

9. 해시 테이블

해시 테이블

- 자료 구조의 두 가지 접근 패턴 : 순차 접근, 직접 접근
- *순차 접근(sequential access)*
 - 연결 리스트에 의해서 제공
 - 구조의 한쪽 끝에서 시작해 각 원소를 하나씩 살펴보면서 목표에 도달하거나 다른 끝에 도달할 때까지 탐색을 진행
 - 탐색 알고리즘은 선형 시간에 실행
 - 예, 오디오 테이프나 비디오 테이프의 자료 구조
- *직접 접근(direct access)* , *임의 접근(random access)*
 - 배열에 의해서 제공
 - 인덱스 i 를 이용해 직접 $a[i]$ 원소를 찾음. i 는 $a[i]$ 의 주소
 - 순차 접근보다 훨씬 더 빠르고 상수 시간에 실행
 - 원소의 인덱스에 대한 사전 지식을 필요
 - 예, 오디오 CD나 비디오 DVD의 자료 구조

- *해시 테이블(hash table)*
 - 원소의 인덱스에 대한 사전 지식 없이 직접 접근을 제공
 - 원소의 내용으로부터 원소의 인덱스를 계산하는 *해시 함수(hash function)*를 이용
 - “해시(hash)” : 원소들이 아무런 순서 없이 마구 뒤섞여 있다는 의미
 - 이 장에서는 java.util 패키지에 구현되어 있는 해시 테이블을 포함한 여러 종류의 해시 테이블을 기술

9.1 테이블과 레코드

- *레코드(record)*
 - 여러 개의 컴포넌트를 가진 복합적인 자료 구조
 - 각 컴포넌트는 자신의 이름과 타입을 가지고 있음
 - 일부 프로그래밍 언어에서는 레코드가 배열과 같이 표준 타입으로 사용
 - Java에서는 레코드가 객체로서 구현됨
- *테이블(table)*
 - 동일 타입의 레코드의 집합
 - 예, 표 9.1은 Country 타입을 이용한 테이블(6개의 레코드)
 - 테이블은 순서가 없는 자료 구조임
- *키 테이블(keyed table) -> 맵(map) 또는 사전(dictionary)*
 - 테이블에 저장된 레코드 전체에 대해서 값이 유일한 *키 필드* (*key field*)라는 특별한 필드 하나를 레코드 타입이 포함하는 테이블
 - 각 키는 레코드 식별에 사용됨



[표 9.1] 6개 레코드로 구성된 테이블

name	language	area	population
France	French	211,200	58,978,172
Germany	German	137,800	82,087,361
Greece	Greek	50,900	10,707,135
Italy	Italain	116,300	56,735,130
Portugal	Portuguese	35,627	9,918,040
Sweden	Swedish	173,732	8,911,296

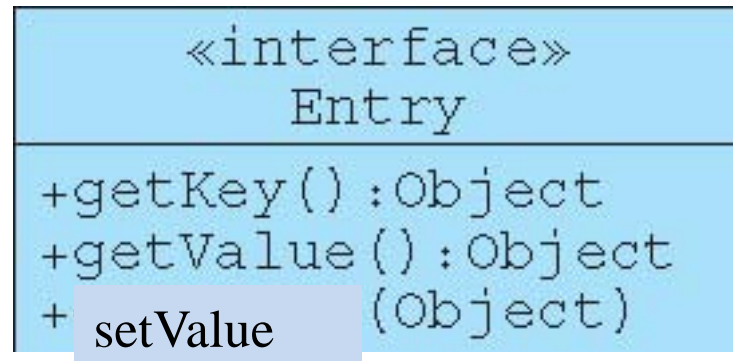


[표 9.2] 7개 레코드로 구성된 테이블

name	iso	language	area	population
Austria	AT	German	32,378	8,139,299
France	FR	Franch	211,200	58,978,172
Germany	DE	German	137,800	82,087,361
Greece	GR	Greek	50,900	10,707,130
Italy	IT	Italain	116,300	56,735,130
Portugal	PT	Portugue	35,672	9,918,040
Sweden	SE	Swedish	173,732	8,911,296

9.2 맵(Maps)에 대한 ADT

- 맵(Map) : 키 보유 레코드의 컬렉션임
- ADT: Keyed Record
 - **키 보유 레코드(keyed record)** : 키(key)와 값(value)이라는 이름을 가진 객체의 순서 쌍
 - 연산
 1. Initialize: 주어진 키와 주어진 값을 가지는 키 보유 레코드를 생성.
 2. Key: 이 레코드의 키 객체를 리턴.
 3. Value: 이 레코드의 값 객체를 리턴.
 4. Update: 이 레코드의 값 객체를 주어진 값 객체로 대체.



Entry 인터페이스

- LISTING 9.2: An Entry Interface

```
1  public interface Entry {  
2      public Object getKey();  
3          // RETURN: key;  
4          // POST: key is the first object in this ordered pair;  
5  
6      public Object getValue();  
7          // RETURN: value;  
8          // POST: value is the second object in this ordered pair;  
9  
10     public void setValue(Object value);  
11         // POST: value is the second object in this ordered pair;  
12 }
```

- ADT : Map

- **맵(map)** : 유일한 키를 갖는 키 보유 레코드의 컬렉션임
- 연산

1. Initialize: 공백 맵을 생성.
2. Search: 주어진 키에 대해 테이블에서 그 키를 가진 레코드를 탐색. 발견시 그 값을 리턴. 그렇지 않으면 null을 리턴.
3. Insert/Update: 주어진 레코드에 대해 테이블에서 그 키를 가진 레코드를 탐색.
발견시 : 그 테이블 레코드의 값을 주어진 레코드의 값으로 대체하고 대체된 값을 리턴.
그렇지 않으면 : 주어진 레코드를 테이블에 삽입.
4. Delete: 주어진 키에 대해 테이블에서 그 키를 가진 레코드를 탐색. 발견시 해당 레코드를 테이블에서 삭제하고 그 값을 리턴. 그렇지 않으면, null을 리턴.
5. Count: 테이블의 레코드 수를 리턴.

