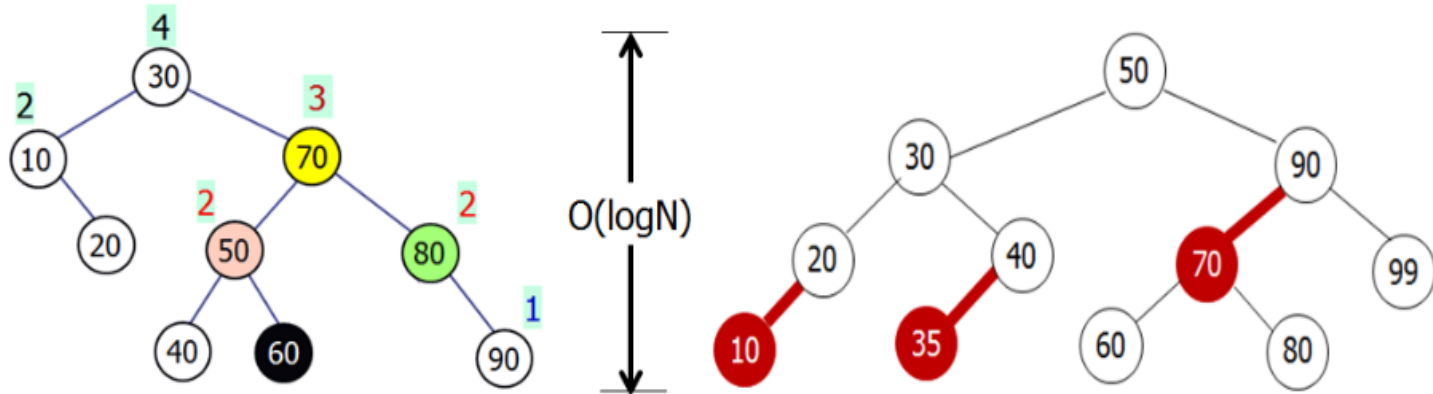


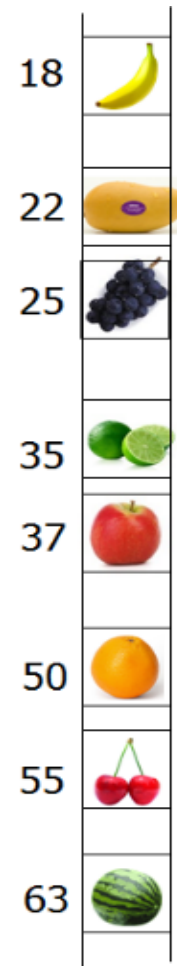
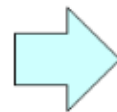
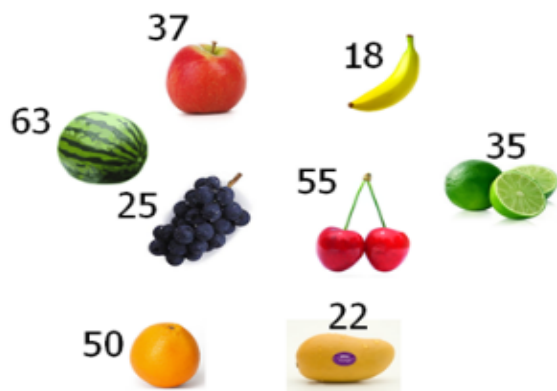
제 6 장 해시 테이블

6.1 해시테이블

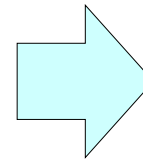
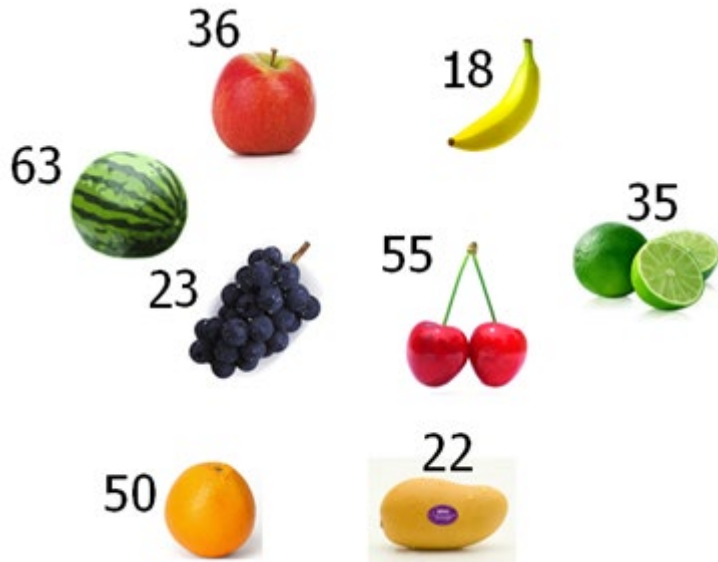
- 이진탐색트리의 성능을 개선한 AVL 트리와 레드블랙트리의 삽입과 삭제 연산의 수행시간은 각각 $O(\log N)$











- 그렇다면 $O(\log N)$ 보다 좋은 성능을 갖는 자료구조는 있을까?



