Applied Algorithms

CSCI-B505 / INFO-I500

Lecture 21.

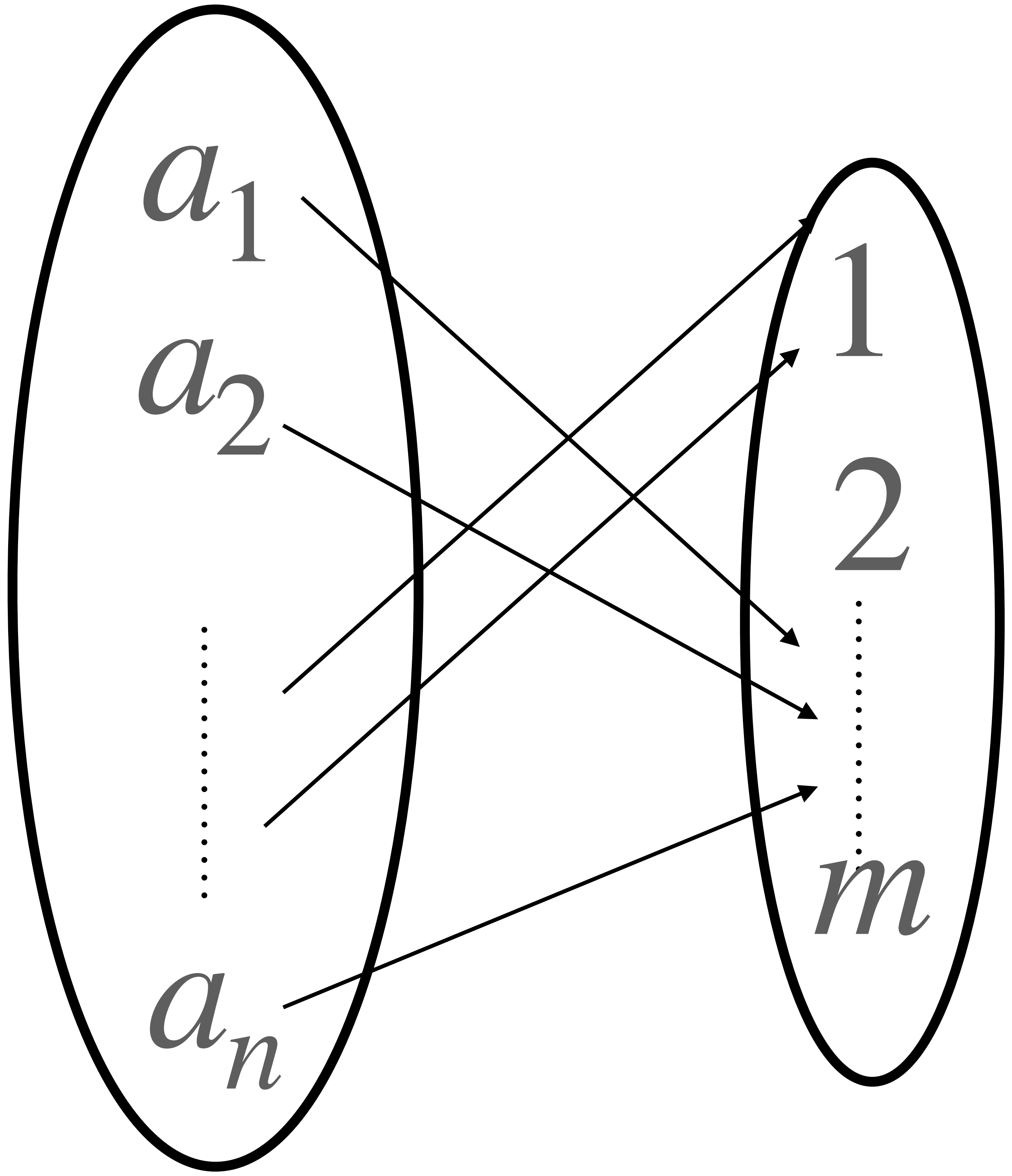
Hashing

M. Oguzhan Kulekci

- Basics of hashing
- Perfect hash schemes

Hash Functions

Map each object from a source space to a target space.



Usually, $n \gg m$.

Why we need such a function ?

- -Efficient indexing, retrieval, search
- Randomizing the distribution (Cryptographic hash functions)
- Many others ….

