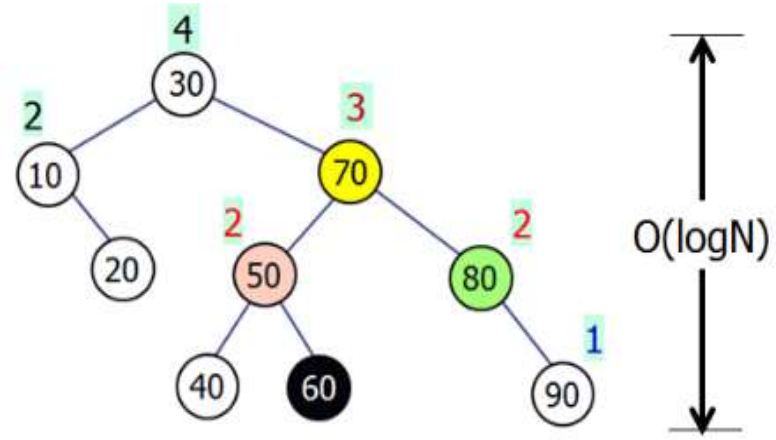
14주차

해시테이블

해시테이블

- 이진탐색트리의 성능을 개선한 AVL 트리의 삽입과 삭제 연산의 수행시간은 각각 O(logN)



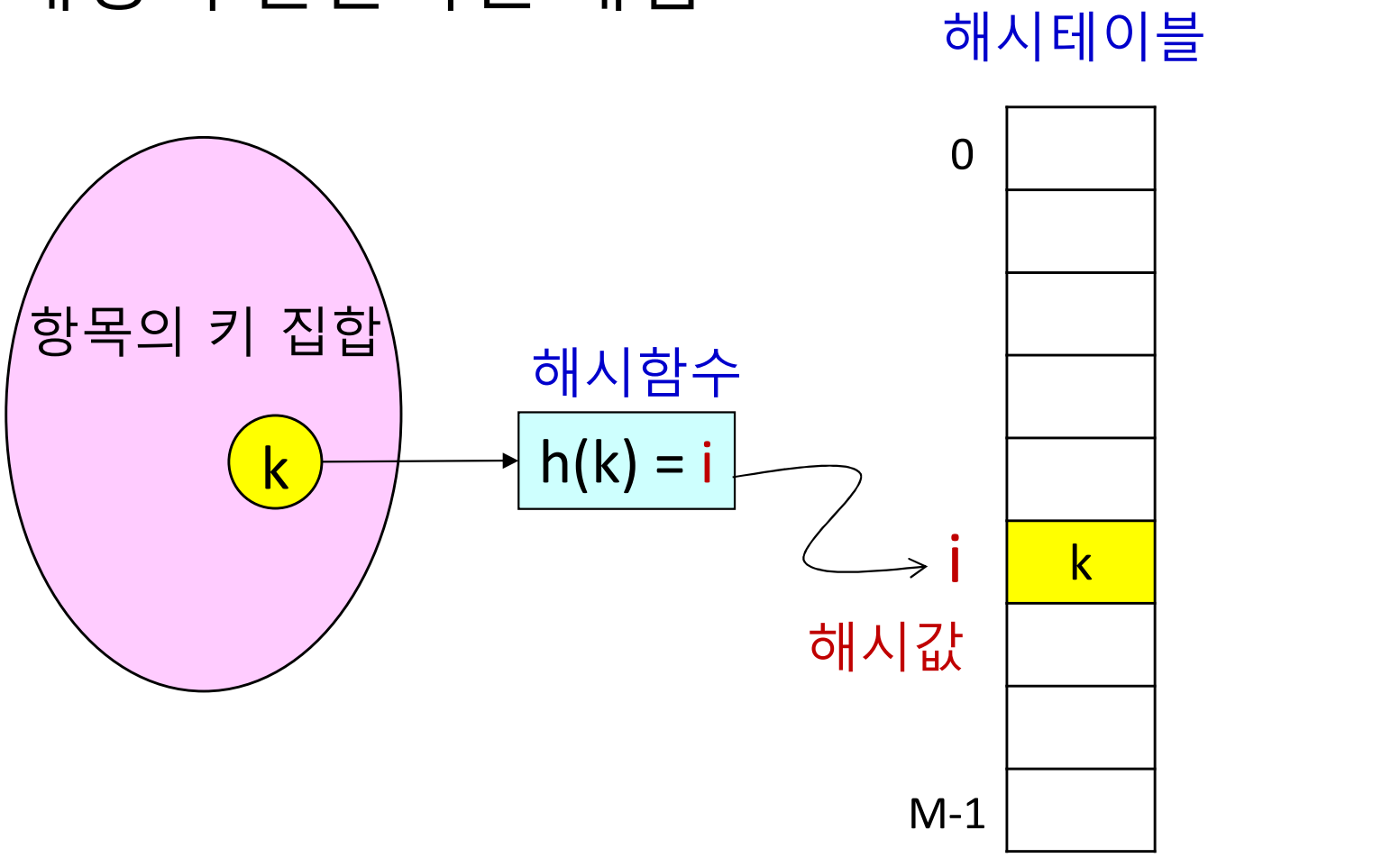
- 그렇다면 O(logN) 보다 좋은 성능을 갖는 자료구조는 없을까?

