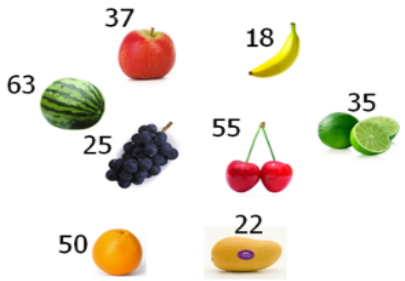
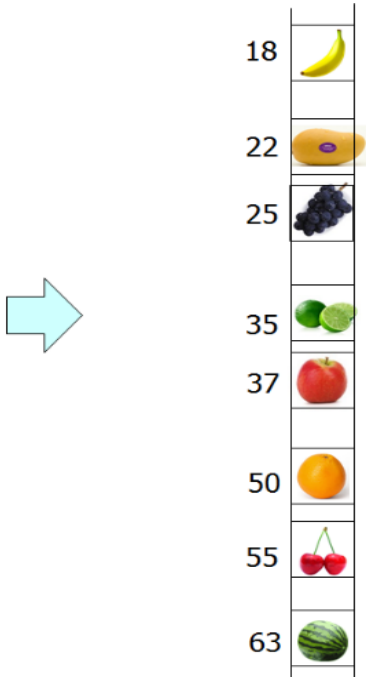
제6장 해시테이블

6.1 해시테이블

- 이진탐색트리의 성능을 개선한 AVL 트리와 레드블랙트리의 삽입과 삭제 연산의 수행시간은 각각 O(logN)
- 그렇다면 O(logN) 보다 좋은 성능을 갖는 자료구조는 없을까?

> [핵심 아이디어] O(logN) 시간보다 빠른 연산을 위해, 키와 1차원 배열의 인덱스의 관계를 이용하여 키(항목)를 저장한다.





h(k)= k % 5

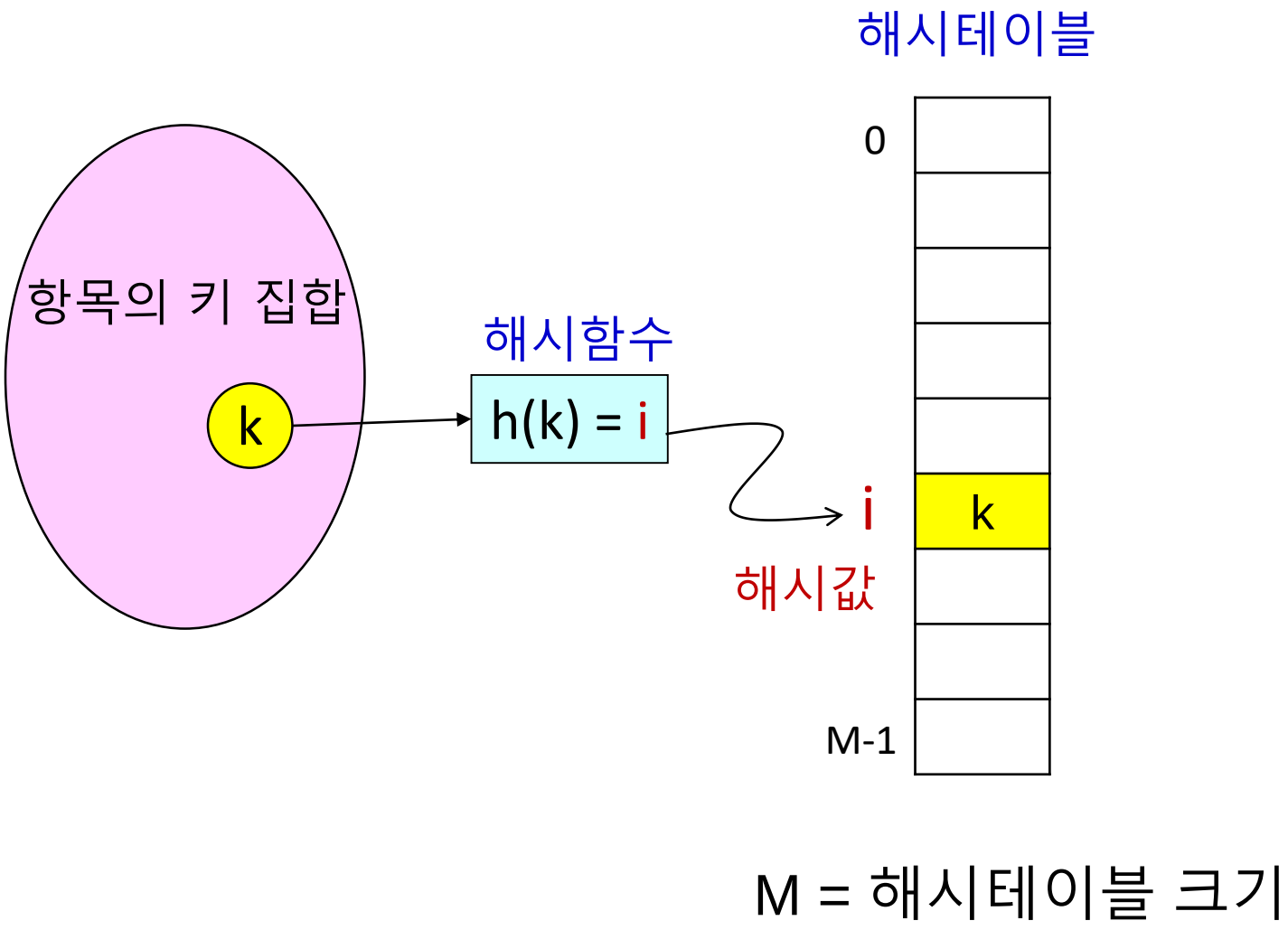
h(10) = 0	10
h(21)= 1	21
h(32)= 2	32
h(8)= 3	8
h(24)= 4	24

해시 테이블

- 키를 배열의 인덱스로 그대로 사용하면 메모리 낭비가 심해질 수 있음
- [문제 해결 방안] 키를 변환하여 배열의 인덱스로 사용

- 해싱(Hashing)
  - 키를 간단한 함수를 사용해 변환한 값을 배열의 인덱스로 이용하여 항목 저장
- 해시함수(Hash Function) : 해싱에 사용되는 함수
- 해시값(Hash value) 또는 해시주소: 해시함수가 계산한 값
- 해시테이블(Hash Table): 항목이 해시값에 따라 저장되는 배열

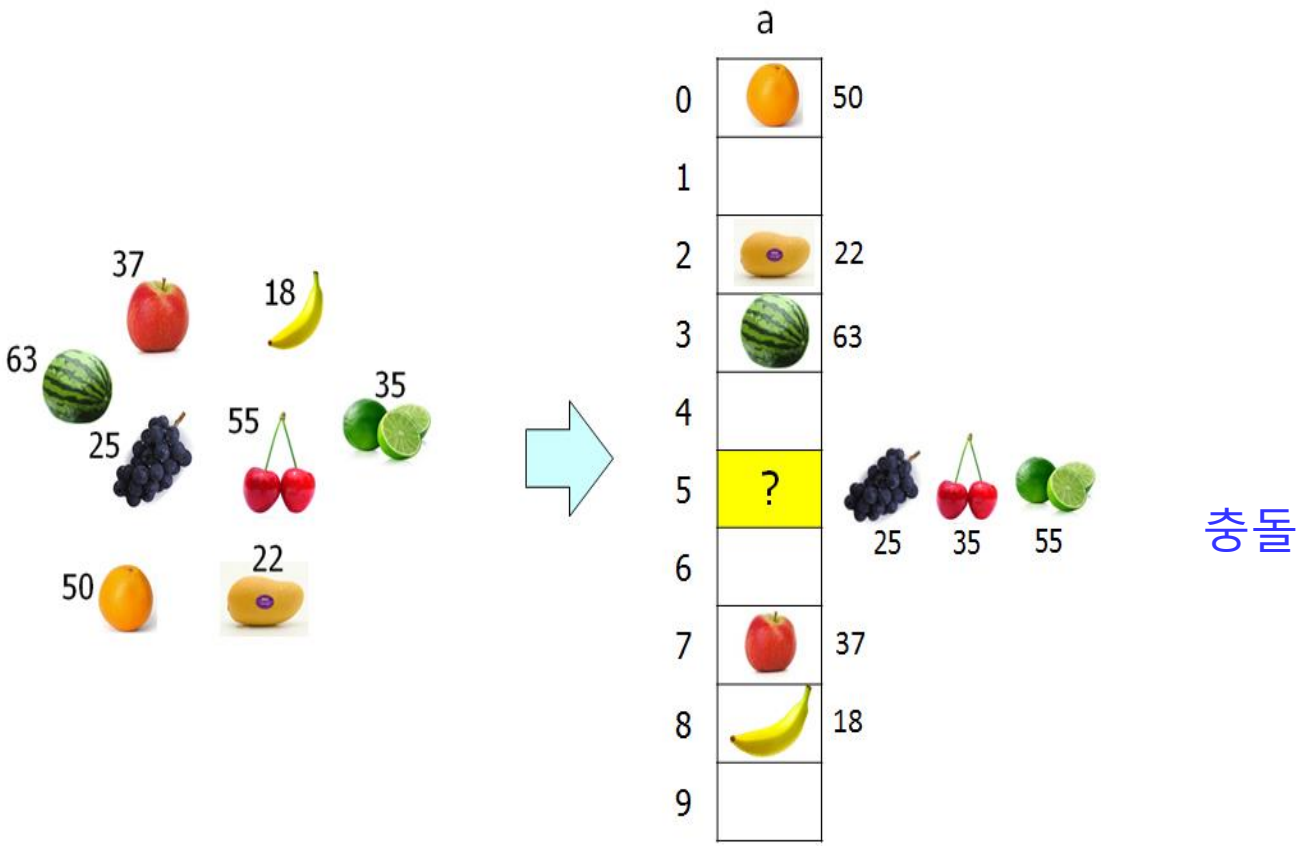
해싱의 전반적인 개념



M = 해시테이블 크기

충돌(Collision)

