Hash Table

1. Introduction
2. Collision Resolution
3. External Hashing

1. Introduction

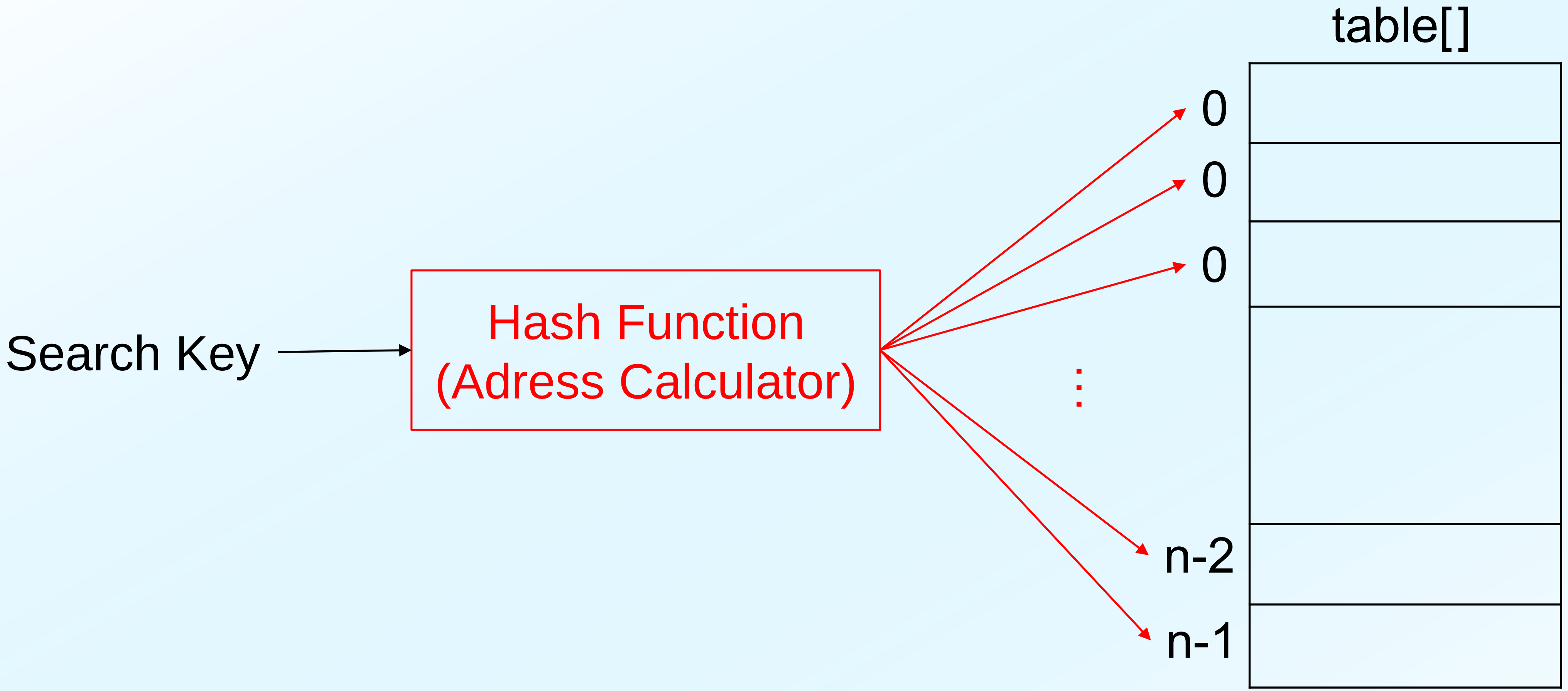
Want $\Theta(1)$-Time Operations

- 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

2. Collision Resolution

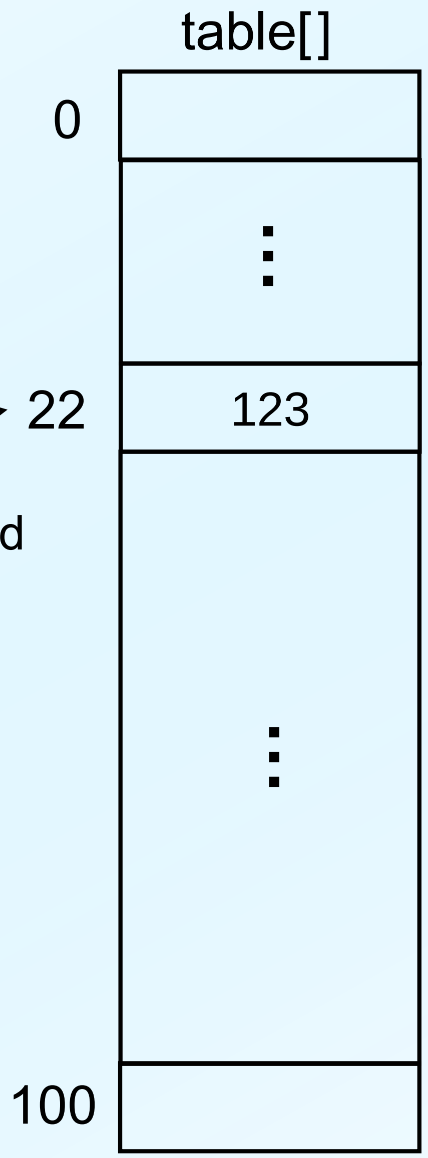
Collision Resolution

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 core of hash tables