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

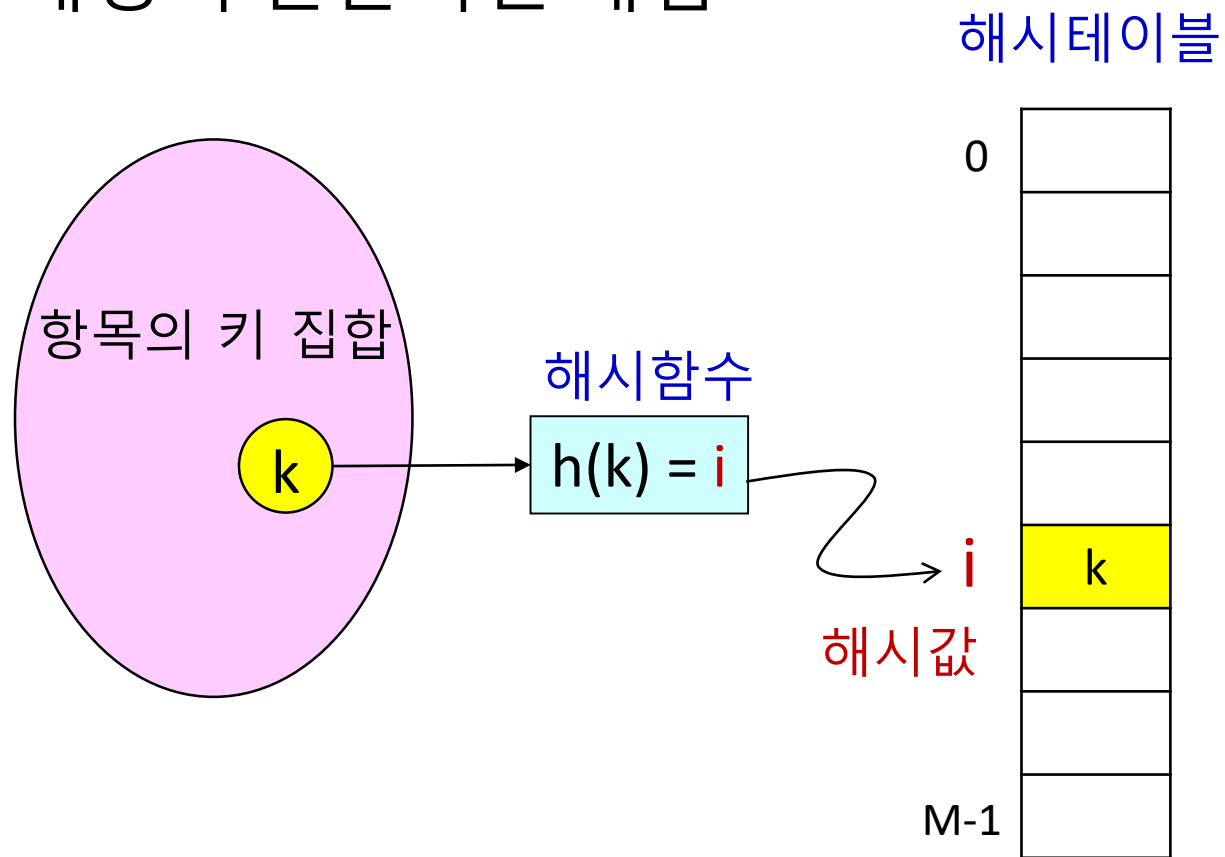


18	
22	
23	
35	
36	
50	
55	
63	

[그림 6-2] 키를 그대로 1차원 리스트의 인덱스로 사용

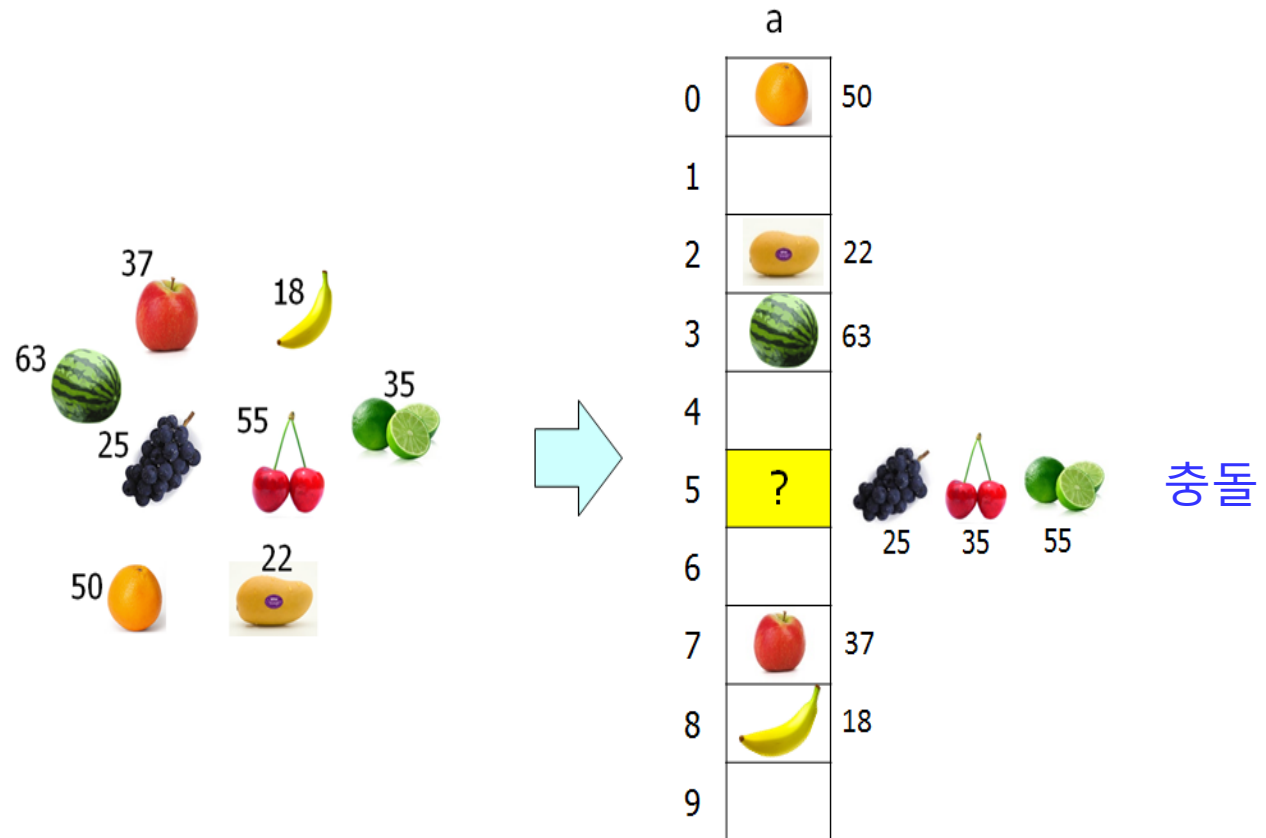
- 그러나 키를 배열의 인덱스로 그대로 사용하면 메모리 낭비가 심해질 수 있음
- [문제 해결 방안] 키를 변환하여 배열의 인덱스로 사용
- 키를 간단한 함수를 사용해 변환한 값을 배열의 인덱스로 이용하여 항목을 저장하는 것을 **해싱(Hashing)**이라고 함
- 해싱에 사용되는 함수를 **해시함수(Hash Function)**라 하고, 해시함수가 계산한 값을 **해시값(Hash value)** 또는 **해시주소**라고 하며, 항목이 해시값에 따라 저장되는 배열을 **해시테이블(Hash Table)**이라고 함

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



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

- 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 제공한 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 $1234 + 5678 + 9012 = 15924$ 를 계산한 후에 해시테이블의 크기가 3이라면 15924에서 3자리 수만을 해시값으로 사용

- 곱셈(Multiplicative) 함수: 1보다 작은 실수 δ 를 키에 곱하여 얻은 숫자의 소수 부분을 테이블 크기 M 과 곱한다. 이렇게 나온 값의 정수 부분을 해시값으로 사용
 - $h(\text{key}) = ((\text{key} * \delta) \% 1) * M$ 이다. Knuth에 의하면 $\delta = \frac{\sqrt{5}-1}{2} \approx 0.61803$ 이 좋은 성능을 보인다.
 - 예를 들면, 테이블 크기 $M = 127$ 이고 키가 123456789인 경우, $123456789 \times 0.61803 = 76299999.30567$, $0.30567 \times 127 = 38.82009$ 이므로 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
 - $h(key) = key \% M$ 이고, 따라서 해시테이블의 인덱스는 0에서 M-1이 됨
 - 여기서 제수로 소수를 사용하는 이유는 나눗셈 연산을 했을 때, 소수가 키들을 균등하게 인덱스로 변환시키는 성질을 갖기 때문

6.3 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)
 - 랜덤조사(Random Probing)
 - 이중해싱(Double Hashing)