- 아무리 우수한 해시함수를 사용하더라도 2 개 이상의 항목을 해시테이블의 동일한 원소에 저장하여야 하는 경우가 발생
- 서로 다른 키들이 동일한 해시값을 가질 때 충돌(Collision) 발생



6.2 해시함수

- **가장 이상적인 해시함수는 키들을 균등하게(Uniformly) 해시테이블의 인덱스로 변환하는 함수**
  - 일반적으로 키들은 부여된 의미나 특성을 가지므로
    키의 가장 앞 부분 또는 뒤의 몇 자리 등을 취하여 해시값으로 사용하는 방식의 해시함수는 많은 충돌을 야기시킴
  - 균등하게 변환한다는 것은 키들을 해시테이블에 랜덤하게 흩어지도록 저장하는 것을 뜻함
  - 해시함수는 키들을 균등하게 해시테이블의 인덱스로 변환하기 위해 의미가 부여되어 있는 키를 간단한 계산을 통해 '뒤죽박죽' 만든 후 해시테이블의 크기에 맞도록 해시값을 계산
  - 하지만 아무리 균등한 결과를 보장하는 해시함수이더라도 함수 계산 자체에 긴 시간이 소요된다면 해싱의 장점인 연산의 신속성을 상실하므로 그 가치를 잃음

대표적인 해시함수

- **중간제곱(Mid-square) 함수**
  - 키를 제곱한 후, 적절한 크기의 중간부분을 해시값으로 사용
- **접기(Folding) 함수**
  - 큰 자릿수를 갖는 십진수를 키로 사용하는 경우,
  - 몇 자리씩 일정하게 끊어서 만든 숫자들의 합을 이용해 해시값을 생성
    - 예를 들어, 123456789012에 대해서 1234 + 5678 + 9012 = 15924를 계산한 후에 해시테이블의 크기가 3이라면 15924에서 3자리 수만을 해시값으로 사용
- **곱셈(Multiplicative) 함수**
  - 1보다 작은 실수 δ를 키에 곱하여 얻은 숫자의 소수 부분을 테이블 크기 M과 곱해서 나온 값의 정수부분을 해시값으로 사용
    - h(key) = (int) (key* δ) % 1) * M.
    - Knuth에 의하면 $\delta = \frac{\sqrt{5}-1}{2} \approx 0.61803$이 좋은 성능을 보임.
    - 예를 들면, 테이블 크기 M = 127이고 키가 123456789인 경우, 123456789 x 0.61803 = 76299999.30567, 0.30567 x 127 = 38.82009이므로 38을 해시값으로 사용

대표적인 해시함수

- **이러한 해시함수들의 공통점:**
  - 키의 모든 자리의 숫자들이 함수 계산에 참여함으로써 계산 결과에서는 원래의 키에 부여된 의미나 특성을 찾아볼 수 없게 됨
  - 계산 결과에서 해시테이블의 크기에 따라 특정부분만을 해시값으로 활용
- **가장 널리 사용되는 해시함수: 나눗셈(Division) 함수**
  - 나눗셈 함수는 키를 소수(Prime) M으로 나눈 뒤, 나머지를 해시값으로 사용
    - h(key) = key % M이고, 따라서 해시테이블의 인덱스는 0에서 M-1
  - 여기서 제수로 소수를 사용하는 이유는 나눗셈 연산을 했을 때, 소수가 키들을 균등하게 인덱스로 변환시키는 성질을 갖기 때문

자바의 hashCode()

- **자바의 모든 클래스는 32비트 int를 리턴하는 hashCode() 포함**
  - hashCode()는 객체를 int로 변환하는 메소드
  - 이론적으로 어떤 종류의 객체(사용자가 정의한 객체를 포함하여)라도 해싱할 수 있도록 지원
  - key1.equals(key2)가 true이면, key1.hashCode() == key2.hashCode()가 성립한다는 조건 하에 구현
    - 2 개의 키가 동일하면 각각의 hashCode 값도 같아야 함
  - Integer객체의 경우 hashCode()는 계산 없이 key를 그대로 리턴
  - Boolean 객체의 hashCode()는 key가 true이면 1231, false이면 1237을 각각 리턴

```
01 public final class Integer {
02     private final int key;
03     public int hashCode()
04     { return key; }
05 }
```

```
01 public final class Boolean {
02     private final boolean key;
03     public int hashCode() {
04         if (key) return 1231;
05         else     return 1237;
06     }
07 }
```

자바의 hashCode()

- Double 객체의 hashCode()는 key를 IEEE 64-bit 포맷으로 변환시킨 후, 모든 bit를 계산에 참여시키기 위해 최상위 32 bit와 최하위 32 bit를 XOR한 결과를 리턴

```
01 public final class Double {
02     private final double key;
03     public int hashCode() {
04         long bits = doubletoLongBits(key);
05         return (int) (bits ^ (bits >>>32));}
06 }
```

- String 객체는 key의 문자(char)를 31진수 숫자로 보고 해시값 계산
  - 예를 들어, key = “ball”이라면
  - hash = $98 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0$

    $= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (98))) = 3016191$ 을 리턴.
  - ‘a’, ‘b’, ‘l’의 unicode값은 각각 97, 98, 108

```
01 public final class String {
02     private final char [ ] s;
03     public int hashCode() {
04         int hash =0;
05         for (int i=0; i < length(); i++)
06             hash = s[i] + (31 * hash);
07         return hash;
08     }
09 }
```

자바의 hashCode()

- **자바의 hashCode()는 제각각 다른 값을 리턴하지만 hashCode() 가 리턴하는 값이 모두 signed 32 bit 정수라는 공통점**
  - 자바를 이용하여 해시테이블을 구현 할 때엔 일반적으로 hashCode()를 override하여 해시함수를 구현
  - 6장에서는hash() 메소드를 다음과 같이 선언하여 사용

```
01 private int hash(Key k){
02     return (k.hashCode() & 0x7fffffff) % M;
03 }
```

    - hashCode()에서 리턴되는 32 bit 정수의 최상위 bit(부호 bit)를 제외시키기 위해 key와 “0x7fffffff” 에 대해 AND 연산을 수행하여 얻은 31 bit 양수를 해시테이블의 크기인 M으로 나눈 나머지를 해시값으로 사용
      - 연산 결과값이 음수인 경우 해시테이블의 인덱스로 사용할 수 없기 때문

6.3 개방주소방식(Open Addressing)

