

# Map / Hash Table

CPE111: Programming with Data Structures

## Reference

Rance D. Necaie. *Data Structures and algorithms using python. Chapter3*. John Wiley&Sons,Inc., 2011

Michael T.Goodrich, Roberto Tamassia, Michael H. Goodwasser. *Data Structures and Algorithms in python. Chapter10*. John Wiley&Sons,Inc. 2013

**Map** (**A s s o c i a t i v e   a r r a y**) is a data structure, called “**M a p**”, since it maps a **key** to a corresponding **value**. In python, this type of data structure is already implemented by “**D i c t i o n a r y**”



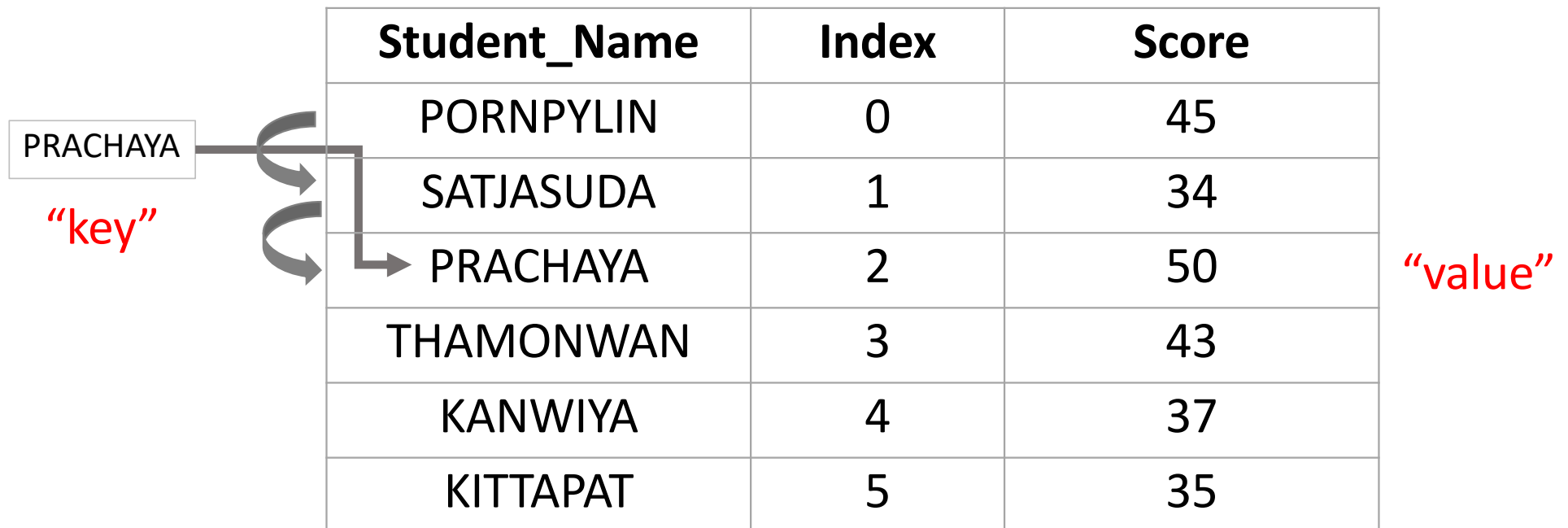
Student_Name	Score
PORNPYLIN	45
SATJASUDA	34
PRACHAYA	50
THAMONWAN	43
KANWIYA	37
KITTAPAT	35
...	...

## Linear map

## binary Search

To search an item with any comparison-based search algorithm in linear map take operation time in term of  $O(\log(n))$ . But actually we expect operation time in  $O(1)$  which is like as **direct access** to the value