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

[그림 6-2] 키를 그대로 1차원 리스트의 인덱스로 사용

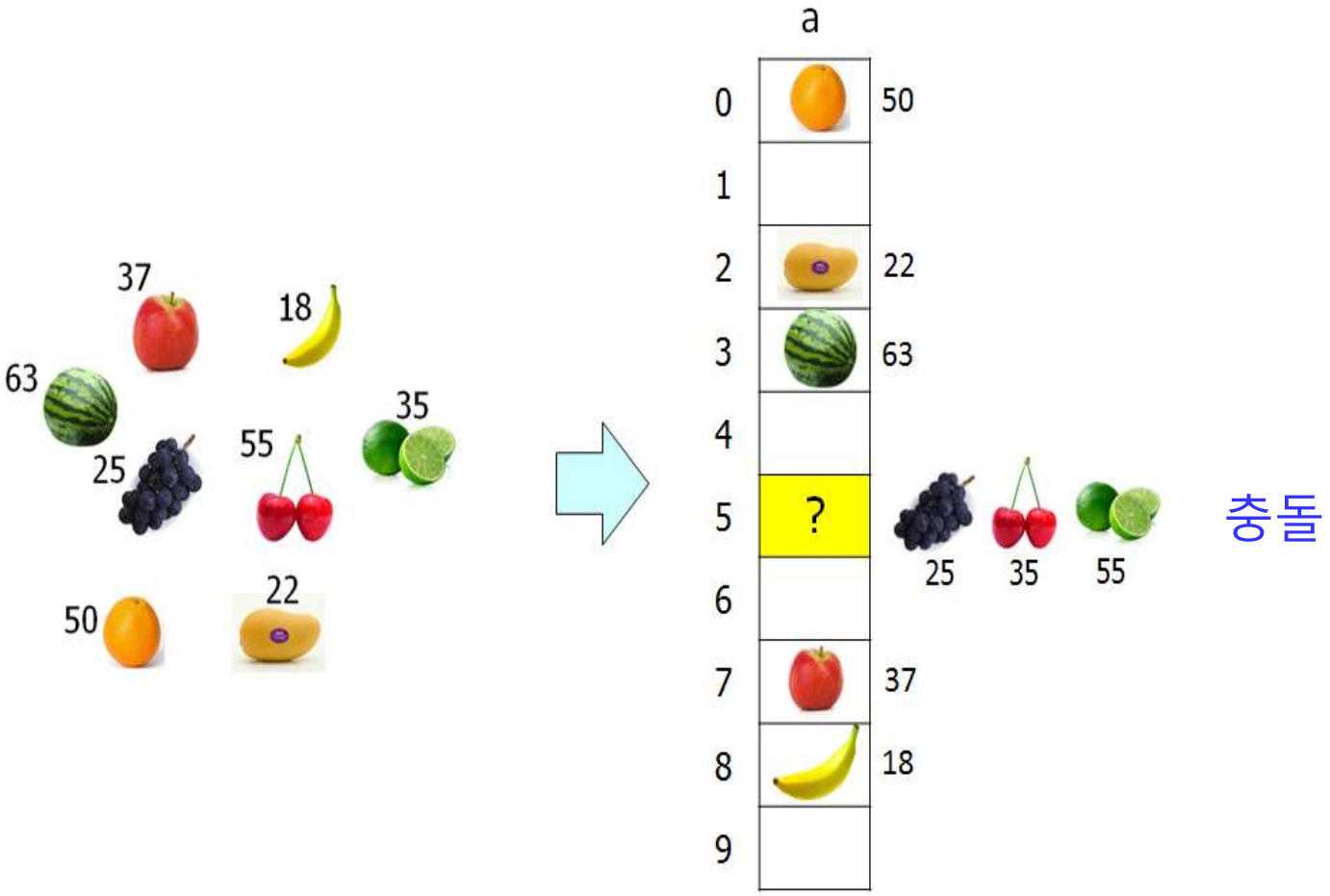
- 그러나 키를 배열의 인덱스로 그대로 사용하면 메모리 낭비가 심해질 수 있음
- [문제 해결 방안] 키를 변환하여 배열의 인덱스로 사용
- 키를 간단한 함수를 사용해 변환한 값을 배열의 인덱스로 이용하여 항목을 저장하는 것을 해싱(Hashing)이라고 함
- 해싱에 사용되는 함수를 해시함수(Hash Function)라 하고, 해시함수가 계산한 값을 해시값(Hash value) 또는 해시주소라고 하며, 항목이 해시값에 따라 저장되는 배열을 해시테이블(Hash Table)이라고 함

해싱의 전반적인 개념



M = 해시테이블 크기

- 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



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

- 아무리 균등한 결과를 보장하는 해시함수라도 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 만든다.

  - 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 곱한다. 이렇게 나온 값의 정수 부분을 해시값으로 사용
  - h(key) = ((key* δ) % 1) * M 이다. Knuth에 의하면 $\delta = \frac{\sqrt{5}-1}{2} \approx 0.61803$이 좋은 성능을 보인다.
  - 예를 들면, 테이블 크기 M = 127이고 키가 123456789인 경우, 123456789 x 0.61803 = 76299999.30567, 0.30567 x 127 = 38.82009이므로 38을 해시값으로 사용

- 이러한 해시함수들의 공통점:
  - 키의 모든 자리의 숫자들이 함수 계산에 참여함으로써 계산 결과에서는 원래의 키에 부여된 의미나 특성을 찾아볼 수 없게 된다는 점
  - 계산 결과에서 해시테이블의 크기에 따라 특정부분만을 해시값으로 활용한다는 점

- 가장 널리 사용되는 해시함수: 나눗셈(Division) 함수
  - 나눗셈 함수는 키를 소수(Prime) M으로 나눈 뒤, 그 나머지를 해시값으로 사용
  - h(key) = key % M이고, 따라서 해시테이블의 인덱스는 0에서 M-1이 됨
  - 여기서 제수로 소수를 사용하는 이유는 나눗셈 연산을 했을 때, 소수가 키들을 균등하게 인덱스로 변환시키는 성질을 갖기 때문

개방주소방식

- 개방주소방식(Open Addressing)은 해시테이블 전체를 열린 공간으로 가정하고 충돌된 키를 일정한 방식에 따라서 찾아낸 empty 원소에 저장
- 대표적인 개방주소방식:

  선형조사(Linear Probing)

  이차조사(Quadratic Probing)

  이중해싱(Double Hashing)

