•자료구조의 두가지 접근 패턴: 순차 접근, 직접 접근

- **순차 접근**(sequential access)
  - 연결리스트에 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 자료구조
- **직접 접근**(direct access), 임의 접근(random access)
  - 배열에 의해서 제공
  - 인덱스i를 이용해 직접a[i] 원소를 찾음. i는a[i]의 주소
  - 순차 접근보다 훨씬 더 빠르고 상수 시간에 실행
  - 원소의 인덱스에 대한 사전지식을 필요
  - 예, 오디오CD나 비디오DVD의 자료구조

- **해시테이블(hash table)**
  - **원소의 인덱스에 대한 사전지식없이 직접접근을제공**
  - 원소의 내용으로부터 원소의 인덱스를 계산하는 해시함수(hash function)를 이용
  - "해시(hash)": 원소들이 아무런 순서없이 마구 뒤섞여 있다는 의미
  - 이장에서는java.util 패키지에 구현되어있는 해시테이블을 포함한 여러 종류의 해시테이블을 기술

테이블에서 주어진 키 값을 가지고있는 레코드의 위치(배열인덱스)를 리턴하는 함수

자료구조에대한 추가적인 구성이 없다면 키 테이블에대한 접근은 순차적임

테이블이 키 값에 의해 정렬되고 배열에 저장되었다면 –> 이진 탐색: 접근 시간을 Θ(n)에서 Θ(lg n)으로개선

해싱(hashing): 정렬하지않고도 더 좋은 성능을 낼 수 있음(정렬 후 이진탐색보다!)

- **레코드**(record)
  - 여러 개의 **컴포넌트를** 가진 복합적인 자료구조
  - 각 컴포넌트는 자신의 이름과 타입을 가지고 있음
  - 일부 프로그래밍언어에서는 레코드가 배열과같이 표준타입으로 사용 –
  - Java에서는 레코드가 객체로서 구현됨
- **테이블**(table)
  - 동일 타입의 레코드의 집합
  - **테이블은 순서가 없는 자료 구조임**
- 키테이블(keyed table) -> 맵(map) 또는사전(dictionary)
  - 테이블에 저장된 레코드 전체에대해서 값이 유일한 키필드(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

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   }
12   public HashTable(int capacity) {
13       this(capacity, 0.75F);
14   }
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  }
```


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

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

나머지 연산

```
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 }
```

제수가 많으면 제수의 배수들은 모두 제수의 배수에 몰림. Ex) 홀수 13은 제수가 1,13 제수의수 적음.

제수의 수 작게하면 성능 증가.(연산에서 배열의 크기로 나누기 때문) 따라서, 홀수 만들어주기 위해 +1 해줌

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/m을 명세한다. 여기서, n=size이고 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 size가 8.25를 초과할 때 (즉, 9번째 레코드가 삽입된 후에) 호출될 것임
  - 이 호출은 해시 테이블을 길이 23으로 재구축하게 됨

NIL 사용이유

삭제할 시 삭제한 곳을 null로 설정하면, null을 발견할 때 탐색이 종료되므로 찾아야 할 key를 결코 찾을 수 없게 됨. 전체 테이블 탐색은 상수시간에 탐색하고자 하는 목적이 해침.

따라서 탐색이 종료되지 않도록 NIL이라는 객체 사용

But 많이 사용하면 충돌의 빈도를 증가(조사 순서 증가), 성능 손상

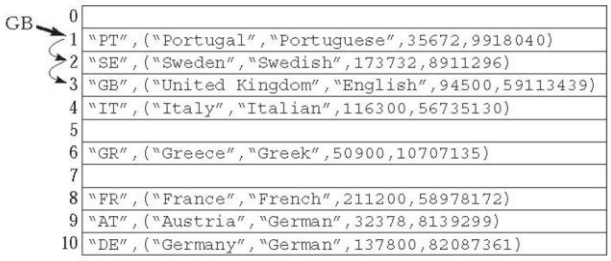
Rehash() 는 NIL을 삭제하므로 성능 개선!

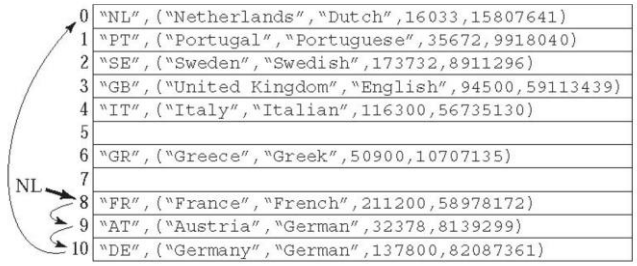
개방주소법(open addressing): 해당 원소가 해시 값에 의해 인덱스된 슬롯에 항상 배치되지는 않고 테이블의 임의에 장소에서 끝날수있음

폐쇄주소법: 반드시 해쉬된 값에 넣는 것

선형 조사(linear probing)

각 "조사"에서 배열의 인덱스를1씩증가하므로 선형 조사(linear probing) 라고 함