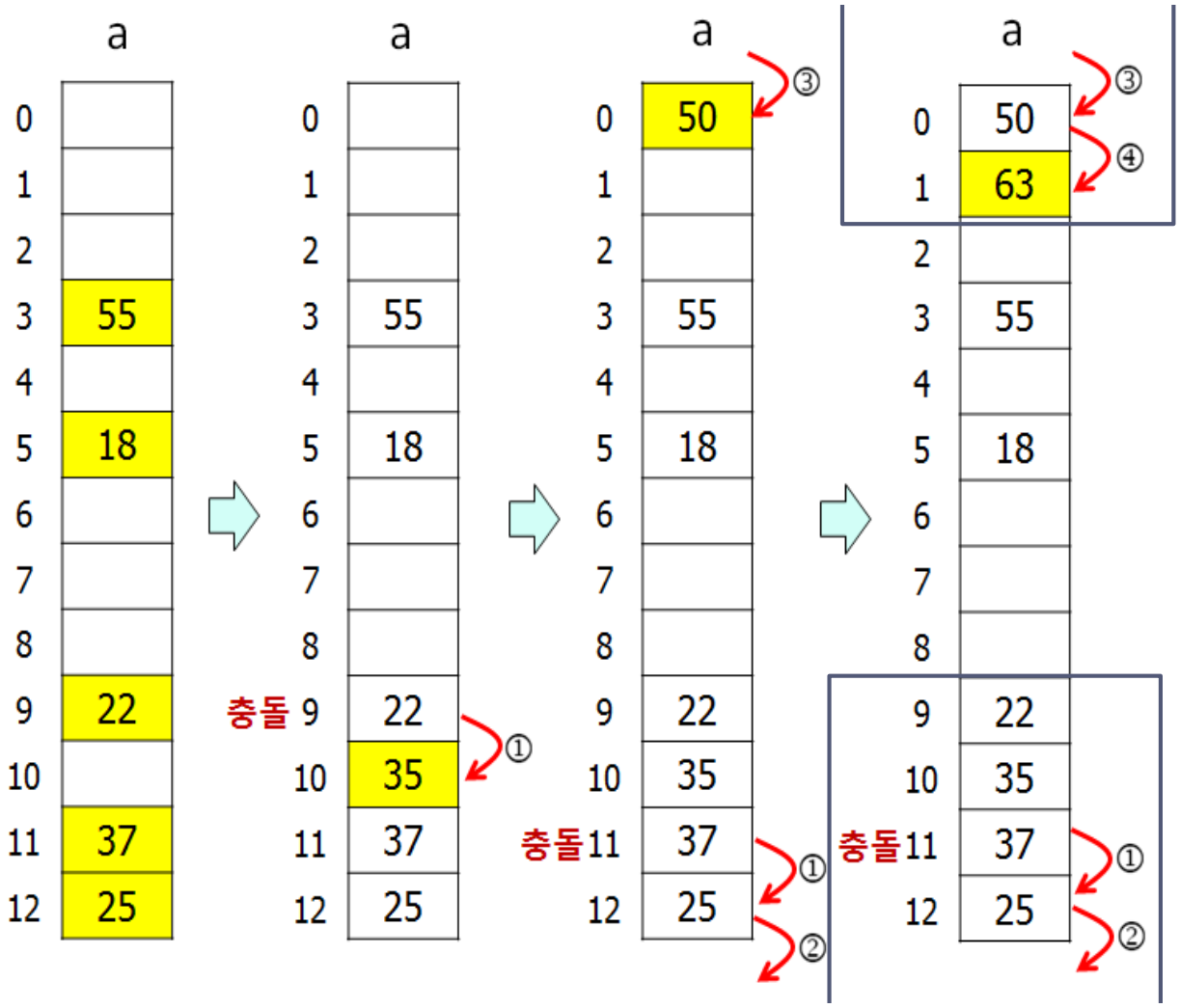
- **개방주소방식은 해시테이블 전체를 열린 공간으로 가정하고 충돌된 키를 일정한 방식에 따라서 찾아낸 empty 원소에 저장**
- **대표적인 개방주소방식:**
  - 선형조사(Linear Probing)
  - 이차조사(Quadratic Probing)
  - 랜덤조사(Random Probing)
  - 이중해싱(Double Hashing)

6.3.1 선형조사(Linear Probing)

- **선형조사는 충돌이 일어난 원소에서부터 순차적으로 검색하여 처음 발견한 empty 원소에 충돌이 일어난 키를 저장**
  - h(key) = i라면, 해시테이블 a[i], a[i+1], a[i+2], …, a[i+j] 를 차례로 검색하여 첫번째로 찾아낸 empty 원소에 key를 저장
  - 해시테이블은 1차원 배열이므로, i + j가 M이 되면 a[0]을 검색
    - (h(key) + j) % M, j = 0, 1, 2, 3, …
- **선형조사의 문제점 - 1차 군집화(Primary Clustering)**
  - 순차탐색으로 empty 원소를 찾아 충돌된 키를 저장하므로 해시테이블의 키들이 빈틈없이 뭉쳐지는 현상이 발생
  - 이러한 군집화는 탐색, 삽입, 삭제 연산 시 군집된 키들을 순차적으로 방문해야 하는 문제점을 야기
  - 군집화는 해시테이블에 empty 원소 수가 적을수록 더 심화되며 해시성능을 극단적으로 저하시킴

선형조사방식의 키 저장 과정

key	h(key) = key % 13
25	12
37	11
18	5
55	3
22	9
35	9
50	11
63	11



```java
public class LinearProbing<K, V> {
    private int M = 13;  // 테이블 크기
    private K[] a = (K[]) new Object[M]; // 해시테이블
    private V[] d = (V[]) new Object[M]; // key관련 데이터 저장
    private int hash(K key){ // 해시코드
        return (key.hashCode() & 0x7fffffff) % M;  // 나눗셈 함수
    }
    private void put(K key, V data) {  // 삽입 연산
        int initialpos = hash(key);    // 초기 위치
        int i = initialpos, j = 1;
        do {
            if (a[i] == null){  // 삽입 위치 발견
                a[i] = key;     // key를 해시테이블에 저장
                d[i] = data;    // key관련 데이터를 동일한 인덱스하에 저장
                return;
            }
            if (a[i].equals(key)) {  // 이미 key 존재:
                d[i] = data;    // 데이터만 갱신
                return;
            }
            i = (initialpos + j++) % M; // i = 다음 위치
        } while (i != initialpos); // 현재 i가 초기위치와 같게되면 루프 종료
    }
```

```java
    public V get(K key) {  // 탐색 연산
        int initialpos = hash(key);
        int i = initialpos, j = 1;
        while (a[i] != null) {  // a[i]가 empty가 아니면
            if (a[i].equals(key))
                return d[i];    // 탐색 성공
            i = (initialpos + j++) % M; // i = 다음 위치
        }
        return null; // 탐색 실패
    }
}
```

Console

<terminated> LinearProbing (1) [Java Application] C:₩Program Files₩J

```
탐색 결과:
50의 data = orange
63의 data = watermelon

해시 테이블:
        0       1       2       3       4       5       6       7       8       9       10      11      12
       50      63    null      55    null      18    null    null    null      22      35      37      25
```

- Line 01: <K, V>는 키와 데이터를 위한generic 타입 선언문
- Line 02의 M은 해시테이블 크기
- Line 03~04: key를 저장할 해시테이블과 key와 관련된 데이터를 저장할 배열의 선언
- Line 06~08: hashCode() 메소드에서 리턴되는 정수를 하위 31 bit 양수로 변환한 후 해시테이블의 크기로 나누는 해시함수를 구현한 메소드
- Line 08~23: key와 데이터 쌍을 저장하는 put() 메소드
  - hash() 메소드로 초기위치인 initialpos를 계산한 후, line 11~22의 do-while루프를 통해 삽입 위치(배열 a에서 empty인 원소)를 발견하면 key와 관련 data를 배열 a와 배열 d에 각각 저장하고, 이미 key가 있는 경우에는 data만 갱신
- Line 21에서는 배열a에서 다음 위치의 조사를 위해 j를 1 증가시킨 후에 (initialpos + j) % M으로 다음 위치를 정함
- Line 22: i가 initialpos와 같게 되면 배열 a에 empty 원소가 없는 것이므로 삽입에 실패한다. 이러한 경우에는 재해싱 (Rehashing) 수행
- Line 24~33: 탐색을 위한 get() 메소드로서 배열 a에 key가 있으면 key와 관련된 data를 리턴
- 25, 37, 18, 55, 22, 35, 50, 63을 차례로 삽입한 후, 50과 63의 data를 각각 출력한 후, 배열 a의 내용 출력

6.3.2 이차조사(Quadratic Probing)