Map 인터페이스

```
«interface»  
Map
```

```
+get (Object) : Object  
+put (Object, Object)  
+remove (Object)  
+size() : int
```

LISTING 9.3 Map 인터페이스

```
public interface Map {  
    public Object get(Object key);  
        // RETURN: value;  
        // POST: if value!=null, then (key,value) is in this map;  
        // if value==null, then no record in this map has the given key;  
    public Object put(Object key, Object value);  
        // RETURN: oldValue;  
        // POST: if oldValue==null, then (key,value) is in this map;  
        // if oldValue!=null, then (key,oldValue) was in this map;  
    public Object remove(Object key);  
        // RETURN: oldValue;  
        // POST: if oldValue==null, no record in this map has the given key;  
        // if oldValue!=null, then (key,oldValue) was in this map;  
    public int size();  
        // RETURN: n;  
        // POST: this map contains n records;
```

```
}
```

9.3 Hash Tables

- 자료구조에 대한 추가적인 구성이 없다면 키 테이블에 대한 접근은 순차적임
- 테이블이 키 값에 의해 정렬되고 배열에 저장되었다면
 - 이진 탐색 : 접근 시간을 $\Theta(n)$ 에서 $\Theta(\lg n)$ 으로 개선
- *해싱(hashing)* : 정렬하지 않고도 더 좋은 성능을 낼 수 있음
- 키 테이블에 대한 *해시 함수*
 - 테이블에서 주어진 키 값을 가지고 있는 레코드의 위치(배열 인덱스)를 리턴하는 함수
 - *해시 테이블* : 해시 함수를 가진 키 테이블임

초보적인 해시 테이블 클래스

- LISTING 9.4: A Naive Hash Table Class

```
1  public class HashTable implements Map {
2      private Entry[] entries = new Entry[11];
3      private int size;
4
5      public Object get(Object key) {
6          return entries[hash(key)].value;
7      }
8
9      public Object put(Object key, Object value) {
10         entries[hash(key)] = new Entry(key,value);
11         ++size;
12         return null;
13     }
```

```
15     public Object remove(Object key) {
16         int h = hash(key);
17         Object value = entries[h].value;
18         entries[h] = null;
19         --size;
20         return value;
21     }
22
23     public int size() {
24         return size;
25     }
26
27     private class Entry {
28         Object key, value;
29         Entry(Object k, Object v) { key = k; value = v; }
30     }
31
32     private int hash(Object key) {
33         return (key.hashCode() & 0x7FFFFFFF) % entries.length;
34     }
35 }
```

hashCode()

- Object 클래스는 임의의 객체에 대해 int를 리턴하는 hashCode() 메소드를 정의하고 있다.
- 2-문자 String iso에 대해 hashCode()는 $31 * iso.charAt(0) + iso.charAt(1)$ 을 리턴

 [표 9.3] hashCode() 메소드

Key	HashCode(key)
AT	2099
FR	2252
DE	2177
GR	2283
IT	2347
PT	2564
SE	2642

- $(key.hashCode() \& 0x7FFFFFFF) \% entries.length$

key.hashCode()의 맨 앞 비트를 0으로 변경하여 양수로 만듦

나머지 연산

크기 7인 해시 테이블

0	
1	"PT", ("Portugal", "Portuguese", 35672, 9918040)
2	"SE", ("Sweden", "Swedish", 173732, 8911296)
3	
4	"IT", ("Italy", "Italian", 116300, 56735130)
5	
6	"GR", ("Greece", "Greek", 50900, 10707135)
7	
8	"FR", ("France", "French", 211200, 58978172)
9	"AT", ("Austria", "German", 32378, 8139299)
10	"DE", ("Germany", "German", 137800, 82087361)

9.4 선형 조사

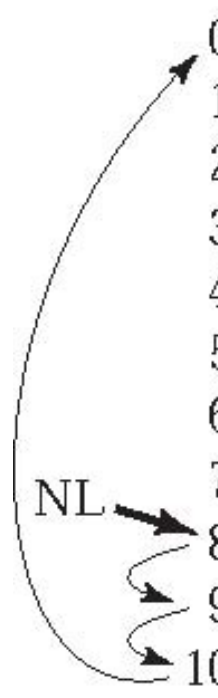
- 충돌(collision) : 해당 키의 버킷이 이미 다른 키에 의하여 점유되어 있을 때
- 해시 테이블에서 충돌을 해결하는 가장 단순한 방법
 - 충돌된 레코드를 배열의 가용한 다음 셀에 저장하는 것
- 이 알고리즘은 각 "조사"에서 배열의 인덱스를 1씩 증가시므로 *선형 조사(linear probing)*라고 함
- 이러한 방법은 해당 원소가 해시 값에 의해 인덱스된 슬롯에 항상 배치되지는 않고 테이블의 임의의 장소에서 끝날 수 있기 때문에 *개방 주소법(open addressing)*이라고도 함

