Hash Tables

Reminder:
Basics of Hash Tables

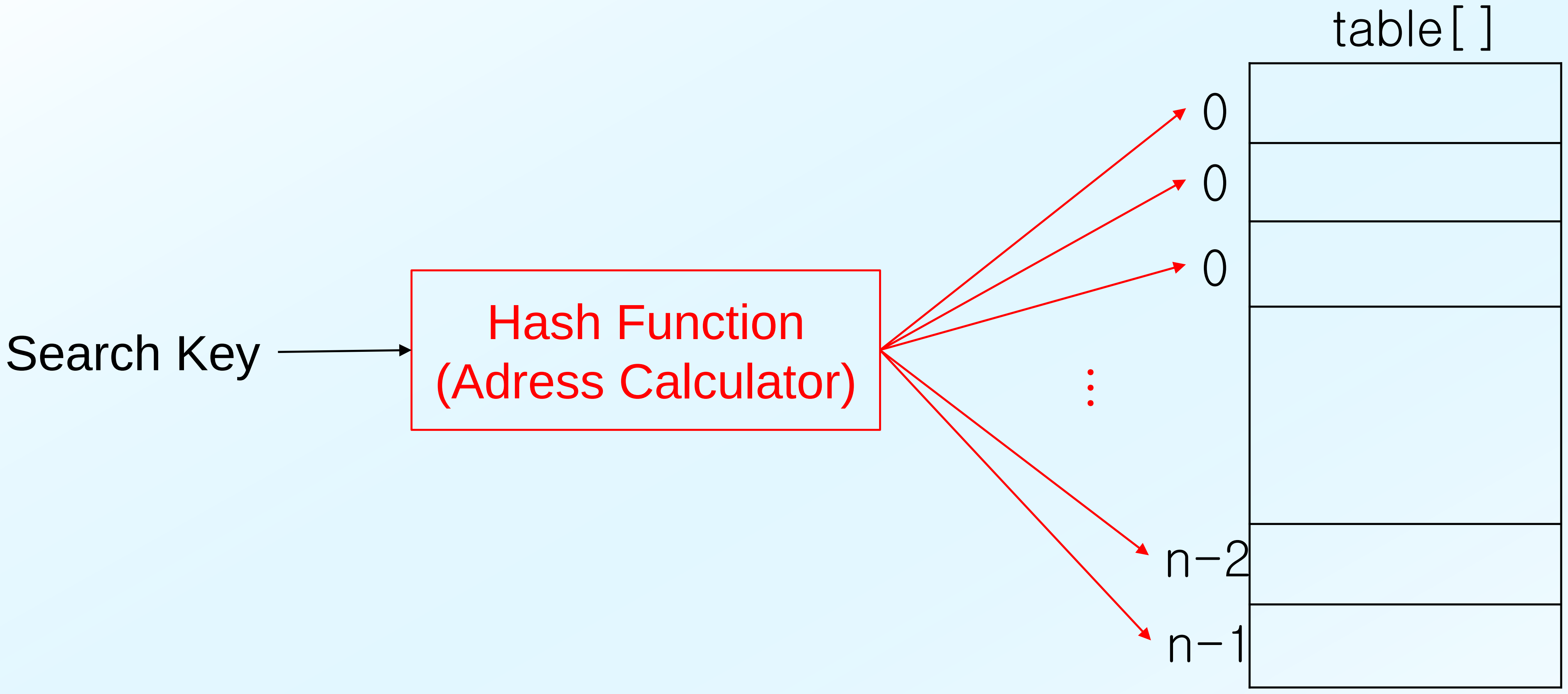
Want $\Theta(1)$-Time Operations

- Array or linked list
  - Overall $O(n)$ time
- Binary search trees
  - Expected $\theta(\log n)$-time search, insertion, and deletion
  - But, $\theta(n)$ in the worst case
- Balanced binary search trees
  - Guarantees $O(\log n)$-time search, insertion, and deletion
  - Red-black tree, AVL tree
- Balanced k-ary trees
  - Guarantees $O(\log n)$-time search, insertion, and deletion w/ smaller constant factor
  - 2-3 tree, 2-3-4 tree, B-trees
- Hash table
  - Expected $\theta(1)$-time search, insertion, and deletion

Hash Tables

- Stack, queue, priority queue
  - do not support *search* operation
- Hash table support quick search, insertion, and deletion
  - But, does not support finding the minimum (or maximum) element
- Applications that need very fast operations
  - 119 emergent calls and locating caller's address
  - Air flight information system
  - 주민등록 시스템