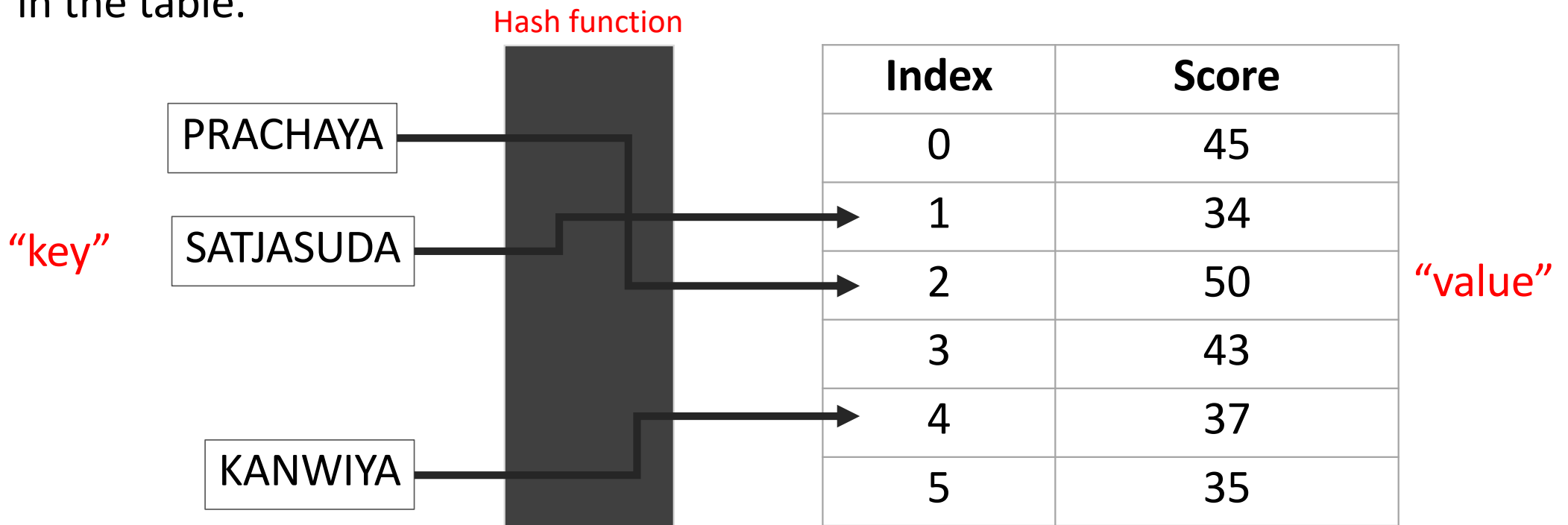
For example: we search **"Score"(value)** by **"Student\_Name" (key)**. First we search "Student\_Name" and found "index", then using "index" to point to value



Student_Name	Index	Score
PORNPYLIN	0	45
SATJASUDA	1	34
PRACHAYA	2	50
THAMONWAN	3	43
KANWIYA	4	37
KITTAPAT	5	35

## Hash Table

**H a s h i n g** is the process of mapping a **search key** to **a limited range of array indices** with the goal of providing **direct access** to the keys. The keys are stored in an array called a **hash table** and a **hash function** is associated with the table. The function converts or maps the search keys to specific entries in the table.



## Hash function

The purpose of a hash function is to **map** a set of **search keys** to **a range of index values [0,..,N-1]** corresponding to entries in a hash table.

- The Computation should be **simple** in order to produce quick results.
- The resulting index **cannot be random**. When a hash function is applied to the same key, it must always return the same index value.
- if The key consists of multiple parts, every part should **contribute** in the computation of the resulting index value.
- The Table size should be a **prime number**, especially when using the modulus operator. This can produce better distributions and fewer collisions as it tends to reduce the number of keys that share the same divisor.

