# 해싱(Hashing)

# 사전(dictionary)

## ◆ADT 사전 (dictionary) (key-element)

A dictionary is a list of key-element pairs, where keys are used to locate elements in the list. key-value

## ◆Operations (methods) on dictionaries:

size () Returns the size of the dictionary

IsEmpty () Returns **true** is the dictionary is empty

retrieve (key) 검색 Find the item with the specified key.

if no such key exists, sentinel value NO_SUCH_KEY is returned.

delete(key) Removes the item with the specified key

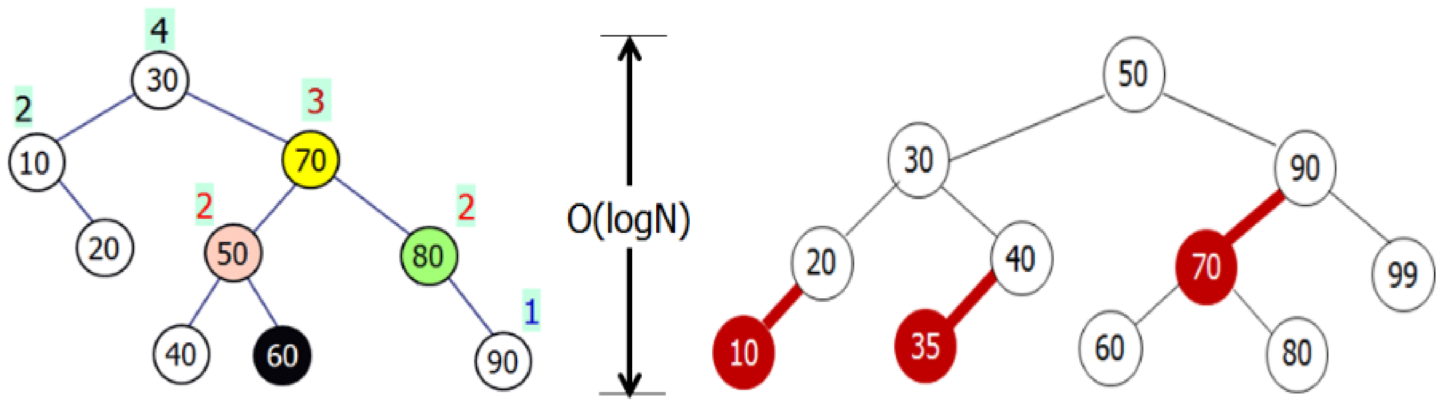
Insert (key, element) Inserts a new key-item pair

# 사전(dictionary)

* n은 item(항목, 원소) 개수

| | insert (put) | delete | retrieve (get) | |
|---|---|---|---|---|
| Unsorted list(array) | O(1) | O(n) | O(n) | |
| Unsorted linked list | O(1) | O(n) | O(n) | |
| 이진탐색트리 | O(h) | O(h) | O(h) | 최악의 경우<br>h = n<br>평균적인 경우<br>h = O(log n) |
| AVL 트리<br>Red-black 트리 | O(log n) | O(log n) | O(log n) | 찾는 연산의 목표는 O(1)로 만들기 위함 |