Collision-Resolution Methods

Open addressing (resolves in the array)

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

Simple version

Full version:

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

Separate chaining

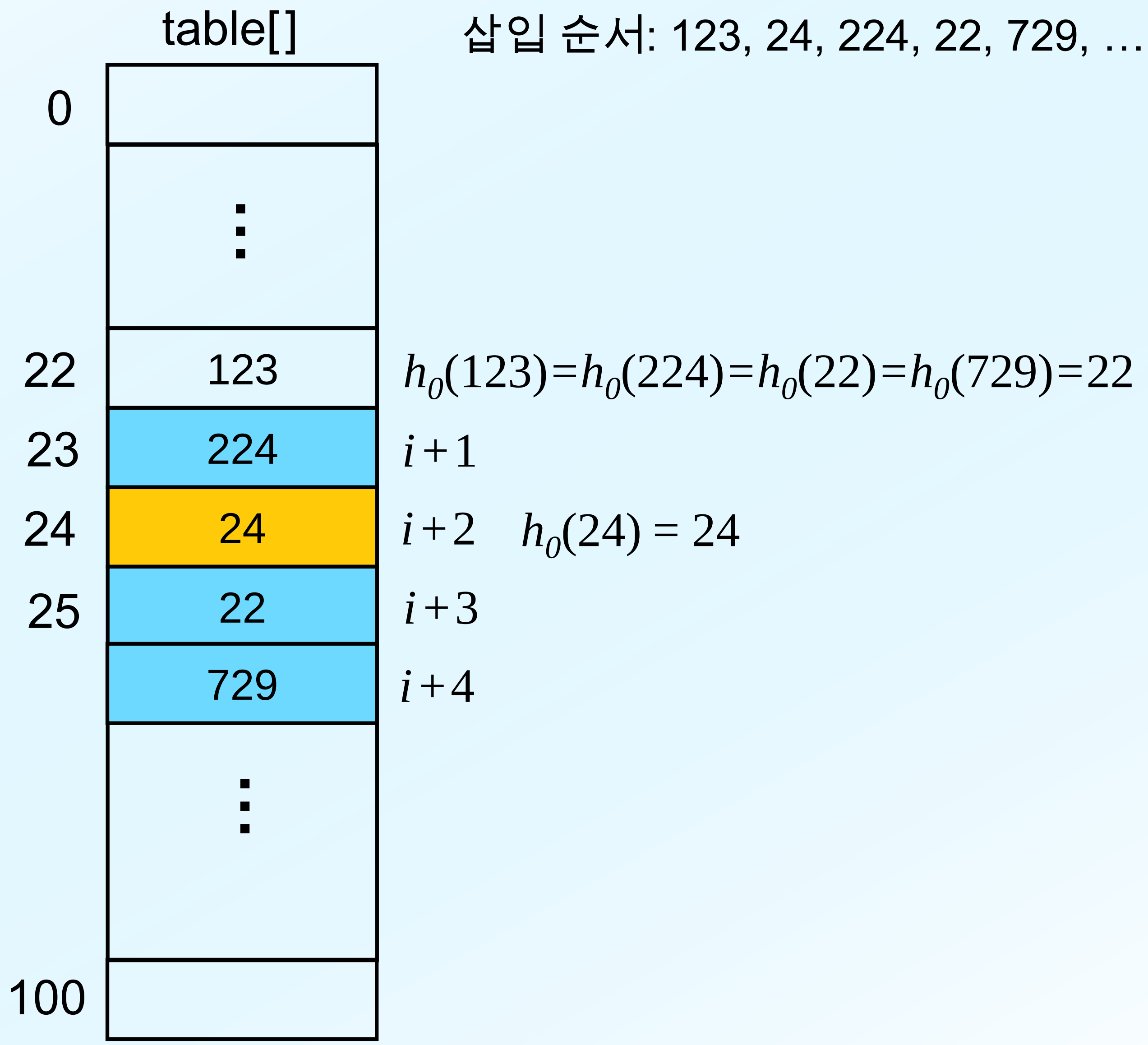
- Each $table[i]$ is maintained by a linked list