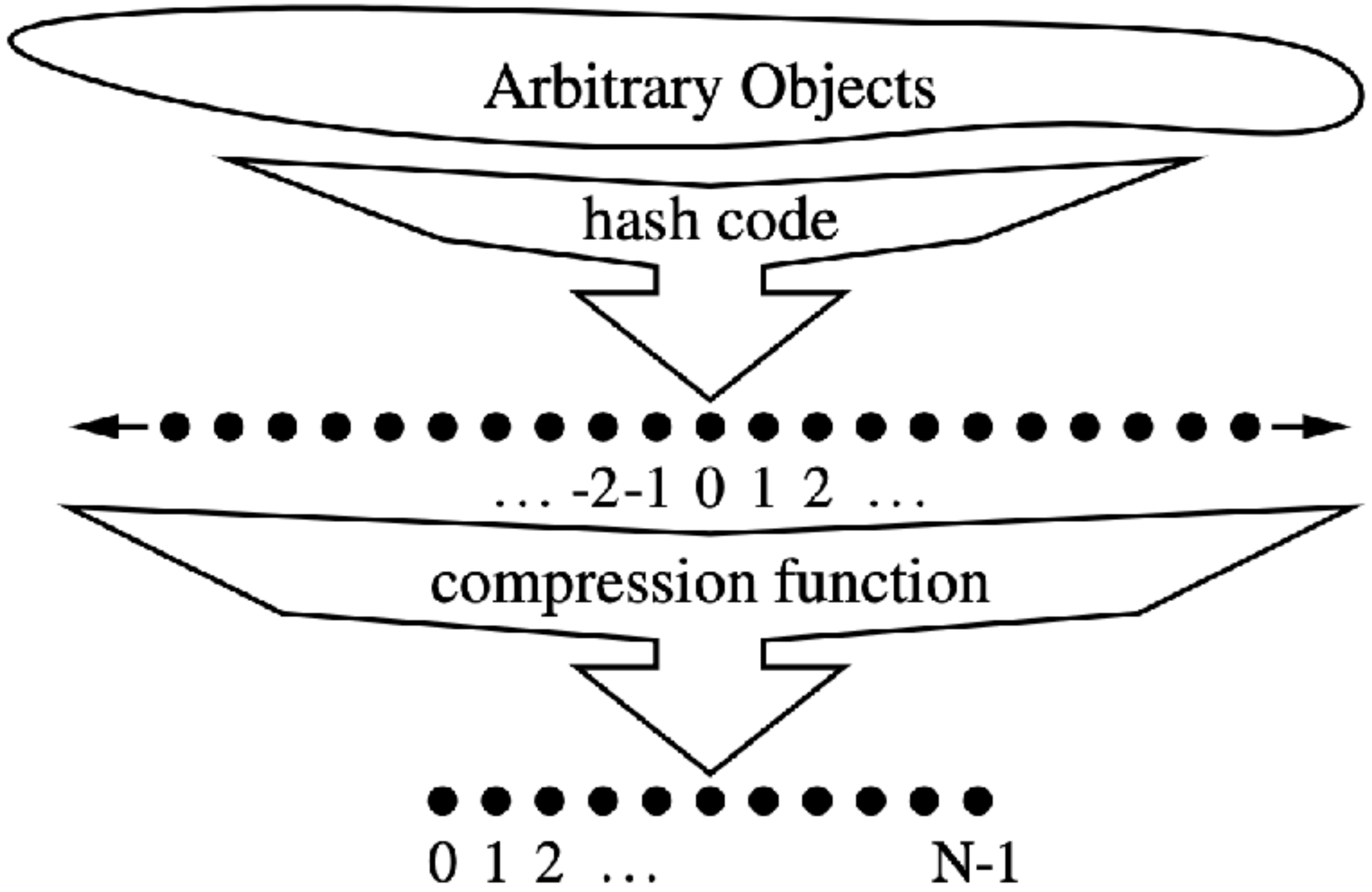
Hash Functions

What makes a good hash function ?

- **Fast**, easy to compute within less space
- **As less collision as possible**, in other words, randomly distributing, similar objects should not fall into same bins *(there are cases contrary to that though)*

Example: Assume we would like to make sure that a downloaded file is correctly received. How can we do this?

Hash Functions



STEP 1. Represent objects to integers, *without aiming to squeeze.*

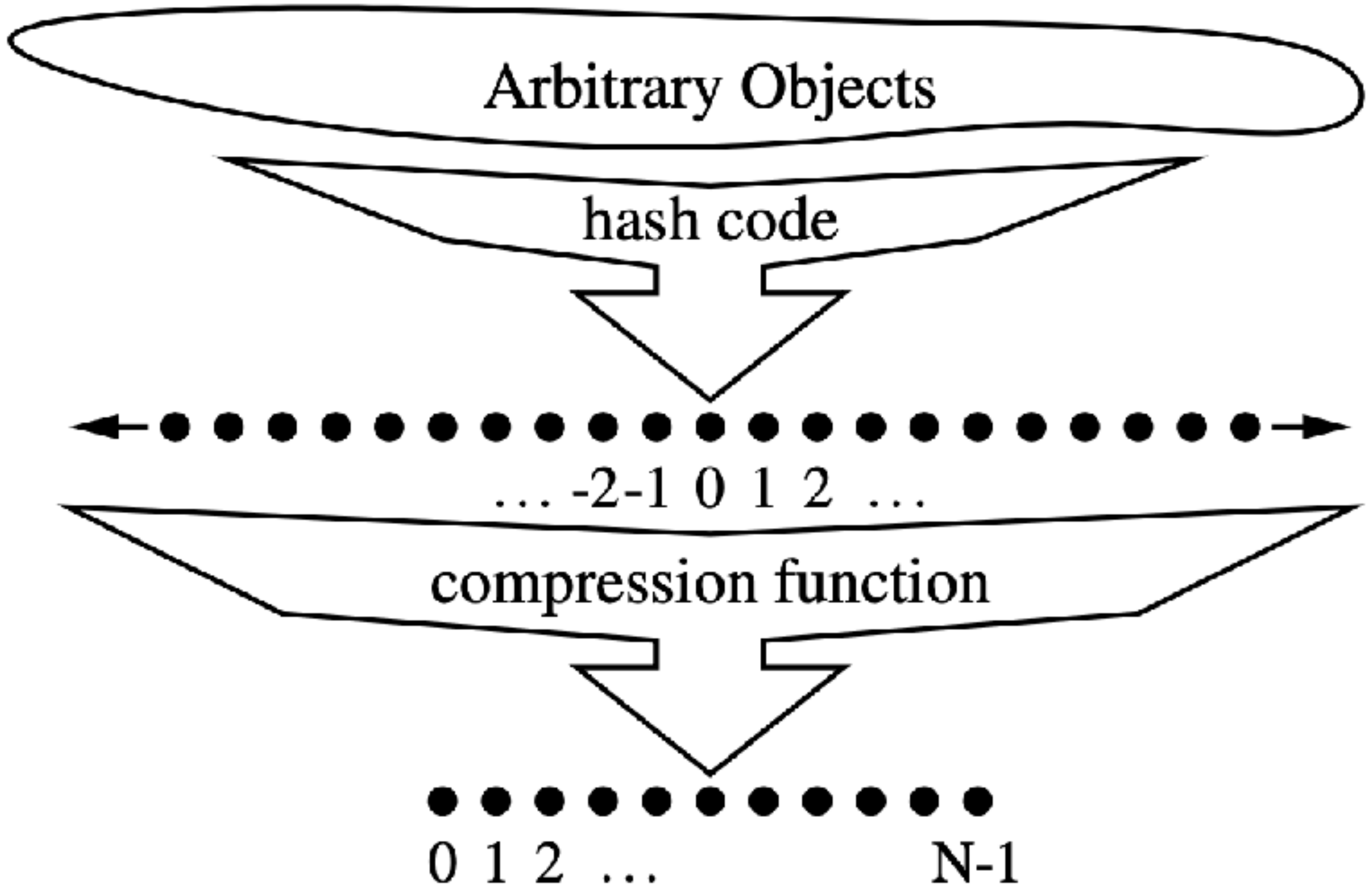
- Bit-representations:

$$\underbrace{0110}_{x_1}\underbrace{010}_{x_2}\ldots\ldots\ldots\ldots\ldots\underbrace{11010}_{x_k}$$

$$HashCode = x_1 + x_2 + .. + x_k$$

$$HashCode = x_1 \oplus x_2 \oplus .. + \oplus x_k$$

Hash Functions



STEP 1. Represent objects to integers,
without aiming to squeeze.