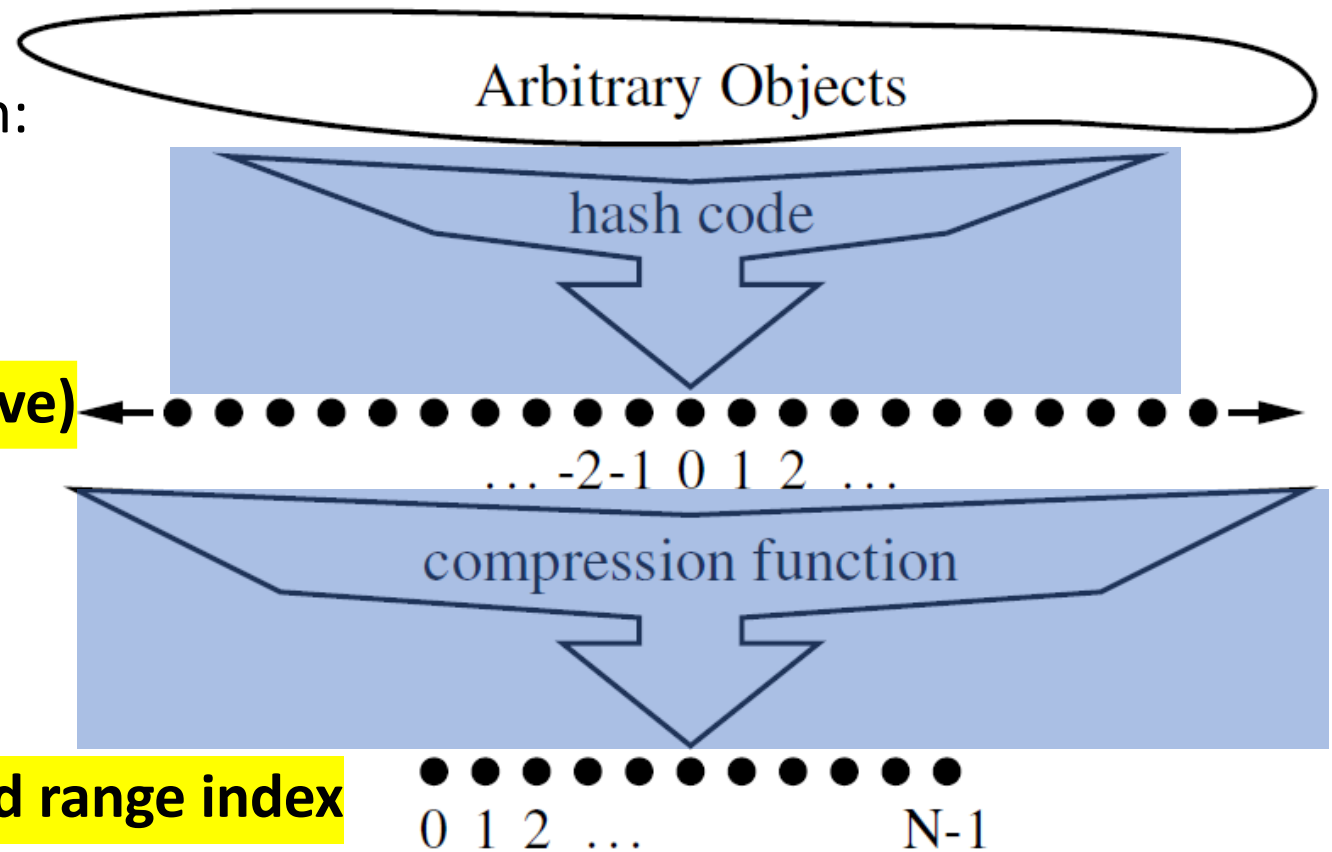
$$x = h(k)$$

A hash function,  $h(k)$ , consists of two parts—a hash code that maps a key  $k$  to an integer, and a compression function.

Two parts of a hash function:  
a hash code and  
a compression function

an integer (negative to positive)

a limited range index




Hash function is normally an one-way function

## Polynomial hash code

$$x_0a^{n-1} + x_1a^{n-2} + \cdots + x_{n-2}a + x_{n-1}$$

or

$$x_{n-1} + a(x_{n-2} + a(x_{n-3} + \cdots + a(x_2 + a(x_1 + ax_0)) \cdots ))$$


character      constant (prime number)

from experimental studies suggest that 33, 37, 39, and 41 are particularly good choices for  $a$  when working with character strings that are English words (list of over 50,000 English words). They produced less than 7 collisions in each case.

## Cyclic-Shift Hash Codes

A variant of the polynomial hash code replaces multiplication by a with a cyclic shift of a partial sum by a certain number of bits.

For example, a **5-bit cyclic shift** of the 32-bit value is achieved by taking the leftmost five bits and placing those on the rightmost side of the representation.

00111101100101101010100010101000



10110010110101010001010100000111

**5-bit cyclic shift**



```
def hash_code(s):
    mask = (1 << 32) - 1
    h = 0
    for character in s:
        h = (h << 5 & mask) | (h >> 27)
        h += ord(character)
    return h
```

when using a **cyclic-shift hash code**, we must wisely choose the amount to shift by for each new character. Our choice of a 5-bit shift is justified by experiments run on a list of just over 230,000 English words.

