


Hash Table

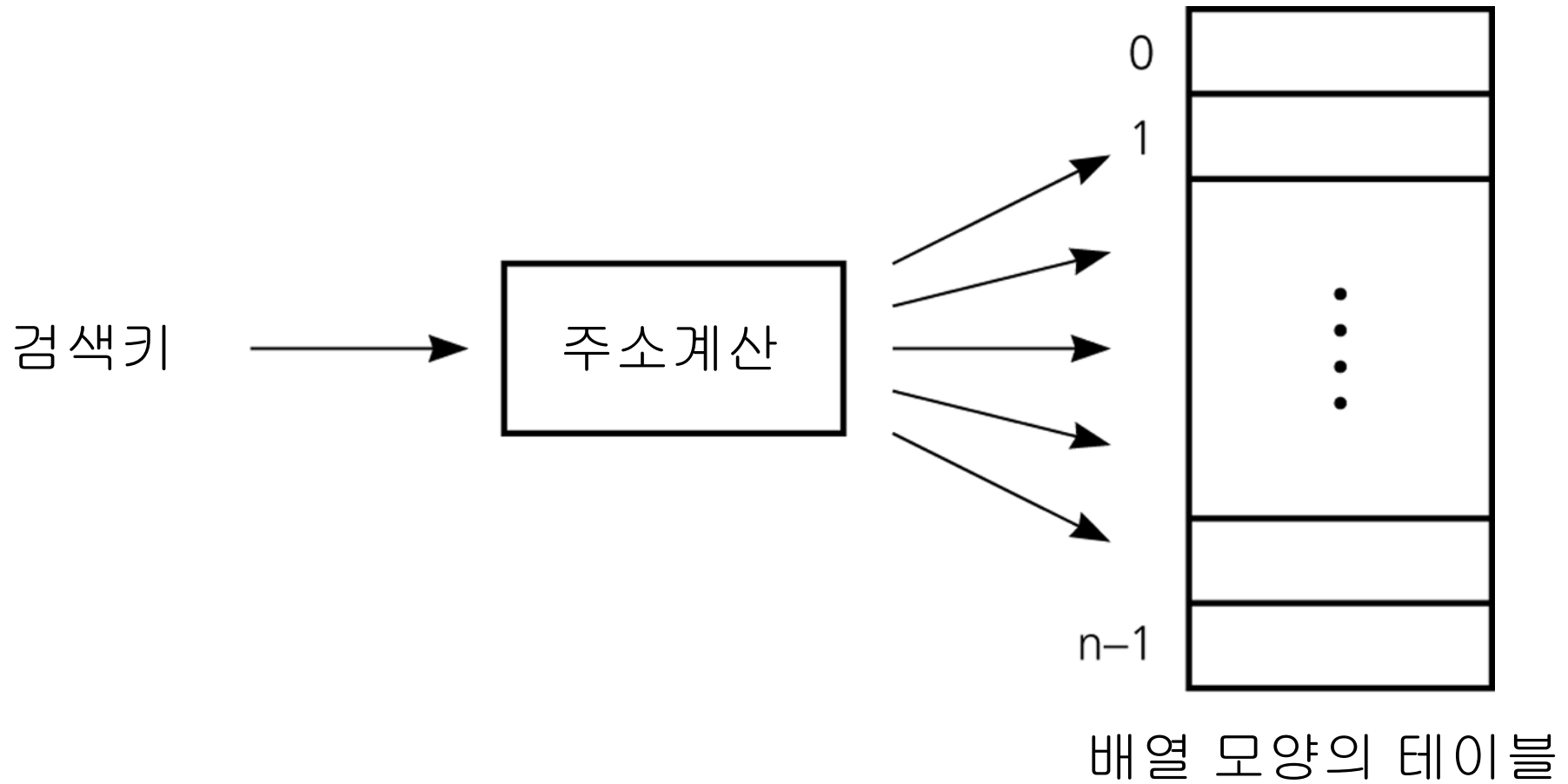
저장/검색의 복잡도

- 
- Array
 - $O(n)$
 - Binary search tree
 - 최악의 경우 $\Theta(n)$
 - 평균 $\Theta(\log n)$
 - Balanced binary search tree(e.g. red-black tree)
 - 최악의 경우 $\Theta(\log n)$
 - B-tree
 - 최악의 경우 $\Theta(\log n)$
 - Balanced binary search tree보다 상수 인자가 작다
 - Hash table
 - 평균 $\Theta(1)$