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




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

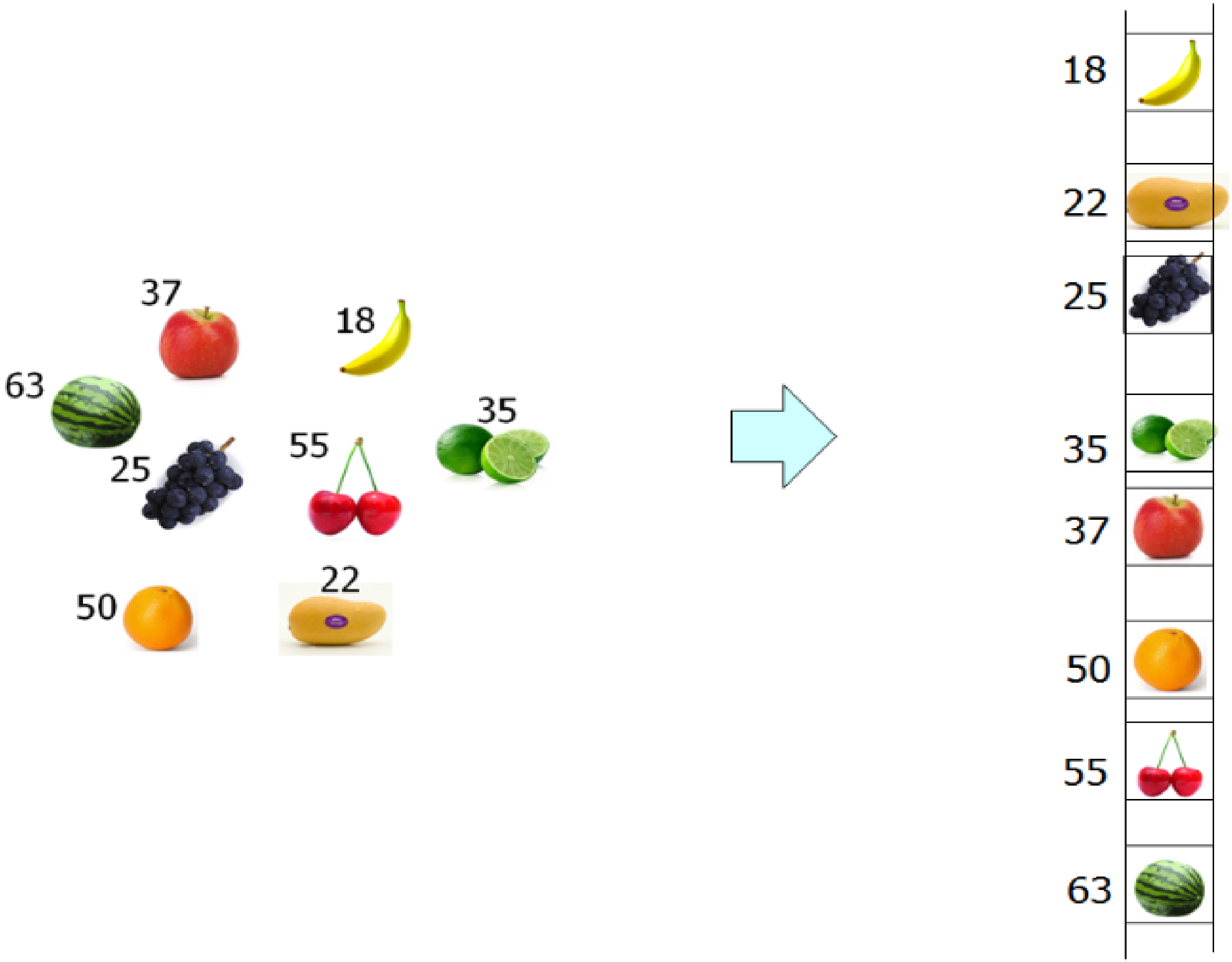
함수: 해시함수.