- **이차조사는 선형조사와 근본적으로 동일한 충돌해결 방법**
  - 충돌 후 배열a에서

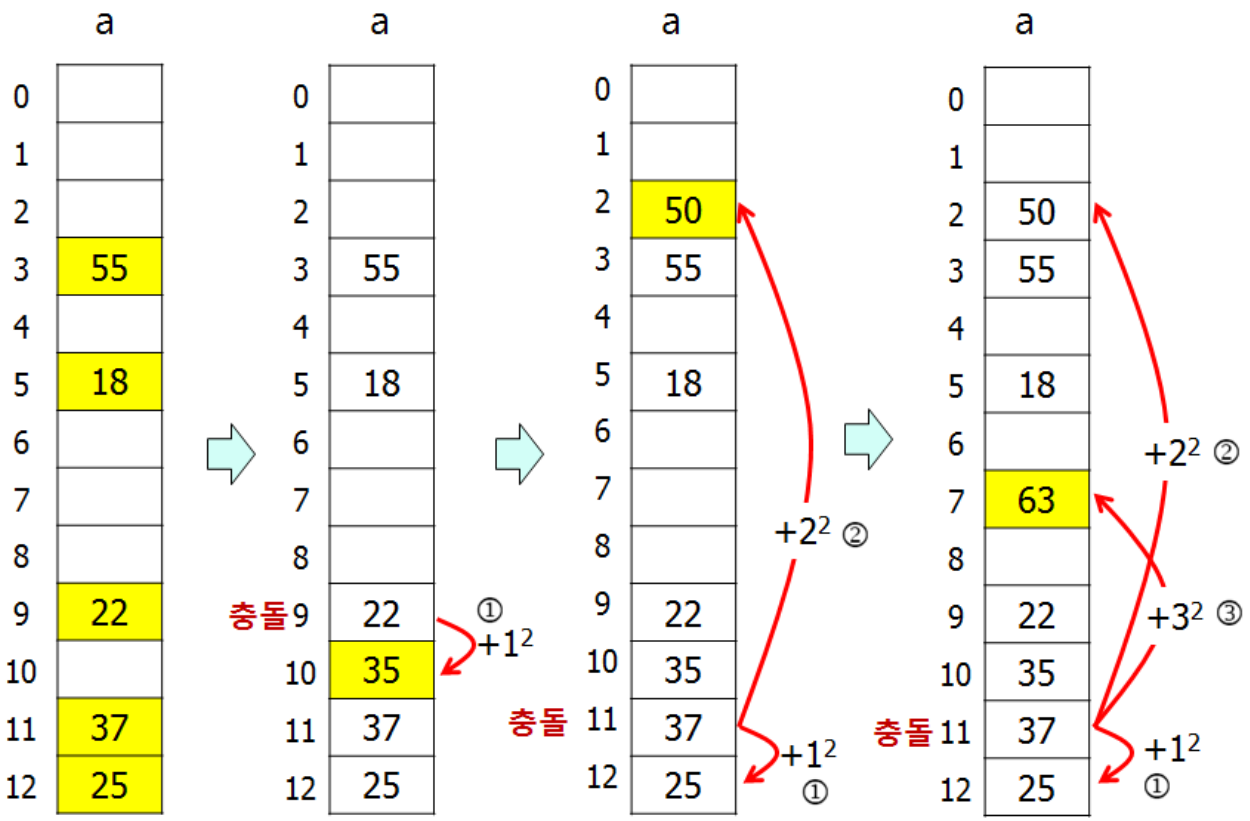
    $(h(key) + j^2)$ % M, j = 0, 1, 2, 3, …

    으로 선형조사보다 더 멀리 떨어진 곳에서 empty 원소를 찾음

  - 이차조사는 이웃하는 빈 곳이 채워져 만들어지는 1차 군집화 문제를 해결
  - 하지만, 2차 군집화(Secondary Clustering)의 문제를 가짐
    - 같은 해시값을 갖는 서로 다른 키들인 동의어(Synonym)들이 똑같은 점프 시퀀스(Jump Sequence)를 따라 empty 원소를 찾아 저장하므로 결국 또 다른 형태의 군집화인 2차 군집화를 야기
    - 점프 크기가 제곱만큼씩 커지므로 배열에 empty 원소가 있는데도 empty 원소를 건너뛰어 빈공간 탐색에 실패하는 경우도 피할 수 없음

이차조사방식의 키 저장 과정

key	h(key) = key % 13
25	12
37	11
18	5
55	3
22	9
35	9
50	11
63	11



```
01 public class QuadProbing<K, V>{
 ⋮      LinearProbing 클래스의 line 02~07 과 동일   private int N = 0, M = 13;  // 항목 수 N, 테이블 크기 M
08     private void put(K key, V data) { // 삽입 연산
09         int initialpos = hash(key);   // 초기 위치
10         int i = initialpos, j = 1;
11         do {
12             if (a[i] == null){  // 삽입 위치 발견
13                 a[i]  = key;    // key를 해시테이블에 저장
14                 d[i] = data; N++; // key관련 데이터 저장
15                   return;
16             }
17             if (a[i].equals(key)) {  // 이미 key 존재
18                 d[i] = data;         // data데이터만 갱신
19             }
20             i = (initialpos + j * j++) % M; // i = 다음 위치
21         } while (N < M);
22     }
23
24     public V get(K key) {  // 탐색 연산
25         int initialpos = hash(key);
26         int i = initialpos, j = 1;
27             while (a[i] != null) {  // a[i]가 empty가 아니면
28                 if (a[i].equals(key))
29                     return d[i];        // 탐색 성공
30                 i = (initialpos + j * j++) % M; // i = 다음 위치
31             }
32             return null; // 탐색 실패
33     }
34 }
```

- Line 02 ~ 07: LinearProbing 클래스의 02 ~ 07과 동일
  - 단, 02 `private int N = 0, M = 13;  // 항목 수 N, 테이블 크기 M`
- Line 08 ~ 22: key와 데이터 쌍을 저장하는 put() 메소드로서 LinearProbing 클래스의 put() 메소드와 거의 동일
  - 단, line 20에서는 배열 a의 다음 위치를 조사하기 위해 (initialpos + j*j) % M으로 증가
- 이후 그 다음 위치를 찾기 위해 j를 1 증가

Console Problems Javadoc Declaration

<terminated> QuadProbing [Java Application] C:₩Program Files₩Java₩jdk1.8.0_40₩bin₩javaw.exe

```
탐색 결과
50의 data = orange
63의 data = watermelon

해시 테이블
          0     1     2     3     4     5     6     7     8     9    10    11    12
       null  null    50    55  null    18  null    63  null    22    35    37    25
```

6.3.3 랜덤조사(Random Probing)

- **랜덤조사는 선형조사와 이차조사의 규칙적인 점프 시퀀스와는 달리 점프 시퀀스를 무작위화 하여 empty 원소를 찾는 충돌해결방법**
- **랜덤조사는 의사 난수 생성기를 사용하여 다음 위치를 찾음**
  - 동일한 seed를 사용하여 동일한 난수열을 생성
- **랜덤조사 방식도 동의어들이 똑같은 점프 시퀀스에 따라 empty 원소를 찾아 키를 저장하게 되고, 이 때문 3차 군집화(Tertiary Clustering)가 발생**

```
01 import java.util.Random;
02 public class RandProbing <K, V>{
:      LinearProbing 클래스의 line 02~07 과 동일  private int N = 0, M = 13;  // 항목 수 N, 테이블 크기 M
09     private void put(K key, V data) { // 삽입 연산
10         int initialpos = hash(key);    // 초기 위치
11         int i = initialpos;
12         Random rand = new Random();
13         rand.setSeed(10);
14         do {
15             if (a[i] == null){ // 삽입 위치 발견
16                 a[i] = key;     // key를 해시테이블에 저장
17                 d[i] = data; N++; // key관련 데이터 저장
18                 return;
19             }
20             if (a[i].equals(key)) { // 이미 key 존재
21                 d[i] = data;  // 데이터만 갱신
22                 return;
23             }
24             i = (initialpos + rand.nextInt(1000)) % M;  // i = 다음 위치
25         } while (N < M);
26     }
27     public V get(K key) {  //탐색 연산
28         Random rand = new Random();
29         rand.setSeed(10);  // 삽입때와 같은 seed값 사용
30         int initialpos = hash(key);    // 초기 위치
31         int i = initialpos;
32         while (a[i] != null) {
33             if (a[i].equals(key))
34                 return d[i];    // 탐색 성공
35             i = (initialpos + rand.nextInt(1000)) % M; // i = 다음 위치
36         }
37         return null;  // 탐색 실패
38     }
39 }
```

- Line 01: 난수 생성을 위한 java.util.Random 라이브러리를 사용하기 위한 import 문

```
02 private int N = 0, M = 13;  // 항목 수 N, 테이블 크기 M
```

- Line 09~26: 키와 데이터 쌍을 저장하는 put() 메소드로 LinearProbing 클래스의 put() 메소드와 거의 동일
- Line 12~13: Random rand = new Random()와 rand.setSeed(10)으로 난수 생성 준비를 마치고, line 24에서 배열 a의 다음 위치를 조사하기 위해
- (initialpos + rand.nextInt(1000)) % M을 계산
  - 초기값 10과 난수 범위 인자로 1000을 사용하여 생성된 난수
  - 113, 380, 293, 290, 246, 456, 797, 888, 981, 214, …

Console | Problems | Javadoc | Declaration

```
<terminated> RandProbing [Java Application] C:₩Program Files₩Java₩jdk1.8.0_40₩bin₩javaw.exe
탐색 결과
50의 data = orange
63의 data = watermelon

해시 테이블
   0    1    2    3    4    5    6    7    8    9   10   11   12
  35   63 null   55 null   18 null   50 null   22 null   37   25
```

