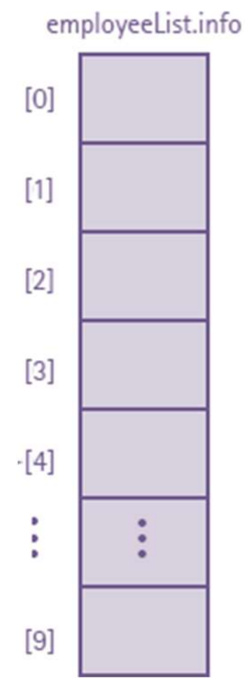
A list of employees of a fairly small company. Each of 100 employees has an ID number in the range 0 to 99, and we want to access the employee records using the key `idNum`.



Example

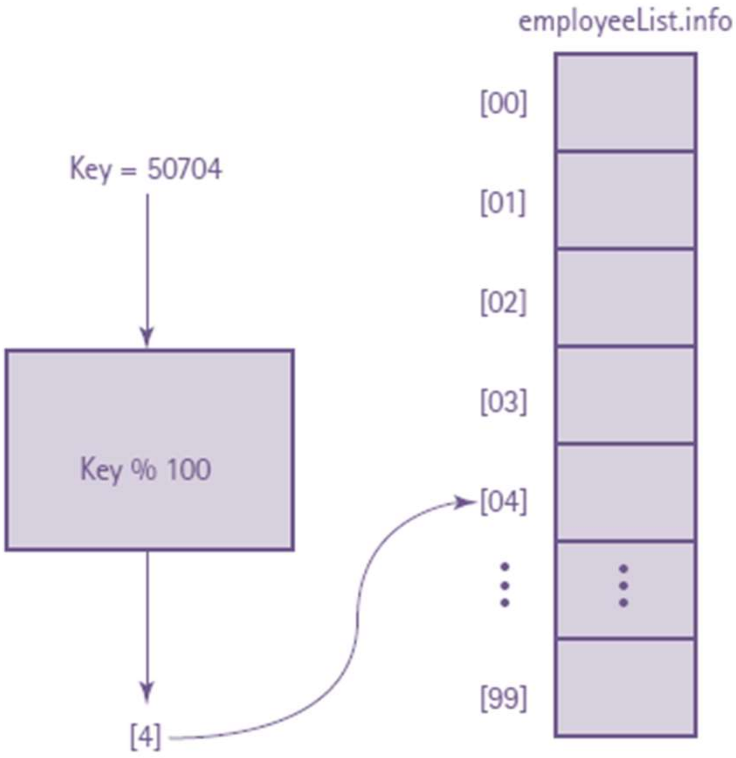
Consider a similar small company that uses its employees' five-digit ID number as the primary key.

Now the range of key values goes from 00000 to 99999.



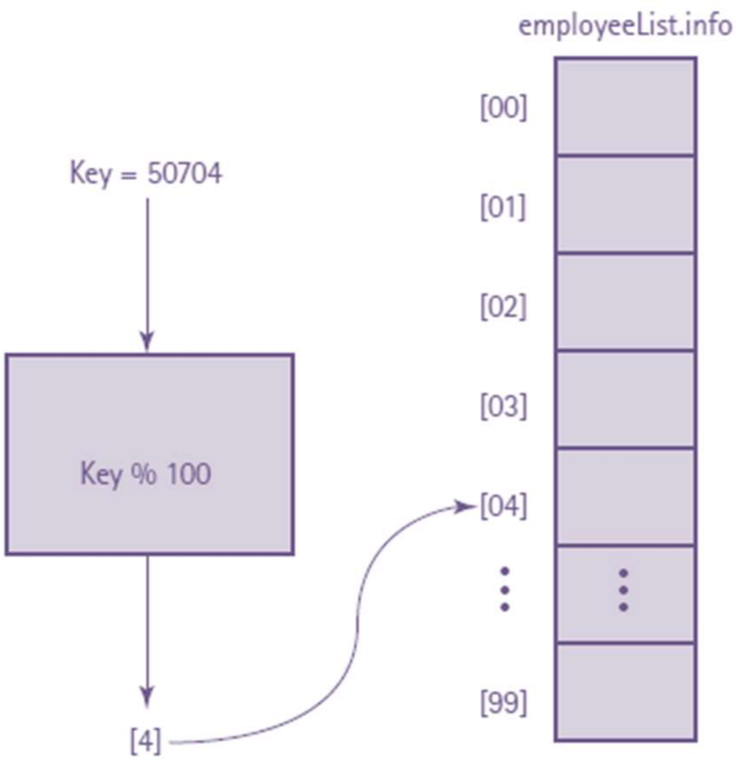
Solution

- Keep the array size down to what we need (an array of 100 elements)
- Use the last two digit of the key to identify each employee



Hashing

Hashing



- **Hash Function**: A function used to manipulate the key of an element in a list to identify its location in the list.
- **Hash Table**: The array that holds the records/elements.
- **Hashing**: The technique used for ordering and accessing elements in a list in a relatively constant amount of time by manipulating the key to identify its location in the list.