선형 조사

GB	0	
	1	"PT" , ("Portugal" , "Portuguese" , 35672 , 9918040)
	2	"SE" , ("Sweden" , "Swedish" , 173732 , 8911296)
	3	"GB" , ("United Kingdom" , "English" , 94500 , 59113439)
	4	"IT" , ("Italy" , "Italian" , 116300 , 56735130)
	5	
	6	"GR" , ("Greece" , "Greek" , 50900 , 10707135)
	7	
	8	"FR" , ("France" , "French" , 211200 , 58978172)
	9	"AT" , ("Austria" , "German" , 32378 , 8139299)
	10	"DE" , ("Germany" , "German" , 137800 , 82087361)

$\text{hashCode}(\text{"GB"}) = 31 \cdot \text{iso.charAt}(0) + \text{iso.charAt}(1)$
 $= 31 \cdot 71 + 66 = 2267$
 $(\text{key.hashCode()} \& 0x7FFFFFFF) \% \text{entries.length}$
 $= 2267 \% 11 = 1$

맨 뒤에서 맨 앞으로 연결



0	"NL", ("Netherlands", "Dutch", 16033, 15807641)
1	"PT", ("Portugal", "Portuguese", 35672, 9918040)
2	"SE", ("Sweden", "Swedish", 173732, 8911296)
3	"GB", ("United Kingdom", "English", 94500, 59113439)
4	"IT", ("Italy", "Italian", 116300, 56735130)
5	
6	"GR", ("Greece", "Greek", 50900, 10707135)
7	
8	"FR", ("France", "French", 211200, 58978172)
9	"AT", ("Austria", "German", 32378, 8139299)
10	"DE", ("Germany", "German", 137800, 82087361)

- put() 메소드에 대한 수정된 코드는 다음과 같음

- ```
public Object put(Object key, Object value) {
 int h = hash(key);
 for (int i = 0; i < entries.length; i++) {
 int j = (h+i)%entries.length;
 Entry entry=entries[j];
 if (entry == null) {
 entries[j] = new Entry(key,value);
 ++size;
 ++used;
 return null; // insertion success
 }
 }
 throw new IllegalStateException(); // failure: table overflow
 return null;
}
```

- 그 결과 수정된 remove() 메소드는 다음과 같은 형태가 됨

- ```
public Object remove(Object key) {  
    int h = hash(key);  
    for (int i = 0; i < entries.length; i++) {  
        int j = (h+i)%entries.length;  
        if (entries[j] == null) break;  
        if (entries[j].key.equals(key)) {  
            Object value = entries[j].value;  
            entries[j] = NIL;  
            --size;  
            return value;  
        }  
    }  
    return null; // failure: key not found  
}
```

```
private final Entry NIL = new Entry(null, null);
```

- put() 메소드에 대한 다시 수정된 코드는 다음과 같음

- ```
public Object put(Object key, Object value) {
 int h = hash(key);
 for (int i = 0; i < entries.length; i++) {
 int j = (h+i)%entries.length;
 Entry entry=entries[j];
 if (entry == null || entry == NIL) {
 entries[j] = new Entry(key,value);
 ++size;
 return null; // insertion success
 }
 }
 throw new IllegalStateException(); // failure: table overflow
 return null;
}
```

- `get(key)` 메소드에 대한 수정된 코드는 ?

## 9.5 재해싱(*rehashing*)

- 테이블 오버플로 문제를 해결하는 방법 : 더 큰 배열을 사용하여 테이블을 재구축함
  - 경험에 의하면 테이블이 포화 상태에 이르기 전에 `rehash()`를 호출하는 것이 좋은 전략이라고 알려져 있음
  - 이는 `rehash()`에 대한 호출을 발생시키는 임계 크기를 설정해 수행할 수 있음
  - 임계 값을 저장하는 대신에 최대 비율  $r = n/m$ 을 명시한다. 여기서,  $n=size$ 이고  $m=entries.length$  임
  - 이 비율을 *적재율(load factor)*이라고 하며, 그 상한 값은 대개 75%나 80% 근처로 설정함
  - 예, 적재율이 75%일 때 배열 길이를 11로 하여 시작하면 `rehash()`는 `size`가 8.25를 초과할 때 (즉, 9번째 레코드가 삽입된 후에) 호출될 것임
  - 이 호출은 해시 테이블을 길이 23으로 재구축하게 됨

- 기존 배열보다 크기가 2배인 배열로 이동시키는 rehash() 메소드

```
private void rehash() {
 Entry[] oldEntries = entries;
 Entries = new Entry[2*oldEntries.length+1];
 for (int k = 0; k < oldEntries.length; k++) {
 Entry entry = oldEntries[k];
 if (entry == null || entry == NIL) continue;
 int h = hash(entry.key);
 for (int i = 0; i < entries.length; i++) {
 int j = nextProbe(h,i);
 if (entries[j]==null) {
 entries[j] = entry;
 break;
 }
 }
 }
 used = size;
}
```