# 1. 해싱

키 – 해시테이블

인덱스

해시테이블

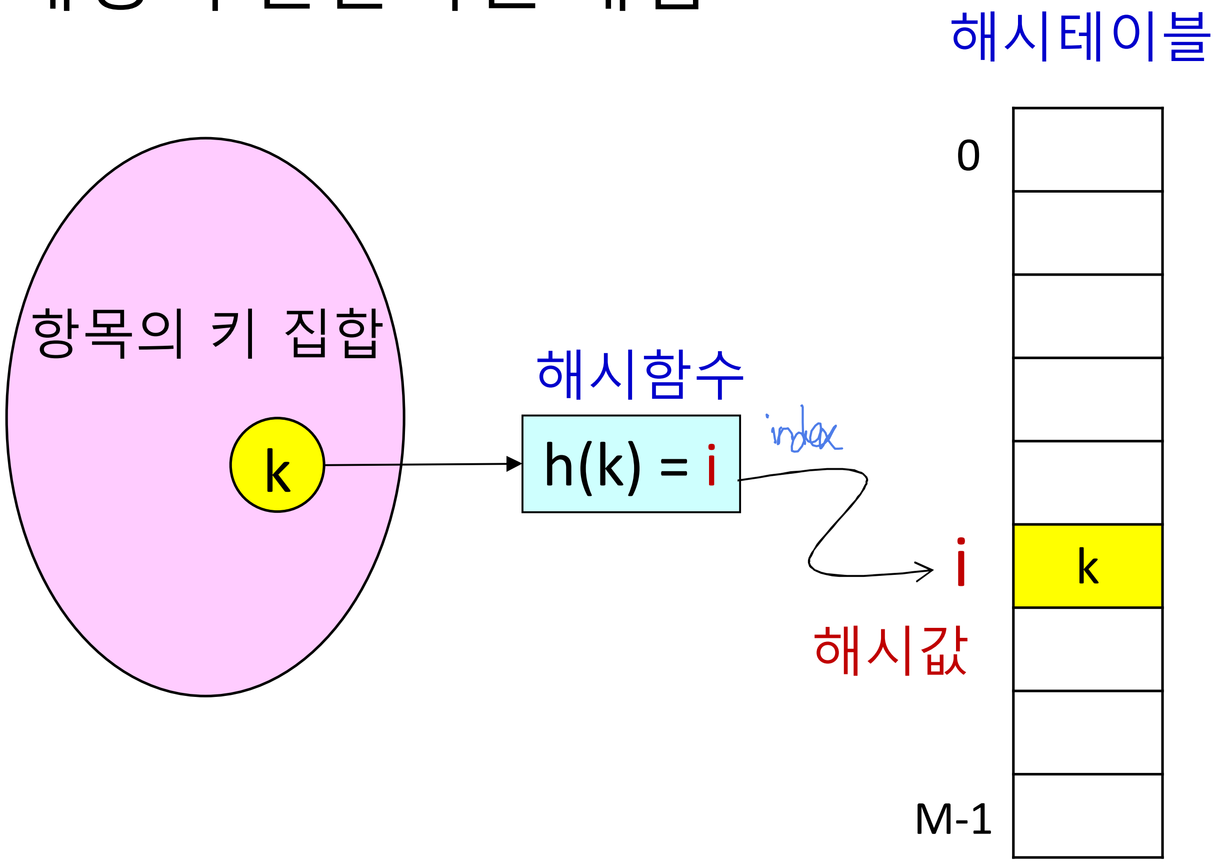


[핵심 아이디어] O(log n) 시간보다 빠른 연산을 위해, 키와 1차원 리스트의 인덱스의 관계를 이용하여 키(항목)를 저장한다.


[그림 6-2] 키를 그대로 1차원 리스트의 인덱스로 사용

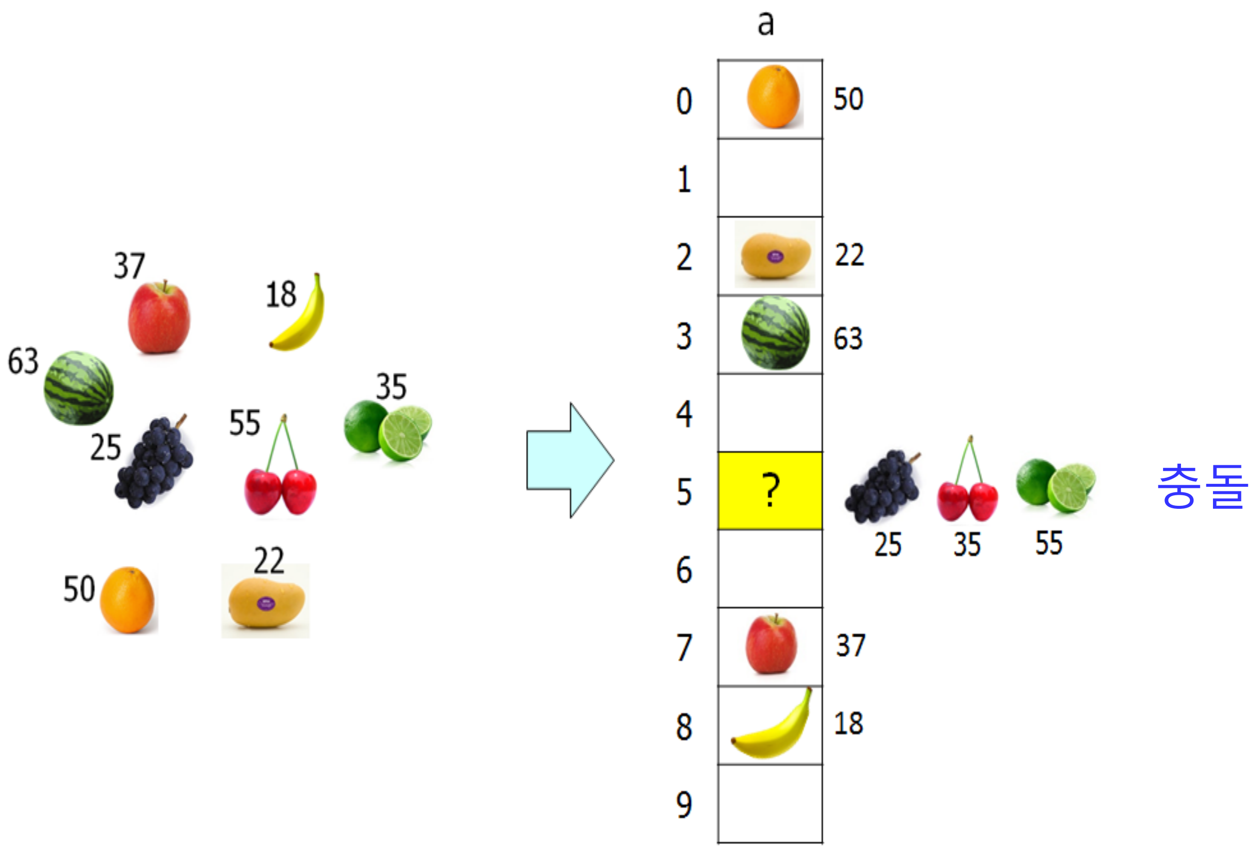
- 그러나 키를 배열의 인덱스로 그대로 사용하면 메모리 낭비가 심해질 수 있음
- [문제 해결 방안] 키를 변환하여 배열의 인덱스로 사용
- 키를 간단한 함수를 사용해 변환한 값을 배열의 인덱스로 이용하여 항목을 저장하는 것을 해싱(Hashing)이라고 함
- 해싱에 사용되는 함수를 해시함수(Hash Function)라 하고, 해시함수가 계산한 값을 해시값(Hash value) 또는 해시주소라고 하며, 항목이 해시값에 따라 저장되는 배열을 해시테이블(Hash Table)이라고 함