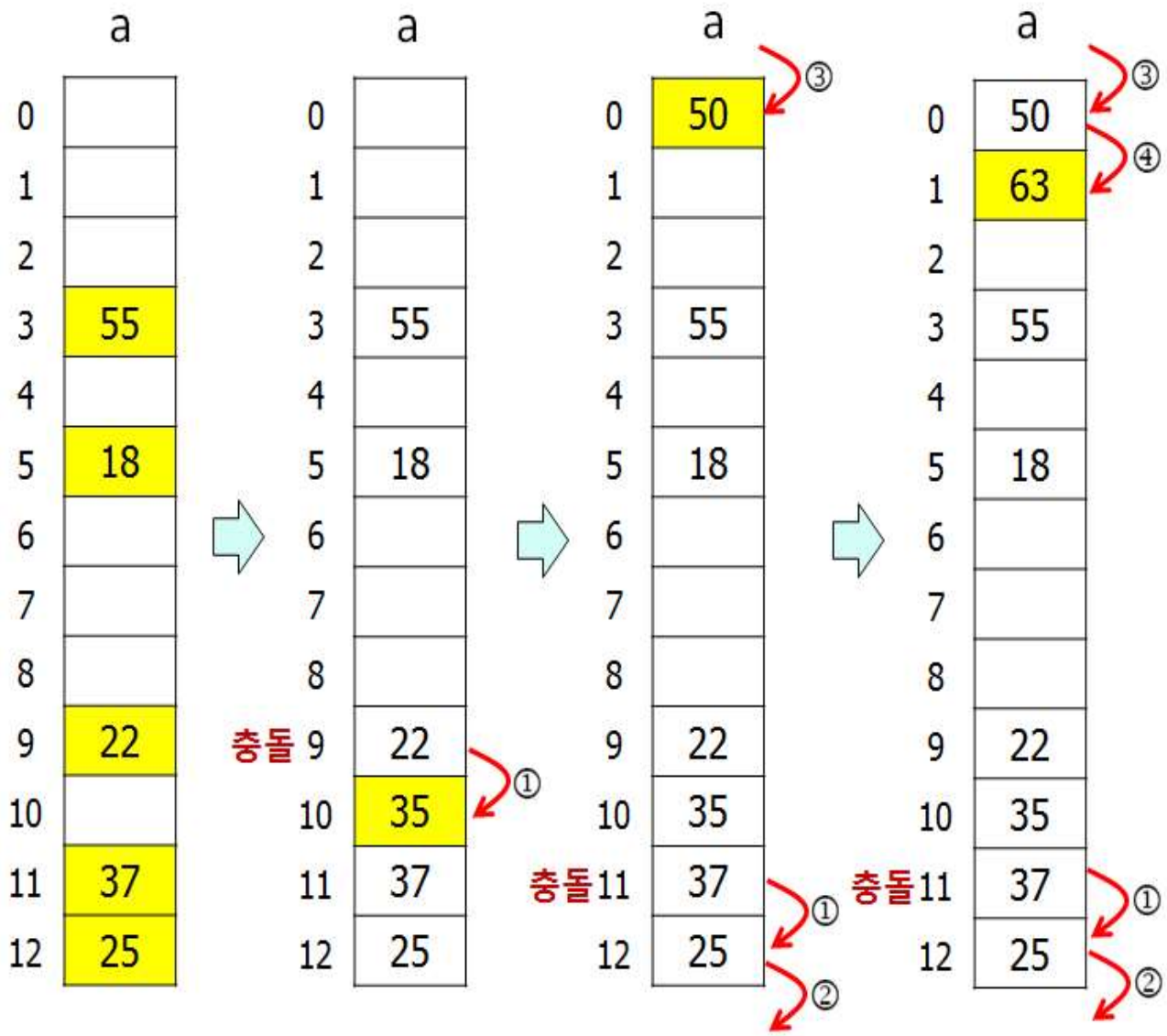
선형조사

- 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 처음으로 찾아낸 empty 원소에 key를 저장
- 해시테이블은 1차원 리스트이므로, i + j가 M이 되면 a[0]을 검색

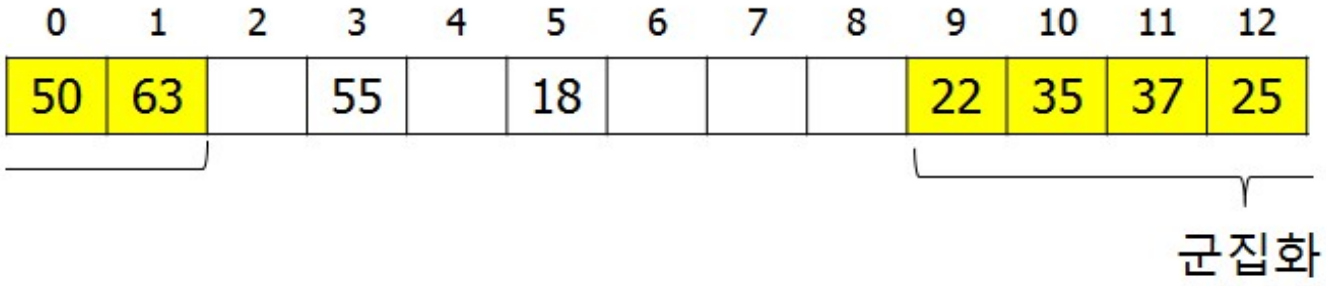
(h(key) + j) % M, j = 0, 1, 2, 3, …

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



- 선형조사는 순차탐색으로 empty 원소를 찾아 충돌된 키를 저장하므로 해시테이블의 키들이 빈틈없이 뭉쳐지는 현상이 발생[1차 군집화(Primary Clustering)]
- 이러한 군집화는 탐색, 삽입, 삭제 연산 시 군집된 키들을 순차적으로 방문해야 하는 문제점을 야기



- 군집화는 해시테이블에 empty 원소 수가 적을수록 더 심화되며 해시 성능을 극단적으로 저하시킴

```
01 class LinearProbing:
02     def __init__(self, size):
03         self.M = size
04         self.a = [None] * size
05         self.d = [None] * size
06
07     def hash(self, key):
08         return key % self.M
09
```

객체 생성자
테이블 크기 M
해시테이블 a
데이터 저장용 d

나눗셈 해시함수

```
10      def put(self, key, data): # 삽입 연산
11          initial_position = self.hash(key)
12          i = initial_position
13          j = 0
14          while True:
15              if self.a[i] == None:
16                  self.a[i] = key
17                  self.d[i] = data
18                  return
19              if self.a[i] == key:
20                  self.d[i] = data
21                  return
22              j += 1
23              i = (initial_position + j) % self.M
24              if i == initial_position:
25                  break
26
```