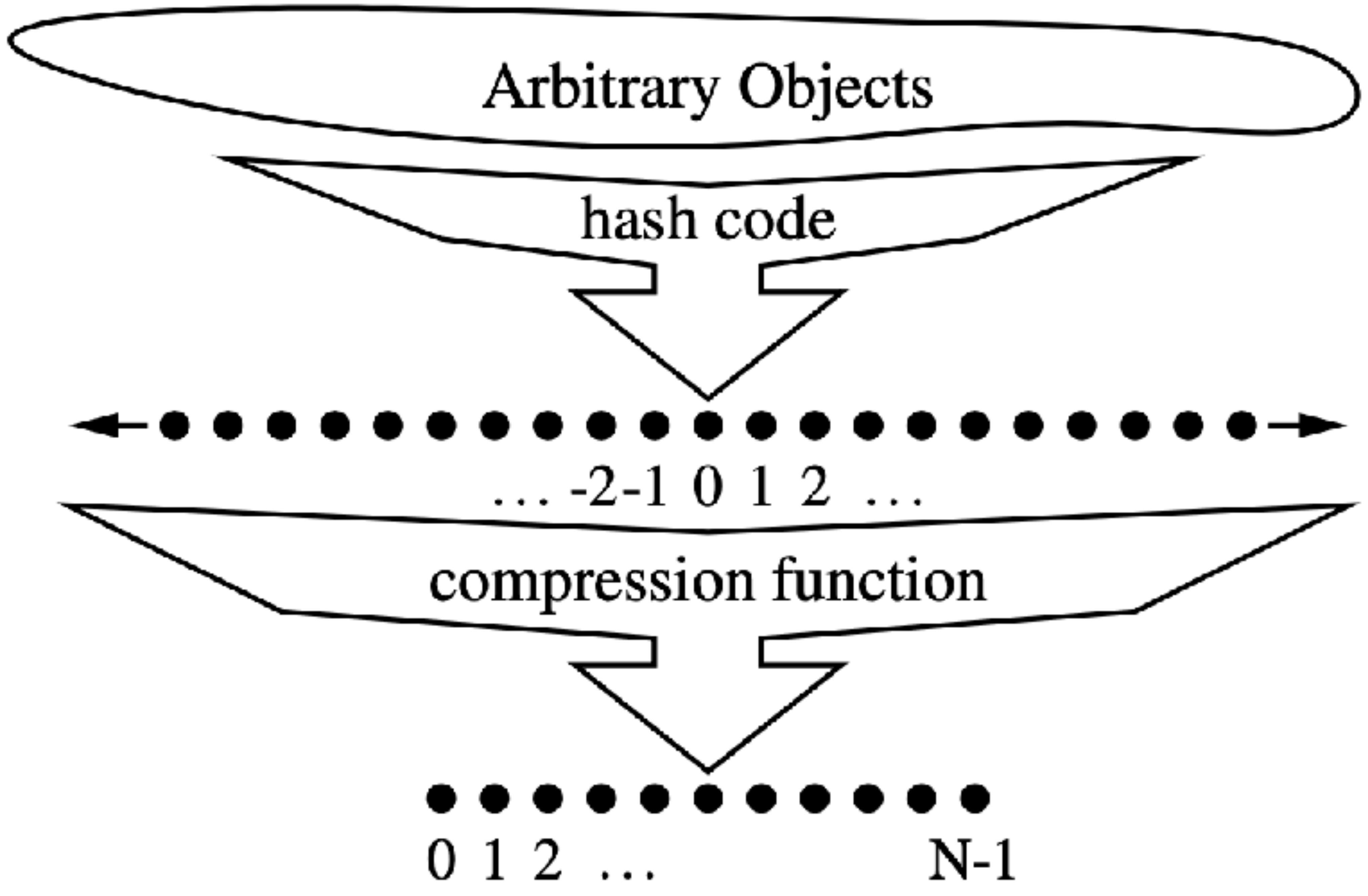
- Polynomial codes:

0110010................11010

x_1 x_2 x_k

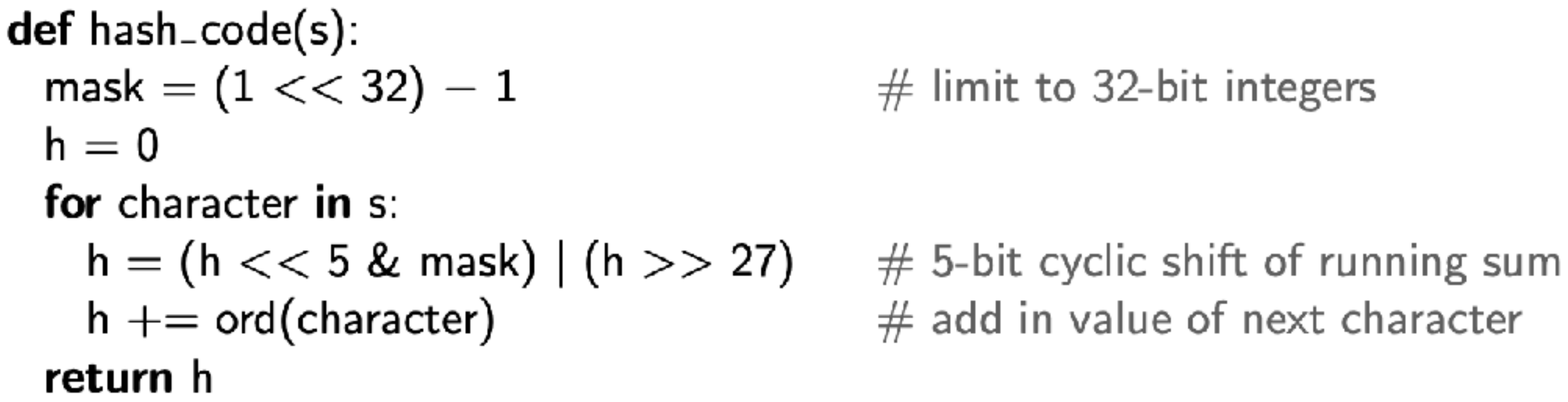
$$HashCode = x_1 + x_2 \cdot a + x_2 \cdot a^2 + .. + x_k \cdot a^{k-1}$$

Hash Functions



STEP 1. Represent objects to integers, *without aiming to squeeze.*

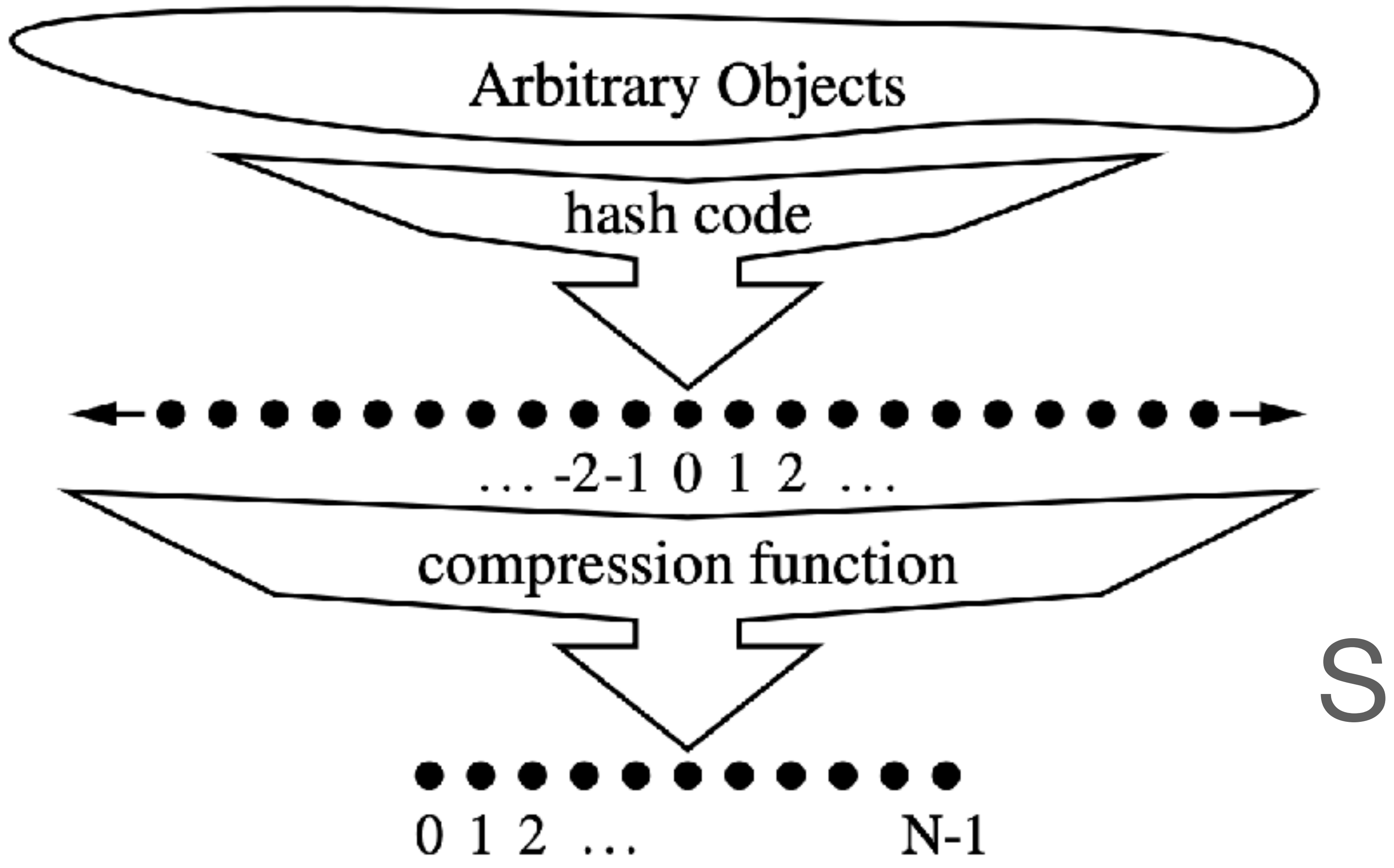
- Cyclic Codes:



```
def hash_code(s):
  mask = (1 << 32) − 1                  # limit to 32-bit integers
  h = 0
  for character in s:
    h = (h << 5 & mask) | (h >> 27)     # 5-bit cyclic shift of running sum
    h += ord(character)                 # add in value of next character
  return h
```