Hash Table

- 원소가 저장될 자리가 원소의 값에 의해 결정되는 자료구조
- 평균 상수 시간에 삽입, 삭제, 검색
- 매우 빠른 응답을 요하는 응용에 유용
 - 예:
 - 119 긴급구조 호출과 호출번호 관련 정보 검색
 - 주민등록 시스템
- Hash table은 최소 원소를 찾는 것과 같은 작업은 지원하지 않는다

주소 계산



크기 13인 Hash Table에 5 개의 원소가 저장된 예

입력: 25, 13, 16, 15, 7

0	13
1	
2	15
3	16
4	
5	
6	
7	7
8	
9	
10	
11	
12	25

Hash function $h(x) = x \bmod 13$

Hash Function

- Hash ft h , universe U , hash table of size m
 - $h : U \rightarrow \{0, 1, \dots, m-1\}$
- 입력 원소가 hash table에 고루 저장되어야 한다
- 계산이 간단해야 한다
- 여러가지 방법이 있으나 가장 대표적인 것은 division method와 multiplication method이다

- Division Method

- $h(x) = x \bmod m$

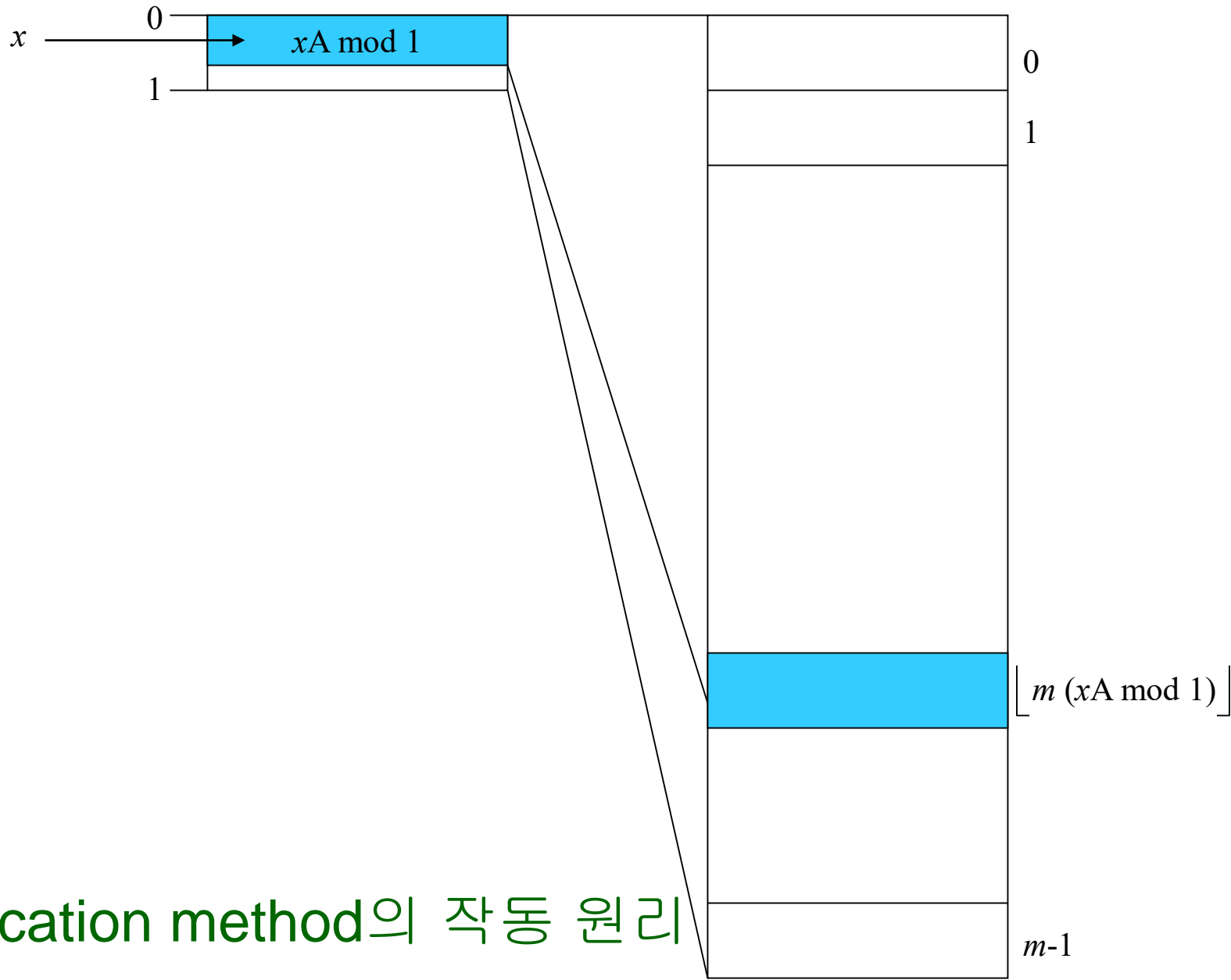
- m : table 사이즈. 대개 prime number(소수)임.