초기 위치

빈 곳 발견

key는 해시테이블에
data는 리스트 d에 저장

key가 이미 해시테이블에
있으므로 data만 갱신

다음 원소 검사를 위해

다음 위치가 초기 위치와 같으면
루프 벗어나기 [저장 실패]

```
27      def get(self, key): # 탐색 연산
28          initial_position = self.hash(key)
29          i = initial_position
30          j = 1
31          while self.a[i] != None:
32              if self.a[i] == key:
33                  return self.d[i]
34              i = (initial_position + j) % self.M
35              j += 1
36              if i == initial_position:
37                  return None
38          return None
39
40      def print_table(self):
41          for i in range(self.M):
42              print('{:4}'.format(str(i)), ' ', end='')
43          print()
44          for i in range(self.M):
45              print('{:4}'.format(str(self.a[i])), ' ', end='')
46          print()
```

초기 위치

탐색 성공

다음 원소 검사를 위해

탐색 실패

해시테이블 출력

[프로그램 6-1] linear_prob.py

```
01 from linearprob import LinearProbing
02 if __name__ == '__main__':
03     t = LinearProbing(13)
04     t.put(25, 'grape')
05     t.put(37, 'apple')
06     t.put(18, 'banana')
07     t.put(55, 'cherry')
08     t.put(22, 'mango')
09     t.put(35, 'lime')
10     t.put(50, 'orange')
11     t.put(63, 'watermelon')
12     print('탐색 결과:')
13     print('50의 data = ', t.get(50))
14     print('63의 data = ', t.get(63))
15     print('해시테이블:')
16     t.print_table()
```

해시테이블 크기가 13인 객체 생성

8개의 항목 삽입

탐색과 테이블 출력

[프로그램 6-2] main.py

Console PyUnit

<terminated> main.py [C:₩Users₩sbyang₩AppData₩Local₩Programs₩Python₩Python36-32₩python.exe]

```
탐색 결과:
50의 data =  orange
63의 data =  watermelon
해시테이블:
0     1     2     3     4     5     6     7     8     9     10    11    12
50    63    None  55    None  18    None  None  None  22    35    37    25
```

이차조사

- 이차조사(Quadratic Probing)는 선형조사와 근본적으로 동일한 충돌해결 방법

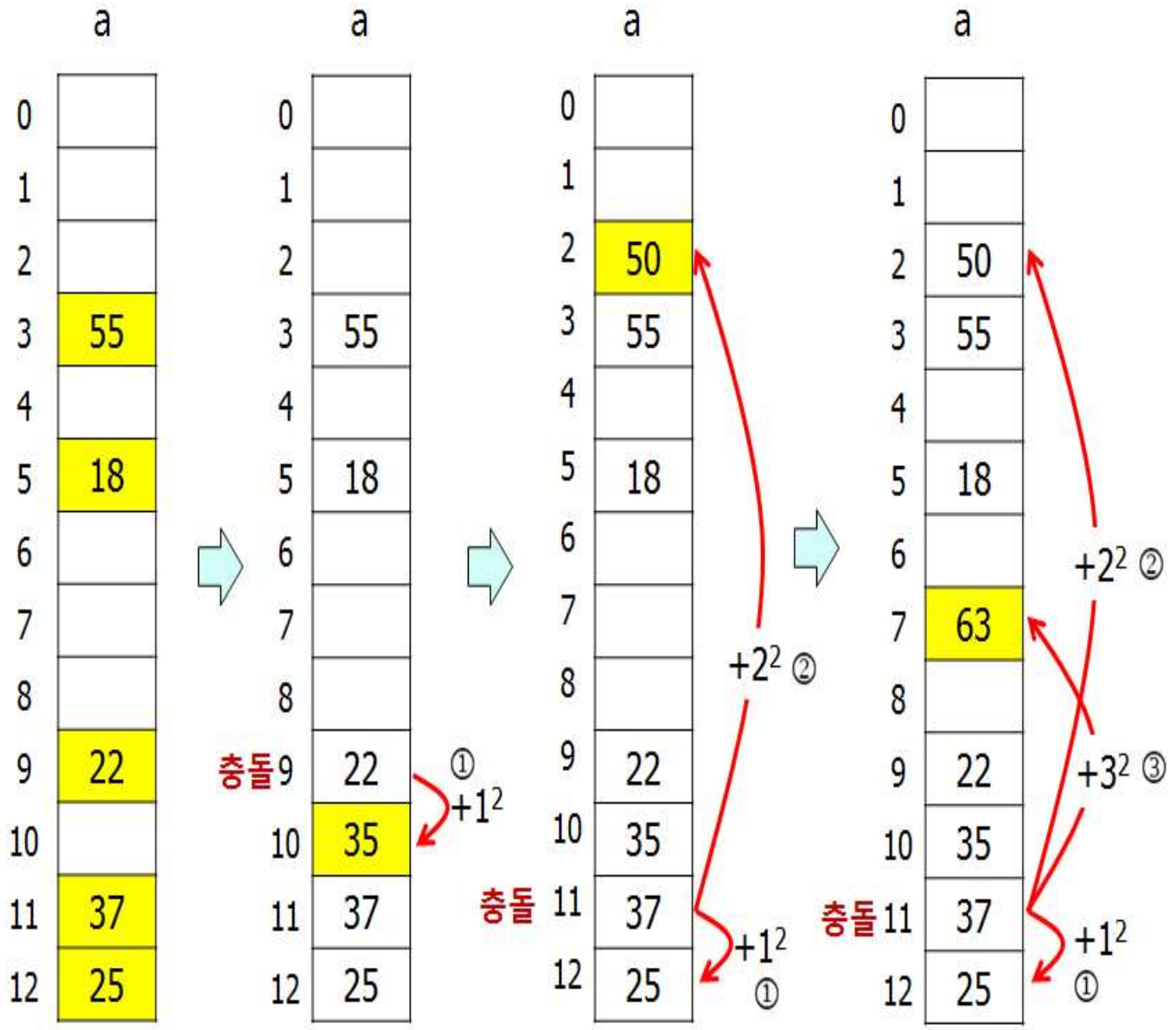
- 충돌 후 배열a에서

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

으로 선형조사보다 더 멀리 떨어진 곳에서 empty 원소를 찾음

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



- 이차조사는 이웃하는 빈 곳이 채워져 만들어지는 1차 군집화 문제를 해결하지만,

- 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 군집화(Secondary Clustering)를 야기