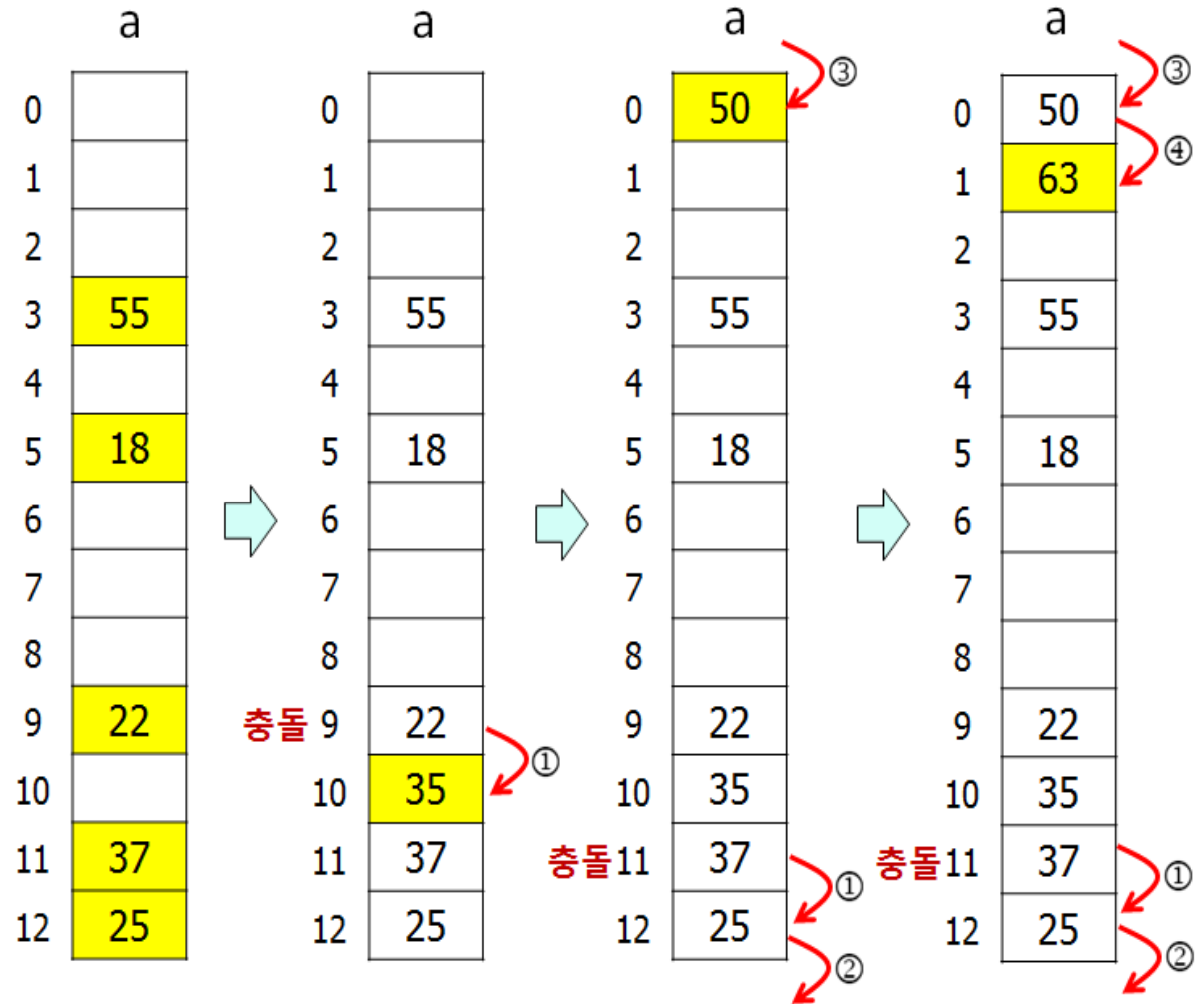
6.3.1 선형조사

- 선형조사는 충돌이 일어난 원소에서부터 순차적으로 검색하여 처음 발견한 empty 원소에 충돌이 일어난 키를 저장
- $h(\text{key}) = i$ 라면, 해시테이블 $a[i], a[i+1], a[i+2], \dots, a[i+j]$ 를 차례로 검색하여 처음으로 찾아낸 empty 원소에 key를 저장
- 해시테이블은 1차원 리스트이므로, $i + j$ 가 M 이 되면 $a[0]$ 을 검색

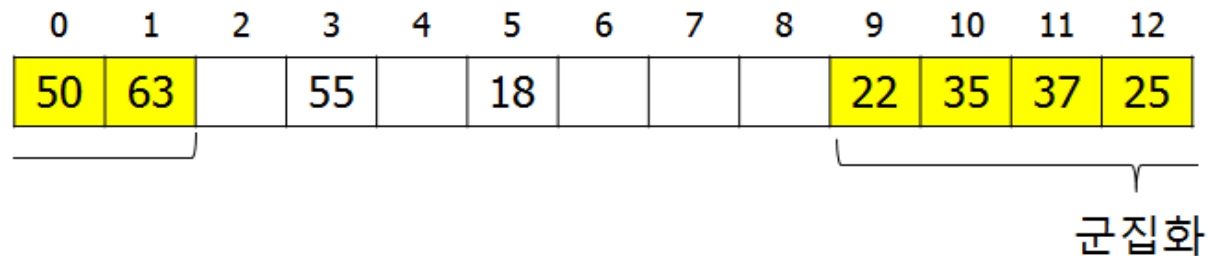
$$(h(\text{key}) + j) \% M, j = 0, 1, 2, 3, \dots$$

선형조사방식의 키 저장 과정

key	$h(\text{key}) = \text{key} \% 13$
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 해시성능을 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
```

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) ● 초기 위치
29     i = initial_position
30     j = 1
31     while self.a[i] != None:
32         if self.a[i] == key:
33             return self.d[i] ● 탐색 성공
34         i = (initial_position + j) % self.M ●
35         j += 1
36         if i == initial_position:
37             return None ●
38     return None ● 탐색 실패
39
40 def print_table(self): ● 해시테이블 출력
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()

```

```

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')

```

해시테이블 크기가 13인
객체 생성

8
개
의
항
목
삽
입

12
13
14
15
16

```

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

```

탐
색
과
테
이
블
출
력

[프로그램 6-2] main.py

Console PyUnit

<terminated> main.py [C:\Users\wsbyang\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

6.3.2 이차조사

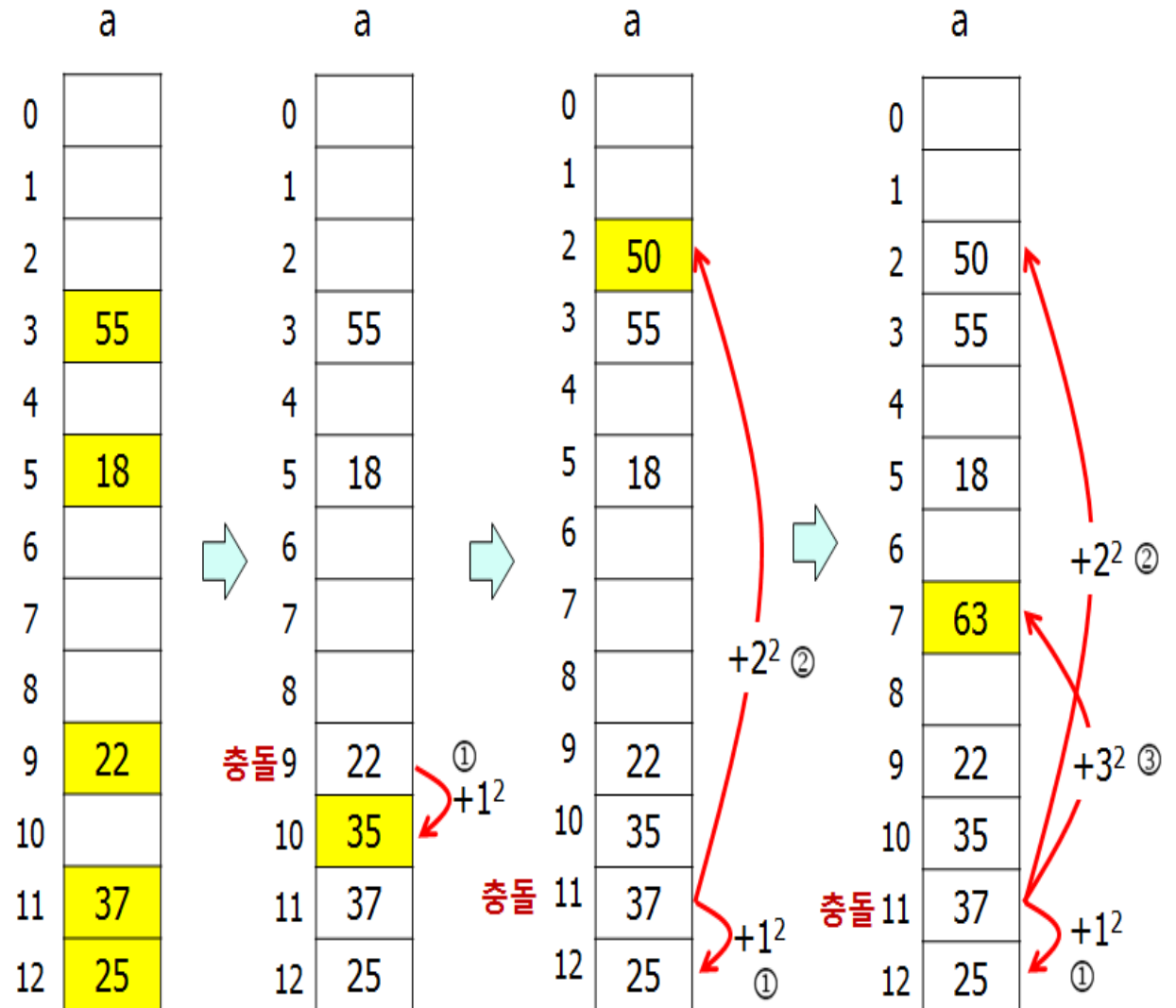
- 이차조사(Quadratic Probing)는 선형조사와 근본적으로 동일한 충돌해결 방법
- 충돌 후 배열 a 에서