Hash Function

```
int hashing(int idNum)
{
    return(idNum % MAX_ITEM);
}
```

```
int retrieveItem(int item)
{
    int location;

    location = hashing(item);
    return (location);
}
```

```
int inserItem(int item, int employeeList[])
{
    int location;

    location = hashing(item);
    employeeList[location] = item;
    length++;
}
```

Hashing VS List

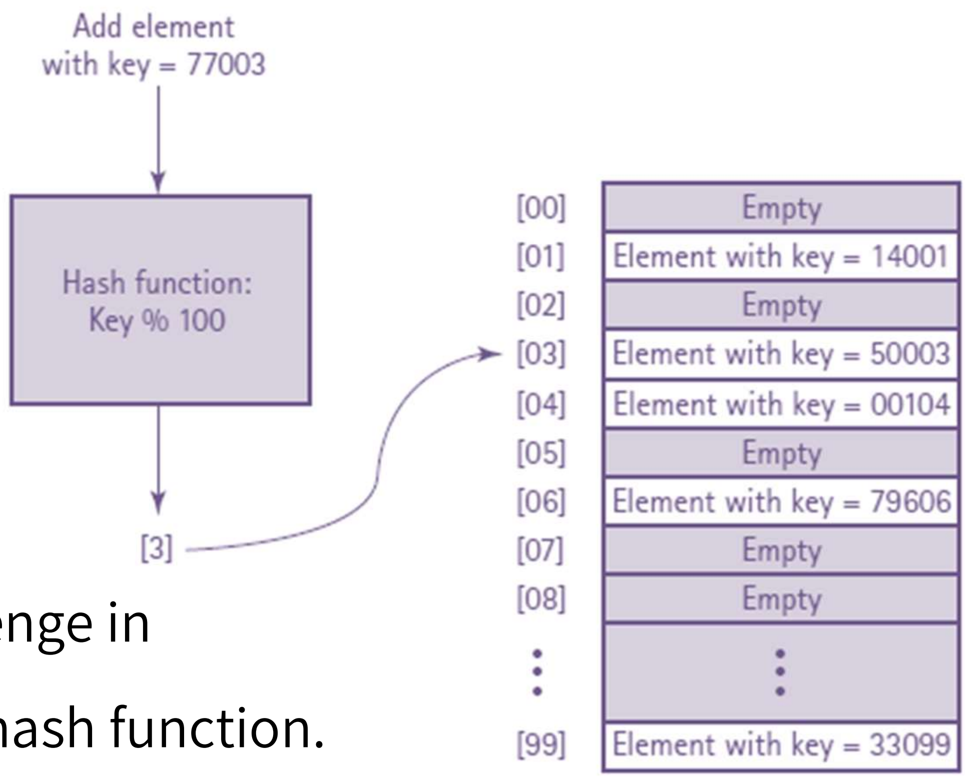
Hashed

[00]	31300
[01]	49001
[02]	52202
[03]	Empty
[04]	12704
[05]	Empty
[06]	65606
[07]	Empty
⋮	⋮

Linear

[00]	12704
[01]	31300
[02]	49001
[03]	52202
[04]	65606
[05]	Empty
[06]	Empty
[07]	Empty
⋮	⋮

Problem



The **biggest** challenge in

designing a good hash function.

Collisions

Collisions

The condition resulting when two or more keys produce the same hash location.