# 해싱의 전반적인 개념




M = 해시테이블 크기

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



# 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

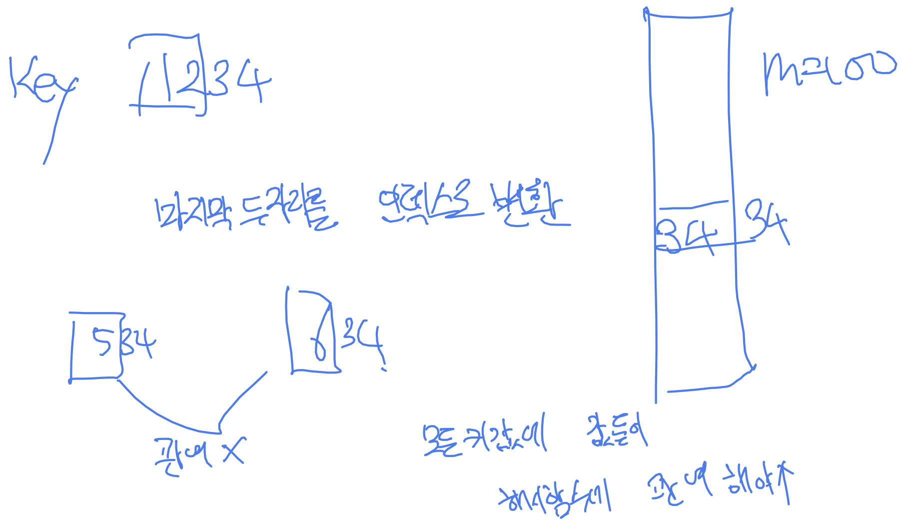
- 아무리 균등한 결과를 보장하는 해시함수라도 함수 계산 자체에 긴 시간이 소요된다면 해싱의 장점인 연산의 신속성을 상실하므로 그 가치를 잃음

- Hash 함수의 조건

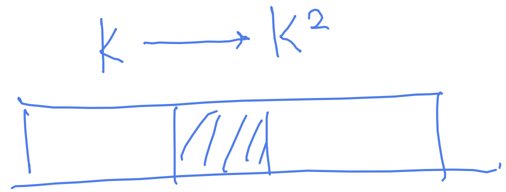
- 주소 계산이 빨라야 함
- 가급적 서로 다른 키의 해시 함수 값이 중복되어서는 안됨
  - ✓ 다른 키 값들에 대한 해시함수 값이 같은 충돌(collision) 확률이 작아야 함
  - ✓ 키 값들이 해시테이블에 골고루 분포될 수 있도록 해 주어야 함