- 점프 크기가 제곱 만큼씩 커지므로 배열에 empty 원소가 있는데도 empty 원소를 건너뛰어 탐색에 실패하는 경우도 피할 수 없음

```
01 class QuadProbing:
02     def __init__(self, size):
03         self.M = size
04         self.a = [None] * size
05         self.d = [None] * size
06         self.N = 0
07
08     def hash(self, key):
09         return key % self.M
10
```

객체 생성자
테이블 크기 M
해시테이블 a
데이터 저장용 d
저장된 항목 수 N

나눗셈 해시함수

```
11      def put(self, key, data):
12          initial_position = self.hash(key)
13          i = initial_position
14          j = 0
15          while True:
16              if self.a[i] == None:
17                  self.a[i] = key
18                  self.d[i] = data
19                  self.N += 1
20                  return
21              if self.a[i] == key:
22                  self.d[i] = data
23                  return
24              j += 1
25              i = (initial_position + j*j) % self.M
26              if self.N > self.M:
27                  break
28
```

초기 위치

빈 곳 발견

key는 해시테이블에
data는 리스트 d에 저장

key가 이미 해시테이블에
있으므로 data만 갱신

다음 원소 검사를 위해

저장된 항목 수가 테이블
크기보다 크면 [저장 실패]

```
29      def get(self, key): # 탐색 연산
30          initial_position = self.hash(key)
31          i = initial_position
32          j = 1
33          while self.a[i] != None:
34              if self.a[i] == key:
35                  return self.d[i]
36              i = (initial_position + j*j) % self.M
37              j += 1
38          return None # 탐색 실패
```

초기 위치

탐색 성공

다음 원소 검사를 위해

[프로그램 6-3] quad_prob.py

```python
from quad_prob import QuadProbing
if __name__ == '__main__':
    t = QuadProbing(13)
    t.put(25, 'grape')
    t.put(37, 'apple')
    t.put(18, 'banana')
    t.put(55, 'cherry')
    t.put(22, 'mango')
    t.put(35, 'lime')
    t.put(50, 'orange')
    t.put(63, 'watermelon')

    print('탐색 결과:')
    print('50의 data = ', t.get(50))
    print('63의 data = ', t.get(63))
    print('해시테이블:')
    t.print_table()
```

해시테이블 크기가 13인 객체 생성

8개의 항목 삽입

탐색과 테이블 출력

[프로그램 6-4] main.py

```
Console  PyUnit
<terminated> main.py [C:₩Users₩sbyang₩AppData₩Local₩Programs₩Python₩Python36-32₩python.exe]
탐색 결과:
50의 data =  orange
63의 data =  watermelon
해시테이블:
0     1     2     3     4     5     6     7     8     9     10    11    12
None  None  50    55    None  18    None  63    None  22    35    37    25
```

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 <u>모든 군집화 문제를 해결</u>

- 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
- 하지만 해시테이블 크기 M을 소수로 선택하면, 이 제약 조건을 만족

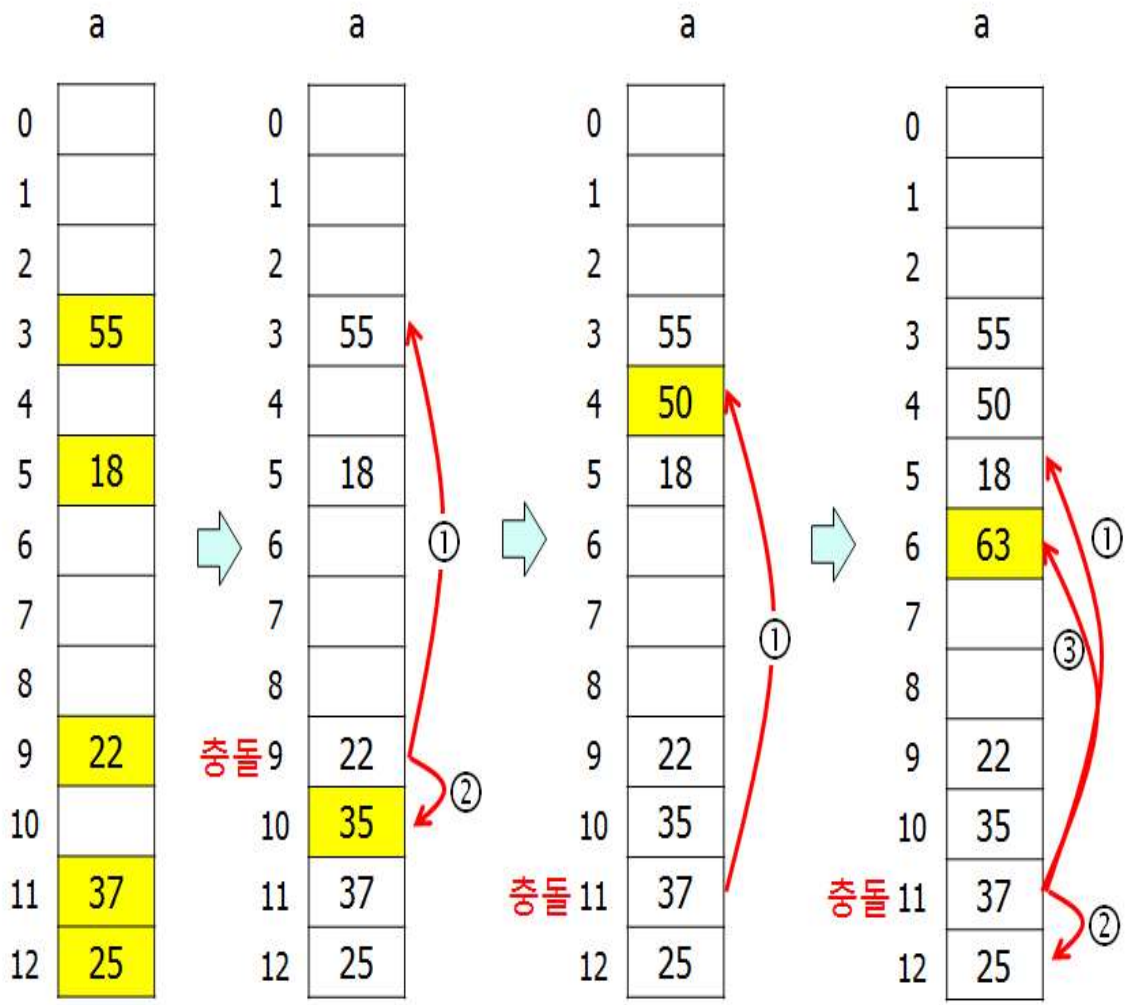
- h(key) = key % 13과 d(key) = 7-(key % 7) 에 따라, 25, 37, 18, 55, 22, 35, 50, 63을 해시테이블에 차례로 저장하는 과정