$$(h(\text{key}) + j^2) \% M, j = 0, 1, 2, 3, \dots$$

으로 선형조사보다 더 멀리 떨어진 곳에서 empty 원소를 찾음

이차조사방식의 키 저장 과정

key	$h(\text{key}) = \text{key} \% 13$
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

```

01 from quad_prob import QuadProbing
02 if __name__ == '__main__':
03     t = QuadProbing(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')

```

해시테이블 크기가 13인
객체 생성

8
개
의
항
목
삽
입

```

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()

```

탐
색
과
테
이
블
출
력

[프로그램 6-4] main.py

Console x PyUnit

<terminated> main.py [C:\Users\wsbyang\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

6.3.3 랜덤조사

- 랜덤조사(Random Probing)는 선형조사와 이차조사의 규칙적인 점프 시퀀스와는 달리 점프 시퀀스를 무작위화하여 empty 원소를 찾는 충돌해결방법
- 랜덤조사는 의사 난수 생성기를 사용하여 다음 위치를 찾음
- 랜덤조사 방식도 동의어들이 똑같은 점프 시퀀스에 따라 empty 원소를 찾아 키를 저장하게 되고, 이 때문에 3차 군집화(Tertiary Clustering)가 발생

```
01 import random
02 class RandProbing:
03 :
12     def put(self, key, data): # 삽입 연산
13         initial_position = self.hash(key)
14         i = initial_position
15         random.seed(1000)
16         while True:
17             if self.a[i] == None:
22                 if self.a[i] == key:
25                     j = random.randint(1, 99)
26                     i = (initial_position + j) % self.M
27                     if self.N > self.M:
28                         break
30     def get(self, key): # 탐색 연산
31 :
33         random.seed(1000)
34         while self.a[i] != None:
37             i = (initial_position + random.randint(1, 99)) % self.M
38         return None
```

random 패키지 불러오기

초기 위치

난수 생성 초기값


난수 크기 범위 지정

다음 원소 검사를 위해

저장할 때와 동일한 난수 생성 초기값

난수 크기 동일한 범위 지정

- `random.seed(1000)`에서 초기값 1000은 임의로 정한 것
- `random.randint(1, 99)`는 1에서 99 사이에서 난수를 생성

Console  PyUnit

<terminated> main.py [C:\Users\wsbyang\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	50	None	55	35	18	63	None	None	22	None	37	25

6.3.4 이중해싱

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

$$(h(\text{key}) + j \cdot d(\text{key})) \bmod M, j = 0, 1, 2, \dots$$

- 이중해싱은 동의어들이 저마다 제2 해시함수를 갖기 때문에 점프 시퀀스가 일정하지 않음
- 따라서 이중해싱은 모든 군집화 문제를 해결

- 제 2의 함수 $d(\text{key})$ 는 점프 크기를 정하는 함수이므로 0을 리턴해선 안됨
- 그 외의 조건으로 $d(\text{key})$ 의 값과 해시테이블의 크기 M 과 서로소(Relatively Prime) 관계일 때 좋은 성능을 보임
- 하지만 해시테이블 크기 M 을 소수로 선택하면, 이 제약 조건을 만족

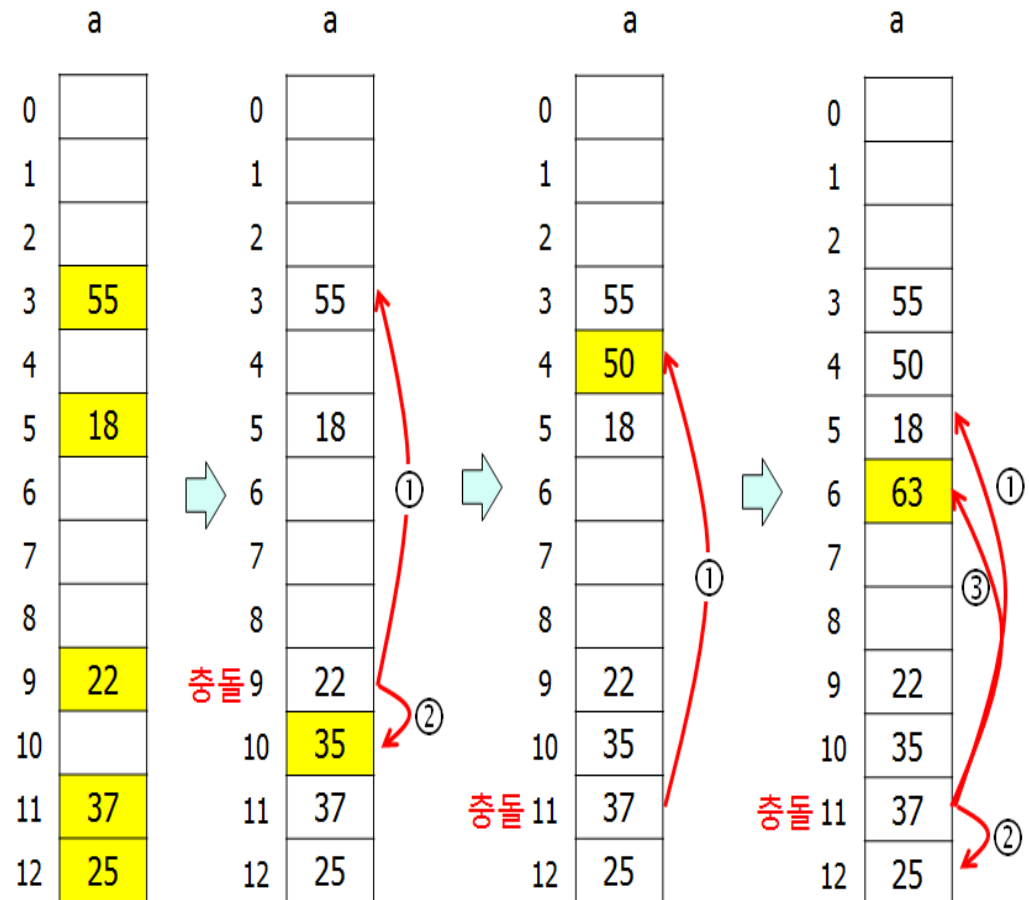
- $h(\text{key}) = \text{key} \% 13$ 과 $d(\text{key}) = 7 - (\text{key} \% 7)$ 에 따라, 25, 37, 18, 55, 22, 35, 50, 63을 해시테이블에 차례로 저장하는 과정