6.3.4 이중해싱

- **이중해싱(Double Hashing)은 2 개의 해시함수를 사용**
  - 하나는 기본적인 해시함수h(key)로 키를 해시테이블의 인덱스로 변환하고, 제2의 함수 d(key)는 충돌 발생 시 다음 위치를 위한 점프 크기를 다음의 규칙에 따라 정함

$$(h(key) + j \cdot d(key)) \bmod M,\ j = 0, 1, 2, \cdots$$

  - 이중해싱은 동의어들이 저마다 제2 해시함수를 갖기 때문에 점프 시퀀스가 일정하지 않음. 따라서 이중해싱은 모든 군집화 문제를 해결
  - 제 2의 함수 d(key)는 점프 크기를 정하는 함수이므로 0을 리턴해선 안됨
  - 그 외의 조건으로 d(key)의 값과 해시테이블의 크기 M과 서로소(Relatively Prime) 관계일 때 좋은 성능을 보임
    - 해시테이블 크기 M을 소수로 선택하면, 이 제약 조건을 만족

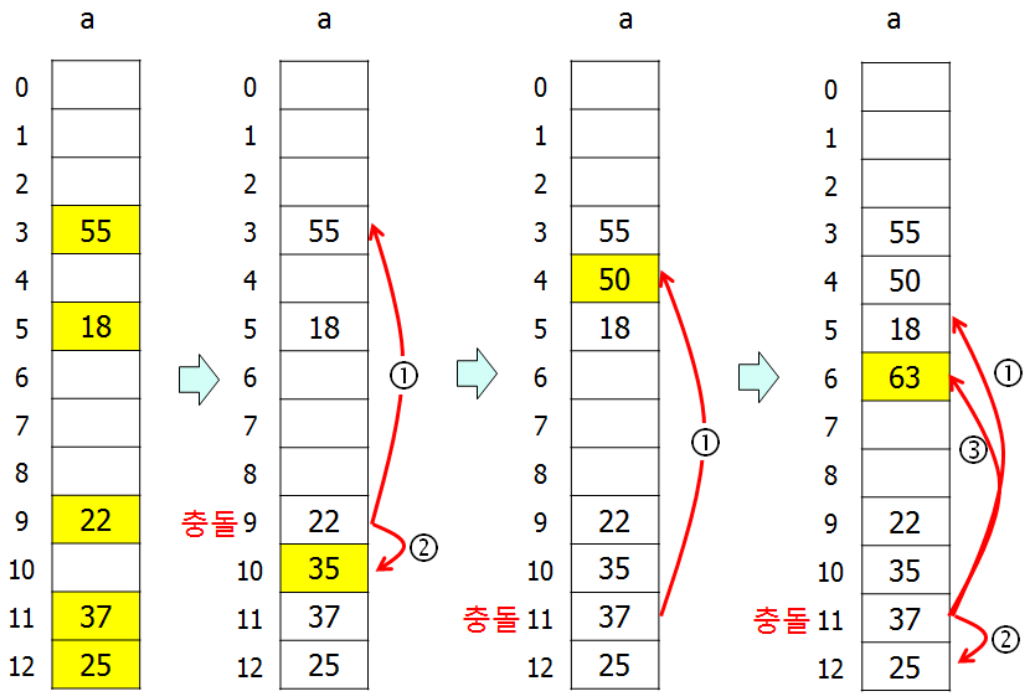
- h(key) = key % 13과 d(key) = 7-(key % 7) 에 따라, 25, 37, 18, 55, 22, 35, 50, 63을 해시테이블에 차례로 저장하는 과정

key	h(key)	d(key)	(h(key) + j*d(key)) % 13		
			j=1	j=2	j=3
25	12				
37	11				
18	5				
55	3				
22	9		①	②	
35	9	7	3	10	
50	11	6	4		③
63	11	7	5	12	6

h(key) = key % 13

d(key) = 7 – (key % 7)

(h(key) + j*d(key)) % 13, j = 0, 1, …



```
01 public class DoubleHashing <K, V>{
 ⋮      LinearProbing 클래스의 line 02~07 과 동일   private int N = 0, M = 13;  // 항목 수 N, 테이블 크기 M
08     private void put(K key, V data) {
09         int initialpos = hash(key);    // 초기 위치
10         int i = initialpos;
11         int j=1;
12         int d=(7-(int)key % 7);  // 두번째 해시 함수, d(key)=7-key%7
13         do {
14             if (a[i] == null){     // 삽입 위치 발견
15                 a[i]  = key;       // key를 해시테이블에 저장
16                 dt[i] = data; N++; // key관련 데이터 저장
17                 return;
18             }
19             if (a[i].equals(key)) { // 이미 key 존재
20                 dt[i] = data;    // 데이터만 갱신
21                 return;
22             }
23             i = (initialpos + j*d) % M; // i = 다음 위치
24             j++;
25         } while (N < M);
26     }
27     public V get(K key) {
28         int initialpos = hash(key);      // 초기 위치
29         int i = initialpos;
30         int j=1;
31         int d=(7-(int)key % 7);
32         while (a[i] != null) {
33             if (a[i].equals(key))
34                 return dt[i];     // 탐색 성공
35             i = (initialpos + j*d) % M; // i = 다음 위치
36             j++;
37         }
38         return null;  // 탐색 실패
39     }
40 }
```

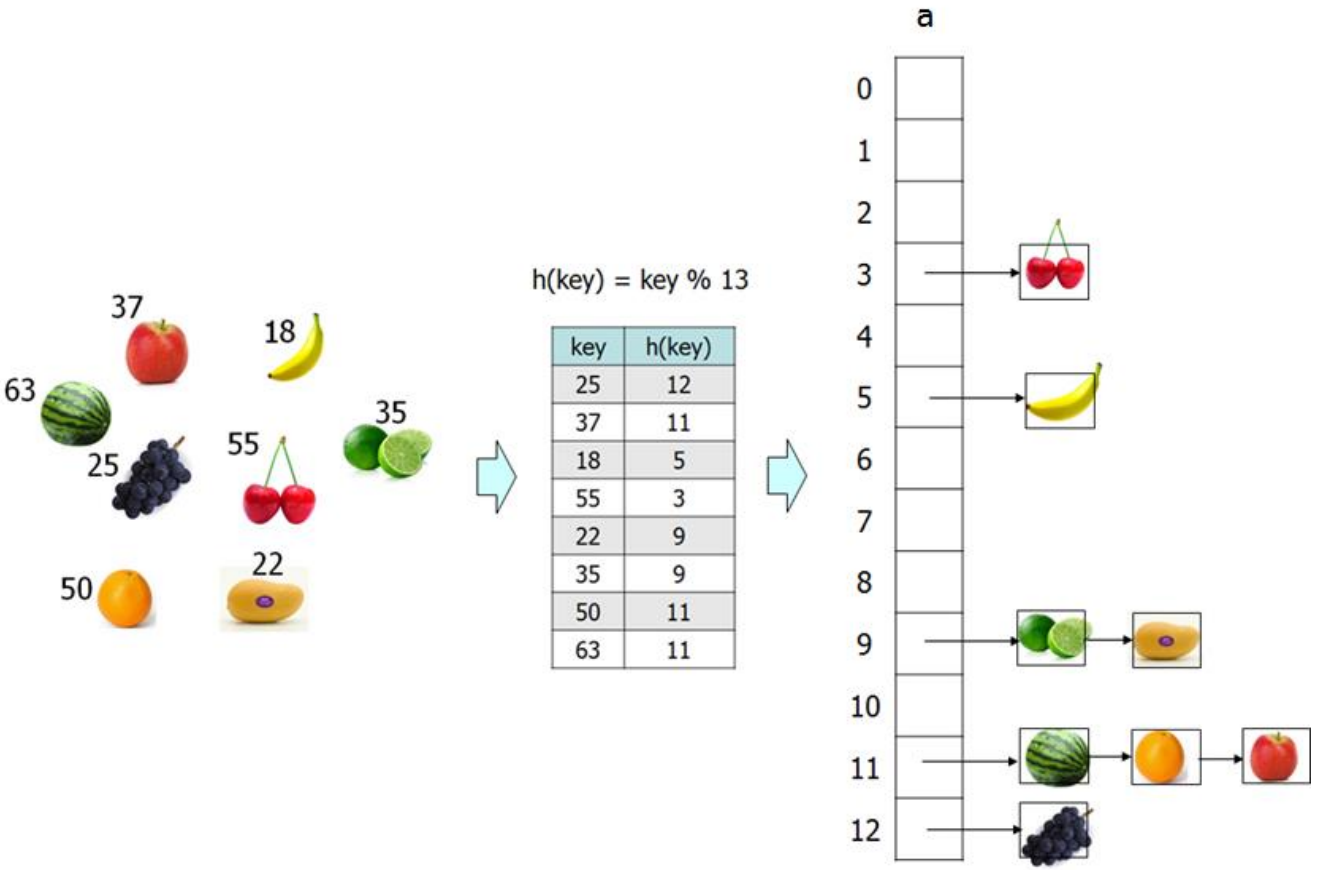
- Line 08~26: key와 데이터 쌍을 저장하는 put() 메소드로서 LinearProbing 클래스의 put() 메소드와 거의 동일
- Line 12: 제 2 함수 d = 7 - key % 7을 계산
- Line 23: 배열 a의 다음 위치를 조사하기 위해 (initialpos + j*d) % M 계산
- 완성된 프로그램에서 25, 37, 18, 55, 22, 35, 50, 63을 차례로 삽입한 후, 50과 63의 data와 배열 a의 내용을 출력한 결과

```
Console  Problems  Javadoc  Declaration
<terminated> DoubleHashing [Java Application] C:₩Program Files₩Java₩jdk1.8.0_40₩bin₩javaw.exe
탐색 결과
50의 data = orange
63의 data = watermelon

해시 테이블
   0    1    2    3    4    5    6    7    8    9   10   11   12
null null null  55   50   18   63 null null  22   35   37   25
```