Hash Function

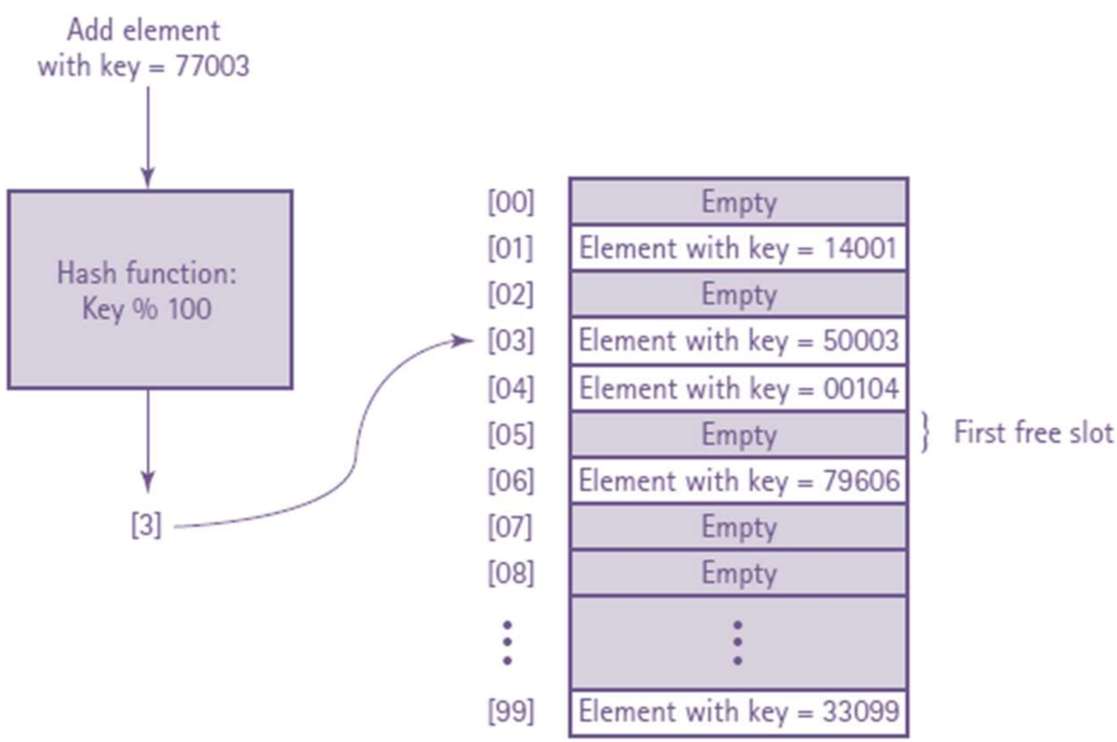
A good hash function **minimize collisions** by spreading the element uniformly throughout the array.

Because it is extremely difficult to avoid them completely.

Linear Probing

Linear Probing

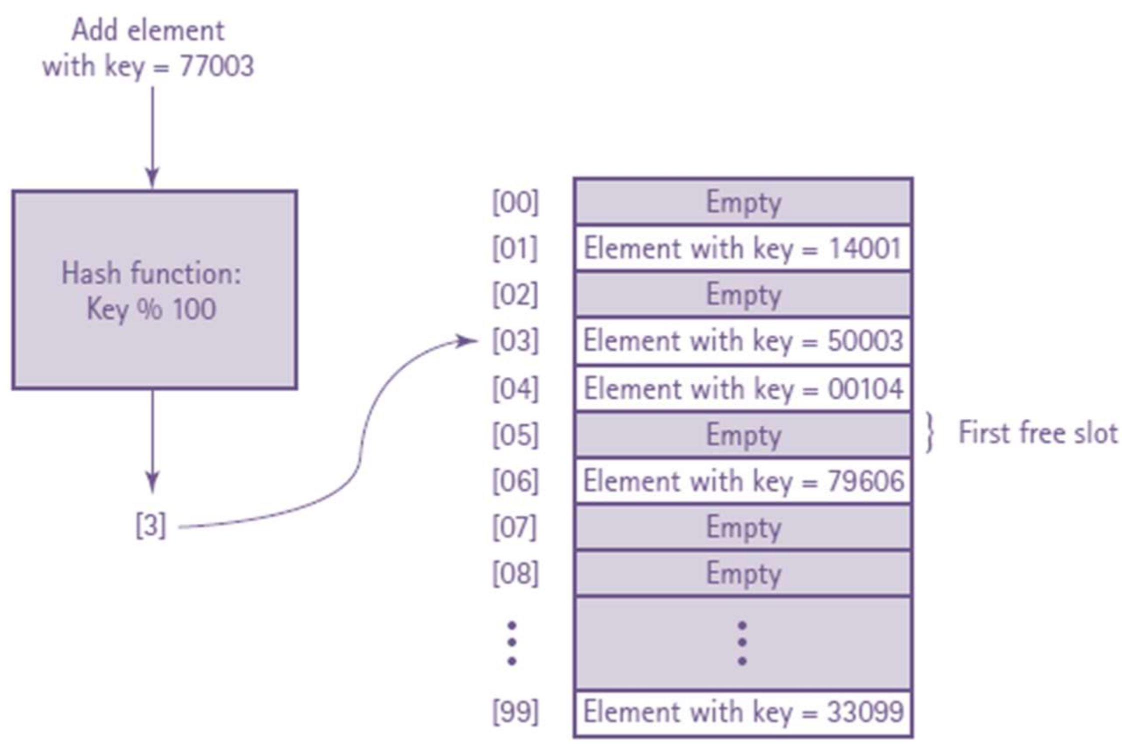
Resolving a hash collision by sequentially searching a hash table beginning at the location returned by the hash function,



Empty Array Slot

Use value:

- Syntactically legal
- Semantically illegal



Linear Probing - Insert

```
const int emptyItem = -1;

int inserItem(int item, int employeeList[])
{
    int location;

    location = hashing(item);
[redacted]
    employeeList[location] = item;
    length++;
}
```