**NIL 참조는 복사되지 않음** ←

← **hash 메소드를 재 정의함**

**//overflow 처리**



## LISTING 9.5 정확한 해시 테이블 클래스

```
1 public class HashTable implements Map {
2 private Entry[] entries;
3 private int size, used;
4 private float loadFactor;
5 private final Entry NIL = new Entry(null, null);
6
7 public HashTable(int capacity, float loadFactor) {
8 entries = new Entry[capacity];
9 this.loadFactor = loadFactor;
10 }
11
12 public HashTable(int capacity) {
13 this(capacity, 0.75F);
14 }
15
16 public HashTable() {
17 this(101);
18 }
```

```

20 public Object get(Object key) {
21 int h = hash(key);
22 for (int i = 0; i < entries.length; i++) {
23 int j = nextProbe(h,i);
24 Entry entry=entries[j];
25 if (entry == null) break;
26 if (entry == NIL) continue;
27 if (entry.key.equals(key)) return entry.value;
28 }
29 return null; // failure: key not found
30 }
32 public Object put(Object key, Object value) {
33 if (used > loadFactor*entries.length) rehash();
34 int h = hash(key);
35 for (int i = 0; i < entries.length; i++) {
36 int j = nextProbe(h,i);
37 Entry entry = entries[j];
38 if (entry == null) {
39 entries[j] = new Entry(key, value);
40 ++size;
41 ++used;
42 return null; // insertion success
43 }

```

```

44 if (entry == NIL) continue;
45 if (entry.key.equals(key)) {
46 Object oldValue = entry.value;
47 entries[j].value = value;
48 return oldValue; // update success
49 }
50 }
51 return null; // failure: table overflow
52 }
54 public Object remove(Object key) {
55 int h = hash(key);
56 for (int i = 0; i < entries.length; i++) {
57 int j = nextProbe(h,i);
58 Entry entry = entries[j];
59 if (entry == null) break;
60 if (entry == NIL) continue;
61 if (entry.key.equals(key)) {
62 Object oldValue = entry.value;
63 entries[j] = NIL;
64 --size;
65 return oldValue; // success
66 }
67 }

```

```
68 return null; // failure: key not found
69 }
71 public int size() {
72 return size;
73 }
74
75 private class Entry {
76 Object key, value;
77 Entry(Object k, Object v) { key = k; value = v; }
78 }
79
80 private int hash(Object key) {
81 if (key == null) throw new IllegalArgumentException();
82 return (key.hashCode() & 0x7FFFFFFF) % entries.length;
83 }
84
85 private int nextProbe(int h, int i) {
86 return (h + i)%entries.length; // Linear Probing
87 }
```

```
89 private void rehash() {
90 Entry[] oldEntries = entries;
91 entries = new Entry[2*oldEntries.length+1];
92 for (int k = 0; k < oldEntries.length; k++) {
93 Entry entry = oldEntries[k];
94 if (entry == null || entry == NIL) continue;
95 int h = hash(entry.key);
96 for (int i = 0; i < entries.length; i++) {
97 int j = nextProbe(h,i);
98 if (entries[j] == null) {
99 entries[j] = entry;
100 break;
101 }
102 }
103 }
104 used = size;
105 }
106 }
```

## 9.6 기타 충돌 해결 알고리즘

- 선형 조사(linear probing)는 충돌의 해결에 있어서 단순하고 어느 정도 효율적인 방법임
- 선형 조사의 문제점
  - 기본 집중(primary clustering) 이 발생 :
    - 해시 함수가 테이블 전체에 대해 레코드를 균일하게 분배하는 데 실패하면 선형 조사는 함께 묶인 레코드의 긴 체인을 만드는 경우
  - 예, 다음의 9개 국가에 대한 레코드를 길이  $m=101$ 인 빈 테이블에 삽입한다고 가정해 보자.

# 선형 조사의 삽입 예

- 길이  $m=101$ 인 빈 테이블에 삽입한다고 가정해 보자.

```
put("FI", new Country("Finland", "Finnish", 130100, 5158372));
put("IQ", new Country("Iraq", "Arabic", 168754, 22427150));
put("IR", new Country("Iran", "Farsi", 636000, 65179752));
put("SK", new Country("Slovakia", "Slovak", 18859, 5396193));
put("CA", new Country("Canada", "English", 3851800, 31006347));
put("LY", new Country("Libya", "Arabic", 679400, 4992838));
put("IT", new Country("Italy", "Italian", 116300, 56735130));
put("PE", new Country("Peru", "Spanish", 496200, 26624582));
put("IS", new Country("Iceland", "Islenska", 40000, 272512));
```

# 선형 조사에서 삽입후의 충돌 예

- 선형 조사의 경우, 아래와 같은 순서로 9개의 레코드를 삽입하면 26번의 충돌이 발생 : 해결방법은 제곱 조사

"FI" → 21

"IQ" → 21 → 22

"IR" → 22 → 23

"SK" → 22 → 23 → 24

"CA" → 21 → 22 → 23 → 24 → 25

"LY" → 21 → 22 → 23 → 24 → 25 → 26

"IT" → 24 → 25 → 26 → 27

"PE" → 24 → 25 → 26 → 27 → 28

"IS" → 23 → 24 → 25 → 26 → 27 → 28 → 29



# 제공 조사

- 선형 조사의 문제점 : 집중에 의해 성능이 손상됨
  - 하나의 클러스터는 그 안에 갭이 없다. 그러므로 "PE"와 "IS"처럼 클러스터의 시작 부근에 해시되는 새 레코드들은 한 번에 한 단계씩 여러 레코드를 거쳐야 하므로 충돌이 늘어나고 시간도 낭비됨
- *제곱 조사(quadratic probing)* :
  - 이 알고리즘은 매번 1씩 증가하는 대신에 점진적으로 더 큰 폭으로 증가시켜 충돌을 해결함
  - 이를 구현하기 위해서는
$$\text{int } j = (h + i) \% \text{entries.length}$$
를
$$\text{int } j = (h + i*i) \% \text{entries.length};$$
로 대체함