There are many other alternatives, *depending on the applications…*

Hash Functions

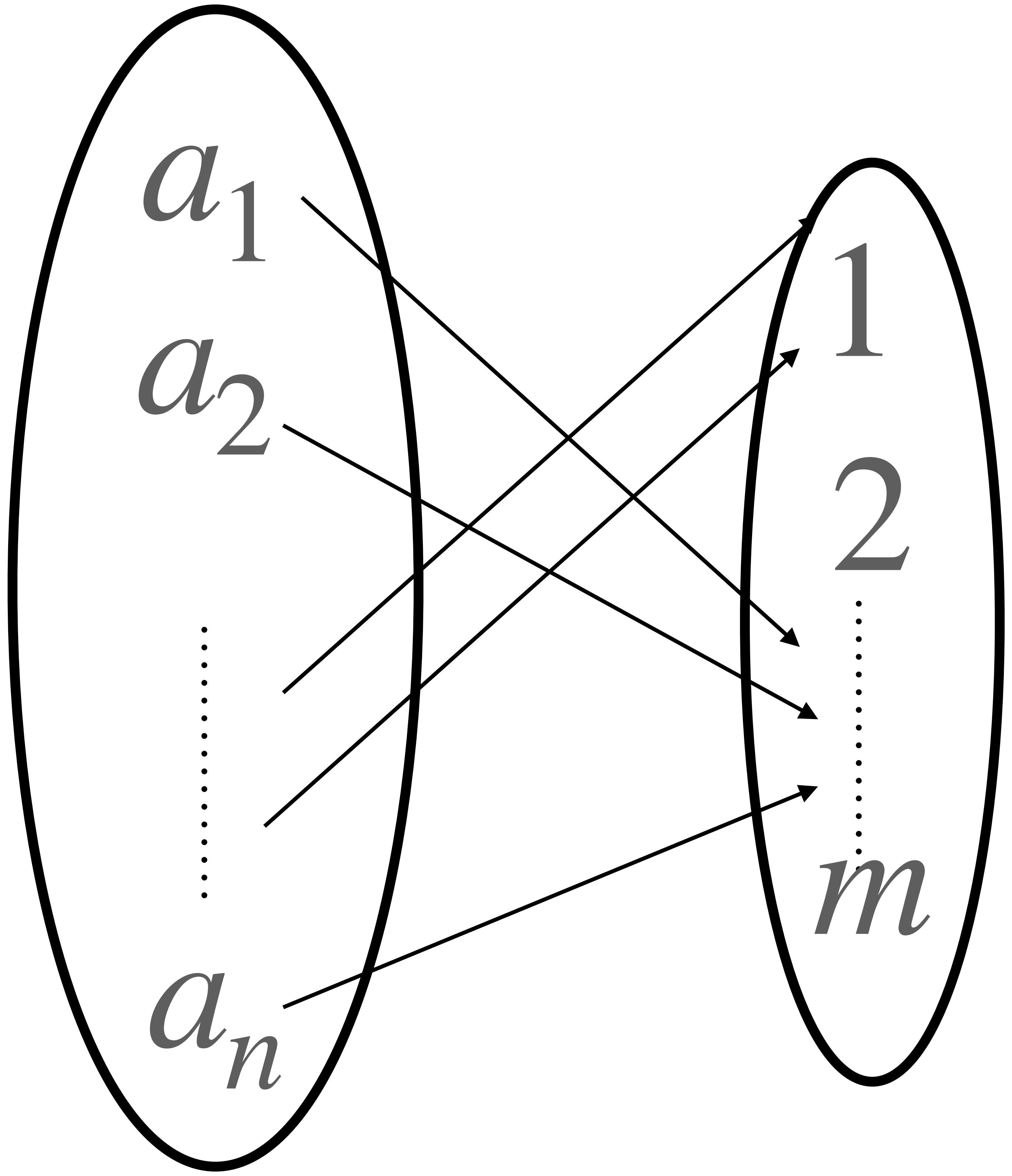


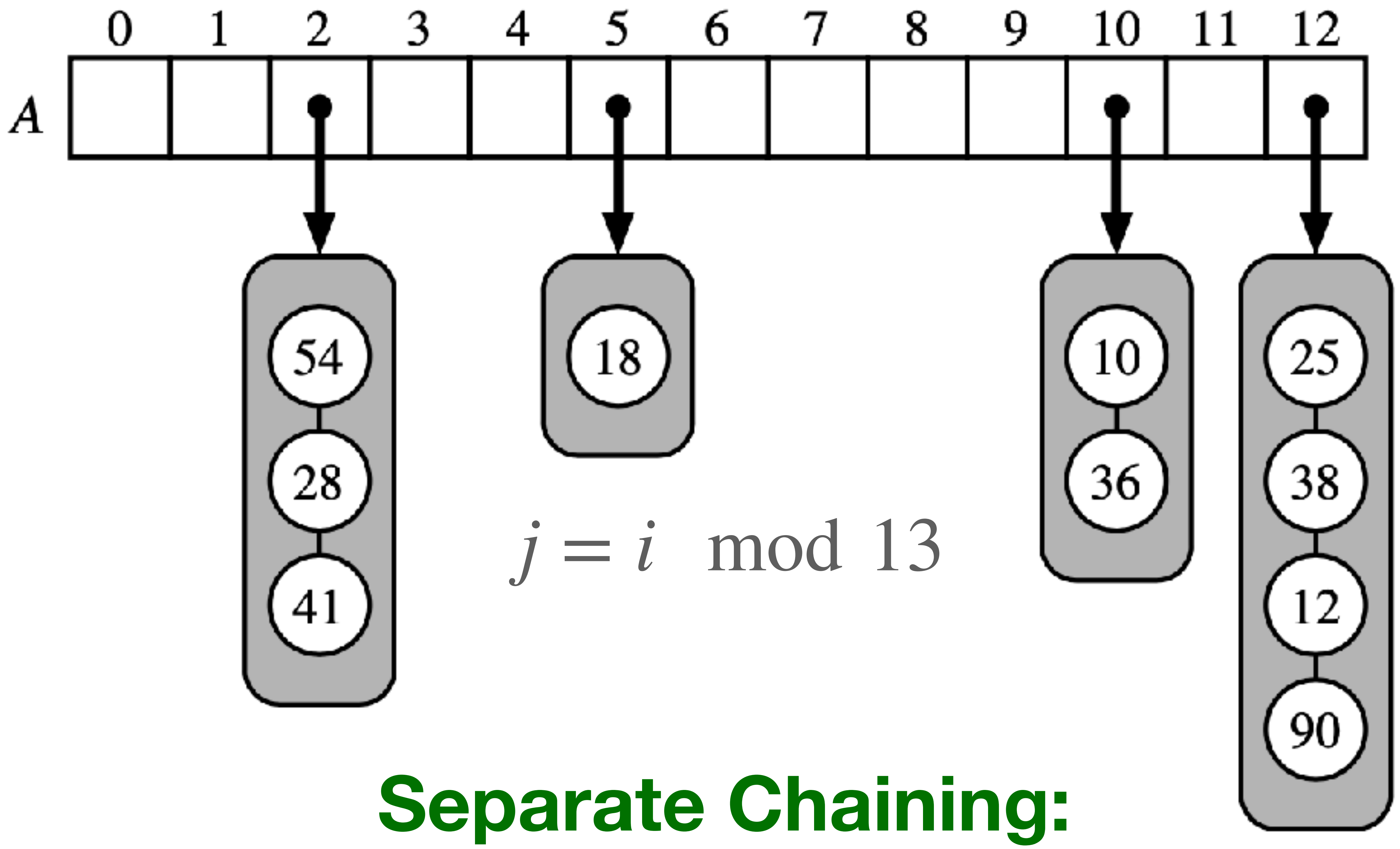
STEP 2. Map the hash code to a smaller universe, *now we aim compression !*

Simply we convert n-bit integer i onto a m-bit integer j

- $j = i \mod 2^m$
- $j = ((a \cdot i + b) \mod p) \mod 2^m$
- Or some other functions …..

Collision Handling





Separate Chaining:

For each bucket on the **hash table,** maintain a list.

- Pros: Easy to implement and understand
- Cons: Extra link data structures may hurt space efficiency

Collusions are unavoidable assuming $n \gg m$.

We need **collusion handling mechanisms.**

Collision Handling

Open addressing:

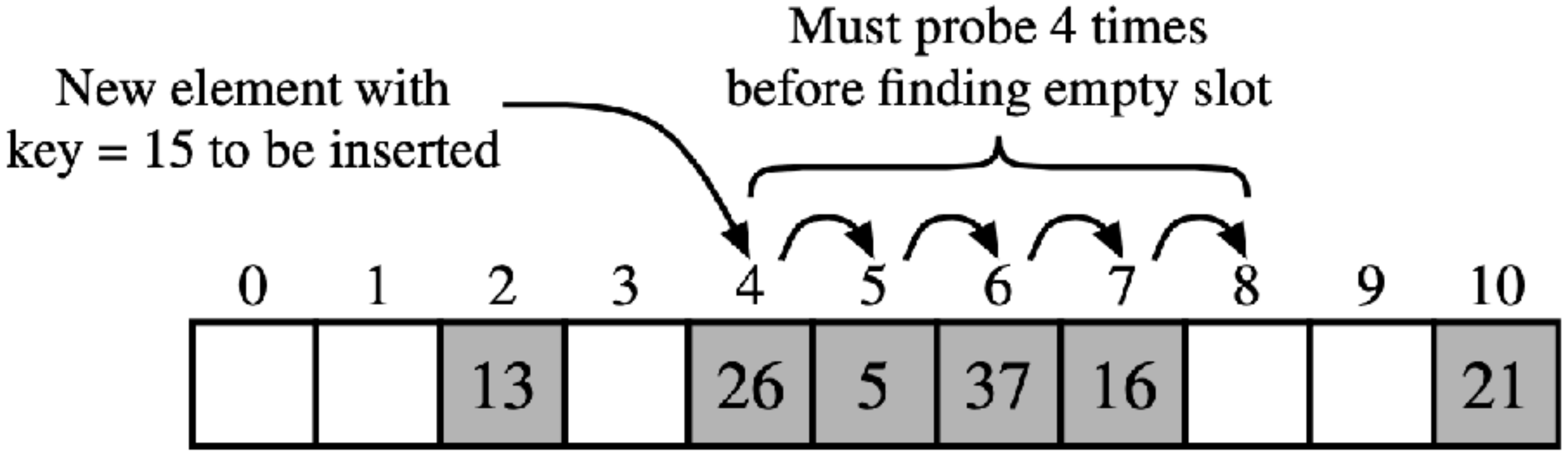
When a collision happens, find another vacancy on the **hash table** to place the item.

What can be the mechanisms to generate the alternative positions?

If occupied, then check the succeeding according to a rule.

$$h(k) + f(i) \mod m, \quad \text{for } i = 0,1,2,3,....$$

Linear Probing: $f(i) = i$



New element with key = 15 to be inserted
Must probe 4 times before finding empty slot
0
1
2
3
4
5
6
7
8
9
10
13
26
5
37
16
21

$$h(k) = k \mod 11$$

How to

- insert
- delete
- search

Collision Handling

$$h(k) + f(i) \mod m, \quad \text{for } i = 0,1,2,3,....$$

Quadratic Probing: $f(i) = i^2$, guarantees to find a vacant position, if hash table size N is prime and less than half full.

- Both linear and quadratic probing techniques result in clustering effects, some positions of the hash table gets more quickly filled.
- Double-hashing mechanisms are used to avoid that clustering bias.