Address Calculator



Hash Functions

- Toy functions
  - Selection digits
    - $h(001364825) = 35$
  - Folding
    - $h(001364825) = 1190$
- Modulo arithmetic
  - $h(x) = x \bmod tableSize$
  - $tableSize$ is recommended to be prime
- Multiplication method
  - $h(x) = (xA \bmod 1) * tableSize$
  - A: constant in (0, 1)
  - $tableSize$ is not critical, usually 2^p for an integer p

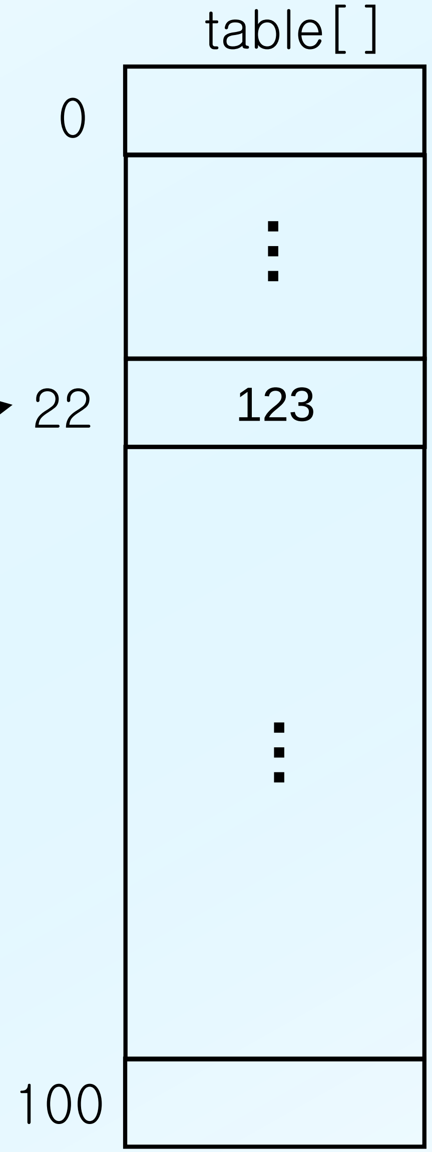
Collision Resolution

Collision:

a key maps to an occupied location in the hash table

$h(224) = 224 \bmod 101 = 22$

table[22] is occupied



An example: $h(x) = x \bmod 101$

Collision resolution

- resolves collision by a seq. of hash values
- $h_0(x)(=h(x)), h_1(x), h_2(x), h_3(x), \ldots$
- The most important in hash tables

Collision-Resolution Methods

Open addressing (resolves in the table)

- Linear probing
  - $h_i(x) = (h_0(x) + i)$ % $tableSize$
- Quadratic probing
  - $h_i(x) = (h_0(x) + i^2)$ % $tableSize$
- Double hashing
  - $h_i(x) = (h_0(x) + i \bullet \beta(x))$ % $tableSize$
  - $\beta(x)$: another hash function

Simple version

Full version:

$h_i(x) = (h_0(x) + ai+b)$ % $tableSize$

Full version:

$h_i(x) = (h_0(x) + ai^2+bi+c)$ % $tableSize$

Separate chaining

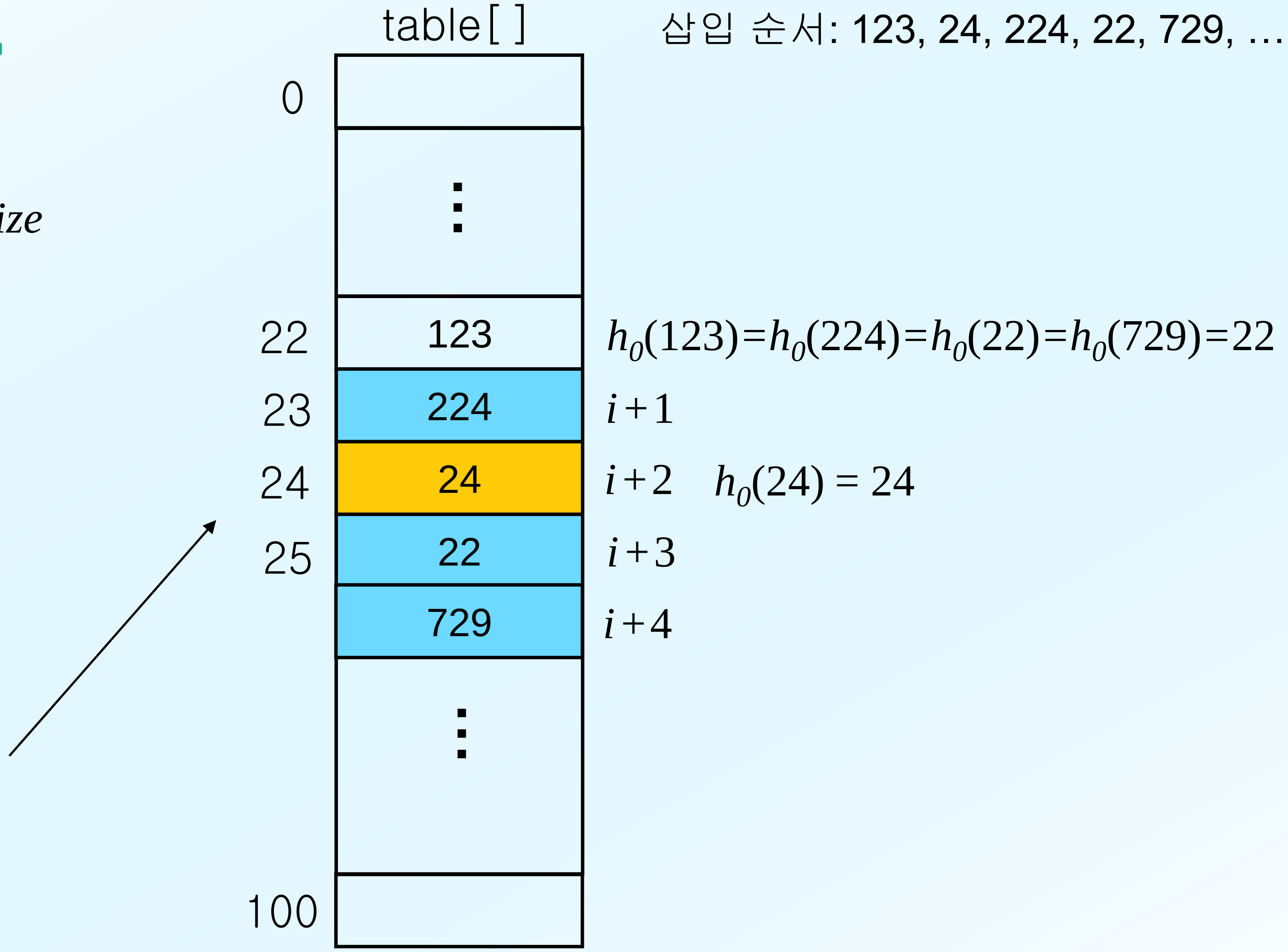
- Each $table[i]$ is maintained by a linked list