- Multiplication Method

- $h(x) = (xA \bmod 1) * m$

- A : $0 < A < 1$ 인 상수

- m 은 굳이 prime number일 필요 없다. 따라서 보통 2^p 으로 잡는다(p 는 정수)



Multiplication method의 작동 원리

Collision

- Hash table의 한 주소를 놓고 두 개 이상의 원소가 자리를 다투는 것
 - Hashing을 해서 삽입하려 하니 이미 다른 원소가 자리를 차지하고 있는 상황
- Collision resolution 방법은 크게 두 가지가 있다
 - Chaining
 - Open Addressing

Collision의 예

입력: 25, 13, 16, 15, 7

0	13
1	
2	15
3	16
4	
5	
6	
7	7
8	
9	
10	
11	
12	25

← $h(29) = 29 \bmod 13 = 3$

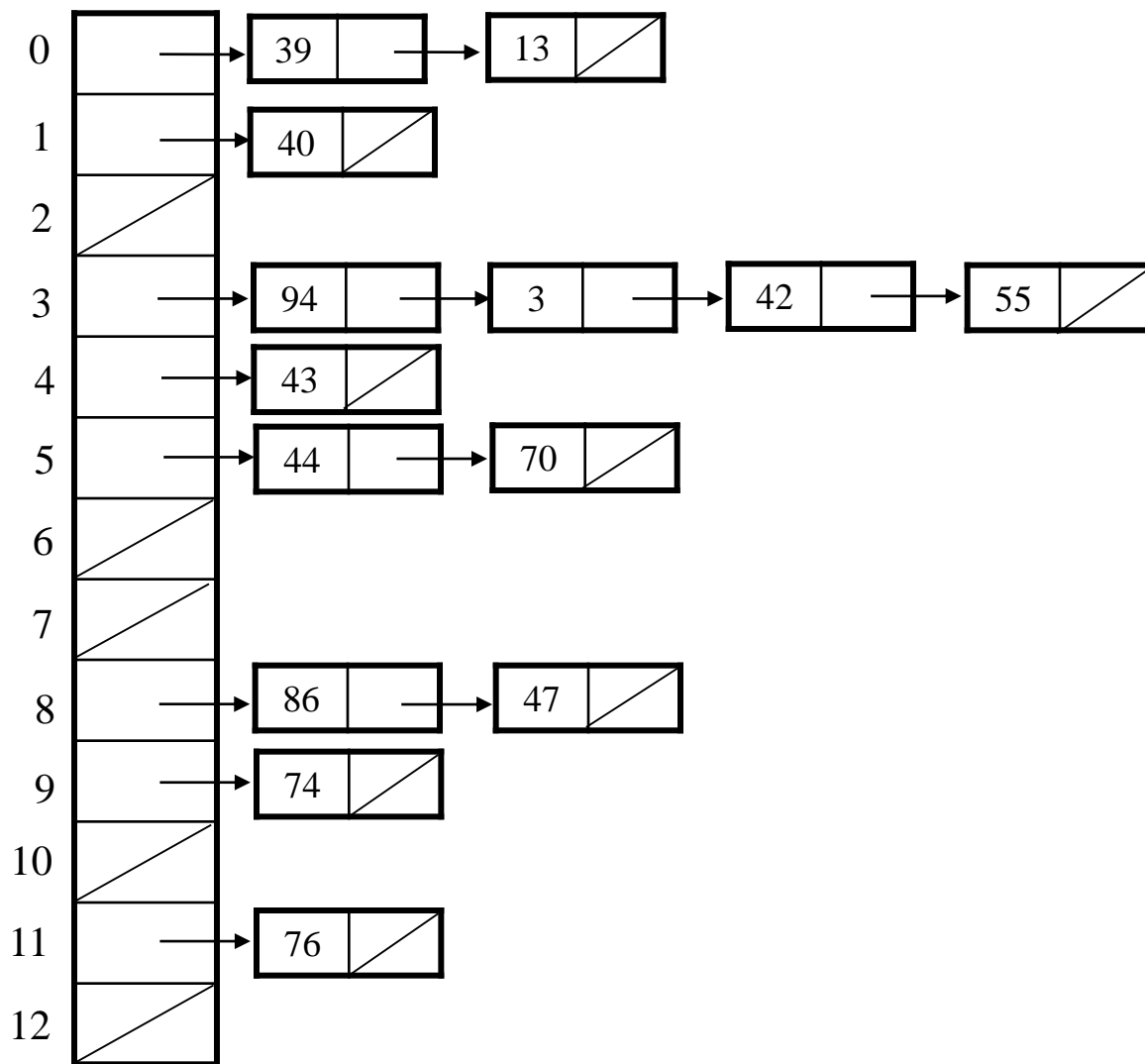
29를 삽입하려 하자 이미
다른 원소가 차지하고 있다!