Double-Hashing:

$f(i) = i \cdot h'(k), \quad \forall k \; h'(k) \neq 0$ or

$f(i) = RNG(i)$. For some random number generator

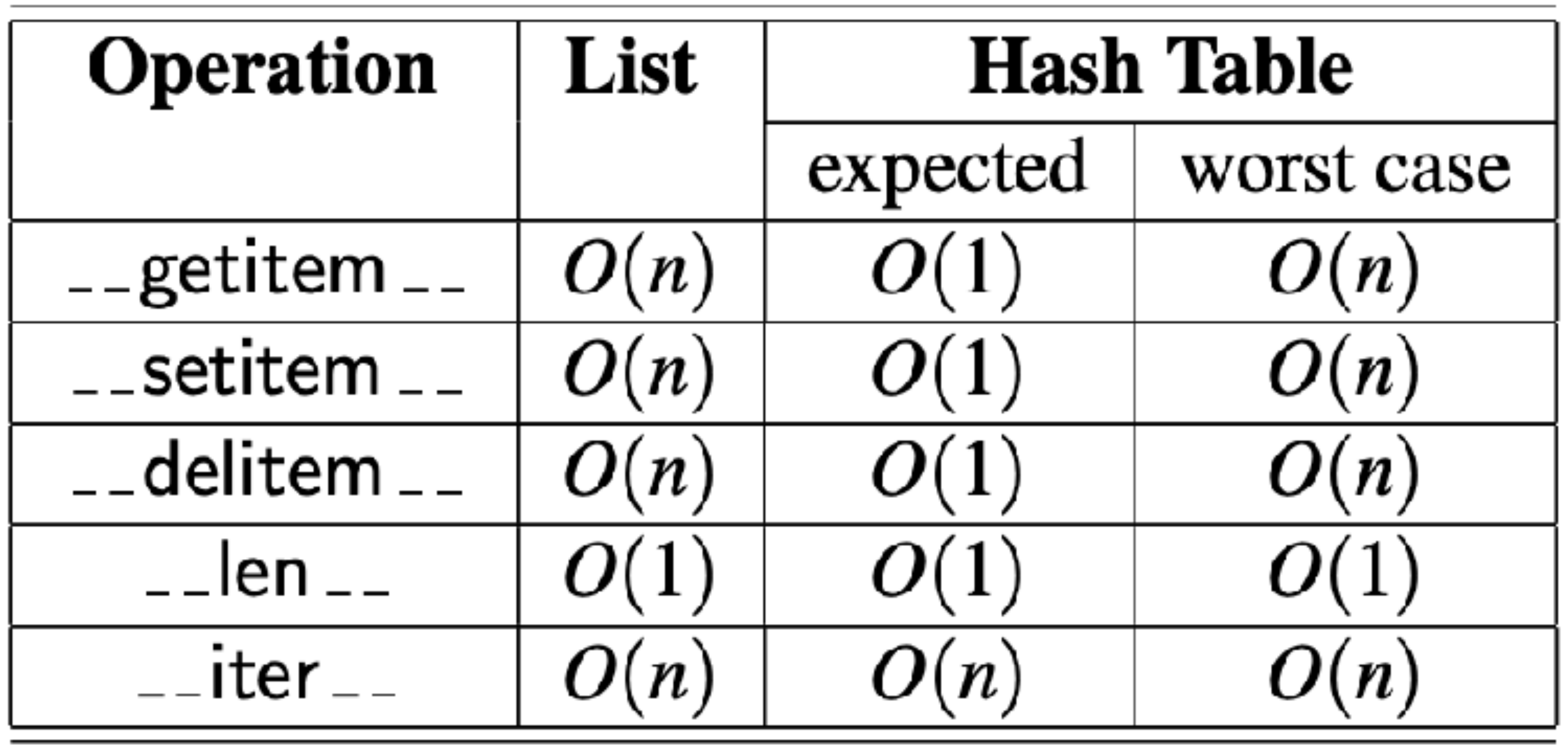
Resizing the hash table

$$\textbf{Load factor} = \lambda = \frac{\textbf{number of occupied cells in the hash table}}{\textbf{hash table capacity}}$$

- Make sure $\lambda < 1$ all the time
- Enlarging the hash table size
  - when $\lambda > 0.9$ with separate chaining
  - when $\lambda > 0.5$ with linear probing,
  - or according to another ratio

  are good practices.
- Resize means we need to **rehash**, which might be heavy
- How to resize ? Why doubling is a good choice ?

Why using a hash makes sense in many applications?



Operation	List	Hash Table	
		expected	worst case
__getitem__	$O(n)$	$O(1)$	$O(n)$
__setitem__	$O(n)$	$O(1)$	$O(n)$
__delitem__	$O(n)$	$O(1)$	$O(n)$
__len__	$O(1)$	$O(1)$	$O(1)$
__iter__	$O(n)$	$O(n)$	$O(n)$

- Hash tables bring **expected** constant time for operating on lists.

- Depending on the selected hash scheme, computing the efficiency and thresholds for resizing may require deep probability calculations.

Example

Find the **longest** subsequence on a given array such that the elements are consecutive, but not necessarily to be in order.

$$A = [8,4,1,9,3,2,0,5,7,10]$$

$$[0,1,2,3,4,5] \qquad\qquad [7,8,9,10]$$

Naive solution	Sorting-based solution	Hash-based solution *(expected)*
$O(n^3)$	$O(n \log n)$	$O(n)$

Example

Find the **longest** subsequence on a given array such that the elements are consecutive, but not necessarily to be in order.

$O(n^3)$-time Naive solution

```
For each $1 \le i \le n$
    1. x= $A[i]$
    2. Repeat searching $(x+1)$ on $A$
       and record the length of the streak  $O(n^2)$
    3. Return the longest streak
```

$$A = [8,4,1,9,3,2,0,5,7,10]$$

9	5	2	10	4	3	1	-	8	-
10		3		5	4	2		9	
		4			5	3		10	
		5				4			
						5			

Example

Find the **longest** subsequence on a given array such that the elements are consecutive, but not necessarily to be in order.

$$A = [8,4,1,9,3,2,0,5,7,10]$$

$O(n \log n)$ solution with sorting

1. Sort $A[]$
2. Scan for the longest consecutive stream

$$A = [\underrightarrow{0,1,2,3,4,5},\overrightarrow{7,8,9,10}]$$

Example

Find the **longest** subsequence on a given array such that the elements are consecutive, but not necessarily to be in order.

Observation: If x is the initial element of a possible subsequence, then $(x-1)$ is not in the array.

$$A = [8,4,1,9,3,2,0,5,7,10]$$

$O(n)$-time solution with hashing

1. **Detect the first elements of possible subsequences ?**
2. Find max length

WHY LINEAR ?
Every element in the array is at most visited twice

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
5		9			0	7	**10**	1	8		2			3			4		

1. Insert $A[]$ on a hash table,
   e.g. $h(i) = 3i + 5 \mod 20$
2. For each item, check whether x-1 is on the list. If so, it cannot be initial.
3. Compute the length of the subsequences by the detected initials.