**“Total”** is The total number of words that collide with at least one other.  
**“Max”** is the maximum number of words colliding at any one hash code.

Shift	Collisions	
	Total	Max
0	234735	623
1	165076	43
2	38471	13
3	7174	5
4	1379	3
5	190	3
6	502	2
7	560	2
8	5546	4
9	393	3
10	5194	5
11	11559	5
12	822	2
13	900	4
14	2001	4
15	19251	8
16	211781	37

## Compression Functions

The hash code for a key **k** typically is not suitable for immediately use, it may be negative or may exceed the capacity of the bucket array. So it is necessary to map that integer into the limited range  $[0, N-1]$ .

A **compression function**, is the second action performed as part of an overall hash function. A good compression function is minimizes the number of **collisions** of hash codes.

## The MAD Method

*Multiply-Add-and-Divide* (or “MAD”) method maps an integer  $i$  to

$$[(ai+b) \bmod p] \bmod N$$

where  $N$  is the size of the bucket array,

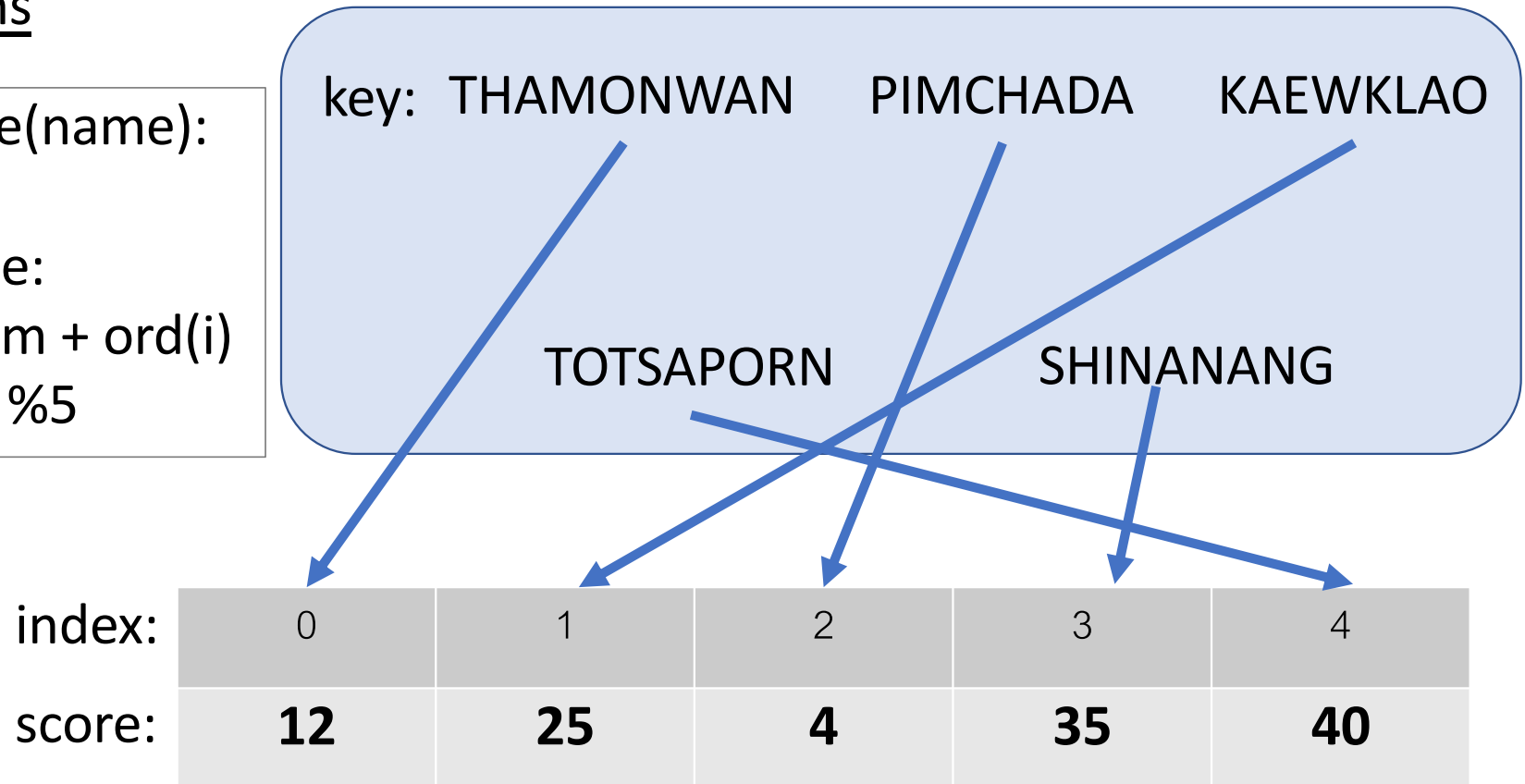
$p$  is a prime number larger than  $N$ , and