## 제곱 조사(2)

```
int j = (h + i*i)%entries.length;
```

- 즉, 각 충돌 후에 h에 i를 더하는 대신에  $i^2$ 를 더한다.  
(quadratic은 변수의 제곱을 의미한다.)
- 따라서 증분의 순서는 1, 4, 9, 16, 25, ...가 된다.
- 제곱 조사는 선형 조사의 충돌 횟수를 반으로 줄임
  - 결과가 개선되는 이유는 제곱 조사가 사용이 안 된 갭을 중간에 남겨 두어 선형 조사 보다 적은 집중을 가져오기 때문이다.

# 제공 조사에서 삽입후의 충돌 예

- 제공 조사를 이용하면 동일한 9개 레코드의 순서적인 입력에 대해 13번의 충돌만이 발생한다.

"FI" → 21

"IQ" → 21 → 22

"IR" → 22 → 23

"SK" → 22 → 23 → 26

"CA" → 21 → 22 → 25

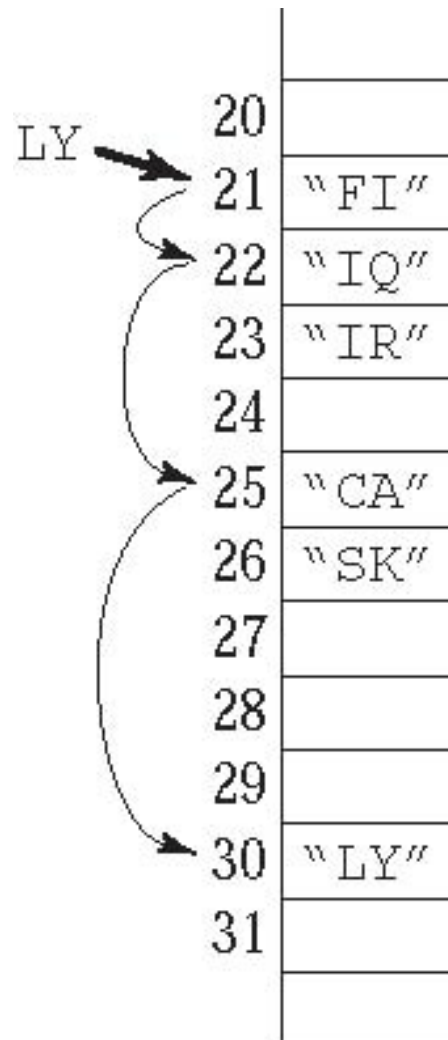
"LY" → 21 → 22 → 25 → 30

"IT" → 24

"PE" → 24 → 25 → 28

"IS" → 23 → 24 → 27

# "LY"에 대한 제공 조사 순서



# 제공 조사의 문제점

- 길이가  $m=11$ 인 테이블에 어떤 키가 인덱스  $j=3$ 으로 해시될 때,
- 순서는 3, 4, 7, 1, 8, 6, 6, 8, 1, 7, 4, 3, 4, 7, 1, 8, 6, 6, 8, 1, 7, 4, 3...됨
- 이는 희소 주기 순서이다. 이는 11개 셀 중 6개 셀이 모두 점유되고 나면 다른 5개의 셀이 비어 있더라도 put()은 실패하게 된다.
- 해결책 : 임계 적재율을 50%로 설정함. 즉,  $6 > 11/2$  가 됨
- 만약, 적재율을 50%로 제한한다해도, 2차 집중 문제가 발생함

# 이중 해싱

- **2차 집중(*secondary clustering*)** : 동일한 값으로 해시되는 2개의 상이한 키가 동일한 조사 순서를 가지게 되는 것이다.
- 해결책 : **이중 해싱(*double hashing*)**
- **이중 해싱(*double hashing*)**
  - 조사 순서를 결정해주는 제 2의 독립적인 해시 함수를 사용
  - 2번째 해시 함수에 의한 상수 증분은 대개 1 보다 큼
  - 제곱 조사처럼 이중 해싱도 조사 순서를 넓게 확대해서 기본 집중을 피함

# 이중해싱을 위한 리스팅 9.5의 조정

```
20 public Object get(Object key) {
21 int h = hash(key);
22 for (int i = 0; i < entries.length; i++) {
23 int j = nextProbe(h,i);
 :
 }
```

→ `int h = hash(key); int d = hash2(key);`

→ `int j = nextProbe(h,d,i)`

```
80 private int hash(Object key) {
81 if (key == null) throw new IllegalArgumentException();
82 return (key.hashCode() & 0x7FFFFFFF) % entries.length;
83 }
84 hash2()에서는
85 1+ (key.hashCode() & 0x7FFFFFFF) % (entries.length-1)
86 private int nextProbe(int h, int i) {
87 return (h + i)%entries.length; // Linear Probing
 }
```

`nextProbe(h,d,i)`에서는 ???