Some other examples

- In a given array find the subarrays (or longest subarray) that sum up to zero ?
- Given an array A[] and a number x, check for pairs in A[] that sum up to x
- Find elements which are present in first array and not in second
- Group words with same set of characters
- Find missing elements of a range

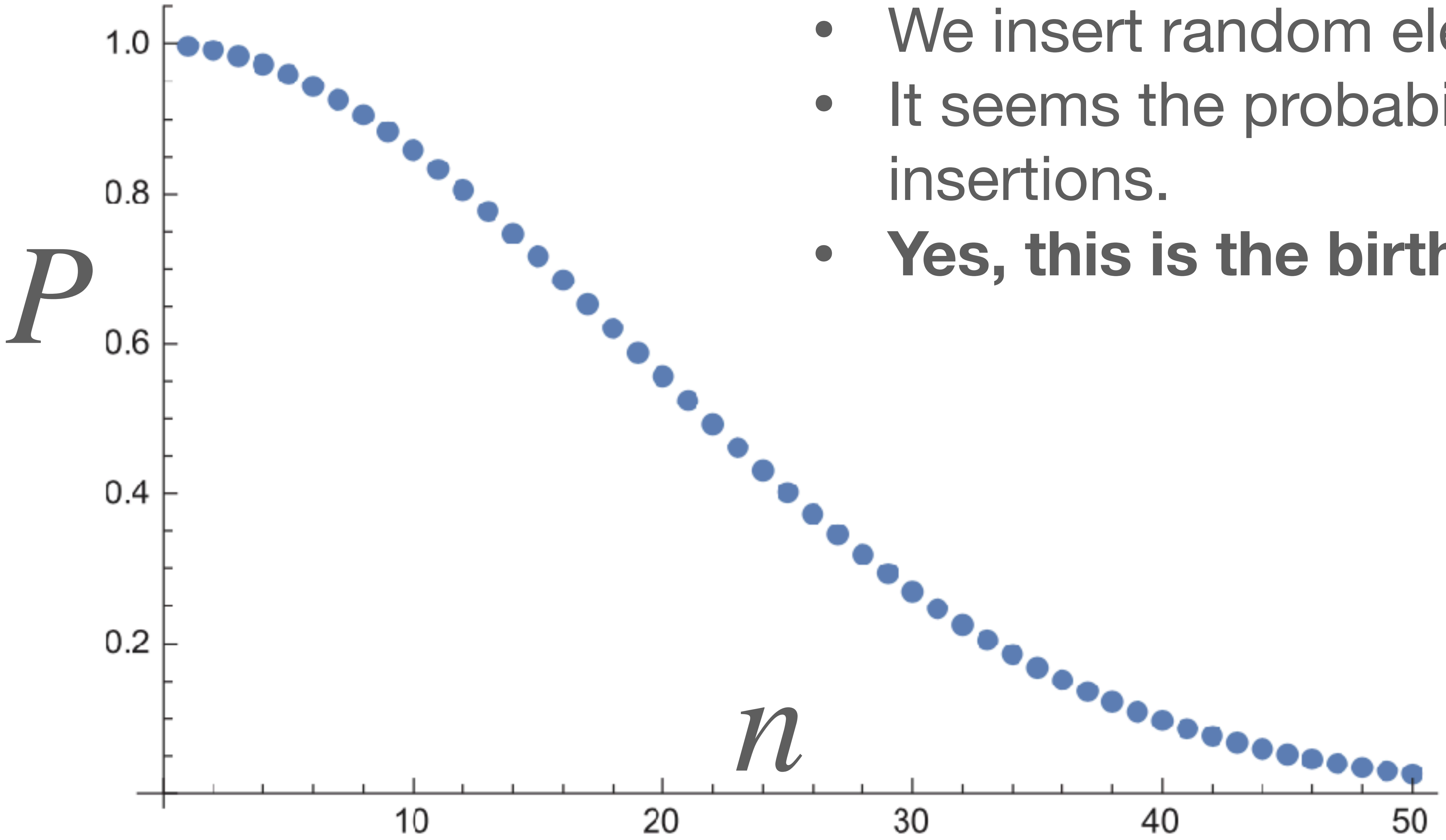
Perfect Hashing: Free of collisions

- **The down side of hashing is we may need to $O(n)$-time to search for an item since collisions may appear.**
- This may be unavoidable in the dynamic case, but how about the **static case**.
- Static case: We have n items and hash table size m. We do not allow insertions /alterations
- Can we find a hash function free of collisions, or at least collision probability is very low ?

- We aim to avoid collisions, and thus, curious about the hash table size m to achieve this goal.
- The probability that the $(i+1)$th insertion does not collide with the previous ones is $(m-i)/m$.
- Then, the probability of obtaining a collision-free hash table of size m is

$$P(\text{no collision}) = \prod_{i=0}^{n-1} \left(\frac{m-i}{m} \right) = \frac{m!}{m^n((m-n)!)}$$

Perfect Hashing: Free of collisions



- Hash table size is **m = 365**.
- We insert random elements from 1 to 365 via a good (!) hash functions
- It seems the probability of collision becomes more than 50% after 23 insertions.
- **Yes, this is the birthday paradox :)**

$$P(\text{no collision}) = \prod_{i=0}^{n-1}\left(\frac{m-i}{m}\right) = \frac{m!}{m^n((m-n)!)}$$

- We are curious about the relation between the elements and the hash table size.
- What should be the size of the hash table that stores n items and have a low collision probability?

$$n = \theta(\sqrt{m}) \rightarrow m = \Theta(n^2)$$

We need quadratic space to keep the items almost collision free !

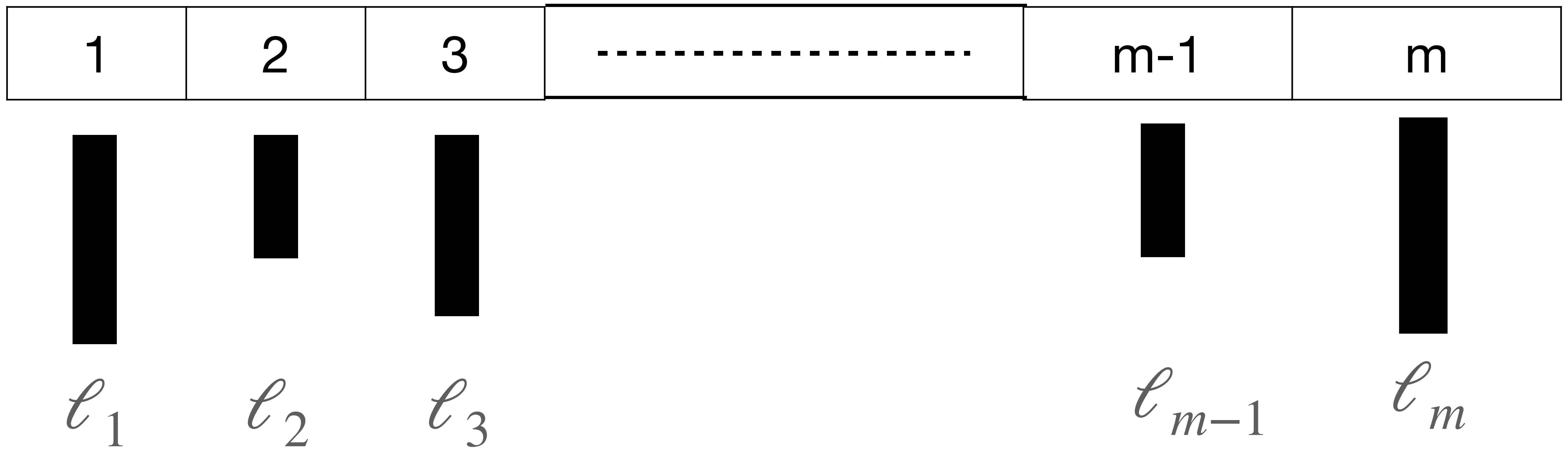
Perfect Hashing: Free of collisions

$$n = \theta(\sqrt{m}) \rightarrow m = \Theta(n^2)$$

We need quadratic space to keep the items almost collision free !