- 균일 해시 함수(uniform hash function)
  - ➢ h(k) = i 가 될 확률은 모든 slot의 인덱스 i에 대해 i/M이 됨

    (k : 키 공간에서 임의로 선택된 키)
  - ➢ M개의 slot 각각에 임의의 k가 대응될 확률은 모두 같게 됨



# 3. 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

키의 모든 숫자들이 해시 주소값을 구하는데 사용되어야 충돌↓

- 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 제수(M)로 소수를 사용하는 이유는 나눗셈 연산을 했을 때, <u>소수가 키들을 균등하게 인덱스로 변환</u>시키는 성질을 갖기 때문

M : 100 일 경우

k = 125

k = 1325

충돌.

- 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 δ = $\frac{\sqrt{5}-1}{2} \approx$ 0.61803이 좋은 성능을 보인다.
  - 예를 들면, 테이블 크기 M = 127이고 키가 123456789인 경우, 123456789 x 0.61803 = 76299999.<u>30567</u>, 0.30567 x 127 = <u>38</u>.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

# 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) = 제곱조사

  랜덤조사(Random Probing)

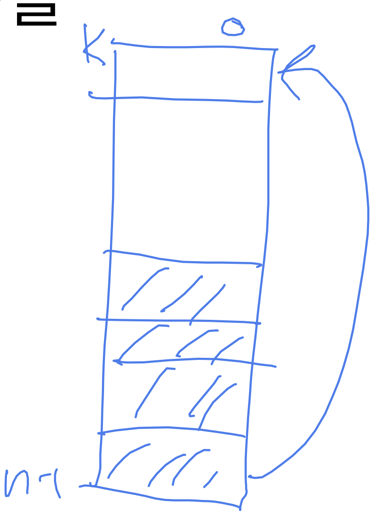
  이중해싱(Double Hashing)

# 3.1 선형조사 (Linear probing)

- 선형조사는 충돌이 일어난 원소에서부터 <u>순차적으로 검색하여</u> 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

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

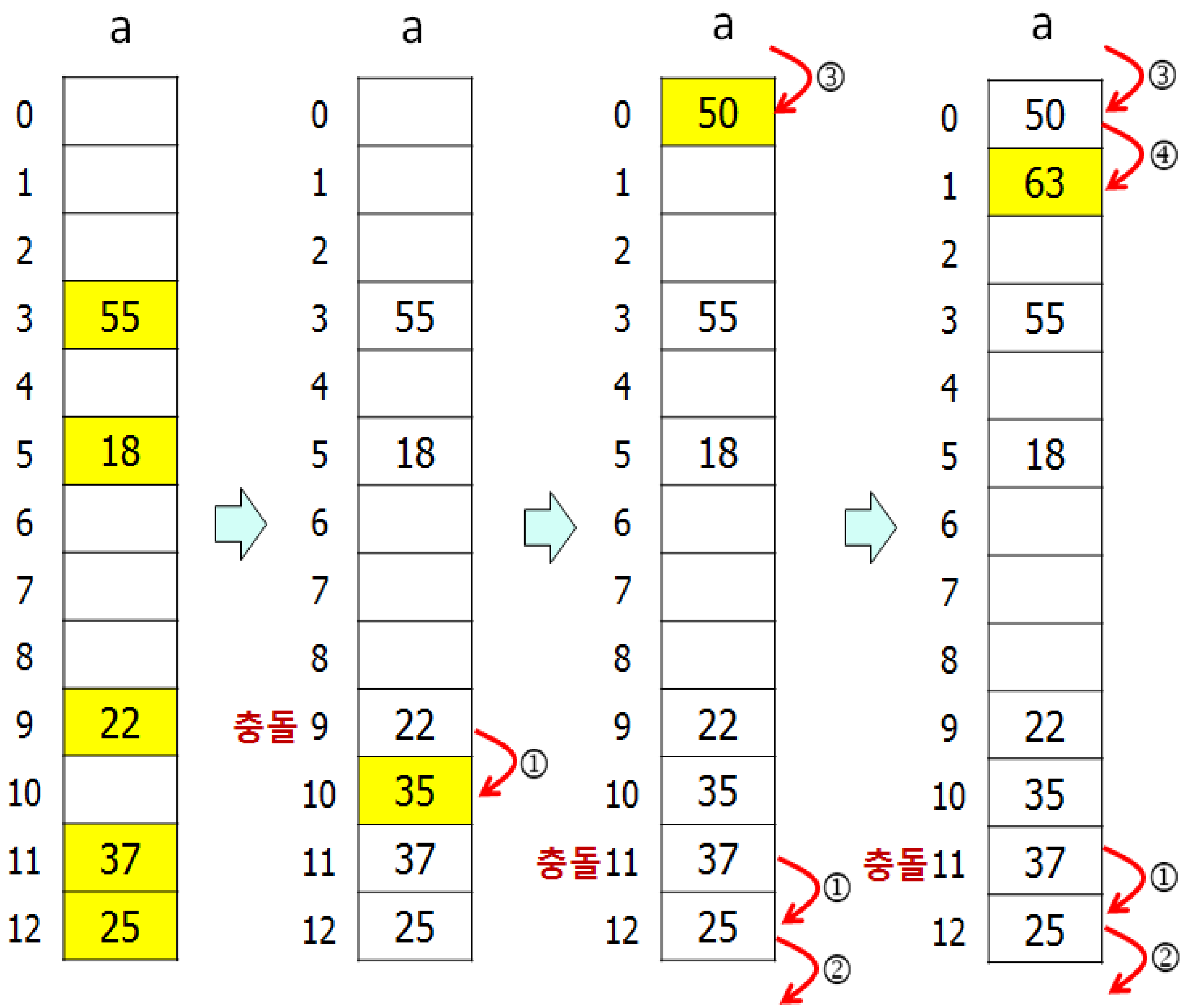
비어있는 공간을 순차적으로 찾음.