Linear Probing - Search

```
int retrieveItem(int item, int employeeList[])
{
    int location;
    int startLoc;
    bool moreToSearch = true;

    location = hashing(item);
[redacted]
        return (location);
}
```

Linear Probing - Delete

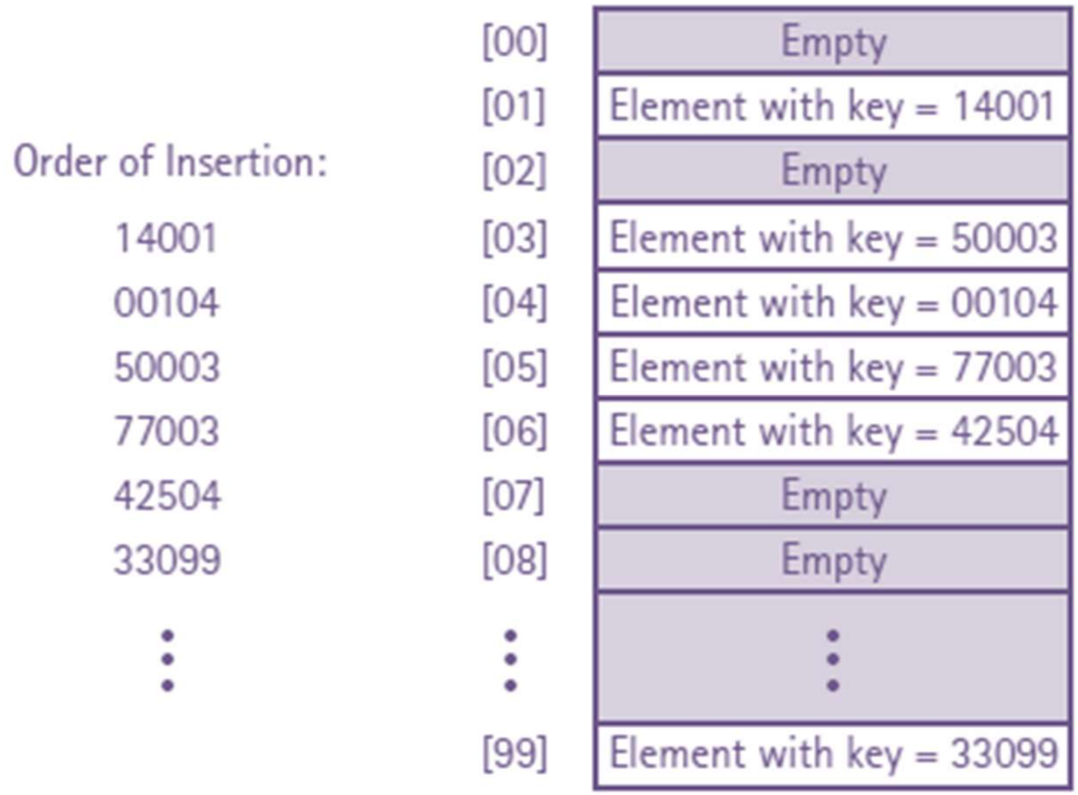
```
void deleteItem(int item, int employeeList[])
{
    int location;

    location=retrieveItem(item, employeeList);
    employeeList[location] = emptyItem;
}
```

Problem

- Empty slot ends the loop in searching
- Use constant value `deleteItem`
- Modify insert and search
  - Insertion treats `emptyItem` and `deleteItem` in the same way
  - `emptyItem` halt the search in `retrieveItem` but `deleteItem` does not

Clustering

The tendency of elements to become unevenly distributed in the hash table , with many elements clustering around a single hash location