```
private int nextProbe(int h, int i) {
    return (h + i)%entries.length;     // Linear Probing
}
```

선형조사는 충돌의 해결에 있어서 단순하고 어느 정도 효율적.

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

선형 조사에서 삽입후의 충돌 예

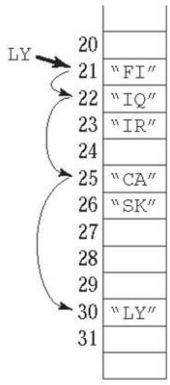
- 선형 조사의 경우, 아래와 같은 순서로 9개의 레코드를 삽입하면 26번의 충돌이 발생 : 해결방법은 제곱 조사

"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

집중에 의해 성능 손상!

충돌이 늘어나고 시간도 낭비됨 > 제곱 조사로 해결

제곱 조사(quadratic probing)



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

증분 순서는 1, 4, 9, 16, 25...

제곱 조사가 사용이 안 된 갭을 중간에 남겨 두어 선형 조사보다 적은 집중을 가져옴

> **선형조사의 충돌 횟수를 반으로 감소**

제곱 조사에서 삽입후의 충돌 예

- 제곱 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

문제점

제곱 조사의 문제점

- 길이가 m=11인 테이블에 어떤 키가 인덱스 j=3으로 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

2차집중(secondary clustering): 동일한 값으로 해시되는 2개의 상이한키가 동일한 조사순서를 가지게되는것이다. **>이중 해싱으로 해결**

이중 해싱(double hashing)

- 조사 순서를 결정해주는 제2의 독립적인 해시함수를 사용
- 2번째 해시함수에의한 상수 증분은 대개 1보다 큼
- 제곱 조사처럼 이중해싱도 조사 순서를 넓게 확대해서 기본 집중을 피함

이중해싱을 위한 리스팅 9.5의 조정