n-1 자리 다 차있을경우 그다음 n을 봐야함 → 0으로 돌아감

# 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

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 50 | 63 | | 55 | | 18 | | | | 22 | 35 | 37 | 25 |

군집화

- 군집화는 해시테이블에 empty 원소 수가 적을수록 더 심화되며 해시성능을 극단적으로 저하시킴

```python
class Dictionary:
    def __init__(self, size):
        self.M = size
        self.keyList = [None]*size
        self.valueList = [None]*size

    def hashFunc(self, key):
        return key % self.M

    def put(self, key, value): # 삽입: linear probing
        initialPos = self.hashFunc(key)
        i =  initialPos
        while True:
            if self.keyList[i] == None:
                self.keyList[i] = key
                self.valueList[i] = value
                return
            if self.keyList[i] == key:
                self.valueList[i] = value
                return
            i = (i + 1) % self.M
            if i == initialPos:
                return
```

```python
def get(self, key): # 검색(탐색)
        initialPos = self.hashFunc(key)
        i =  initialPos
        while self.keyList[i] != None:
            if self.keyList[i] == key:
                return self.valueList[i]
            i = (i + 1) % self.M
            if i == initialPos:
                return
```

# 3.2 이차조사 (Quadratic probing)

=제곱.

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

- 충돌 후 배열 a에서

방법 1)