Hash function $h(x) = x \bmod 13$

Collision Resolution

- Chaining
 - 같은 주소로 hashing되는 원소를 모두 하나의 linked list로 관리한다
 - 추가적인 linked list 필요
 - 단 한번의 hashing으로 okay
- Open addressing
 - Collision이 일어나더라도 어떻게든 주어진 테이블 공간에서 해결한다
 - 추가적인 공간이 필요하지 않다
 - 여러 번의 hashing이 필요할 수 있다

Chaining을 이용한 Collision Resolution의 예



Open Addressing

- 빈자리가 생길 때까지 해시값을 계속 만들어낸다
 - $h_0(x), h_1(x), h_2(x), h_3(x), \dots$
- 중요한 세가지 방법
 - Linear probing
 - Quadratic probing
 - Double hashing

Linear Probing

$$h_i(x) = (h(x) + ci) \bmod m$$

예(: 입력 순서 25, 13, 16, 15, 7, 28, 31, 20, 1, 38)

0	13
1	
2	15
3	16
4	28
5	
6	
7	7
8	
9	
10	
11	
12	25

0	13
1	
2	15
3	16
4	28
5	31
6	
7	7
8	20
9	
10	
11	
12	25

0	13
1	1
2	15
3	16
4	28
5	31
6	38
7	7
8	20
9	
10	
11	
12	25

$$h_i(x) = (h(x) + i) \bmod 13$$

Linear Probing은 Primary Clustering에 취약하다

Primary clustering: 특정 영역에 원소가 몰리는 현상

0	
1	
2	15
3	16
4	28
5	31
6	44
7	
8	
9	
10	
11	37
12	

← Primary clustering의 예

Quadratic Probing

$$h_i(x) = (h(x) + c_1 i^2 + c_2 i) \bmod m$$

예: 입력 순서 15, 18, 43, 37, 45, 30

0	
1	
2	15
3	
4	43
5	18
6	45
7	
8	30
9	
10	
11	37
12	



$$h_i(x) = (h(x) + i^2) \bmod 13$$

Quadratic Probing은 Secondary Clustering에 취약하다

Secondary clustering: 여러 개의 원소가 동일한 초기 해시 함수값을 갖는 현상

0	
1	
2	15
3	28
4	
5	54
6	41
7	
8	21
9	
10	
11	67
12	

← Secondary clustering의 예

$$h_i(x) = (h(x) + i^2) \bmod 13$$

Double Hashing

$$h_i(x) = (h(x) + if(x)) \bmod m$$

예: 입력 순서 15, 19, 28, 41, 67

0	
1	
2	15
3	
4	67
5	
6	19
7	
8	
9	28
10	
11	41
12	

$$h_0(15) = h_0(28) = h_0(41) = h_0(67) = 2$$

$$h_1(67) = 3$$

$$h_1(28) = 8$$

$$h_1(41) = 10$$

$$h(x) = x \bmod 13$$

$$f(x) = (x \bmod 11) + 1$$

$$h_i(x) = (h(x) + if(x)) \bmod 13$$

삭제시 조심할 것

0	13
1	1
2	15
3	16
4	28
5	31
6	38
7	7
8	20
9	
10	
11	
12	25

(a) 원소 1 삭제

0	13
1	
2	15
3	16
4	28
5	31
6	38
7	7
8	20
9	
10	
11	
12	25

(b) 38 검색, 문제발생

0	13
1	DELETED
2	15
3	16
4	28
5	31
6	38
7	7
8	20
9	
10	
11	
12	25

(c) 표식을 해두면 문제없다

Hash Table에서의 검색 시간


- Load factor α
 - Hash table 전체에서 얼마나 원소가 차 있는지를 나타내는 수치
 - Hash table에 n 개의 원소가 저장되어 있다면
$$\alpha = \frac{n}{m} \text{ 이다}$$
- Hash table에서의 검색 효율은 load factor와 밀접한 관련이 있다

Assumption