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

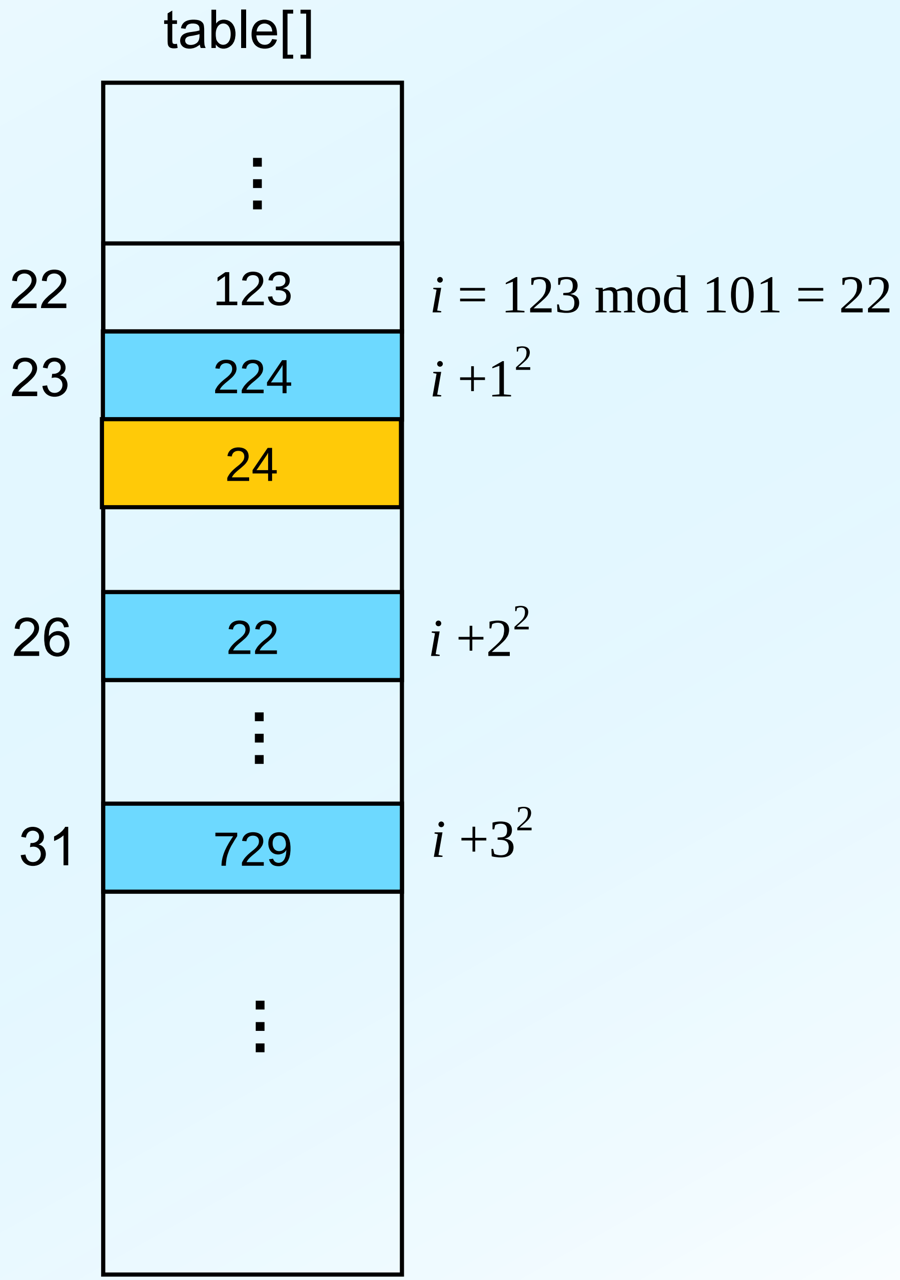
Quadratic probing

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

bad w/ secondary clustering

Quadratic probing with

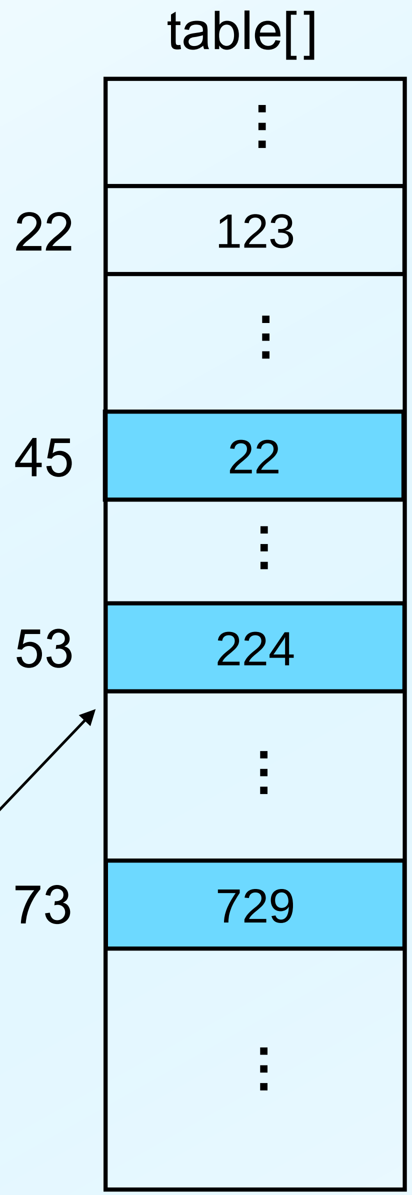
$h_i(x) = (h_0(x) + i^2) \bmod 101$



Open Addressing

Double hashing

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



$h_0(123) = h_0(224) = h_0(22) = h_0(729) = 22$

$\beta(22) = 23,\ h_1(22) = 45$

$\beta(224) = 31,\ h_1(224) = 53$

$\beta(729) = 51,\ h_1(729) = 73$

Double hashing with

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

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

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 i by a collision-resolution method
        table[i] ← x
    numItems++
```

Deletion

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

Increasing the size of hash table

- Load factor α
  - The rate of occupied slots in the table
  - A high load factor harms performance
    - We need to increase the size of hash table