Open Addressing

Linear probing

$h_i(x) = (h_0(x) + i) \bmod tableSize$

bad w/ primary clustering



Linear probing with

$h_i(x) = (h_0(x) + i) \bmod 101$

Open Addressing

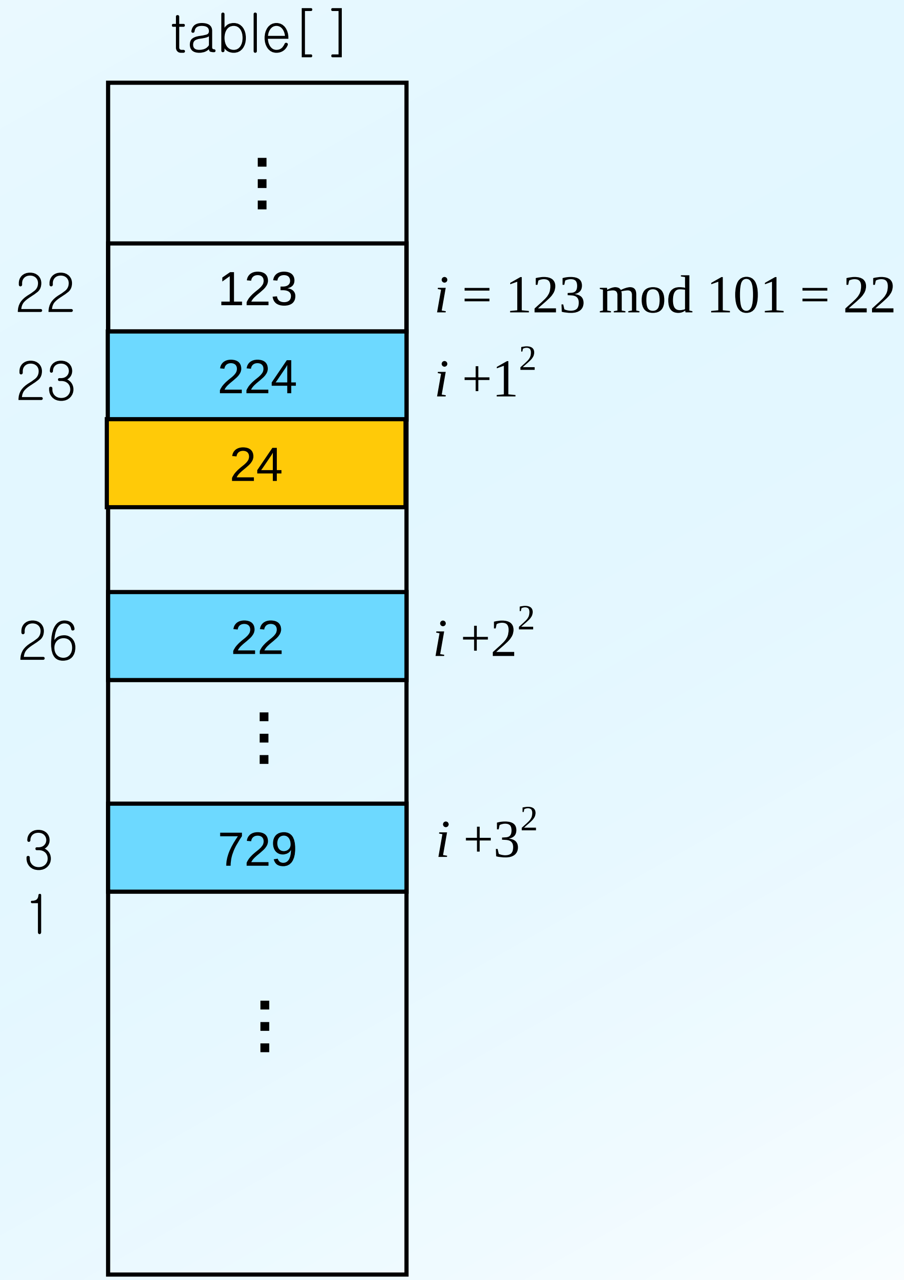
Quadratic probing

$h_i(x) = (h_0(x) + i^2) \bmod tableSize$

bad w/ secondary clustering

Quadratic probing with

$h_i(x) = (h_0(x) + i^2) \bmod 101$



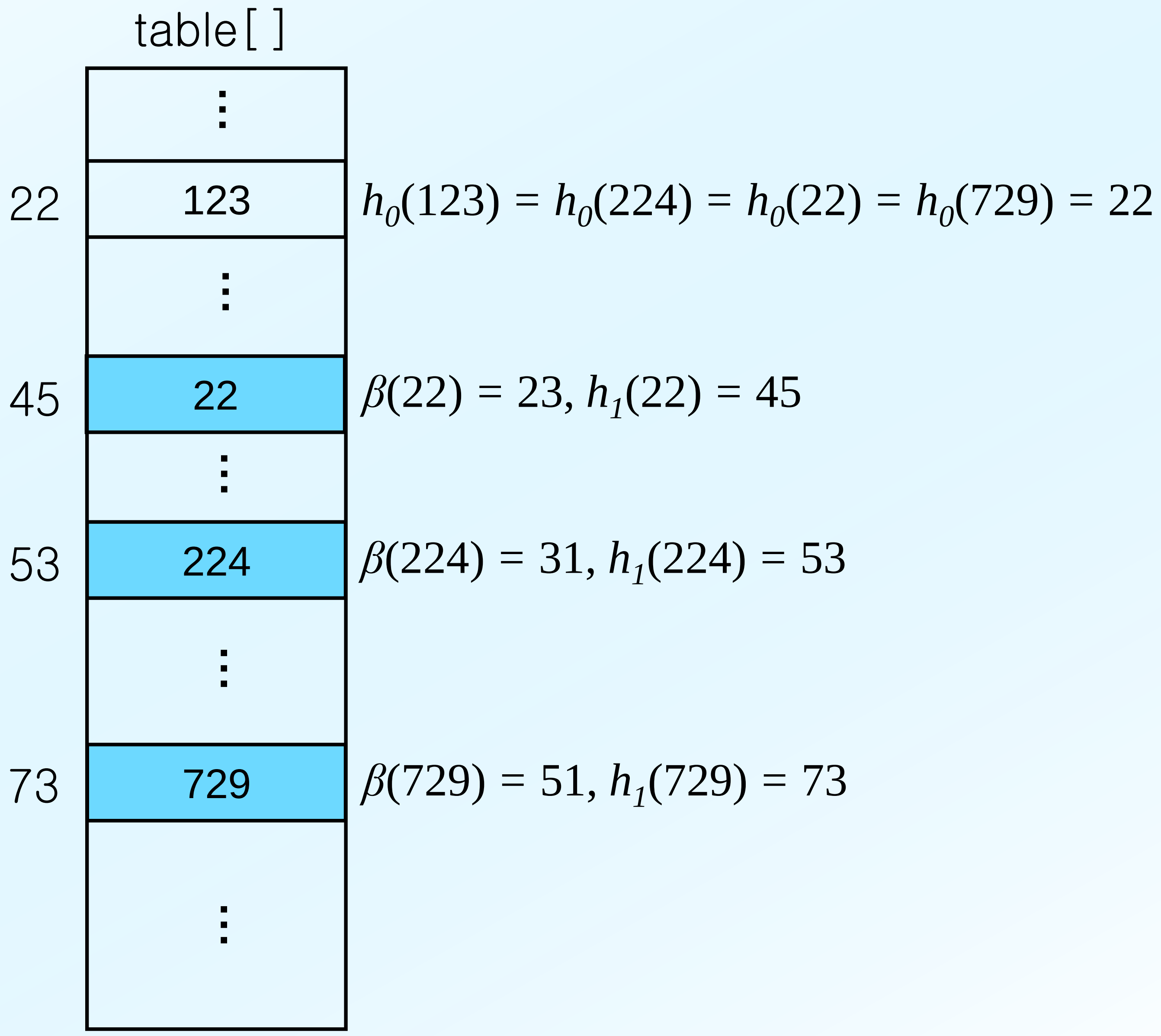
Open Addressing

Double hashing

$$h_i(x) = (h_0(x) + i\beta(x)) \bmod 101$$

Double hashing with

$$h_0(x) = x \bmod 101$$
$$\beta(x) = 1 + (x \bmod 97)$$