- Hash ft h is computable in $O(1)$ time
- h distributes keys uniformly in the table
- All elements of U occurs w/ equal probability as inputs

Chaining에서의 검색 시간

Under the above assumption,
a search takes $\theta(1 + \alpha)$ on average


$$\theta(\max(1, \alpha))$$

Open Addressing에서의 검색 시간

Assumption (**uniform hashing**)

$h_0(x), h_1(x), \dots, h_{m-1}(x)$ 가 $\{0, 1, \dots, m-1\}$ 의 permutation을 이루고, 모든 permutation은 같은 확률로 일어난다

Open Addressing에서의 검색 시간

Thm1: The expected #probes in an unsuccessful search or an insertion is at most $\frac{1}{1-\alpha}$

<proof>

$p_i = \text{Pr}(\text{exactly } i \text{ probes access occupied slots})$

$q_i = \text{Pr}(\text{at least } i \text{ probes access occupied slots})$

$$\begin{aligned}
 \text{Expected \# probes} &= 1 + \sum_{i \geq 1} i p_i \\
 &= 1 + \sum_{i \geq 1} i (q_i - q_{i+1}) \\
 &= 1 + \sum_{i \geq 1} q_i \\
 &\leq 1 + \sum_{i \geq 1} \alpha^i \longleftarrow q_i = \frac{n}{m} \frac{n-1}{m-1} \cdots \frac{n-i+1}{m-i+1} \leq \left(\frac{n}{m}\right)^i = \alpha^i \\
 &= \frac{1}{1-\alpha}
 \end{aligned}$$

Open Addressing에서의 검색 시간

Thm 2: The expected #probes in a successful search is

$$\text{at most } \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

<proof>

- The load factor α right after i^{th} key had been inserted was $\frac{i}{m}$
- If x is the $(i + 1)^{th}$ key inserted, then the expected #probes in a successful search for x is, by the previous thm, at most $\frac{1}{1-\frac{i}{m}}$
- Average over all keys

$$\begin{aligned} \frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} &= \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} \\ &\leq \frac{1}{\alpha} \int_0^1 \frac{1}{1-x} dx \\ &= \frac{1}{\alpha} \ln \frac{1}{1-\alpha} \end{aligned}$$

Load Factor가 우려스럽게 높아지면

- Load factor가 높아지면 일반적으로 hash table의 효율이 떨어진다
- 일반적으로, threshold을 미리 설정해 놓고 load factor가 이에 이르면
 - Hash table의 크기를 두 배로 늘린 다음 hash table에 저장되어 있는 모든 원소를 다시 hashing하여 저장한다



Hash Table의 창의적 이용예: Minhash

- Suggested by Andrei Broder, 1997
- Min-wise locality sensitive permutation hashing
- 두 집합의 유사성을 빨리 판별하게 함
 - Vector의 유사성
 - 문서의 유사성
 - 웹페이지의 유사성
 - 주식 패턴의 유사성
 - ...