Rehashing

Rehashing – Linear Probing

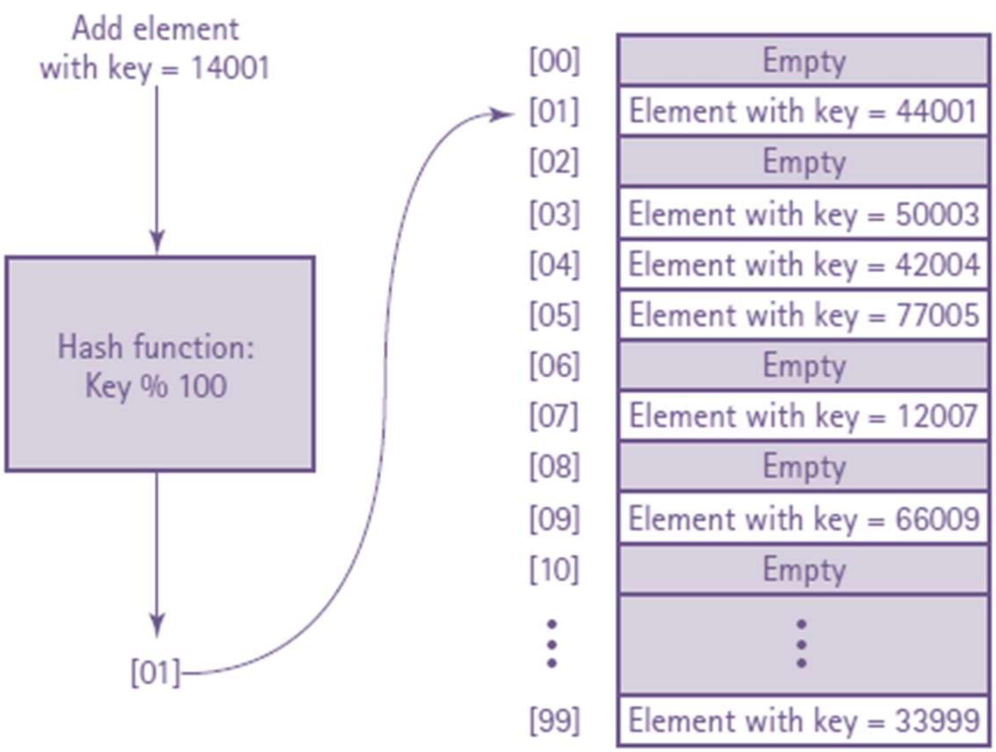
Resolving the collision by computing a new hash location from a hash function that manipulates the original location rather than the element's key

$$(hashValue + 1)\%\ 100$$

$$(hashValue + constant)\%\ array\ size$$

As long as `constant` and `array size` are relatively prime

Rehashing – Linear Probing



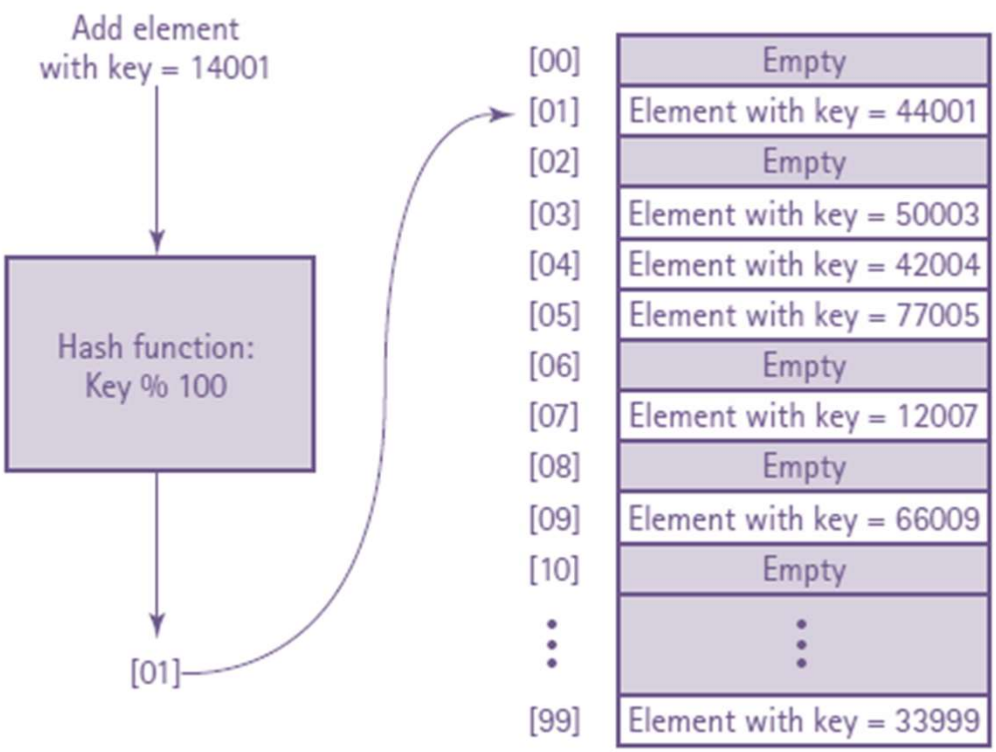
$$(hashValue + 1)\% \ 100$$

$(1+3)\%100$

$(4+3)\%100$

$(7+3)\%100$

Rehashing – Linear Probing



$(hashValue + 2)\%\ 100$

Only examining successive odd-numbered indexes

Rehashing – Linear Probing

- Does not eliminate clustering.