## 이중 해싱에서 삽입후의 충돌 예

- 동일한 순서의 9개 레코드의 입력에 대해 5번의 충돌만 발생

"FI" → 21

"IQ" → 21 → 89

"IR" → 22

"SK" → 22 → 97

"CA" → 21 → 85

"LY" → 21 → 91

"IT" → 24

"PE" → 24 → 99

"IS" → 23



## 9.7 별도 체인

- *개방 주소법*: 이제까지 기술된 해시 알고리즘
  - 개방 주소법은 충돌 해결을 위해 배열 내부에서 개방된 위치를 탐색하기 때문이다.
- *폐쇄 주소법(closed addressing)*
  - 폐쇄 주소법은 하나의 해시 위치에 1개 보다 많은 레코드를 허용함으로써 충돌을 피한다. 따라서 복잡한 자료구조가 필요.
- *별도 체인(separate chaining)*
  - 레코드의 배열 대신에 버킷의 배열을 사용
  - *버킷(bucket)*이란 일종의 레코드의 컬렉션임
  - 가장 단순한 자료 구조는 하나의 버킷에 대해 하나의 연결 리스트를 사용하는 것

# 체인을 이용한 폐쇄 주소법

- 그림 9.7은 용량을 11로, 적재율을 2.0으로 하고 15개의 유럽 국가를 적재한 후의 해시 테이블을 보이고 있다. 키만을 보이고 있고 전체 레코드는 보이지 않고 있다.

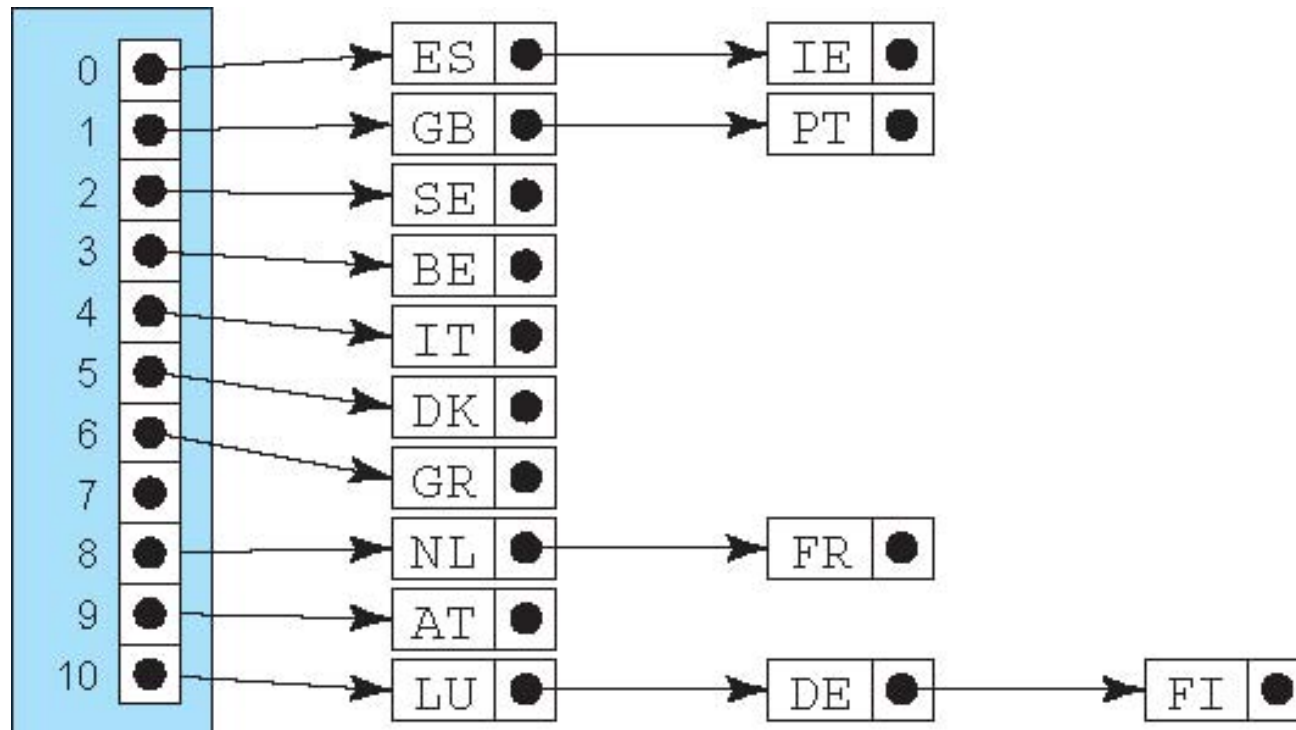


그림 9.7

# 폐쇄 주소법

- 폐쇄 주소법을 사용하는 해시 테이블에 대한 자료 구조가 더 복잡하기는 하지만 코드는 약간 더 짧다. 빈 NIL 항목도 필요하지 않고 충돌 해결 알고리즘도 필요하지 않다.
- 폐쇄 주소법의 경우 체인의 길이에 제한이 없으므로 적재율이 지원 배열의 길이를 초과할 수 있다. 그렇지만, 긴 체인을 허용하게 되면 해시 테이블의 성능이 저하된다. 그러므로 테이블 size가 주어진 임계값을 넘어가면 재해싱을 수행함.
- 폐쇄 주소법은 충돌을 방지하는 명확한 장점을 가지고 있다. 각 버킷이 임의 개수의 다수의 키를 저장할 수 있다면 오버플로는 일어나지 않게 될 것이다.
- 단점은 일부 체인이 매우 길어질 수 있다는 것이다. 매우 불균형한 버킷 체인의 배열은 해싱이 제공해야 할 상수 시간 접근의 장점을 파괴할 수도 있다.