```
20  public Object get(Object key) {
21      int h = hash(key);          → int h = hash(key);  int d = hash2(key);
22      for (int i = 0; i < entries.length; i++) {
23          int j = nextProbe(h,i);  → 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      }   hash2( ) 에서는
84          1+ (key.hashCode() & 0x7FFFFFFF) % (entries.length-1)
85      private int nextProbe(int h, int i) {
86          return (h + i)%entries.length; // Linear Probing
87      }   nextProbe(h,d,i) 에서는 ???
```

+1 해주는 이유: 증분이므로 0이 나오면 안됨

nextProbe(h,d,i)는 j = (h + i*d)%m; 으로 변경

이중 해싱에서 삽입후의 충돌 예

- 동일한 순서의 9개 레코드의 입력에 대해 5번의 충돌만 발생

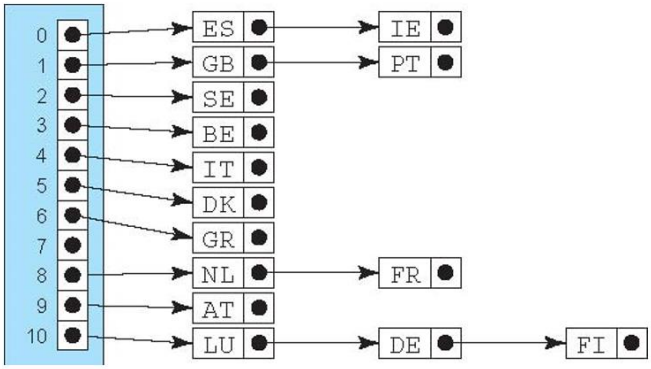
"FI" → 21
"IQ" → 21 → 89
"IR" → 22
"SK" →22 → 97
"CA" → 21 → 85
"LY" → 21 → 91
"IT" → 24
"PE" → 24 → 99
"IS" → 23

개방 주소법: 선형 조사, 제곱 조사, 이중 해싱이 이에 해당. 충돌 해결을 위해 배열 내부에서 개발된 위치를 탐색, 사용

폐쇄 주소법: 별도 체인이 이에 해당. 하나의 해시 위치에 1개보다 많은 레코드를 허용하여 충돌 피함. 복잡한 자료구조 필요. 하지만 NIL사용x 충돌 해결 알고리즘도 사용하지 않으므로 코드는 약간 더 짧다. ***충돌을 방지하는 명확한 장점을 가짐, 일부 체인이 매우 길어질 수 있다는 단점.***

별도 체인(separate chaining)

레코드의 배열 대신 버켓(일종의 레코드의 컬렉션)의 배열 사용



java.util.HashMap 클래스가 폐쇄주소법을 사용하는 이유

일반적으로 폐쇄주소법이 개방주소법 알고리즘 보다 성능이 우수하다.

- *참고*

– 해시 테이블에 대한 적재율이 비율 r=n/m이고, 이때 n은 테이블의 레코드 수, m은 지원 배열의 길이 (entries.length), q = 1/(1-r) = m/(m-n)으로 정의된다

		키를 찾음	키를 찾지 못함
개방 주소법 (r<1)	선형 조사 제곱 조사 이중 해싱	(1+q)/2 1+ln q−r/2 (ln q)/r	(1+q²)/2 q+ln q−r q
폐쇄 주소법	별도 체인	1+r/2	r

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

완전 해시 함수(perfect hash function)

: 해시 함수가 모든 가능한 키의 집합에 대해 일대일이 될 때

➔ 이게 발견된다면 개방 주소법이 가장 좋다. 이런 자료구조를 조사표(lookup table)이라고 함.

- 이 방법은 공간을 낭비한다. 한 예에서는 길이 77인 배열이 15 레코드 저장에 사용되어 적재율이 20% 보다 낮아진다.

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.

제산(division)해시 함수

추출(extraction)해시 함수

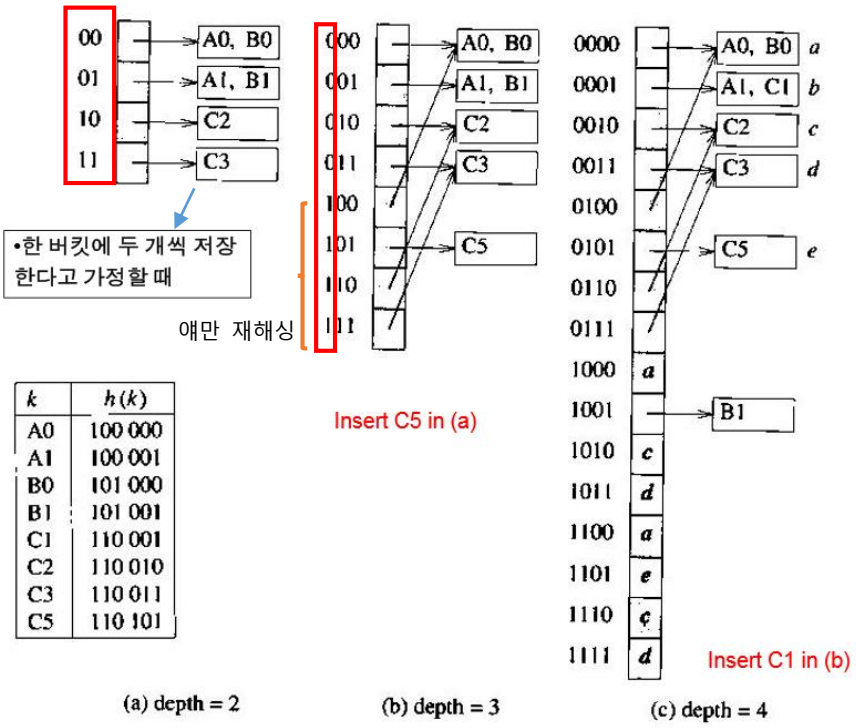
중첩(folding hash function)해시 함수

동적해싱(dynamic hashing)

- 동적해싱(dynamic hashing)은 재조정을 한 번 할 때마다 오직 하나의 버킷 안에 있는 엔트리들에 대해서만 홈버킷을 변경하게 하여 재조정 시간을 줄이는 방법이다

- 디렉토리(directory)를 사용하는 동적 해싱

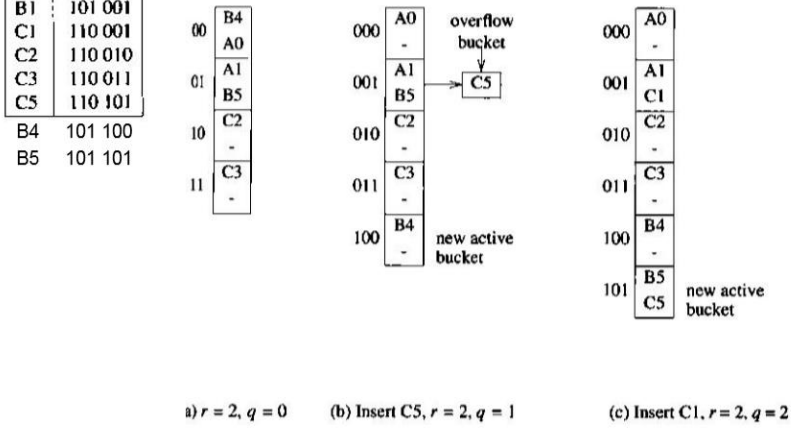
  테이블에 있는 모든 엔트리들을 재해싱 하는 것이 아니라 오버플로우 된 버킷에 있는 엔트리들만 재해싱



k	h(k)
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C1	110 001
C2	110 010
C3	110 011
C5	110 101

- 디렉토리가 없는 동적 해싱

k	h(k)
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C1	110 001
C2	110 010
C3	110 011
C5	110 101
B4	101 100
B5	101 101



a) $r = 2, q = 0$ (b) Insert C5, $r = 2, q = 1$ (c) Insert C1, $r = 2, q = 2$

-