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	7	①	②	③
35	9		3	10	
50	11		4		
63	11	7	5	12	6

$$h(\text{key}) = \text{key} \% 13$$

$$d(\text{key}) = 7 - (\text{key} \% 7)$$

$$(h(\text{key}) + j * d(\text{key})) \% 13, j = 0, 1, \dots$$




```
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\wsbyang\AppData\Local\Programs\Python\Python36-32\python.exe]

탐색 결과:

50의 data = orange

63의 data = watermelon

해시테이블:

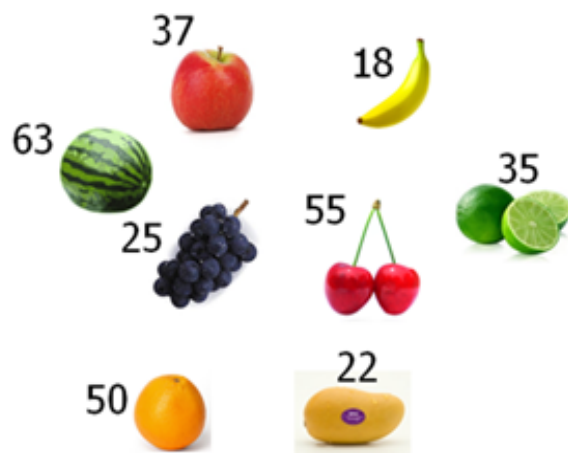
0	1	2	3	4	5	6	7	8	9	10	11	12
None	None	None	55	50	18	63	None	None	22	35	37	25

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.

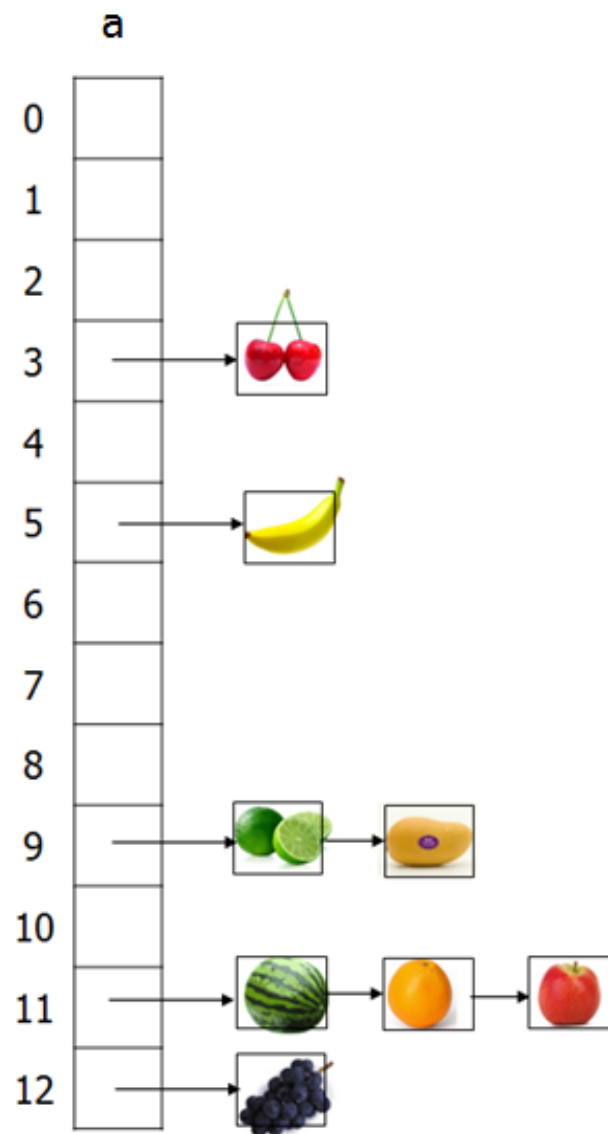
6.4 폐쇄주소방식

- 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)



$$h(\text{key}) = \text{key} \% 13$$

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
```

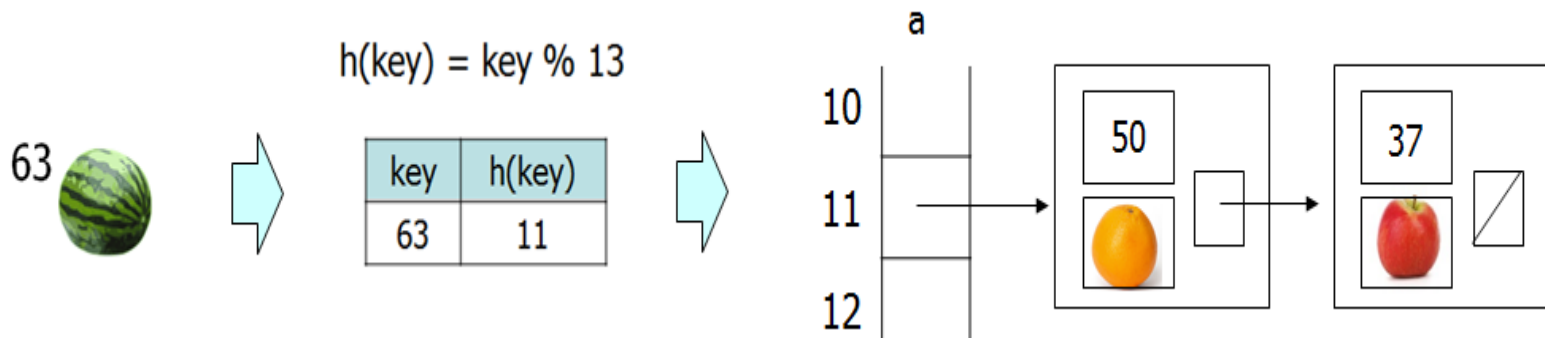
key가 이미 있으면
data만 갱신

새 노드 생성

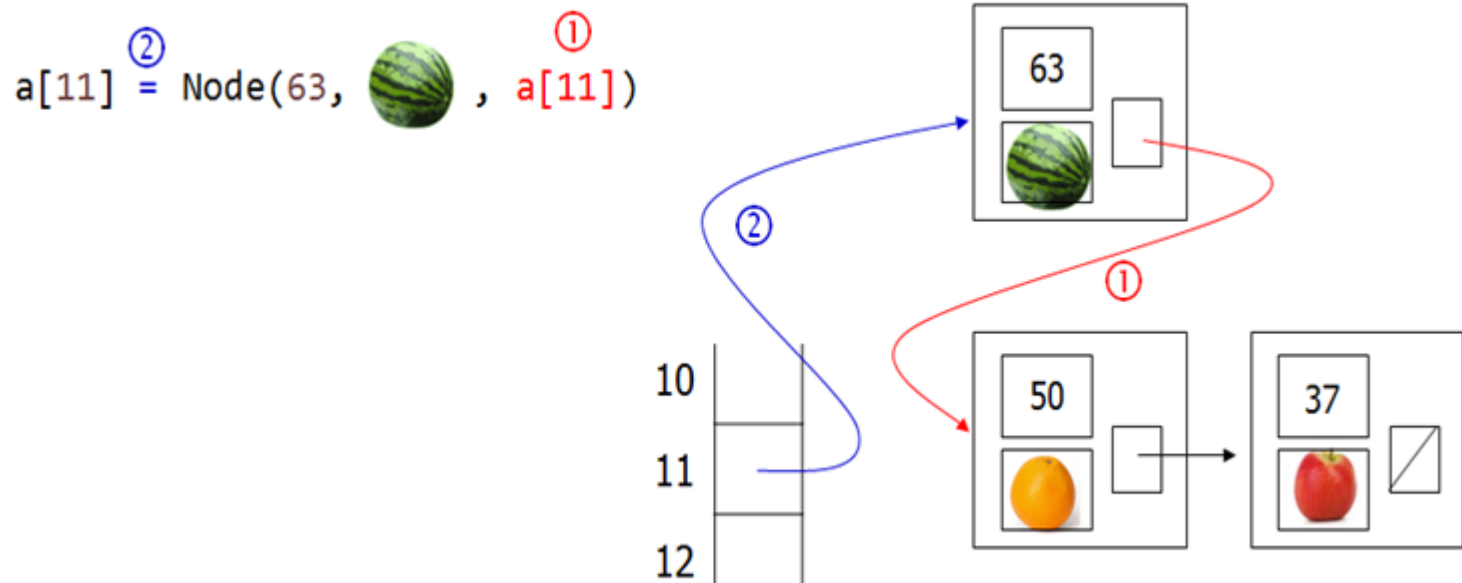
단순연결리스트
맨 앞에 삽입

```
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()
```

The diagram illustrates the execution flow of the `get` method. A blue dot on line 29, following the condition `if key == p.key:`, is connected by a line to a box labeled "탐색 성공" (Search Success). Another blue dot on line 32, following the `return None` statement, is connected by a line to a box labeled "탐색 실패" (Search Failure).