Be Careful in Deletion

Hash function:

$$h_i(x) = (h_0(x) + i) \bmod 13$$

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

(a) Delete element 1

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

(b) Search 38, wrong result!

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

(c) Okay: marking with DELETED

Insertion

```
hashInsert(x):
◀ table[]: hash table, x: new key to insert
    if (table[h(x)] is not occupied)
        table[h(x)] ← x
    else
        Find an appropriate location k by a collision-resolution method
        table[k] ← x
    numItems++
```

Deletion

```
hashDelete(x):
◀ table[]: hash table, x: key to delete
    Find the location k of x by search
    if (search was successful)
        table[k] ← DELETED
        numItems--
```

When the Load Factor is Higher than Wanted

$$\alpha = \frac{\text{\# of occupied slots}}{\text{hash table size}}$$

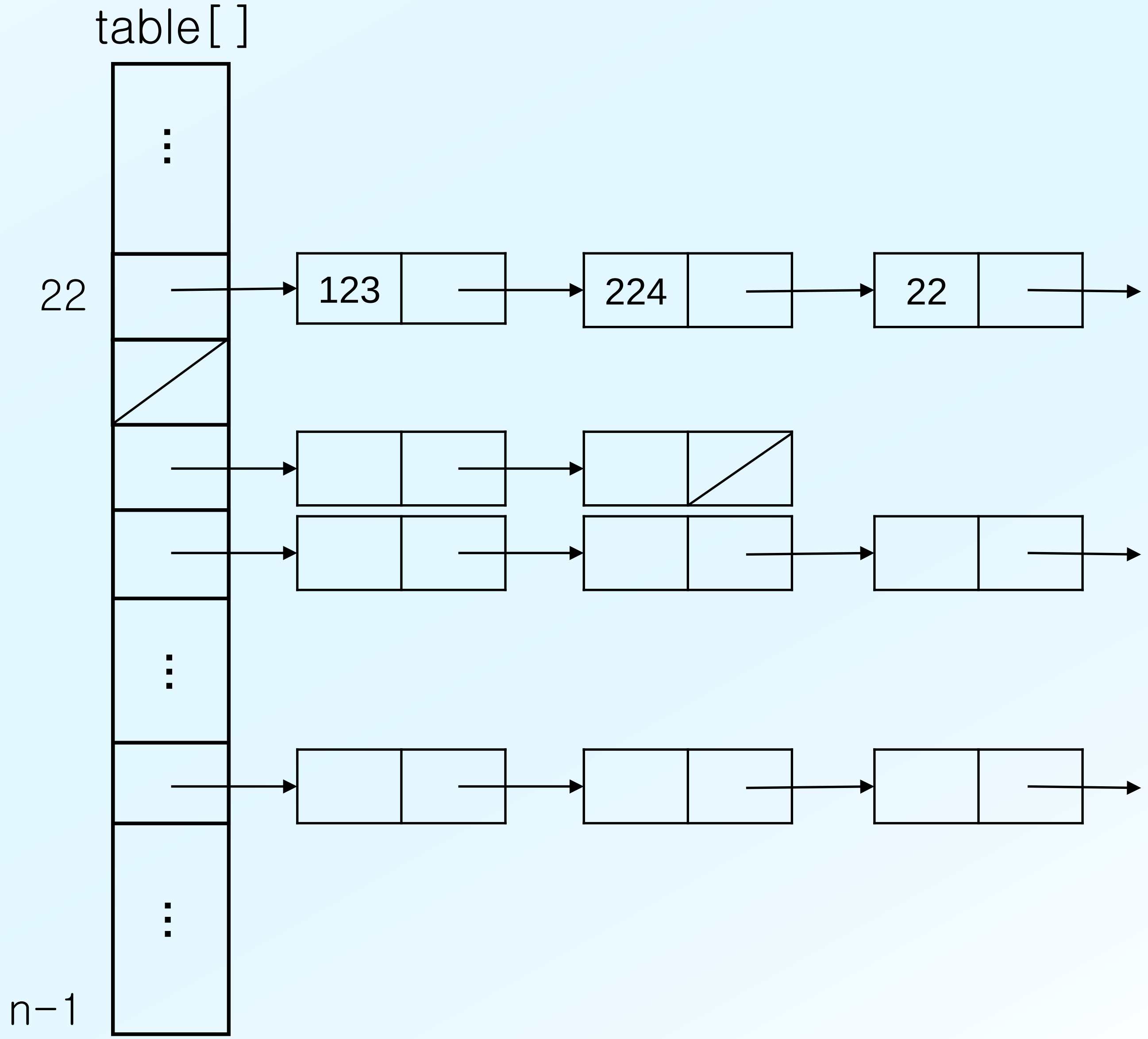
- A hash table performs bad when the load factor(α) is too high
- Generally, set a threshold and if the load factor surpasses it
  - Double the size of the hash table and