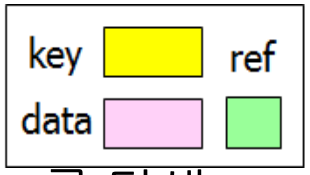
6.4 폐쇄주소방식(Closed Addressing)

- **폐쇄주소방식의 충돌해결 방법은 키에 대한 해시값에 대응되는 곳에만 키를 저장**
  - 충돌이 발생한 키들은 한 위치에 모여 저장
  - 이를 구현하는 가장 대표적인 방법: 체이닝(Chaining)

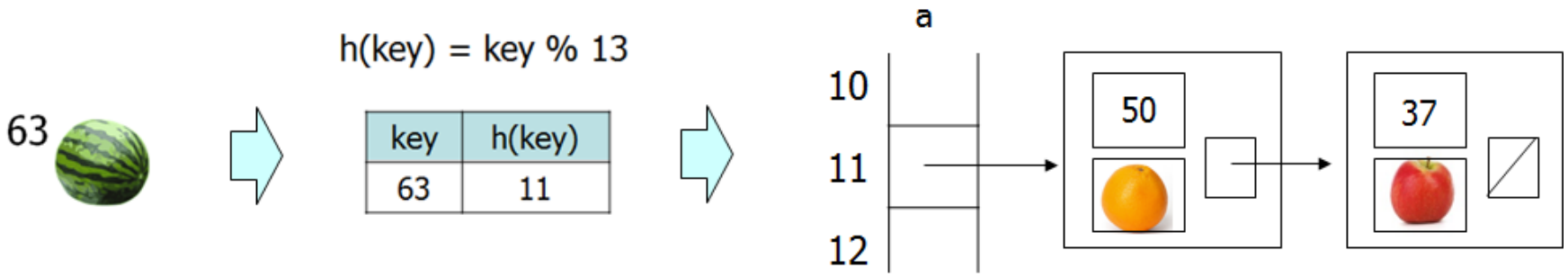


```java
public class Chaining<K, V> {
    private int M = 13; // 테이블 크기
    private Node[] a = new Node[M]; // 해시 테이블
    public static class Node {  // Node 클래스
        private Object key;
        private Object data;
        private Node   next;
        public Node(Object newkey, Object newdata, Node ref){  // 생성자
            key  = newkey;
            data = newdata;
            next = ref;
        }
        public Object getKey()  { return key; }
        public Object getData() { return data;}
    }
    private int hash(K key) { //해시코드
        return (key.hashCode() & 0x7fffffff) % M; } // 나눗셈 연산
    public V get(K key) {  //탐색 연산
        int i = hash(key);
        for (Node x = a[i]; x != null; x = x.next)    // 연결리스트 탐색
            if (key.equals(x.key)) return (V) x.data;  // 탐색 성공
        return null;  // 탐색 실패
    }
    private void put(K key, V data) { // 삽입 연산
        int i = hash(key);
        for (Node x = a[i]; x != null; x = x.next)
            if (key.equals(x.key)) {  // 이미 key 존재
                x.data =  data;      // 데이터만 갱신
                return;
            }
        a[i] = new Node(key, data, a[i]); // 연결 리스트의 첫 노드로 삽입
    }
}
```

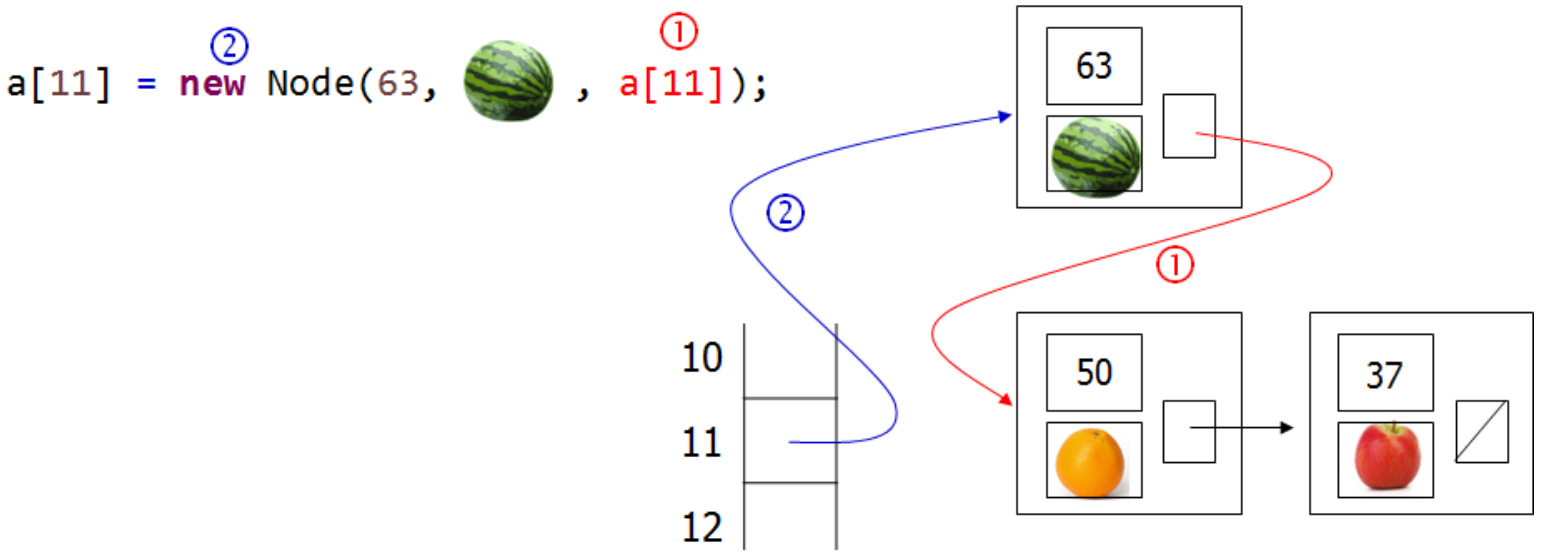
- Line 03: 해시테이블인 배열a를 선언
  - 배열의 원소에는 연결리스트 Node를 가리키는 래퍼런스를 저장
- Line 04~15: 연결리스트의 노드를 위한 Node 클래스
  - Node는 키를 저장하는 key, 데이터를 저장하는 data, 다른 Node를 참조하는 ref 로 구성



- Line 16~17: 해시함수 h(key) = key % 13 구현
- Line 18~23: get() 메소드로서 line 20의 for-루프를 통해 key를 탐색
- Line 24~32: put() 메소드
- Line 26~30: 저장하려고 하는 key가 이미 저장되어 있는지를 get() 메소드의 for-루프와 같은 방법으로 탐색하여 key가 발견되면 data를 갱신
- 실제로 key가 새로운 키일 때, line 31에서 Node를 할당 받아 key와 data 를 저장한 후, 해당 연결리스트의 첫 노드로 삽입