Rehashing – Quadratic Probing

Resolving a hash collision by using the rehashing formula (hashValue $\pm$ I^2) % array size, where I is the number of times that the rehash function has been applied

$$(hashValue + 1)\%\ array\ size$$

$$(hashValue - 1)\%\ array\ size$$

$$(hashValue + 2)\%\ array\ size$$

$$(hashValue - 2)\%\ array\ size$$

Rehashing – Quadratic Probing

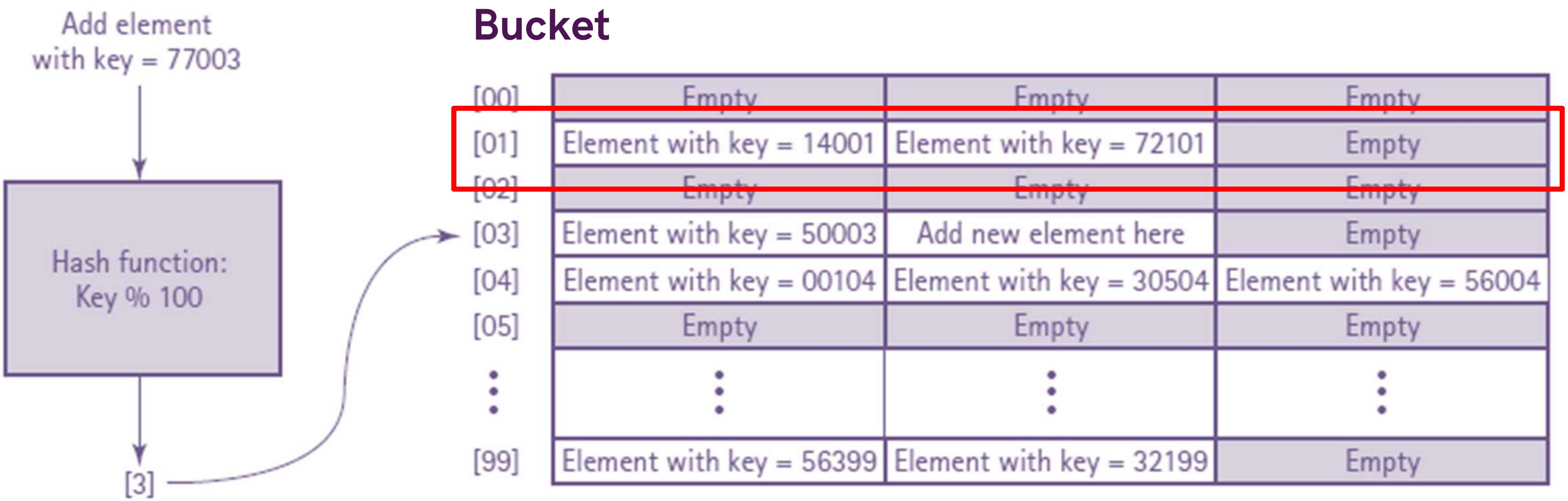
- Reduce clustering
- Does not necessarily examine every slot in the array
- E.g:
  - Array size is the power of 2 (512 or 1024) → relatively few array slots are examined.
  - Array size is a prime number of the form (4*some_integer+3) → examine every slot in the array.

Rehashing – Random Probing

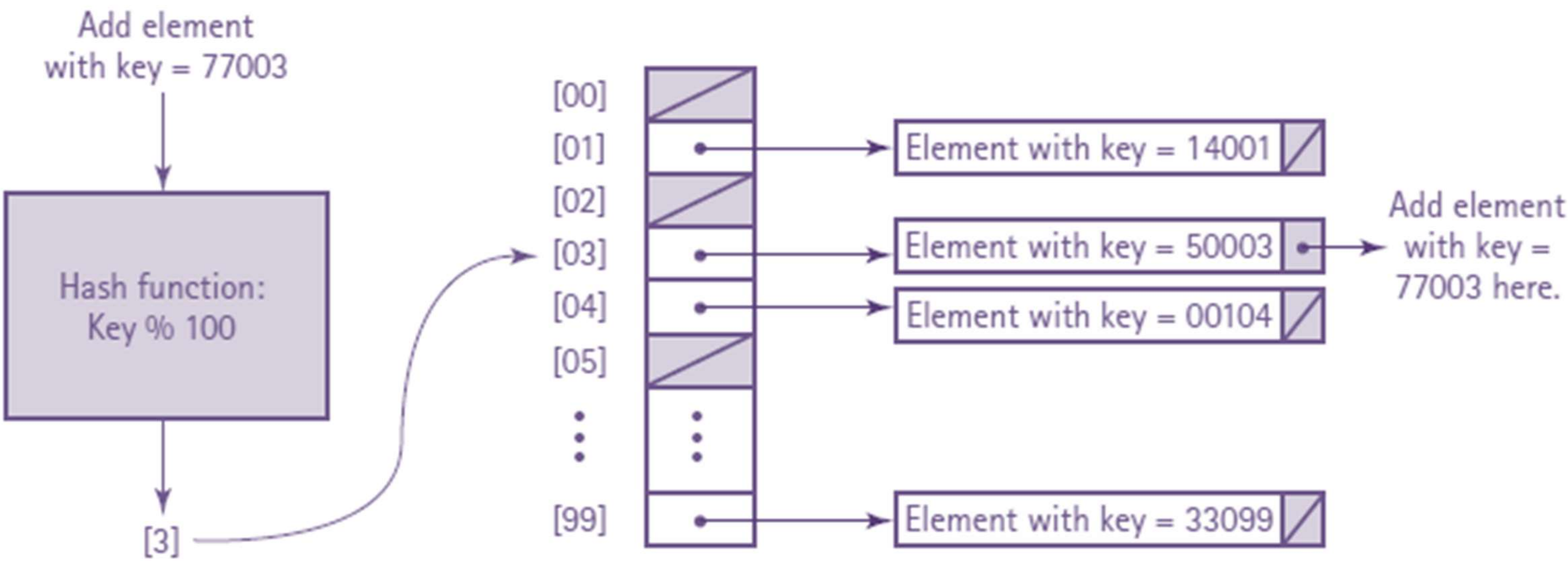
- Resolving a hash collision by generating pseudo-random hash values in successive application of the rehash function.
- Excellent technique for eliminating clustering.
- Tend to be slower than the other techniques discussed before.