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

으로 선형조사보다 더 멀리 떨어진 곳에서 empty 원소를 찾음

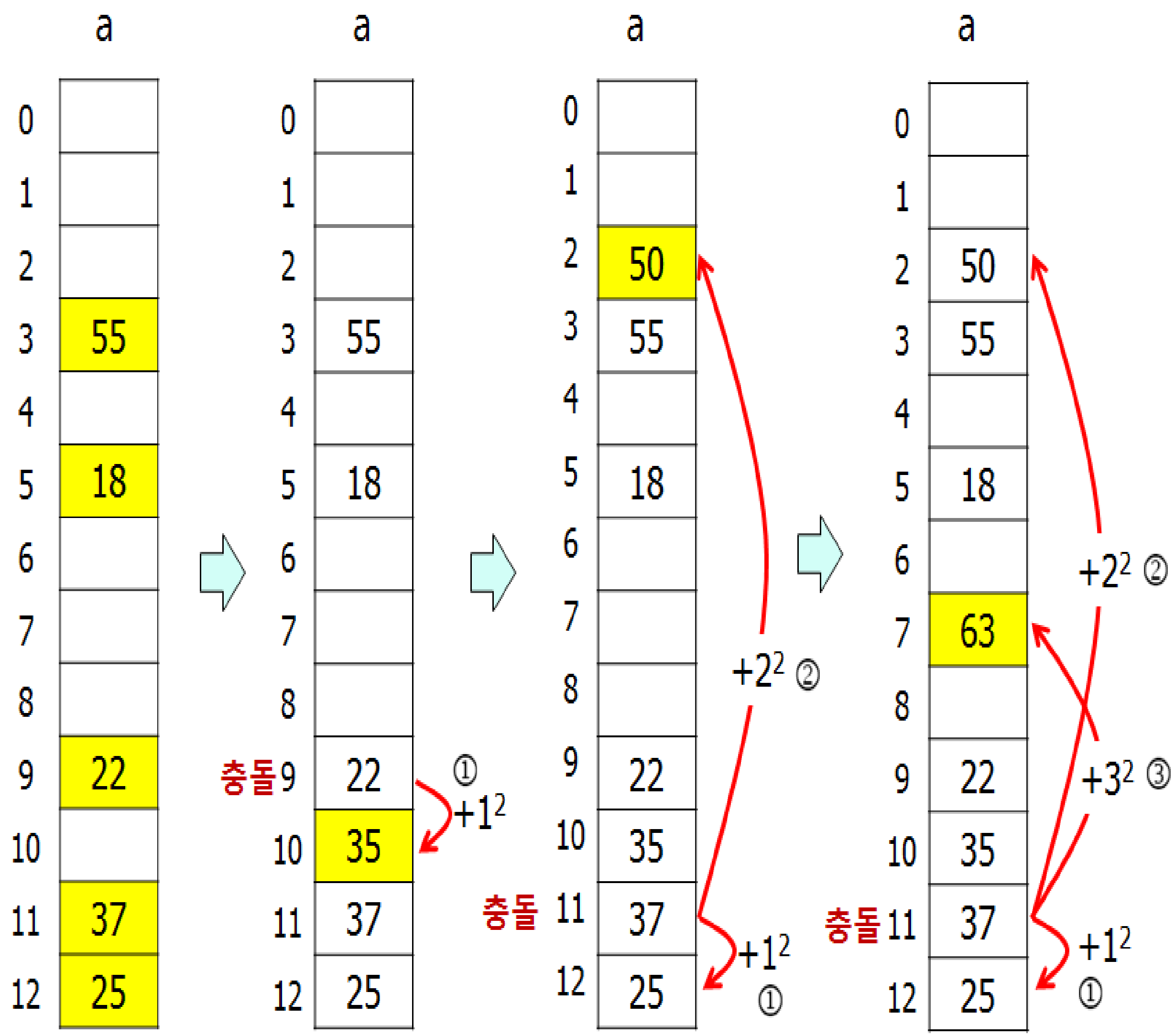
방법 2)

$$(h(key) \pm j^2) \% M, \ \overset{j}{\cancel{i}} = 1, 2, \ldots$$

**M이 4j+3 형태의 소수이면 빈 곳을 항상 찾는다.**

# 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

```python
class Dictionary:
    def __init__(self, size):
        self.M = size      # 테이블 크기
        self.keyList = [None]*size
        self.valueList = [None]*size
        self.N = 0        # 저장된 항목 수

    def hashFunc(self, key):
        return key % self.M

    def put(self, key, value): # 삽입: quadratic probing
        initialPos = self.hashFunc(key)
        i = initialPos
        j = 0
        while True:
            if self.keyList[i] == None:
                self.keyList[i] = key
                self.valueList[i] = value
                self.N += 1
                return
            if self.keyList[i] == key:
                self.valueList[i] = value
                return
            j += 1
            i = (initialPos + j*j) % self.M
            if self.N > self.M:
                return
```

```python
    def get(self, key):
        initialPos = self.hashFunc(key)
        i = initialPos
        j = 0
        while self.keyList[i] != None:
            if self.keyList[i] == key:
                return self.valueList[i]
            j += 1
            i = (initialPos + j*j) % self.M
        return None
```