$a$  and  $b$  are random integers from the interval  $[0, p-1]$ , with  $a > 0$ .

This compression function is chosen in order to eliminate repeated patterns in the set of hash codes and get us closer to having a “good” hash function, that is, one such that the probability any two different keys collide is  $1/N$ . This good behavior would be the same as we would have if these keys were “thrown” into  $A$  uniformly at random.

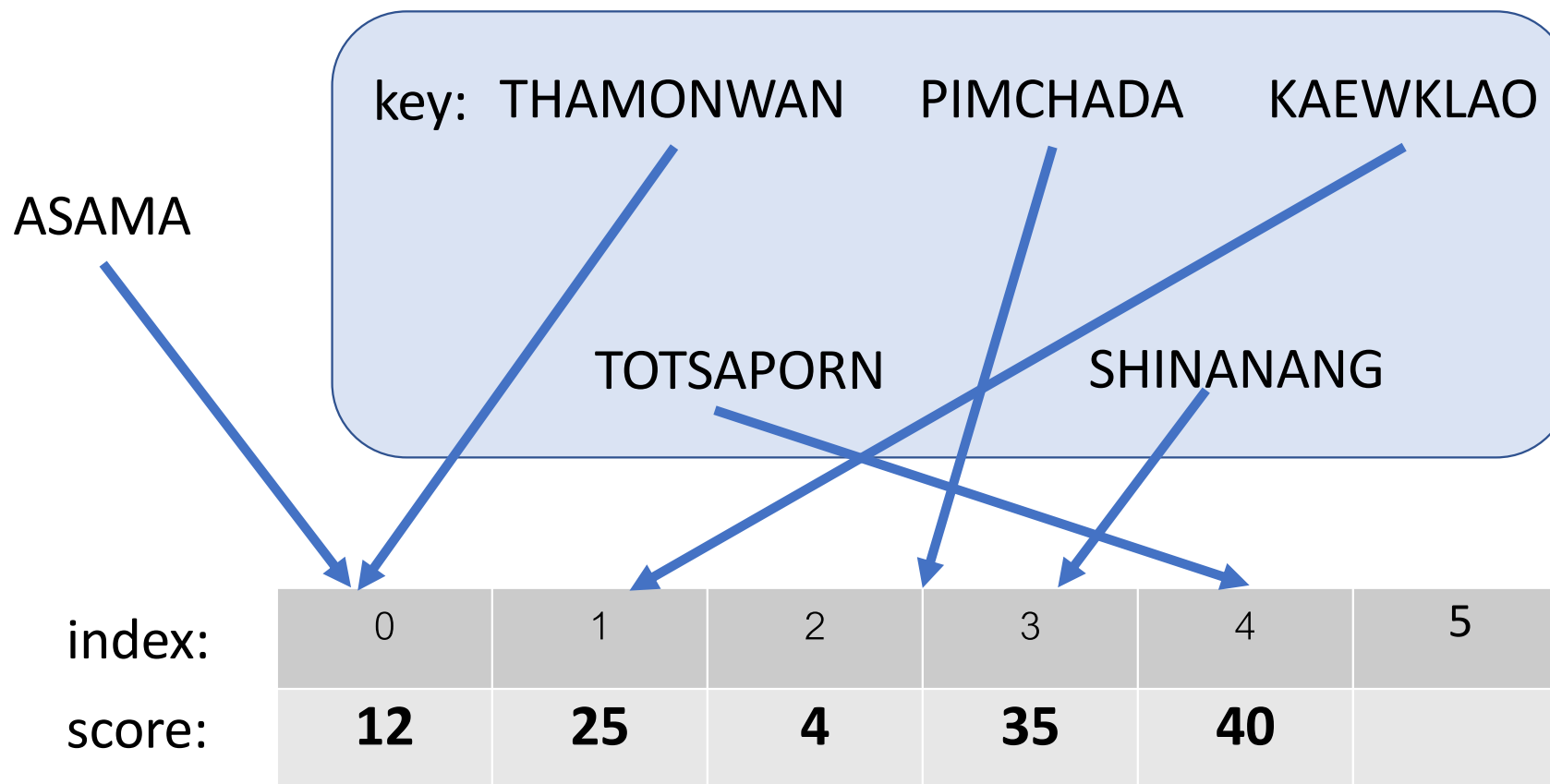
A “**perfect**” hash function will map every key to a different table entry, resulting in no collisions

```
def hashname(name):  
    sum = 0  
    for i in name:  
        sum = sum + ord(i)  
    return sum % 5
```