- Increasing the hash table
  - Roughly double the table size
  - Rehash all the items on the new table

Java Code

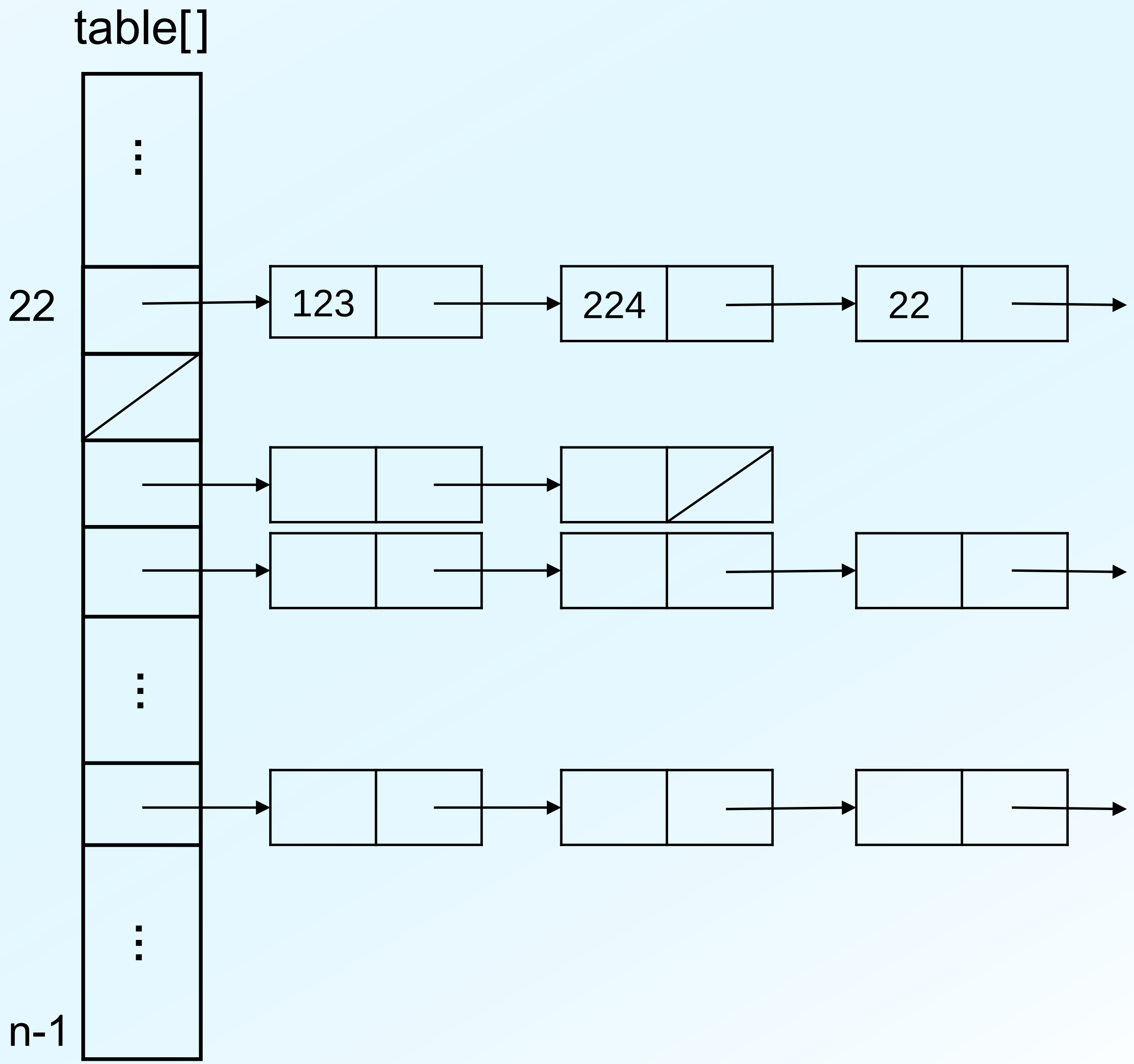
```
public class OpenHashTable implements IndexInterface<Integer>{
        private Integer table[];
        int numItems;
        static final Integer DELETED = -12345, NOT_FOUND = -1; //auto boxing
        public OpenHashTable(int n) {
                table = new Integer[n];
                numItems = 0;
        }
        private int hash(int i, Integer x) {
                return (x + i) % table.length;  // Linear probing
        }
        public Integer search(Integer x) {
                int slot;
                for (int i = 0; i < table.length; i++) {
                        slot = hash(i, x);
                        if (table[slot] == null)
                                return NOT_FOUND;
                        else if (table[slot].compareTo(x) == 0)
                                return slot;  // Found at table[slot]
                }
                return NOT_FOUND;
        }
…
```