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 class DoubleHashing:
02     def __init__(self, size):
03         self.M = size
04         self.a = [None] * size
05         self.d = [None] * size
06         self.N = 0  # 항목 수
07 
08     def hash(self, key):
09         return key % self.M
10 
```

```
11      def put(self, key, data): # 삽입 연산
12          initial_position = self.hash(key)
13          i = initial_position
14          d = 7 - (key % 7)
15          j = 0
16          while True:
17              if self.a[i] == None:
18                  self.a[i] = key
19                  self.d[i] = data
20                  self.N += 1
21                  return
22              if self.a[i] == key:
23                  self.d[i] = data
24                  return
25              j += 1
26              i = (initial_position + j*d) % self.M
27              if self.N > self.M:
28                  break
29
```

```
30      def get(self, key): # 탐색 연산
31          initial_position = self.hash(key)
32          i = initial_position
33          d = 7 - (key % 7)
34          j = 0
35          while self.a[i] != None:
36              if self.a[i] == key:
37                  return self.d[i]
38              j += 1
39              i = (initial_position + j*d) % self.M
40          return None
```

Console PyUnit

<terminated> main.py [C:₩Users₩sbyang₩AppData₩Local₩Programs₩Python₩Python36-32₩python.exe]

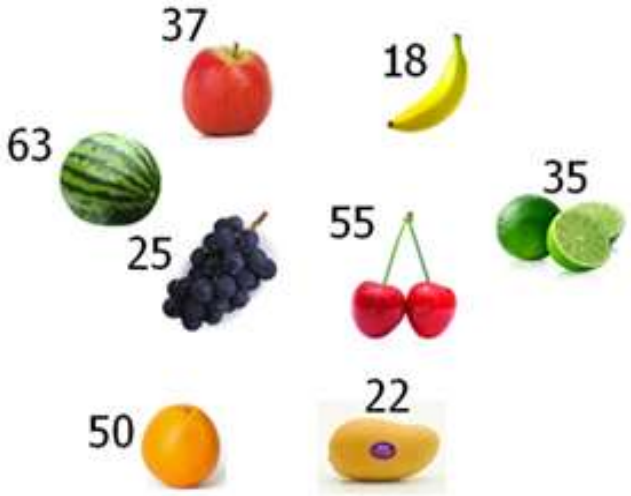
```
탐색 결과:
50의 data =  orange
63의 data =  watermelon
해시테이블:
0     1     2     3     4     5     6     7     8     9     10    11    12
None  None  None  55    50    18    63    None  None  22    35    37    25
```

이중해싱의 장점

- 이중해싱은 빈 곳을 찾기 위한 점프 시퀀스가 일정하지 않으며, 모든 군집화 현상을 발생시키지 않는다.
- 또한 해시 성능을 저하시키지 않는 동시에 해시테이블에 많은 키들을 저장할 수 있다는 장점을 가지고 있다.

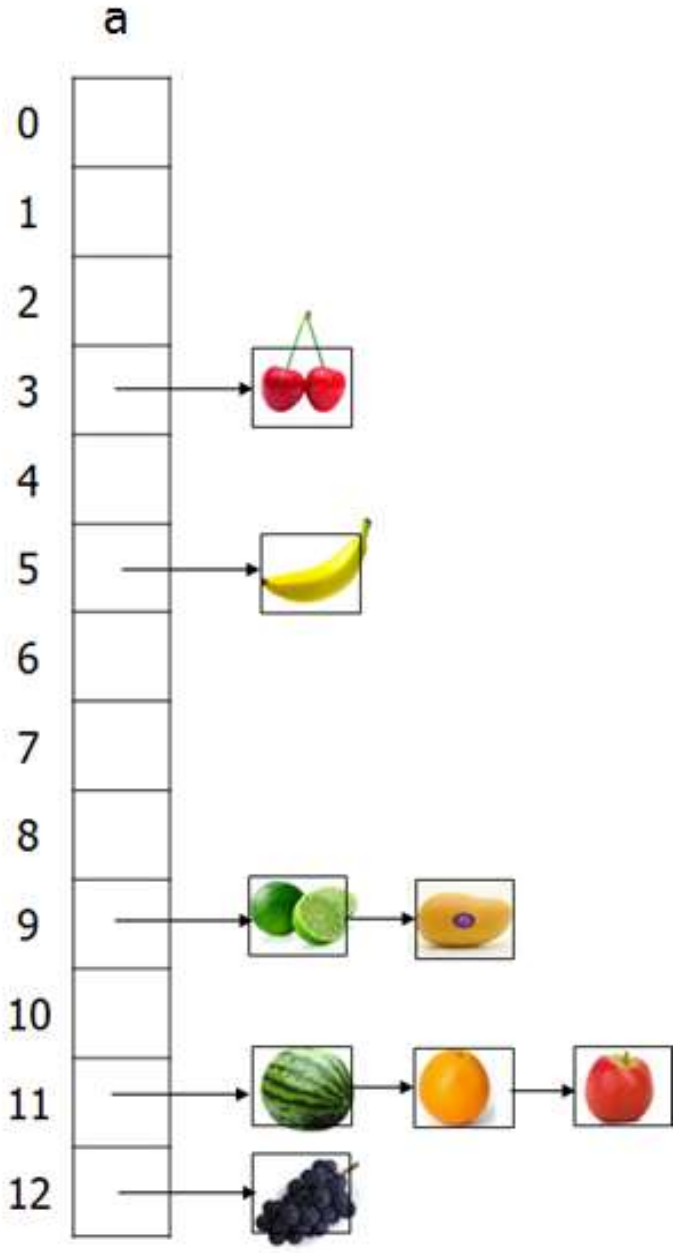
폐쇄주소방식

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



h(key) = key % 13

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



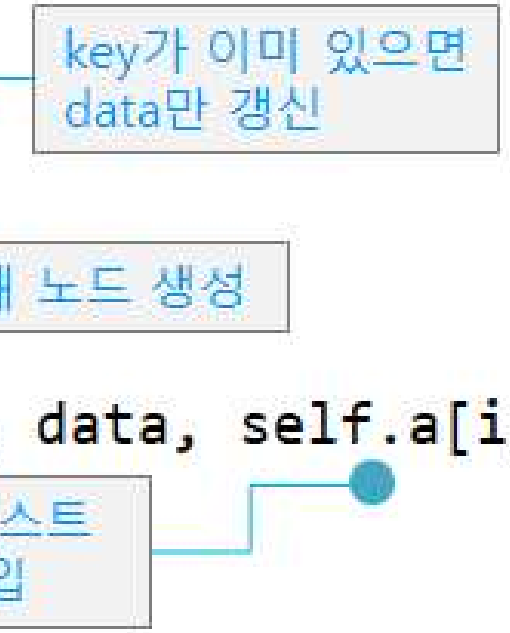
```
01 class Chaining:
02     class Node:
03         def __init__(self, key, data, link):
04             self.key   = key
05             self.data  = data
06             self.next  = link
07
08     def __init__(self, size):
09         self.M = size
10         self.a = [None] * size
11
12     def hash(self, key):
13         return key % self.M
14
```

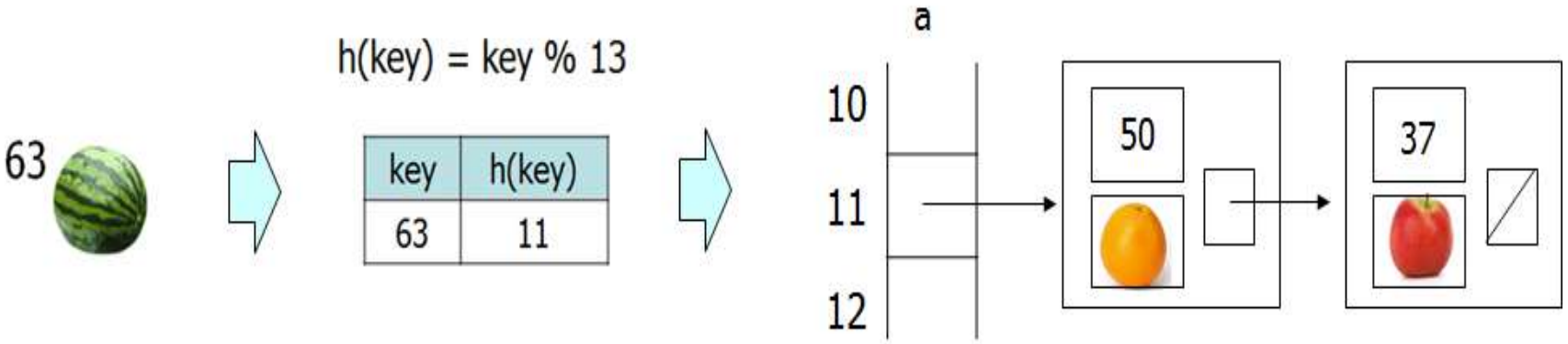
노드 객체 생성자
key, data, next

Chaining 객체 생성자
해시테이블 a

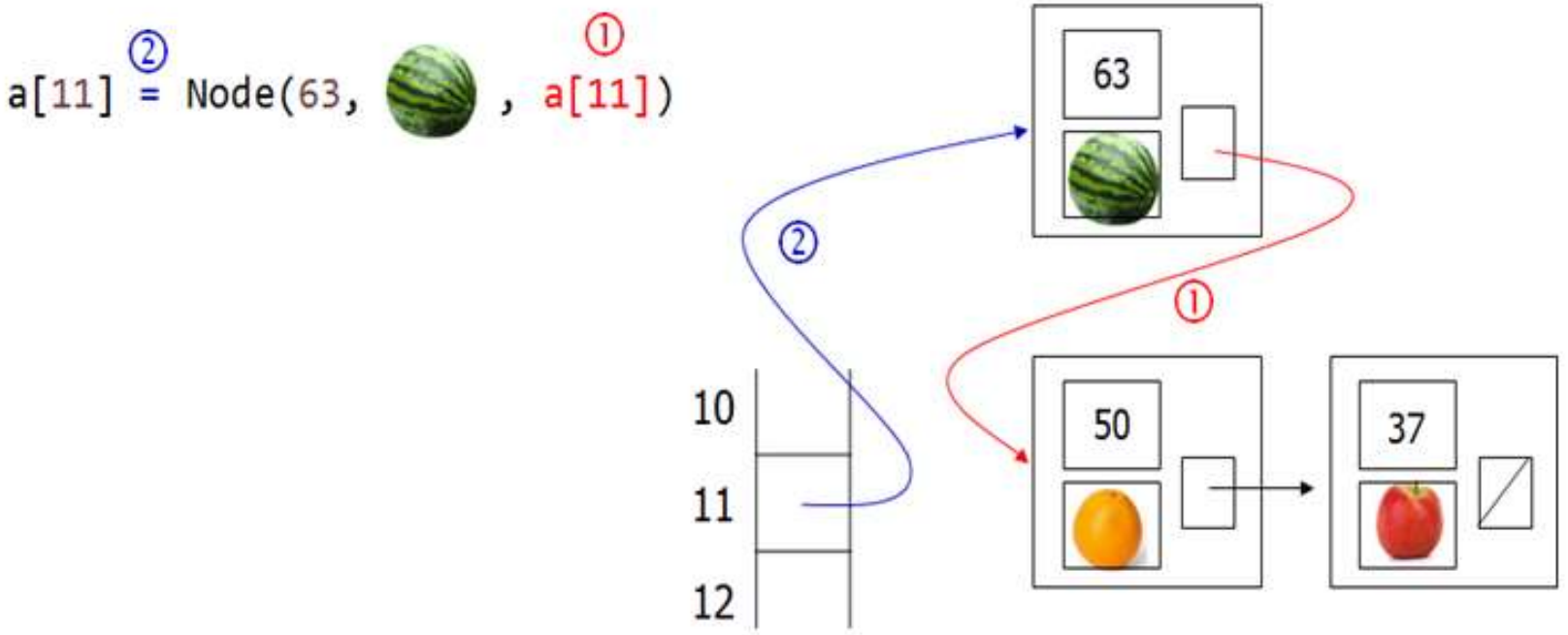
나눗셈 해시함수