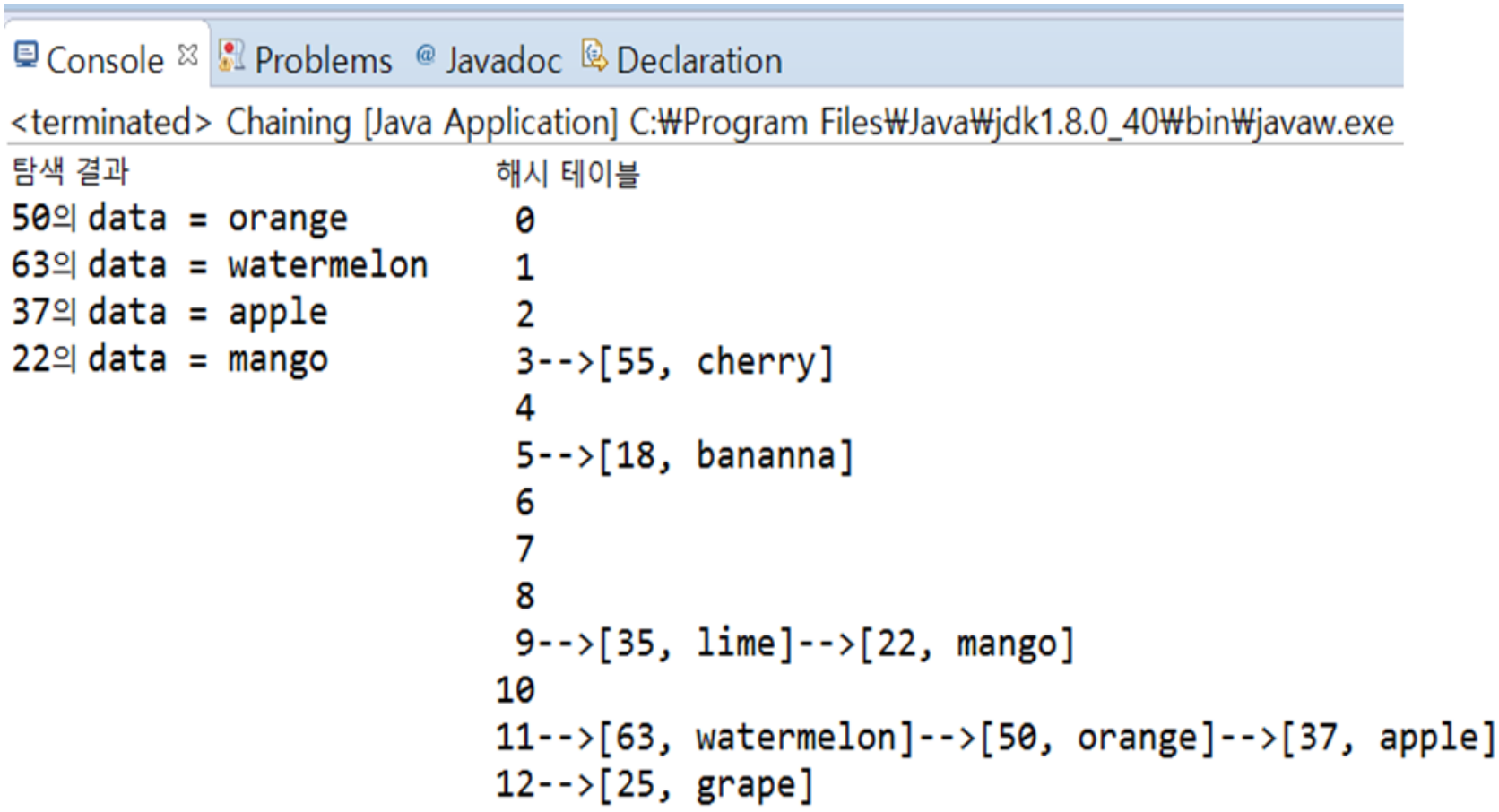


63을 삽입하기 전



63을 삽입한 후

- 완성된 프로그램에서 25, 37, 18, 55, 22, 35, 50, 63을 차례로 삽입한 후, 50, 63, 37, 22의 data와 배열 a의 내용을 출력한 결과



6.5 기타 해싱

- 융합해싱(Coalesced Hashing):
- 2-방향 체이닝(Two-Way Chaining)
- 뻐꾸기 해싱(Cickoo Hashing)

6.6 재해시와 동적해싱

- **어떤 해싱방법도 해시테이블에 비어있는 원소가 적으면,**
  삽입에 실패하거나 해시성능이 급격히 저하되는 현상을 피할 수 없음
- **이 경우, 해시테이블을 확장시키고 새로운 해시함수를 사용하여 모든 키들을 새로운 해시테이블에 다시 저장하는 재해시(Rehash)가 필요**
  - 재해시는 오프라인(Off-line)에서 이루어지고
    모든 키들을 다시 저장해야 하므로 O(N) 시간이 소요

- **재해시 수행 여부는 적재율(Load Factor)에 따라 결정**
  - 적재율 α = (테이블에 저장된 키의 수 N)/ (테이블 크기 M)
  - 일반적으로 $\alpha > 0.75$가 되면 해시테이블 크기를 2 배로 늘리고,
    $\alpha < 0.25$가 되면 해시테이블을 1/2로 줄임

동적해싱(Dynamic Hashing)

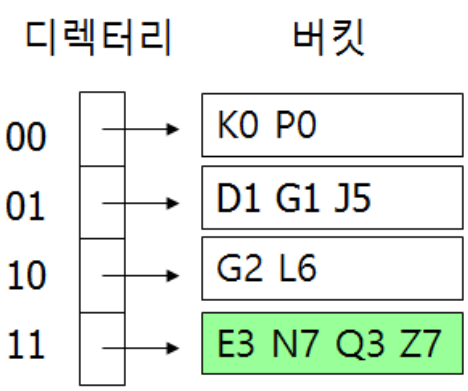
- **대용량의 데이터베이스를 위한 해시방법으로 재해싱을 수행하지 않고 동적으로 해시테이블의 크기를 조절**
- **대표적인 동적해싱**
  - 확장해싱(Extendible Hashing)
  - 선형해싱(Linear Hashing)

확장해싱

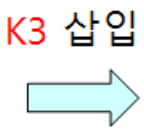
- **디렉터리(Directory)를 메인메모리(Main Memory)에 저장하고, 데이터는 디스크 블록(Disk Block) 크기의 버킷(Bucket) 단위로 저장**
  - 버킷이란 키를 저장하는 곳
  - 확장해싱에서는 버킷에 overflow가 발생하면 새 버킷을 만들어 나누어 저장하며 이때 이 버킷들을 가리키던 디렉터리는 2 배로 확장

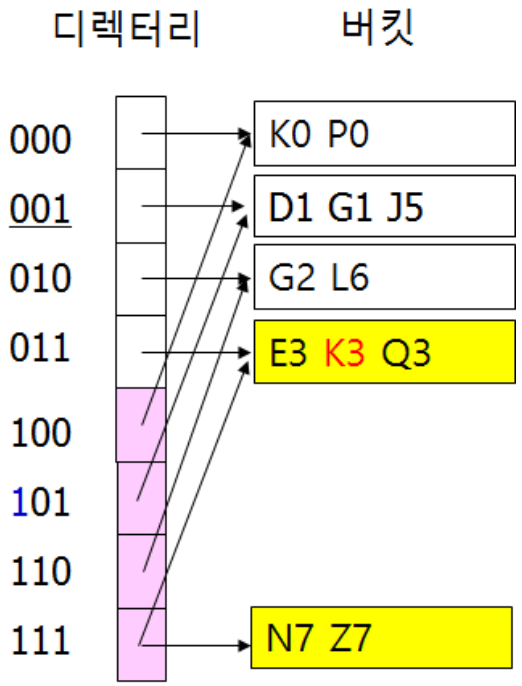
K0	K0000
P0	P0000
D1	D0001
G1	G0001
G2	G0010
E3	E0011
Q3	Q0011
J5	J0101
L6	L0110
N7	N0111
Z7	Z0111
K3	K0011

(a) 키 코드



(b) 디렉터리 확장 전





(c) 디렉터리 확장 후

확장해싱

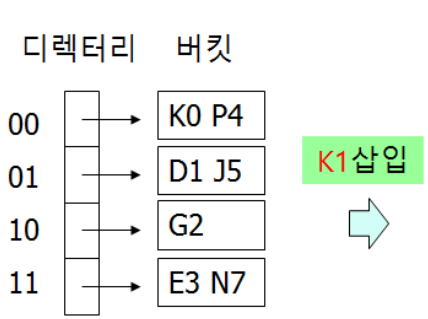
- 예제 설명
  - (a)의 키 코드의 마지막 두 자리를 가지고 키들을 버킷에 저장한다.
    - 이 때의 버킷 크기는 4. 즉, (b)에서 버킷 [E3, N7, Q3, Z7]은 꽉 차있는 상태
  - K3을 삽입하면 K3의 코드의 마지막 2자리가 '11'이므로 [E3, N7, Q3, Z7] 버킷에 저장되어야 하지만 꽉 차있으므로, (c)와 같이 디렉터리를 2배로 확장
  - 이 때 코드의 마지막 세 자리를 가지고 키들을 탐색, 삽입, 삭제 연산을 수행