```java
public void insert(Integer x) {
        int slot;
        if (numItems == table.length) { /* 에러 처리 */ }
        else {
                for (int i = 0; i < table.length; i++) {
                        slot = hash(i, x);
                        if (table[slot] == null || table[slot].compareTo(DELETED) == 0) {
                                table[slot] = x;
                                numItems++;
                                break;
                        }
                }
        }
}
public void delete(Integer x) {
        int slot;
        for (int i = 0; i < table.length; i++) {
                slot = hash(i, x);
                if (table[slot] == null) { /* 에러 처리 */ }
                else if (table[slot] == x) {
                        table[slot] = DELETED;
                        numItems--;
                }
        }
        // x가 존재하지 않으면 아무 영향도 미치지 않고 끝난다.
        // 필요에 따라 다른 방식으로 처리할 수도 있다.
}
// isEmpty() and clear() are trivial
} // end class OpenHashTable
```

Separate Chaining

Hash table은 linked list의 header들

No interference bet'n keys not collided



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[$h(x)$]

Java Code

```
public class ChainedHashTable implements IndexInterface<Node<Integer>> {
        private LinkedList<Integer>[] table;
        int numItems = 0;
        public ChainedHashTable(int n) {
                table = (LinkedList<Integer>) new LinkedList[n];
                                                    //컴파일러는 불평하지만 ok
                for (int i = 0; i < n; i++)
                        table[i] = new LinkedList<>();
                numItems = 0;
        }
        private int hash(Integer x) {
                return x % table.length; // 간단한 예
        }
        public void insert(Integer x) {
                int slot = hash(x);
                table[slot].add(0, x);
                numItems++;
        }
…
```

1/2 of ChainedHashTable