63을 삽입하기 전



63을 삽입한 후

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

Console  PyUnit

<terminated> main.py [C:\Users\wsbyang\AppData\Local\Programs\Python\WP

탐색 결과:

50의 data = orange

63의 data = watermelon

해시테이블:

0

1

2

3-->[55 , cherry]

4

5-->[18 , banana]

6

7

8

9-->[35 , lime]-->[22 , mango]

10

11-->[63 , watermelon]-->[50 , orange]-->[37 , apple]

12-->[25 , grape]

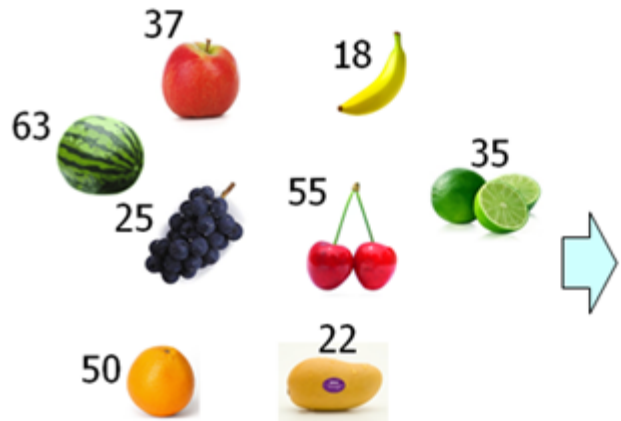
6.5 기타 해싱

- 2-방향 체이닝(Two-Way Chaining)
- 뱀꾸기 해싱(Cickoo Hashing)

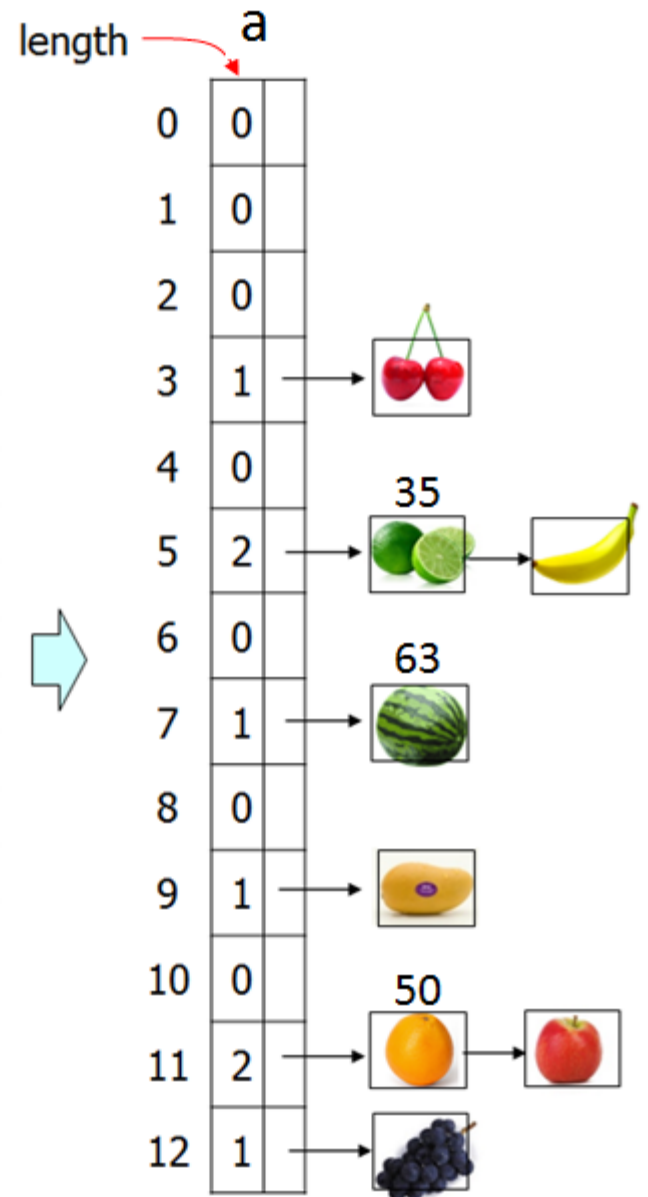
2-방향 체이닝

- 체이닝과 동일하나 2 개의 해시함수를 이용하여 연결리스트의 길이가 짧은 쪽에 새 키를 저장
- 해시테이블의 원소는 Node를 가리키는 래퍼런스 이외에도 연결리스트의 길이(length)를 가짐
- 다음 슬라이드의 그림은 2 개의 해시함수 $h(key)$ 와 $d(key)$ 가 이미 계산되어 있다고 가정한 후, 25, 37, 18, 55, 22, 35, 50, 63을 차례로 저장한 결과

2-방향 체이닝



key	h(key)	d(key)
25	12	
37	11	
18	5	
55	3	
22	9	
35	9	5
50	11	5
63	11	7



- 25, 37, 18, 55, 22까지는 충돌 없이 저장되나, 35를 저장할 때에는 $h(35) = 9$, $d(35) = 5$ 이고 $a[9]$ 와 $a[5]$ 의 연결리스트의 길이가 같으므로, 임의로 $a[5]$ 의 연결리스트에 35를 저장
- 50을 저장할 때는 $h(50) = 11$, $d(50) = 5$ 이므로 $a[11]$ 과 $a[5]$ 의 리스트의 길이를 비교하여 $a[11]$ 의 리스트가 더 짧으므로 $a[11]$ 의 리스트에 50을 저장
- 63을 저장할 때는 $a[11]$ 와 $a[7]$ 의 리스트의 길이를 비교하여 $a[7]$ 의 리스트가 짧으므로 $a[7]$ 의 리스트에 63을 저장
- 새로운 키가 삽입되면 해당 리스트의 길이를 1 증가

- 2-방향 체이닝은 두 개의 해시함수를 계산해야 하고, 연결리스트의 길이를 비교해야 하며, 추후에 탐색을 위해선 두 연결리스트를 탐색해야 하는 경우도 발생
- 연구 결과에 따르면, 연결리스트의 평균 길이는 $O(\log \log N)$ 으로 매우 짧아서 실제로 매우 좋은 성능을 보임

빼꾸기 해싱



- 빼꾸기 해싱(Cuckoo Hashing)은 빼꾸기가 다른 새의 둥지에 알을 낳고, 부화된 빼꾸기 새끼가 다른 새의 알이나 새끼들을 둥지에서 밀어내는 습성을 모방한 해싱 방법
- 빼꾸기해싱은 2 개의 해시함수와 2 개의 해시테이블을 가지고 키들을 다음의 알고리즘에 따라 저장
- 해시함수 $h(key)$ 는 $htable$ 을 위한 것이고, 해시함수 $d(key)$ 는 $dtable$ 을 위한 것

새 키 저장 알고리즘

[1] $key = new_key$

[2] $h(key) = i$ 를 계산하여, $htable[i]$ 에 key 를 저장

[3] **if** key 가 저장된 원소가 비어 있으면:

삽입을 종료

[4] **else:** // key 가 저장되면서 그 자리에 있던 키를 쫓아낸 경우
 key 때문에 쫓겨난 키를 old_key 라고 하자.