## Collision

A collision has occurred when **two or more** keys with **the same hash value**, cause two different items will be mapped to the same bucket in **A**.



## Collision handling

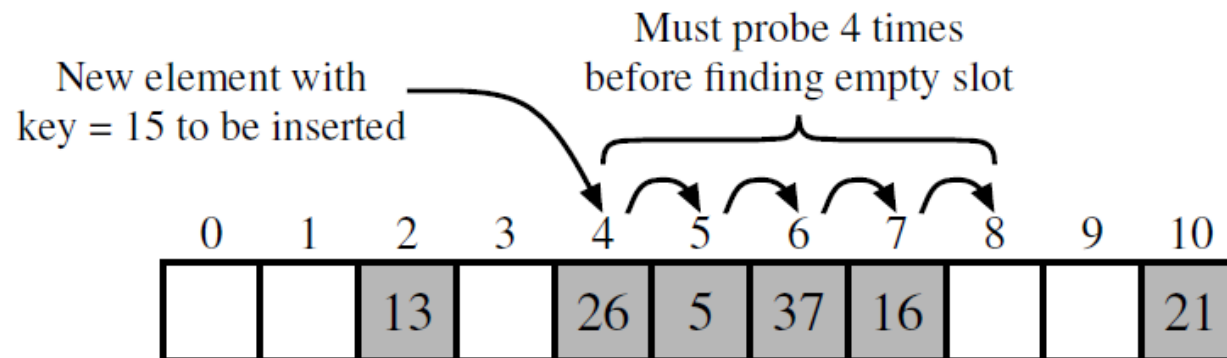
### Open Addressing

This approach has no auxiliary structures, but it requires a bit more complexity to deal with collisions. There are several variants of this approach.

Open addressing requires that the load factor is always at most 1 and that items are stored directly in the cells of the bucket array itself.

### Linear Probing

Ex: Hash function is  $h(k) = k \bmod 11$  and  $k = 15$



## Quadratic probing

iteratively tries the buckets  $A[(h(k) + f(i)) \bmod N]$ , for  $i = 0, 1, 2, \dots$ , where  $f(i) = i^2$ , until finding an empty bucket.