```java
public Node<Integer> search(Integer x) {
        int slot = hash(x);
        if (table[slot].isEmpty()) return null;
        else {
                int i = table[slot].indexOf(x);
                if (i == -1) return null;
                else return table[slot].getNode(i);
        }
}
public void delete(Integer x) {
        if (isEmpty()) { /*에러 처리*/ }
        else {
                int slot = hash(x);
                int i = table[slot].indexOf(x);
                table[slot].remove(i);
                numItems--;
        }
}
public boolean isEmpty() {
        return numItems == o;
}
public void clear() {
        for (int i = 0; i < table.length; i++)
                table[i] = new LinkedList<>();
        numItems = 0;
}
} // end class ChainedHashTable
```

2/2 of ChainedHashTable

Efficiency of Hashing

Approximate average # of comparisons w/ keys for a search

- Linear probing
  - $\frac{1}{2}(1 + {}^{1}/_{(1-\alpha)})$ for a successful search
  - $\frac{1}{2}(1 + {}^{1}/_{(1-\alpha)^2})$ for an unsuccessful search
- Quadratic probing and double hashing
  - ${}^{-\ln(1-\alpha)}/_{\alpha}$ for a successful search
  - ${}^{1}/_{1-\alpha}$ for an unsuccessful search
- Separate chaining (assume sorted lists)
  - $\alpha/2$ for a successful search