[5] **if** old_key 가 있었던 테이블이 $htable$ 이면:

$d(old_key)=j$ 를 계산하여, $dtable[j]$ 에 old_key 를 저장

[6] **else:** // old_key 가 있었던 테이블이 $dtable$ 이면

$h(old_key)=j$ 를 계산하여, $htable[j]$ 에 old_key 를
저장

[7] $key = old_key$, go to step [3]

배꾸기해싱으로 10, 32, 45, 61을 차례로 삽입하는 과정

key	h(key)	d(key)
10	6	3
32	1	3
45	6	9
61	1	7

	htable	dtable
0		
1	32	
2		
3		
4		
5		
6	10	
7		
8		
9		



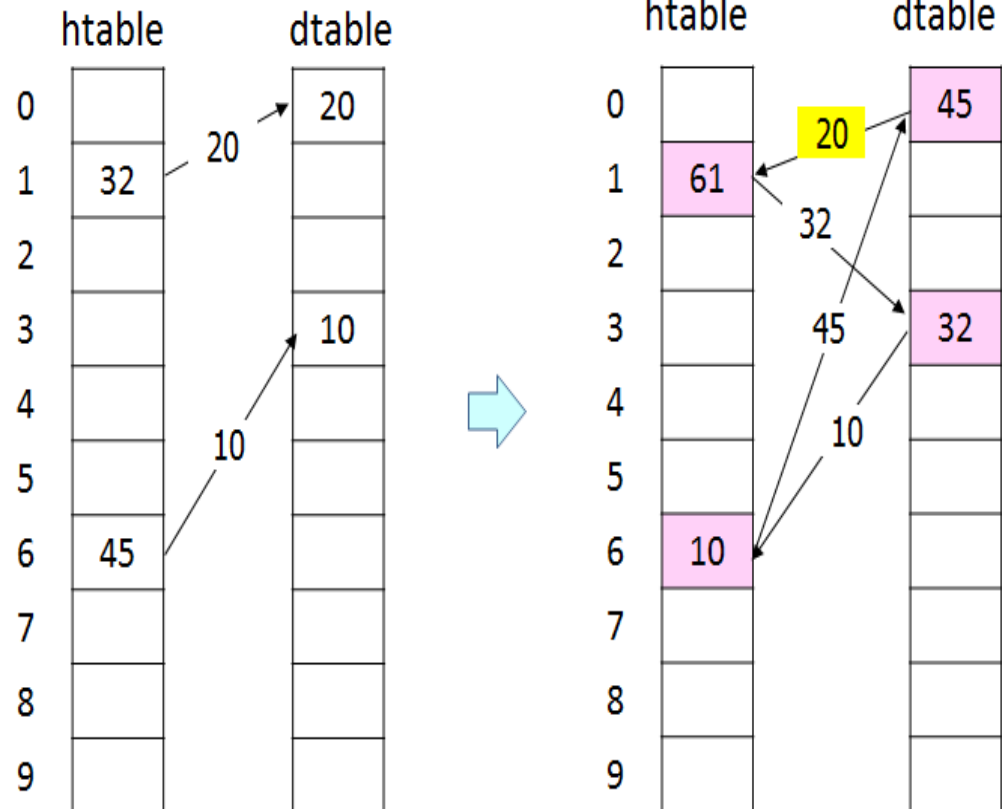
	htable	dtable
0		
1	32	
2		
3		10
4		
5		
6	45	
7		
8		
9		



	htable	dtable
0		
1	61	①
2		
3		32
4		②
5		
6	10	
7		③
8		
9		45

빠꾸기해싱의 삽입 과정 중 발생한 싸이클

key	h(key)	d(key)
10	6	3
20	1	0
45	6	0
32	1	3
61	1	3



- 삽입과정에서 싸이클이 발생할 경우, 빠꾸기해싱은 삽입에 실패한 것으로 간주하여 재해싱을 수행

- 뺨꾸기해싱의 장점은 탐색과 삭제를 $O(1)$ 시간에 보장한다는 것인데, 이런 장점을 갖는 해시함수는 아직 존재하지 않음
- 즉, 최대 2 회의 해시함수 계산으로 각각의 테이블 원소를 찾아 각 연산을 처리
- 단, 삽입은 높은 확률로 $O(1)$ 시간에 수행이 가능

재해시(Rehash)

- 어떤 해싱방법도 해시테이블에 비어있는 원소가 적으면, 삽입에 실패하거나 해시성능이 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 $\alpha = (\text{테이블에 저장된 키의 수 } N) / (\text{테이블 크기 } M)$
- 일반적으로 $\alpha > 0.75$ 가 되면 해시 테이블 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)

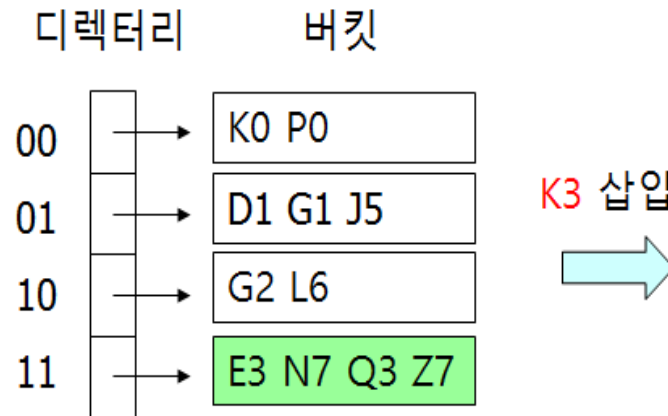
확장 해싱

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

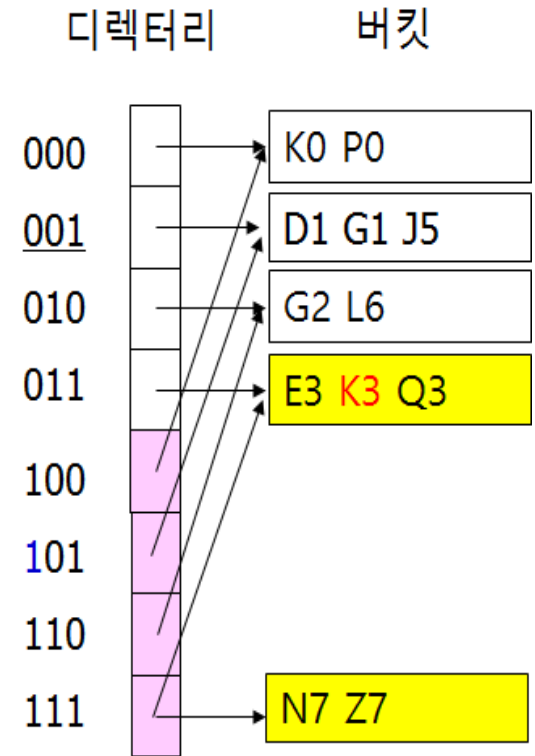
확장 해싱의 디렉터리 확장

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

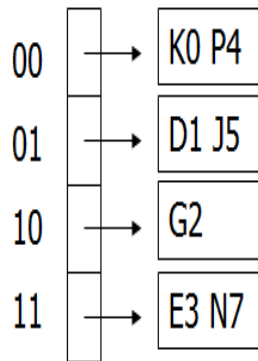
- (a)의 키 코드의 마지막 두 자리를 가지고 키들을 버킷에 저장한다.
- 이 때의 버킷 크기는 4이다. 즉, (b)에서 버킷 [E3, N7, Q3, Z7]은 꽉 차있는 상태
- K3을 삽입하면 K3의 코드의 마지막 2자리가 '11'이므로 [E3, N7, Q3, Z7] 버킷에 저장되어야 하지만 꽉 차있으므로, (c)와 같이 디렉터리를 2배로 확장한다.
- 이 때 코드의 마지막 세 자리를 가지고 키들을 탐색, 삽입, 삭제 연산을 수행