# 별도 체인에 의한 폐쇄 주소법

- LISTING 9.6: Closed Addressing by Separate Chaining

```
1 public class HashTable {
2 private Entry[] entries;
3 private int size;
4 private float loadFactor;
5
6 public HashTable(int capacity, float loadFactor) {
7 entries = new Entry[capacity];
8 this.loadFactor = loadFactor;
9 }
10
11 public HashTable(int capacity) {
12 this(capacity, 0.75F);
13 }
14
15 public HashTable() {
16 this(101);
17 }
```

```

19 public Object get(Object key) {
20 int h = hash(key);
21 for (Entry e = entries[h]; e != null; e = e.next) {
22 if (e.key.equals(key)) return e.value; // success
23 }
24 return null; // failure: key not found
25 }
26
27 public Object put(Object key, Object value) {
28 int h = hash(key);
29 for (Entry e = entries[h]; e != null; e = e.next) {
30 if (e.key.equals(key)) {
31 Object oldValue = e.value;
32 e.value = value;
33 return oldValue; // successful update
34 }
35 }
36 entries[h] = new Entry(key,value,entries[h]);
37 ++size;
38 if (size > loadFactor*entries.length) rehash();
39 return null; // successful insertion
40 }

```

```
42 public Object remove(Object key) {
43 int h = hash(key);
44 for (Entry e = entries[h], prev=null; e!=null; prev=e, e=e.next) {
45 if (e.key.equals(key)) {
46 Object oldValue = e.value;
47 if (prev == null) entries[h] = e.next;
48 else prev.next = e.next;
49 --size;
50 return oldValue; // success
51 }
52 }
53 return null; // failure: key not found
54 }

56 public int size() {
57 return size;
58 }
```

```
60 private class Entry {
61 Object key, value;
62 Entry next;
63 Entry(Object k, Object v, Entry n) {
64 key=k; value=v; next=n;
65 }
66 }
67 }

69 private int hash(Object key) {
70 if (key == null) throw new IllegalArgumentException();
71 return (key.hashCode() & 0x7FFFFFFF) % entries.length;
72 }
```

```
74 private void rehash() {
75 Entry[] oldEntries = entries;
76 entries = new Entry[2*oldEntries.length+1];
77 for (int k = 0; k < oldEntries.length; k++) {
78 for (Entry old = oldEntries[k]; old != null;) {
79 Entry e = old;
80 old = old.next;
81 int h = hash(e.key);
82 e.next = entries[h];
83 entries[h] = e;
84 }
85 }
86 }
87 }
```



## 9.8 java.util.Map 인터페이스

- LISTING 9.7: The java.util.Map Interface

```
1 public interface Map {
2 public void clear();
3 public boolean containsKey(Object key);
4 public boolean containsValue(Object value);
5 public Set entrySet();
6 public boolean equals(Object object);
7 public Object get(Object key);
8 public int hashCode();
9 public boolean isEmpty();
10 public Set keySet();
11 public Object put(Object key, Object value);
 :
23 }
```

java.util.HashMap이 java.util.Map 인터페이스를 구현

- 디폴트 초기 용량 101
- 디폴트 최대 적재율이 0.75인 페쇄주소법 사용

```
1 public class TestMap {
2 public static void main(String[] args) {
3 java.util.Map map = new java.util.HashMap();
4 map.put("AT", new Country("Austria", "German", 32378, 8139299));
5 map.put("BE", new Country("Belgium", "Dutch", 11800, 10182034));
6 map.put("DK", new Country("Denmark", "Danish", 16639, 5356845));
7 map.put("FR", new Country("France", "French", 211200, 58978172));
8 map.put("GR", new Country("Greece", "Greek", 50900, 10707135));
9 map.put("IE", new Country("Ireland", "English", 27100, 3632944));
10 map.put("IT", new Country("Italy", "Italian", 116300, 56735130));
11 map.put("ES", new Country("Spain", "Spanish", 194880, 39167744));
12 System.out.println("map.keySet(): " + map.keySet());
13 System.out.println("map.size(): " + map.size());
14 System.out.println("map.get(\\\"ES\\\") : " + map.get("ES"));
15 Country es = (Country)map.get("ES");
16 es.population = 40000000;
17 System.out.println("map.get(\\\"ES\\\") : " + map.get("ES"));
18 System.out.println("map.remove(\\\"ES\\\") : " + map.remove("ES"));
19 System.out.println("map.get(\\\"ES\\\") : " + map.get("ES"));
20 System.out.println("map.keySet(): " + map.keySet());
21 System.out.println("map.size(): " + map.size()); } }
```

- 출력 결과

```
map.keySet(): [AT, FR, GR, DK, IT, BE, ES, IE]
map.size(): 8
map.get("ES"): (Spain,Spanish,194880,39167744)
map.get("ES"): (Spain,Spanish,194880,40000000)
map.remove("ES"): (Spain,Spanish,194880,40000000)
map.get("ES"): null
map.keySet(): [AT, FR, GR, DK, IT, BE, IE]
map.size(): 7
```