## Double hashing

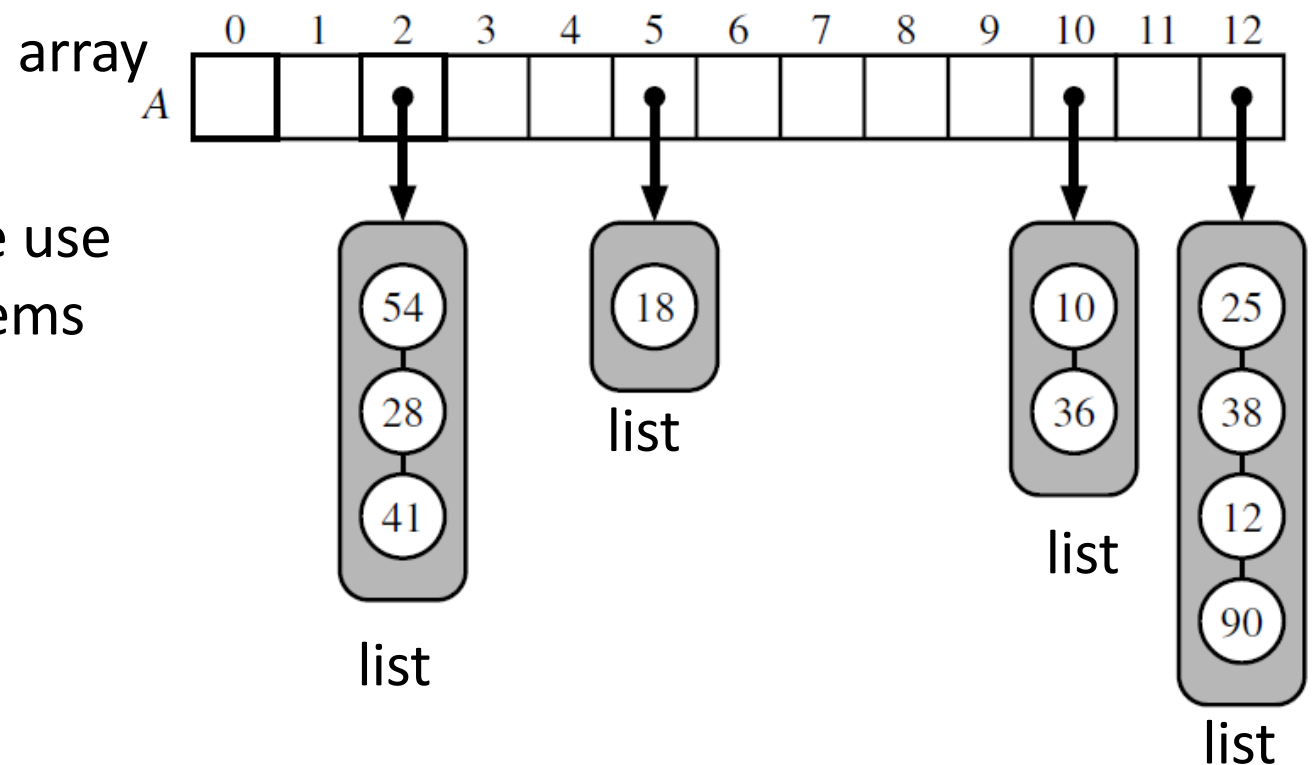
In this approach, we choose a secondary hash function,  $h$ , and if  $h$  maps some key  $k$  to a bucket  $A[h(k)]$  that is already occupied, then we iteratively try the buckets  $A[(h(k) + f(i)) \bmod N]$  next, for  $i = 1, 2, 3, \dots$ , where  $f(i) = i \cdot h'(k)$ .

The efficiency of the hash operations depends on the **hash function**, the **size of table**, and the **type of probe** used to resolve collisions.

## Separate Chaining

A simple and efficient way for dealing with collisions is to have each bucket  $A[j]$  store its own secondary container.

A natural choice for the secondary container is a small map instance implemented using a list



Disadvantage: It requires the use of an **auxiliary list** to hold items with colliding keys.



## Load Factor

The ratio  $\lambda = n/N$ , called the **load factor** of the hash table,  $n$  is number of items and  $N$  is a capacity of bucket arrays. So it should be bounded by a small constant, preferably below than 1.

With **separate chaining**, as  $\lambda$  gets very close to 1, the probability of a collision greatly increases. From the experiments and average-case analyses suggest that we should maintain  $\lambda < 0.9$  for hash tables with separate chaining.

With **open addressing**, Experiments suggest that we should maintain  $\lambda < 0.5$  for an open addressing scheme with **linear probing**, and perhaps only a bit higher for other open addressing schemes.

## Time Complexity of Hash Table

Operation	List	Hash Table	
		expected	worst case
<code>--getitem--</code>	$O(n)$	$O(1)$	$O(n)$
<code>--setitem--</code>	$O(n)$	$O(1)$	$O(n)$
<code>--delitem--</code>	$O(n)$	$O(1)$	$O(n)$
<code>--len--</code>	$O(1)$	$O(1)$	$O(1)$
<code>--iter--</code>	$O(n)$	$O(n)$	$O(n)$