```
15      def put(self, key, data): # 삽입 연산
16          i = self.hash(key)
17          p = self.a[i]
18          while p != None:
19              if key == p.key:
20                  p.data = data
21                  return
22              p = p.next
23          self.a[i] = self.Node(key, data, self.a[i])
24
```





63을 삽입하기 전



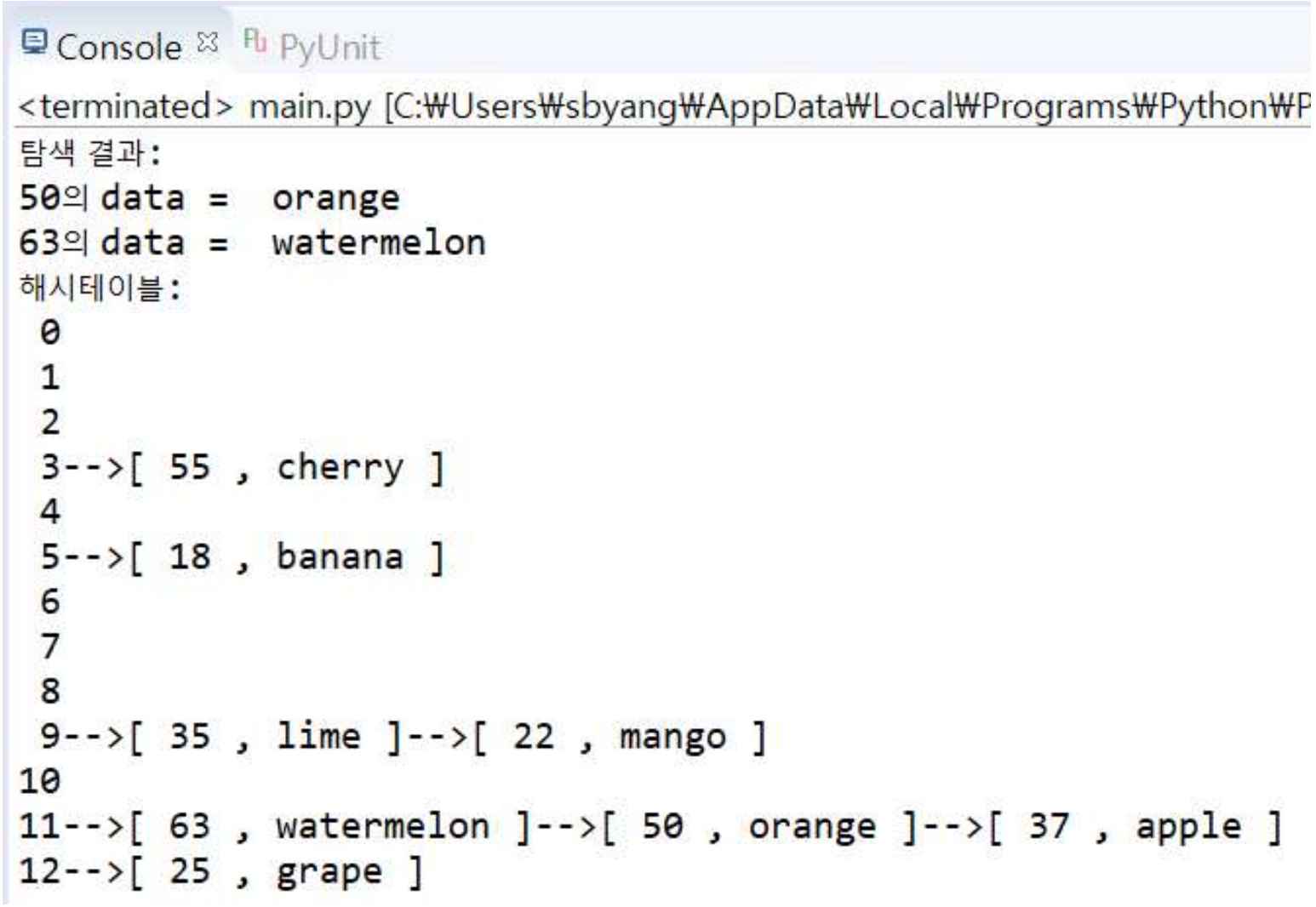
63을 삽입한 후

```
25      def get(self, key): # 탐색 연산
26          i = self.hash(key)
27          p = self.a[i]
28          while p != None:
29              if key == p.key:
30                  return p.data
31              p = p.next
32          return None
33
34      def print_table(self): # 테이블 출력
35          for i in range(self.M):
36              print('%2d' % (i), end='')
37              p = self.a[i];
38              while p != None:
39                  print('-->[', p.key,',', p.data, ']', end='')
40                  p = p.next
41              print()
```

탐색 성공

탐색 실패

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



재해시(Rehash)

- 어떤 해싱방법도 해시테이블에 비어있는 원소가 적으면, 삽입에 실패하거나 해시 성능이 급격히 저하되는 현상을 피할 수 없음

- 이러한 경우, 해시테이블을 확장시키고 새로운 해시함수를 사용하여 모든 키들을 새로운 해시테이블에 다시 저장하는 재해시가 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 해시 테이블 크기를 2 배로 늘리고, $\alpha < 0.25$가 되면 해시테이블을 1/2로 줄임

수고하셨습니다