## 9.9 해싱 알고리즘의 분석

- 4개 해싱 알고리즘의 평균 실행-시간 복잡도
  - 이 공식들은 지원 배열의 길이  $m$ 이 크고  $\text{hash}()$  메소드가 주어진 모든 키 값의 집합에 대해 범위  $0 \leq h < m$  사이에 균일하게 분포된 인덱스 번호  $h$ 를 리턴하는 것으로 가정한다.
  - 해시 테이블에 대한 적재율이 비율  $r = n/m$ 이고, 이때  $n$ 은 테이블의 레코드 수,  $m$ 은 지원 배열의 길이 ( $\text{entries.length}$ ),  $q = 1/(1-r) = m/(m-n)$ 으로 정의된다

|                       |                         | 키를 찾음                                         | 키를 찾지 못함                              |
|-----------------------|-------------------------|-----------------------------------------------|---------------------------------------|
| 개방 주소법<br>( $r < 1$ ) | 선형 조사<br>제곱 조사<br>이중 해싱 | $(1+q)/2$<br>$1 + \ln q - r/2$<br>$(\ln q)/r$ | $(1+q^2)/2$<br>$q + \ln q - r$<br>$q$ |
| 폐쇄 주소법                | 별도 체인                   | $1 + r/2$                                     | $r$                                   |

- 예,  $r=75\%$  (따라서  $q = 4.0$ 이고  $\ln q = 1.386$ )라면, 이 공식들은 다음의 표와 같은 값들을 갖게 된다.
- 주어진 적재율에 대한 평균 실행 시간은 테이블에서 아래로 갈수록 좋아지는 것을 알 수 있음. 일반적으로 별도 체인을 이용한 폐쇄 주소법이 개방 주소법 알고리즘 보다 성능이 우수하다. `java.util.HashMap` 클래스가 이 알고리즘을 사용하는 이유는 이 때문이다.

|                       |       | 키를 찾음 | 키를 찾지 못함 |
|-----------------------|-------|-------|----------|
| 개방 주소법<br>( $r < 1$ ) | 선형 조사 | 2.5   | 8.5      |
|                       | 제곱 조사 | 2.0   | 4.6      |
|                       | 이중 해싱 | 1.8   | 4.0      |
| 폐쇄 주소법                | 별도 체인 | 1.375 | 0.75     |

## 9.10 완전 해시 함수

- 개방 주소법이 폐쇄 주소법 보다 잘 작동하는 경우
  - 완전 해시 함수(*perfect hash function*) 일때
    - 해시 함수가 모든 가능한 키의 집합에 대해 일대일이 될 때
    - 보통 이런 특별한 상황은 매우 드물지만, 전체 키 집합을 미리 알 수 있다면, 해시 테이블과 해시 함수 고안이 가능
  - 완전 해시 함수가 발견되면 개방 주소법(조사 과정이 필요없는)이 가장 좋다. 이러한 자료 구조를 종종 조사표(*lookup table*)라고 한다.
    - 이 방법은 공간을 낭비한다. 한 예에서는 길이 77인 배열이 15 레코드 저장에 사용되어 적재율이 20% 보다 낮아진다.

# 최소 완전 해시 함수

- *최소 완전 해시 함수(minimal perfect hash function)*
  - 일대일이면서 100% 적재율을 가진 완전 해시 함수
  - 이 함수의 발견은 매우 어렵지만, 함수의 탐색 과정을 도와주는 알고리즘 중 하나가 1980년에 R. J. Cichelli에 의해서 제안되었음. 문자열에 대한 완전 해시 함수 발견함
- 최소완전해시함수는 전체 키 집합이 미리 알려져 있고 변하지 않을 때 유용하다. 컴파일러에서 사용되는 프로그래밍 언어의 예약어에 대한 테이블이 이러한 형태이다.

## 9.11 기타 해시 함수

- 완전 해시 함수가 불가능하다면, 최선의 해시 함수는 레코드를 해시 테이블 전체에 고르게 분배하는 것임.
- 제산(*division*) 해시 함수
  - 이 함수는 큰 정수를 테이블의 길이로 나눈 나머지를 사용
  - 예를 들어,  $\text{num}(\text{key}) \% \text{entries.length}$   
 $(\text{key.hashCode()} \& 0x7FFFFFFF) \% \text{entries.length}$
- 또 다른 해싱 방법은 추출(*extraction*)
  - 이는 키의 일부분에 적용된 제산 방법으로 볼 수 있다.  
예, 일주일의 7개 요일의 이름에 대한 해싱은 모두 동일한 접미어 "day"를 가지기 때문에 각 키워드의 마지막 3개 문자를 생략함



- 중첩 해시 함수(*folding hash function*)
  - 키가 단순한 정수일 경우 간단한 솔루션을 제공한다.
  - 이 방법은 숫자 문자열을 여러 부분으로 분리하고 나서 이들을 함께 "중첩"시킨다.
  - 예, 9-자리 코드로 구성된 미국의 사회보장번호 054-36-1729 를 해싱하는 경우, 해시값 37의 계산 방법
    1. 해당하는 긴 정수를 구성: 054361729
    2. 이를 3개 부분으로 분리: 054, 361, 729
    3. 중간 부분을 역순으로 배열: 054, 163, 729
    4. 각 부분들을 더함:  $054+163+729 = 946$
    5. 테이블 길이로 나눈 나머지를 구함:  $946\%101 = 37$
  - 개인 레코드에 대한 용량이 101인 해시 테이블에서 사회보장번호 054-36-1729를 갖는 레코드는 `table[37]`에 저장

# 중첩

054 361 729