선형해싱

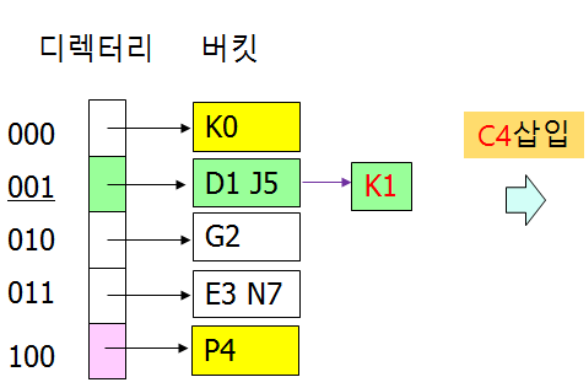
- **디렉터리 없이 삽입할 때 버킷을 순서대로 추가하는 방식**
  - 추가되는 버킷은 삽입되는 키가 저장되는 버킷과 무관하게 순차적으로 추가
  - 만일 삽입되는 버킷에 저장공간이 없으면 overflow 체인에 새 키를 삽입
  - 체인은 단순연결리스트로서 overflow된 키들을 임시로 저장하고, 나중에 버킷이 추가되면overflow 체인의 키들을 버킷으로 이동

K0	K0000
P4	P0100
D1	D0001
G2	G0010
E3	E0011
J5	J0101
N7	N0111
K1	K0001
C4	C0100

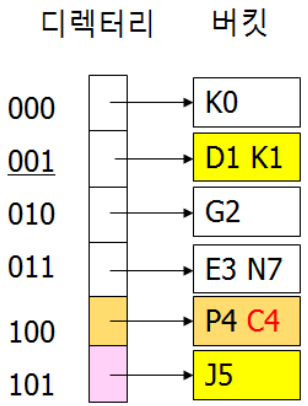
(a) 키 코드



(b) K1 삽입 전



(c) K1 삽입 후



(d) C4 삽입 후

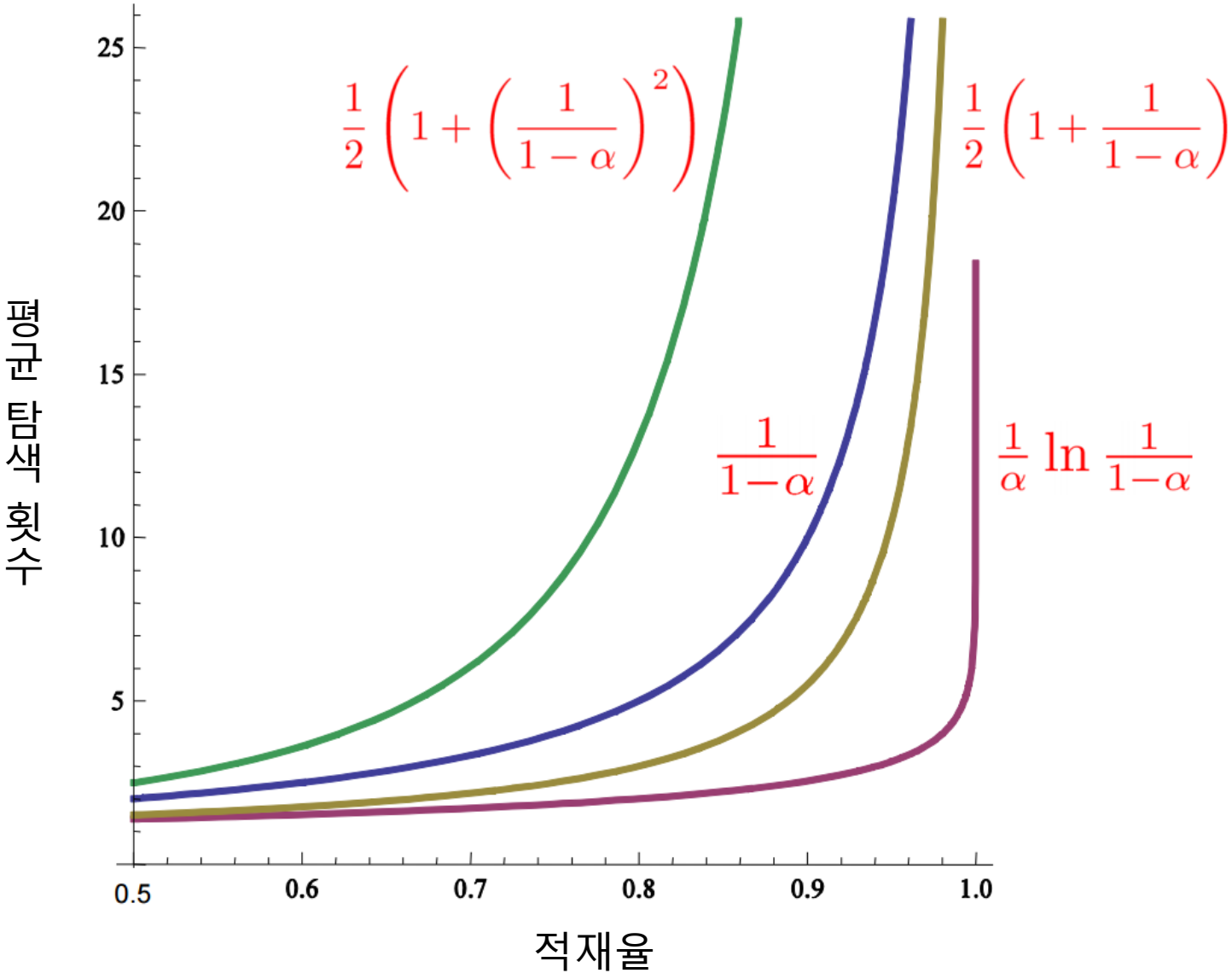
선형해싱

- 예제 설명
  - 버킷 크기가 2이다.
  - (b)는 (a)의 키 코드에 따라 마지막 두 자리를 이용하여 키들을 저장한 상태
  - (c)는 K1을 삽입하려는데 버킷 001 에 저장할 공간이 없어 overflow 체인에 임시로 K1을 저장한 경우
  - 다음으로 추가되는 버킷은 인덱스가 100이며, 이 때 버킷 000에 저장되었던 P4는 버킷 100으로 이동
    - 왜냐하면 P4 코드의 마지막 3 bit가 100이기 때문
  - (d)는 C4를 100 버킷에 삽입한 경우이며, 새롭게 101 버킷이 추가.
  - 001 버킷의 코드가 101로 끝나는 키인 J5가 버킷 101로 이동하고, overflow 체인의 K1은 버킷 001로 이동
  - 다음 키가 삽입될 때에는 버킷110 이 추가

- **선형해싱은 디렉터리를 사용하지 않는다는 장점을 가지며, 인터렉티브 (Interactive) 응용에 적합**

6.7 해시방법의 성능 비교 및 응용

- 해시방법의 성능은 탐색이나 삽입 연산을 수행할 때
  성공과 실패한 경우를 각각 분석하여 측정
- 선형조사는 적재율 α가 너무 작으면 해시테이블에 empty 원소가 너무 많고, α 값이 1.0에 근접할수록 군집화가 심화됨
- 개방주소방식의 해싱은 $\alpha \approx 0.5$, 즉, $M \approx 2N$일 때 상수시간 성능 보임
- 체이닝은 α가 너무 작으면 대부분의 연결리스트들이 empty 가 되고, 너무 큰 α는 연결리스트들의 길이가 너무 길어져 해시성능이 매우 저하
- 일반적으로 M이 소수이고, $\alpha \approx 10$ 정도이면 O(1) 시간 성능을 보임



대표적인 해싱방법의 성능 비교

대표적인 해싱방법의 성능 비교

	탐색 성공	삽입/탐색 실패
선형조사	$\frac{1}{2}[1+\frac{1}{(1-\alpha)}]$	$\frac{1}{2}[1+\frac{1}{(1-\alpha)^2}]$
이중해싱	$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$	$\frac{1}{1-\alpha}$
체이닝	$1+\frac{\alpha}{2}$	α

요약

- 해싱이란 키를 간단한 함수로 계산한 값을 배열의 인덱스로 이용하여 항목을 저장하고, 탐색, 삽입, 삭제 연산을 평균 O(1) 시간에 지원하는 자료구조
- 해시함수는 키들을 균등하게 해시테이블의 인덱스로 변환, 대표적인 해시함수는 나눗셈 함수
- 충돌해결방법들은 두가지로 분류: 개방주소방식, 폐쇄주소방식
- 개방주소방식: 선형조사, 이차조사, 랜덤조사, 이중해싱
- 폐쇄주소방식은 키에 대한 해시값에 대응되는 곳에만 키를 저장
- 체이닝은 해시테이블 크기만큼의 연결리스트를 가지며, 키를 해시값에 대응되는 연결리스트에 저장
  - 군집화 현상이 발생하지 않으며, 구현이 간결하여 실제로 가장 많이 활용되는 해시방법

- 재해시는 삽입에 실패하거나 해시성능이 급격히 저하되었을 때, 해시테이블의 크기를 확장하고 새로운 해시함수를 사용해 모든 키들을 새로운 해시테이블에 저장
- 동적해싱은 대용량의 데이터베이스를 위한 해시방법으로 재해시를 수행하지 않고 동적으로 해시테이블의 크기를 조절: 확장해싱과 선형해싱