    rehash all the elements in the table

Separate Chaining

Table[ ] is a header array of linked lists

No interference bet'n keys not collided
(Open addressing may interfere…)



Operations in Chained Hash Table

search(table[], x):
    Search x in the list table[$h(x)$]

insert(table[], x):
    Insert x in the list table[$h(x)$]

delete(table, x):
    Delete x in the list table[$h(x)$]

Observation

- No difference among probing methods when the load factor is low
- Successful search follows the same path as that of insertion

Internal/External Hashing

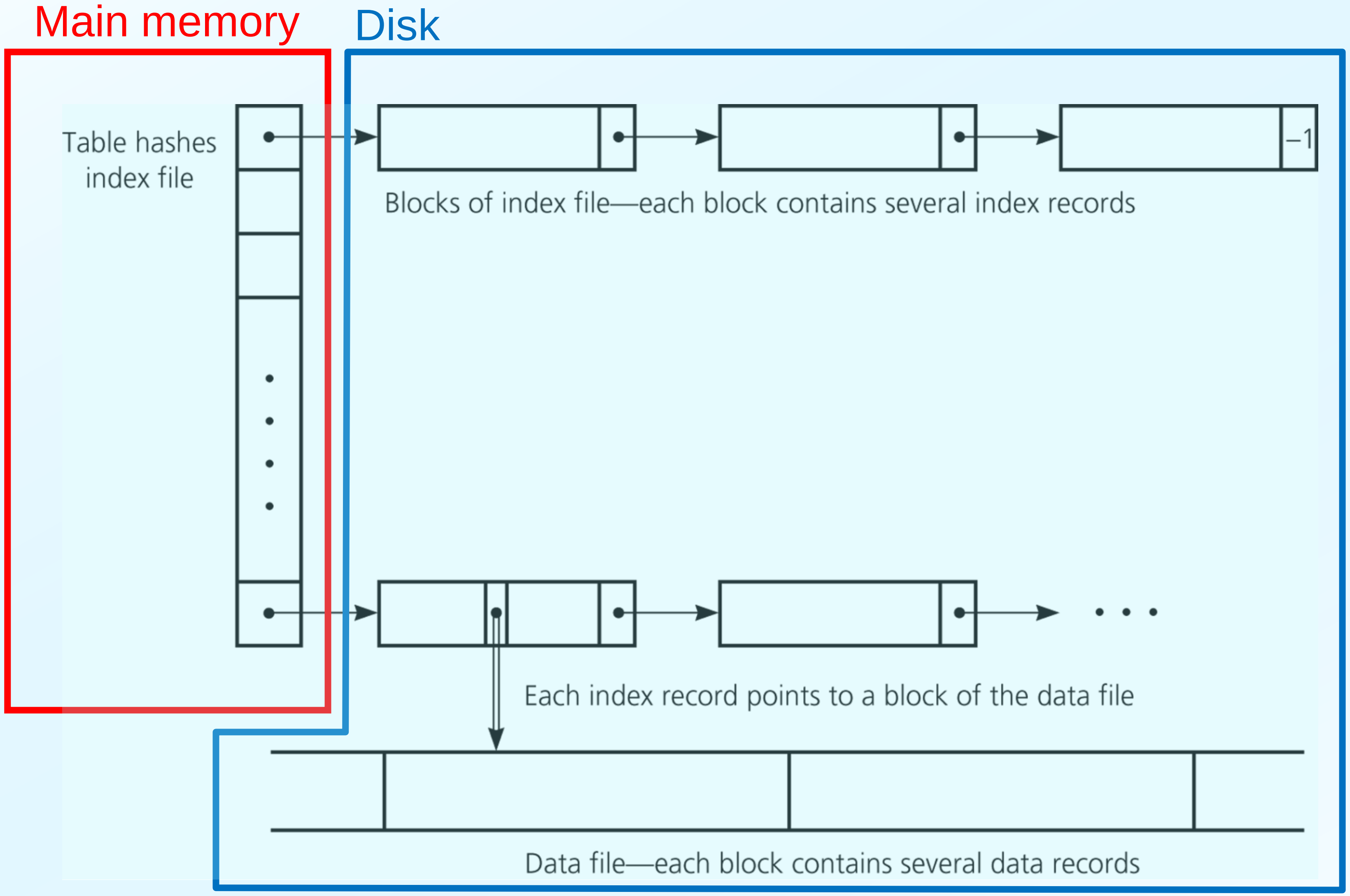
- Hash table is

  in the main memory(internal hashing) or

  in the disk(external hashing)

- In an external hashing, the # of disk accesses is critical

External Hash Table



Efficiency of Hash Tables

Search Time in Chaining

Assuming a uniform distribution of data,

a search takes $\Theta(\max(1, \alpha))$ on average

Search Time in Open Addressing

Assumption (uniform hashing)

- $h_0(x), h_1(x), \ldots, h_{m-1}(x)$ is a permutation of $\{0, 1, \ldots, m-1\}$
- Every permutation is equally likely

Note: collision probability = $\frac{n}{m}$

$\begin{cases} \text{m: table size} \\ \text{n: \# of elements in the table(= \# of occupied slots)} \end{cases}$

[Theorem 1]

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

<proof>

p_i = Pr(exactly i probes access occupied slots)

q_i = Pr(at least i probes access occupied slots)

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

[Theorem 2]

The expected #probes in a successful search
is at most $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$

Note: a successful search exactly follows the path of insertion

<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

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

A Creative Utilization of Hash Tables

Minhash

- Suggested by Andrei Broder, 1997
- Min-wise locality sensitive permutation hashing
- Fast computation of similarity of two sets is possible

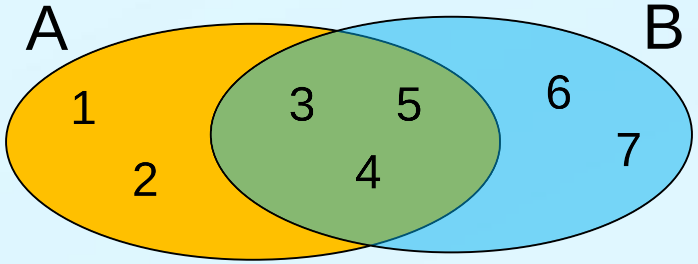
$$\text{Similarity of two} \begin{cases} \text{vectors} \\ \text{documents} \\ \text{web pages} \\ \text{stock patterns} \\ \ldots \end{cases}$$

Jaccard Similarity

Two sets A, B

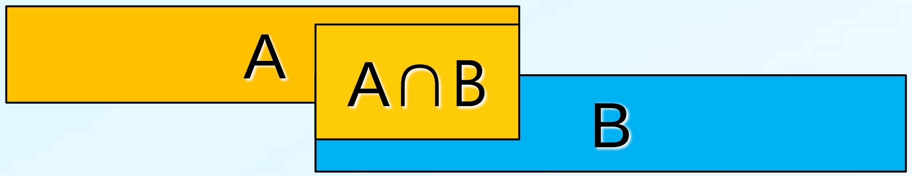
Jaccard similarity

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$



e.g. A={1, 2, 3, 4, 5}
B={3, 4, 5, 6, 7} → J(A,B) = $\frac{3}{7}$

For sets $A_1, A_2, \ldots, A_n$,
we often need to computer their pairwise similarities or similarity to another set B