Bucket and Chaining

Bucket


Add element
with key = 77003
Hash function:
Key % 100
[3]
Bucket
[00]
Empty
Empty
Empty
[01]
Element with key = 14001
Element with key = 72101
Empty
[02]
Empty
Empty
Empty
[03]
Element with key = 50003
Add new element here
Empty
[04]
Element with key = 00104
Element with key = 30504
Element with key = 56004
[05]
Empty
Empty
Empty
[99]
Element with key = 56399
Element with key = 32199
Empty

Chaining

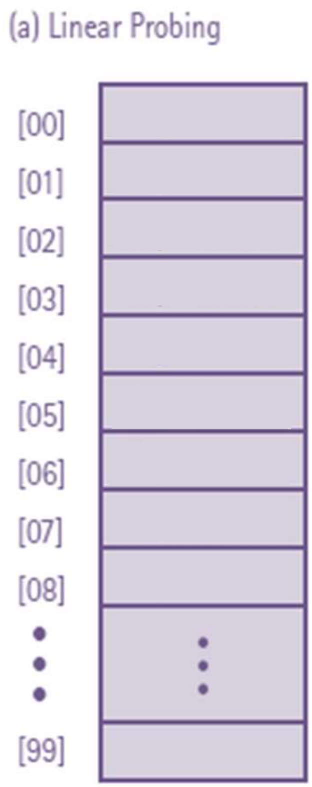

Add element
with key = 77003
Hash function:
Key % 100
[3]
[00]
[01]
[02]
[03]
[04]
[05]
[99]
Element with key = 14001
Element with key = 50003
Element with key = 00104
Element with key = 33099
Add element
with key =
77003 here.

Chaining - Searching

- Apply the hash function to the key
- Search the chain for the element

Chaining

45300
20006
50002
40000
25001
13000
65905
30001
95000



Good Hash Function

Minimize Collisions

- Use more space →space vs time trade off
- Design hash function to minimize collisions → access to each element no longer direct (O(1)), worst case O(N).