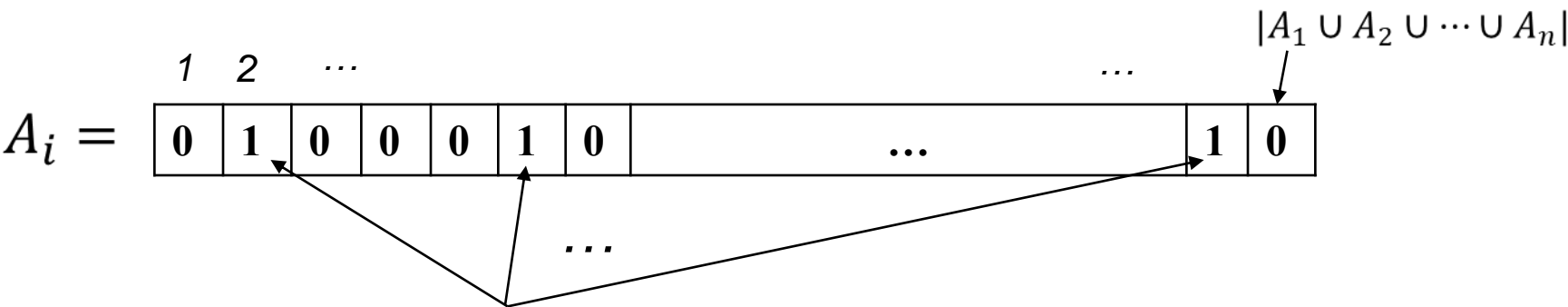
Jaccard Similarity

- 두 집합 A, B
 - $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$
 - 아주 많은 집합 A_1, A_2, \dots, A_n 이 있고 그들간의 pairwise similarity를 다 계산해야 한다면?
 - 예: $n = 10^5$
 - 이들간의 pair는 대략 $50\text{억}(\frac{10^{10}}{2})$ 개
 - 엄청난 시간이 든다
- 각 집합이 크면

Binary Vector, or Signature Vector



$A_1 \cup A_2 \cup \dots \cup A_n$ 의 각 원소당 한 bit



A_i 에 포함된 원소

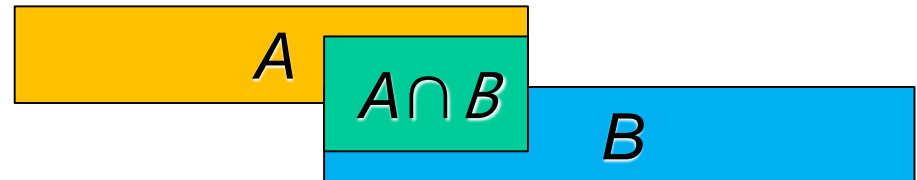
$h_{min}(S)$: A Permutation Hashing

$h(x)$: a hash function

$h_{min}(S)$: a hash ft returning the index of the member $x \in S$ that minimizes $h(x)$

$h_{min}(S) \in \{1, 2, \dots, |S|\}$, $S = A_1 \cup A_2 \cup \dots \cup A_n$

➡ $\text{Prob}(h_{min}(A) = h_{min}(B)) = J(A, B)$



Field에서의 적용

한 개의 $h_{min}()$ 으로는 확률만 맞을 뿐

충분히 많은 $h_{min}^1(), h_{min}^2(), \dots, h_{min}^k()$ 준비

모든 $A_i, i = 1, 2, \dots, n$ 에 대해

$h_{min}^1(A_i), h_{min}^2(A_i), \dots, h_{min}^k(A_i)$ 를 계산한다(단 한번)

$$\begin{aligned} \rightarrow J(A_i, A_j) &= \frac{\text{일치한 } h_{min}() \text{의 수}}{k} \\ &= \frac{\sum_{r=1}^k \delta(h_{min}^r(A_i), h_{min}^r(A_j))}{k}, \end{aligned} \quad \delta(a, b) = \begin{cases} 1, & \text{if } a = b \\ 0, & \text{if } a \neq b \end{cases}$$

Monte Carlo approach의 일종
(random sampling based...)