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

example

$S = \{a, b, c, d, e\}$

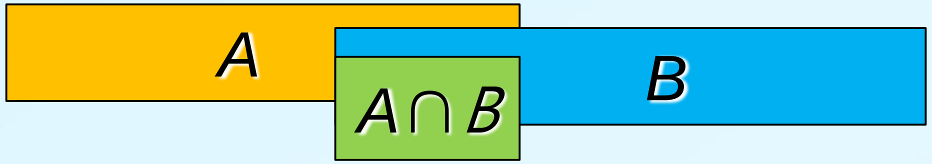
$h(a)\ h(b)\ h(c)\ h(d)\ h(e)\}$

minimum

Then, $h_{min}(S) = d$

$h(x)$: a hash function

$h_{min}(S) = x \in S$ that minimizes $h(x)$



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

Usage in Fields

Using one $h_{min}(\,)$ just probabilistically matches with Jaccard similarity

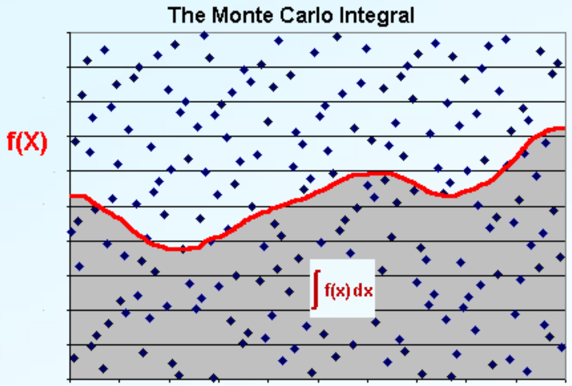
Prepare many enough $h_{min}(\,)$'s: $h^1_{min}(\,), h^2_{min}(\,), \ldots, h^k_{min}(\,)$ ← k different hash functions

For all $A_i, i = 1,2, \ldots, n$, compute (just one time) $h^1_{min}(\,), h^2_{min}(\,), \ldots, h^k_{min}(\,)$

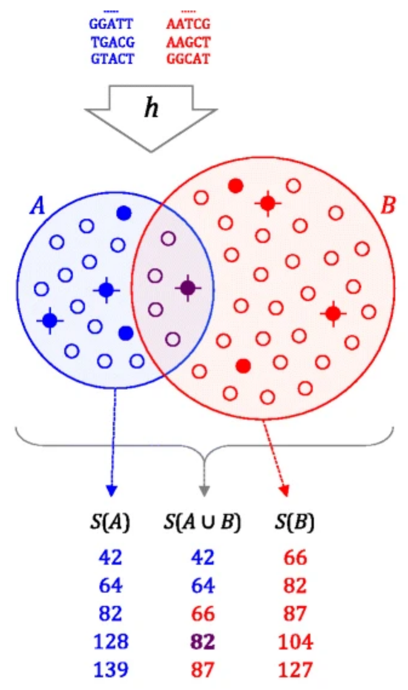
$$J(A_i, A_j) = \frac{\#\text{ of the same } h_{min}\text{'s}}{k}$$

$$= \frac{\sum_{r=1}^{k} \delta(h^r_{min}(A_i),\ h^r_{min}(A_j))}{k}, \qquad \delta(a,b) = \begin{cases} 1, \text{if } a = b \\ 0, \text{if } a \neq b \end{cases}$$

An example of Monte Carlo approximation
(random sampling based…)



Applying Minhash to DNA Pairwise Similarity



$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \approx \frac{|S(A \cup B) \cap S(A) \cap S(B)|}{|S(A \cup B)|}$$

Overview of the MinHash bottom sketch strategy for estimating the Jaccard index. First, the sequences of two datasets are decomposed into their constituent k-mers (*top*, *blue and red*) and each k-mer is passed through a hash function h to obtain a 32- or 64-bit hash, depending on the input k-mer size. The resulting hash sets, A and B, contain $|A|$ and $|B|$ distinct hashes each (*small circles*). The Jaccard index is simply the fraction of shared hashes (*purple*) out of all distinct hashes in A and B. This can be approximated by considering a much smaller random sample from the union of A and B. MinHash sketches $S(A)$ and $S(B)$ of size $s = 5$ are shown for A and B, comprising the five smallest hash values for each (*filled circles*). Merging $S(A)$ and $S(B)$ to recover the five smallest hash values overall for $A \cup B$ (*crossed circles*) yields $S(A \cup B)$. Because $S(A \cup B)$ is a random sample of $A \cup B$, the fraction of elements in $S(A \cup B)$ that are shared by both $S(A)$ and $S(B)$ is an unbiased estimate of $J(A,B)$

A good example,
although they made some variation