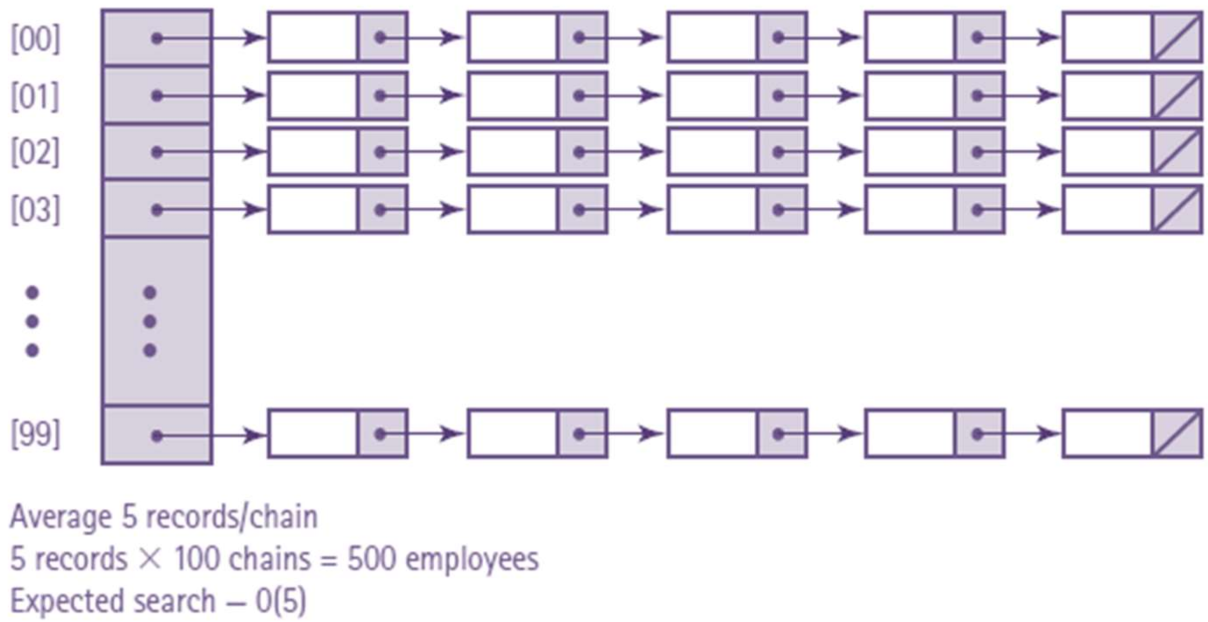
Minimize Collisions

- Use more space →space vs time trade off
- Design hash function to minimize collisions → access to each element no longer direct (O(1)), worst case O(N).
- Need to know the statistical distribution of keys.

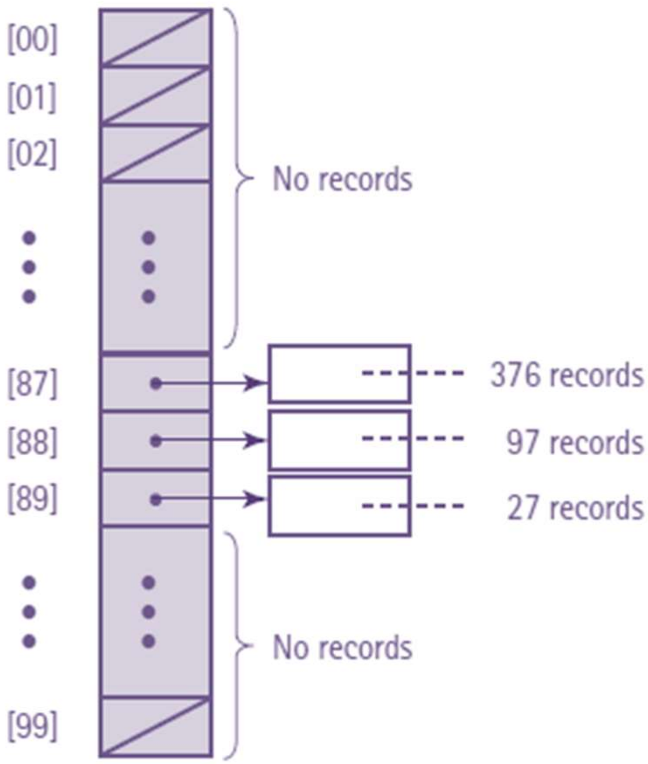
Example - The Plan

The company has 500 employees and decide to use a chained approach to handle collisions.

idNum%100



Example - The Reality



$$\underbrace{XXX}_{\substack{\text{3 digits,}\\ \text{unique number}\\ \text{(000–999)}}}\quad \underbrace{X}_{\substack{\text{1 digit,}\\ \text{dept. number}\\ \text{(0–9)}}}\quad \underbrace{XX}_{\substack{\text{2 digits,}\\ \text{year hired}\\ \text{(e.g., 89)}}}$$

376 employees hired in 1987
97 employees hired in 1988
27 employees hired in 1989

500 employees
Actual search 0(M)

Folding

Folding

If the element key is a string, use internal representation of the string's character to create a number that can serve as an index.

```
int hashing(char letter[])
{
    int sum = 0;
    for (int index = 0; index < 5; index++)
        sum = sum + int(letter[index])
    return sum%MAX_ITEM;
}
```

Folding

A hash method that breaks the key into several pieces and concatenates or exclusive-ORs some of the pieces to form the hash value.

Example

We want to devise a hash function that result in an index between 0 and 255, and internal representation of the `int` key is a bit string of 32 bits.

1. Break the key into **four** bit strings of **8 bits** each.
2. Exclusive-OR the first and last bit strings
3. Exclusive-OR the two middle bit strings
4. Exclusive-OR the results of steps 2 and 3 to produce the 8-bit index into the array.

Example

String

618403

binary representation

00000000000010010110111110100011

00000000 (leftmost 8 bits)

00001001 (next 8 bits)

01101111 (next 8 bits)

10100011 (rightmost 8 bits)

Example

00000000 (leftmost 8 bits)

00001001 (next 8 bits)

01101111 (next 8 bits)

10100011 (rightmost 8 bits)

```
       00000000
(XOR)  10100011
       --------
       10100011
```

```
       00001001
(XOR)  01101111
       --------
       01100110
```

Example

00000000
(XOR) 10100011
10100011

00001001
(XOR) 01101111
01100110

10100011
(XOR) 01100110
11000101 → 197

Complexity

Complexity

- If hash function never produces duplicates or array size if very large compared to the expected number of item → O(1)
- As the number of elements approach the array size, the efficiency of algorithms deteriorates
- A precise analysis of complexity of hashing is beyond the scope of this course.