- Quadratic space usage for hash is counter intuitive in many cases
- Instead, how about using multiple hash functions ?
- We analyze two-level perfect hashing now.

Perfect Hashing: Free of collisions



- Assume we use a hash table of size m and we are inserting n items. The number of items mapped to each location is $\ell_1, \ell_2, \ldots, \ell_m$. Due to collisions some cells may be 0 some may be longer than 1.
- We aim each list to be **enough short** satisfying $W = \sum_{i=1}^{m} \ell_i^2 = \theta(n)$**.** Why ?

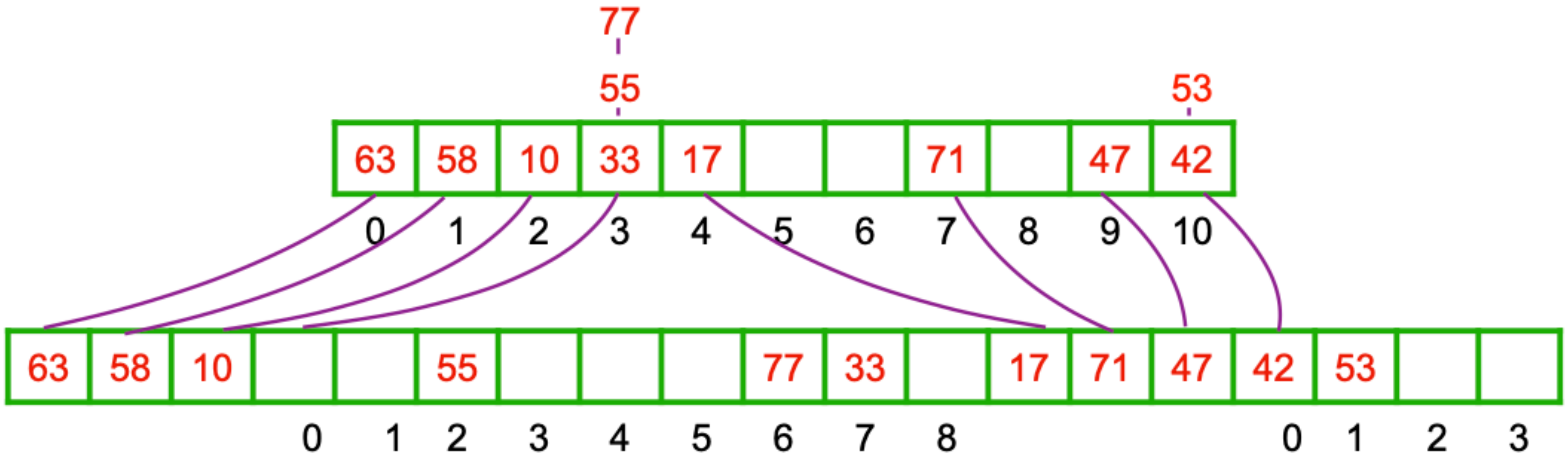
- Skipping the details, it is known that $W < 4n$ can be satisfied in a few trials.

First Level :

- Make the hash table size $m = O(n)$ and expect to have $W < 4n$ with a chosen hash function.
- Try a hash function, compute W. If it is larger than $4n$, try another one, which is expected to have such a one quick.

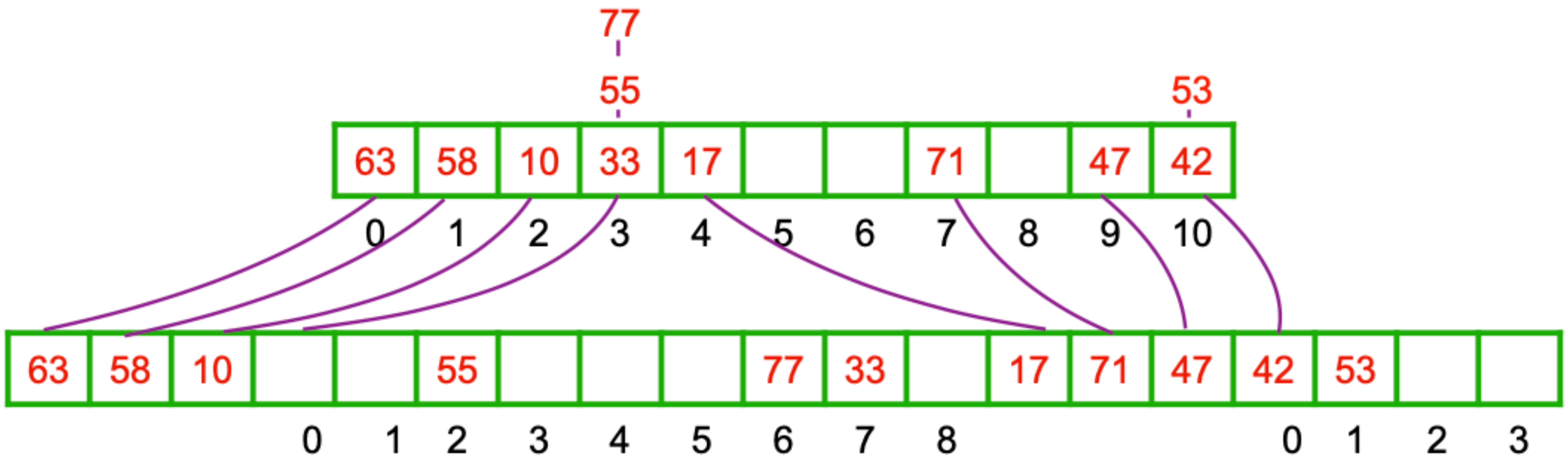
Perfect Hashing: Free of collisions

- Make the hash table size m and expect to have $W < 4m$ with a chosen hash function.
- Try a hash function, compute W. If it is larger than $4m$, try another one, which is expected to have such a one quick.



- Create the second level by allocating ℓ_i^2 element for each bucket i on the first level.
- Notice that for ℓ_i elements we are allocating ℓ_i^2, which is good enough to have a very low collision probability as we discussed in birthday paradox.
- For the second level again choose a hash function, where now it is expected to be a perfect hash with high probability due to space allocated.
- **Now we have a linear space usage and a perfect hash functionality !**

Perfect Hashing: Free of collisions



- How to search for an element?
- Find it is bucket on first level hash with the corresponding hash function.
- On this bucket, we save the locations and the second level hash function for this bucket.
- We apply the second level hash and using the allocated space on the second level, we land on the target position where the search key should be residing.

Constant time search, linear space usage for static dictionaries.

Reading assignment

- Skiena chapter 8.

- Goodrich et al. 10.2