  - α for an unsuccessful search

Good Hash Functions

- should be easy and fast to compute
- should scatter the data evenly on the hash table

Observation

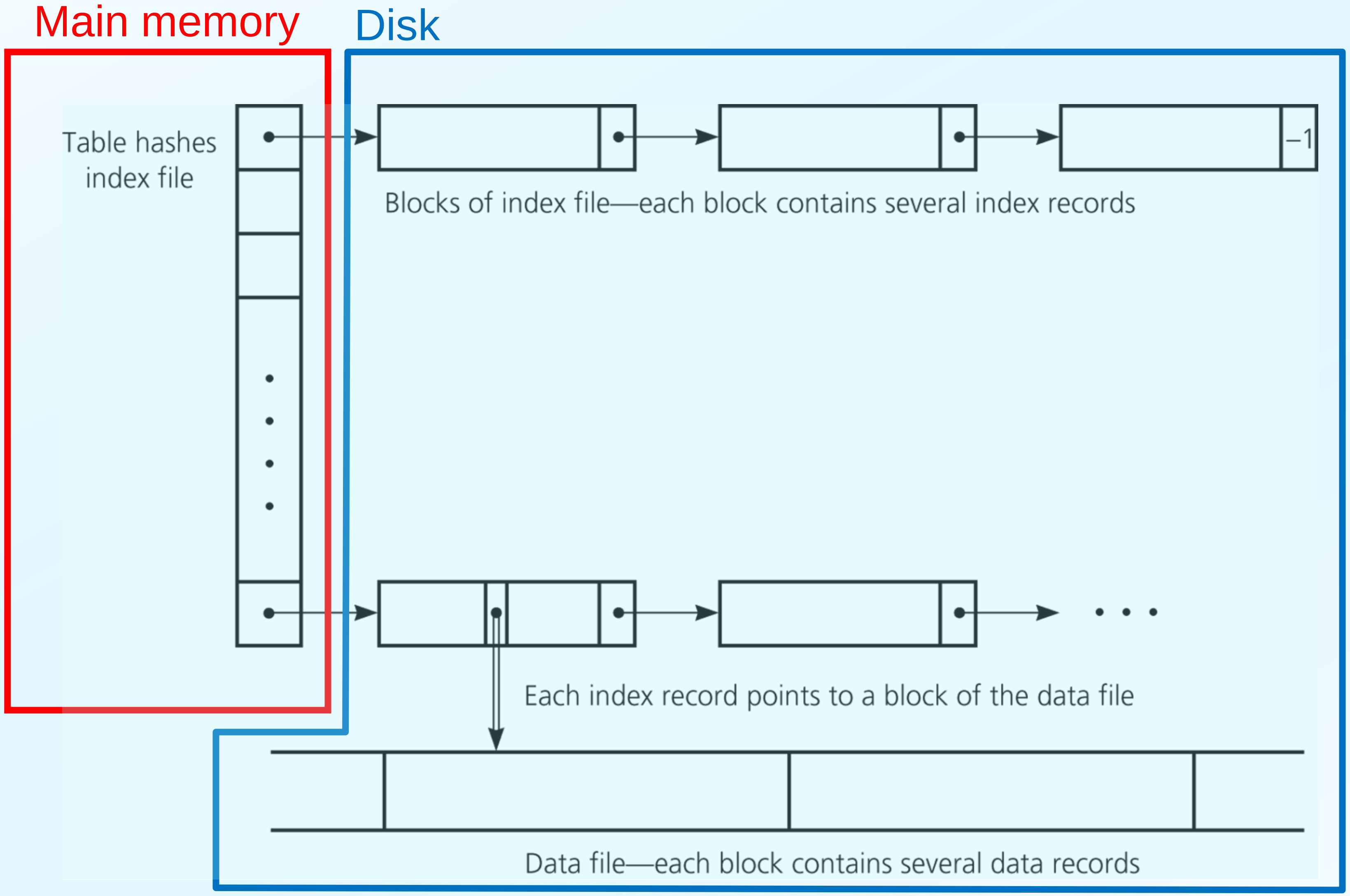
- Load factor가 낮을 때는 probing 방법들은 대체로 큰 차이가 없다.
- Successful search는 insertion할 당시의 궤적을 그대로 밟는다.

3. External Hashing

Internal/External Hashing

- Hash table이 main memory에 있는지 disk에 있는지에 따라 나뉜다
- External hashing은 disk 접근 횟수가 중요하다

External Hash Table



생각 해보기

Index에서 Disk access 몇 번 필요하겠는가?

Given situation

1조개의 records in disk
12 bytes/key
4 bytes for page number
Main memory allowed: 4G bytes
Disk block size: 32K

12+4=16 bytes/key
32K/16=2K keys/block
평균 2/3를 채우면 1.33K keys/block
1T/1.33K ≈ 0.75G blocks
4*0.75G slots = 3G bytes

Index에서 Disk access 한 번이면 충분

B-Tree와 비교해보자

Given situation

1조개의 records in disk
12 bytes/key
4 bytes for page number
Main memory allowed: 4G bytes
Disk block size: 32K

External hash table

12+4=16 bytes/key
32K/16=2K keys/block
평균 2/3를 채우면 1.33K keys/block
1T/1.33K ≈ 0.75G blocks
4*0.75G slots = 3G bytes

B-Tree

12+4+4=20 bytes/key
최대 32K/20=1.6K keys/block
(0.8~1.6K keys)
평균 1.2K key 채우면,
총 1T/1.2K = 0.833G nodes(blocks) 필요
Depth 4이면 충분

Index에서
Disk access 두 번이면 충분
(첫 두 레벨은 main memory에 둘 수 있다)