선형 해싱

- 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 추가되면 overflow 체인의 키들을 버킷으로 이동

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

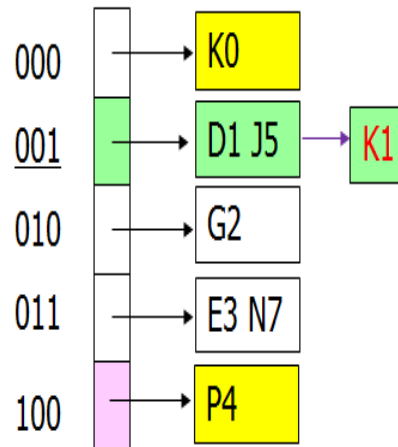
디렉터리 버킷



K1삽입



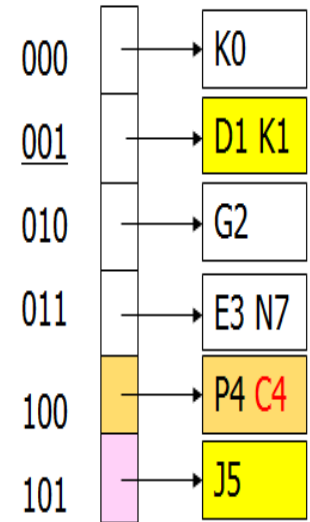
디렉터리 버킷



C4삽입



디렉터리 버킷



(a) 키 코드

(b) K1 삽입 전

(c) K1 삽입 후

(d) C4 삽입 후

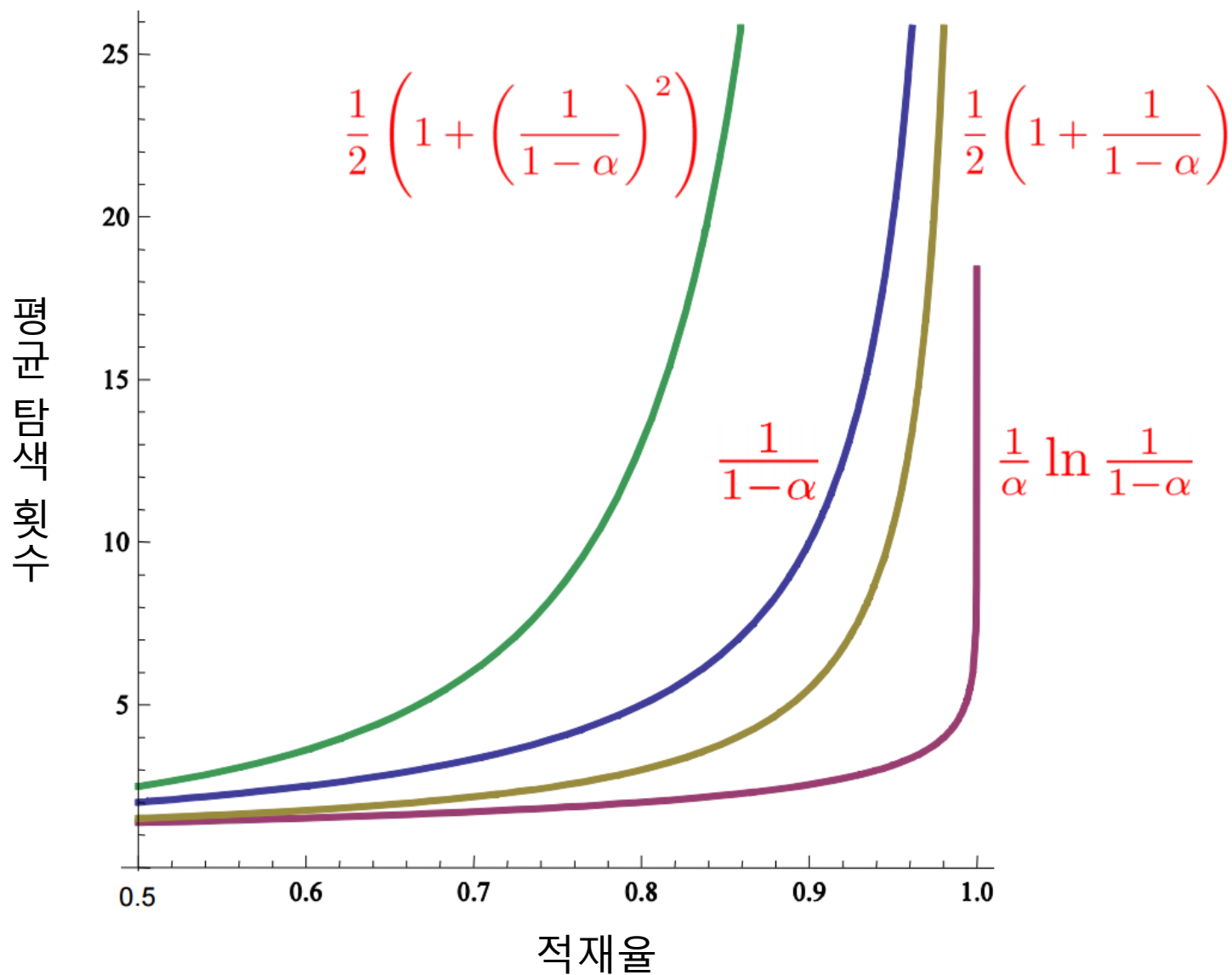
- 버킷 크기가 2이다.
- (b)는 (a)의 키 코드에 따라 마지막 두 자리를 이용하여 키들을 저장한 상태
- (c)는 K1을 삽입하려는 데 버킷 001 에 저장할 공간이 없어 overflow 체인에 임시로 K1을 저장한 경우

- 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 추가되었다.
- 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, **인터랙티브(Interactive) 응용에 적합**

6.6 해시방법의 성능 비교 및 응용

- 해시방법의 성능은 탐색이나 삽입 연산을 수행할 때 성공과 실패한 경우를 각각 분석하여 측정
- 선형조사는 적재율 α 가 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 α 가 너무 작으면 대부분의 연결리스트들이 empty 가 되고, α 가 너무 크면 연결리스트들의 길이가 너무 길어져 해시성능이 매우 저하됨
- 일반적으로 M 이 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 $O(1)$ 시간**에 지원하는 자료구조
- 해시 함수는 키들을 균등하게 해시테이블의 인덱스로 변환, 대표적인 해시 함수는 나눗셈 함수
- 충돌해결방법들은 크게 두가지로 분류: 개방주소방식, 폐쇄주소방식
- 개방주소방식: 선형조사, 이차조사, 랜덤조사, 이중해싱

- 폐쇄주소방식은 키에 대한 해시값에 대응되는 곳에만 키를 저장
- 체이닝은 해시 테이블 크기만큼의 연결리스트를 가지며, 키를 해시값에 대응되는 연결리스트에 저장
 - 군집화 현상이 발생하지 않으며, 구현이 간결하여 실제로 가장 많이 활용되는 해시방법
- 2-방향 체이닝은 2 개의 해시 함수를 이용하여 연결리스트의 길이가 짧은 쪽에 새 키를 저장

- 빼꾸기 해싱은 탐색과 삭제를 $O(1)$ 시간에 보장하는 매우 효율적인 해싱 방법
- 재해시는 삽입에 실패하거나 해시 성능이 급격히 저하되었을 때, 해시테이블의 크기를 확장하고 새로운 해시 함수를 사용해 모든 키들을 새로운 해시테이블에 저장
- 동적 해싱은 대용량의 데이터베이스를 위한 해시방법으로 재해시를 수행하지 않고 동적으로 해시테이블의 크기를 조절: 확장 해싱과 선형 해싱