## 3.3 이중해싱 (Double hashing)

- 이중해싱(Double Hashing)은 2 개의 해시함수를 사용
- 하나는 기본적인 해시함수h1(key)로 키를 해시테이블의 인덱스로 변환하고, 제2의 함수 h2(key)는 충돌 발생 시 다음 위치를 위한 점프 크기를 다음의 규칙에 따라 정함

$$(h1(key) + j \cdot h2(key)) \bmod M, \; j = 0, 1, 2, \cdots$$

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

- 제 2의 함수 h2(key)는 점프 크기를 정하는 함수이므로 0을 리턴해선 안됨
- 그 외의 조건으로 h2(key)의 값과 해시테이블의 크기 M과 서로소(Relatively Prime) 관계일 때 좋은 성능을 보임
- 하지만 해시테이블 크기 M을 소수로 선택하면, 이 제약 조건을 만족
- 전형적인 2차 해시함수 h2

  h2(key)=R – ( key % R ), 여기서 R은 M보다 작은 소수임

- h1(key) = key % 13과 h2(key) = 7-(key % 7) 에 따라, 25, 37, 18, 55, 22, 35, 50, 63을 해시테이블에 차례로 저장하는 과정

M=13

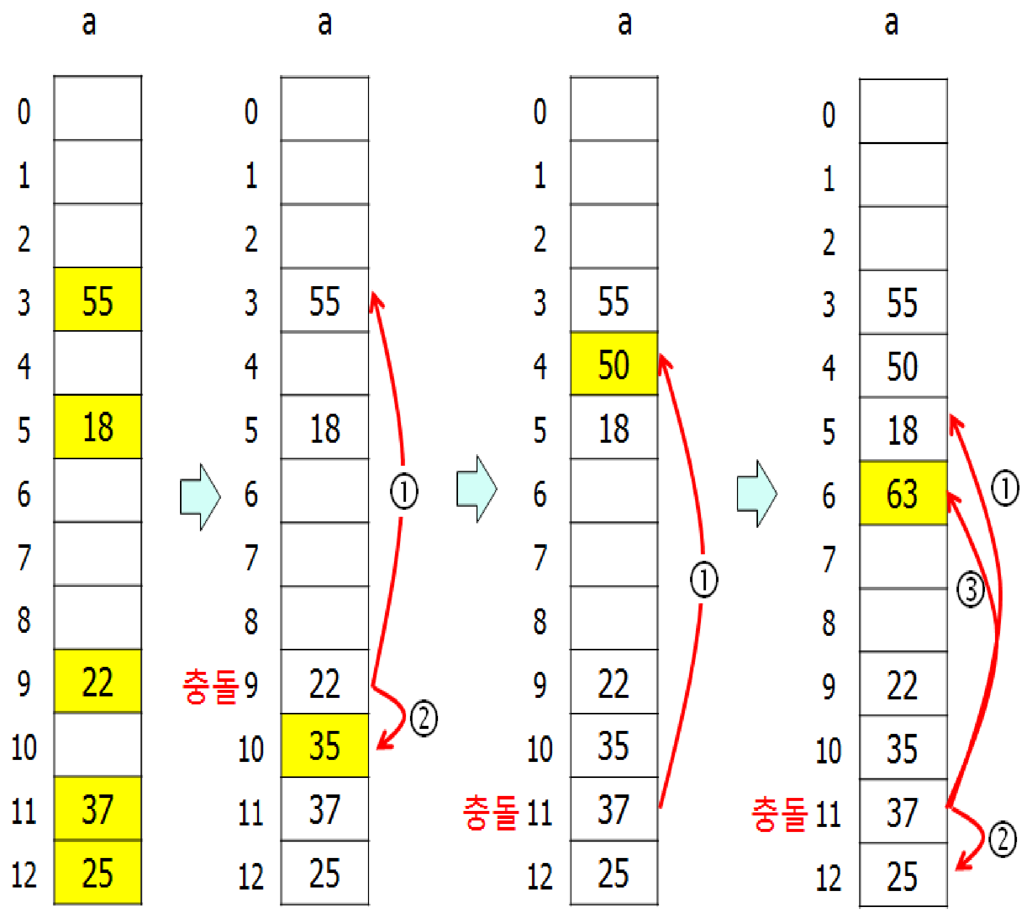
1~7

| 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 |

h1(key) = key % 13

h2(key) = 7 – (key % 7)

(h1(key)+h2(key)) % 13, j = 0, 1, …



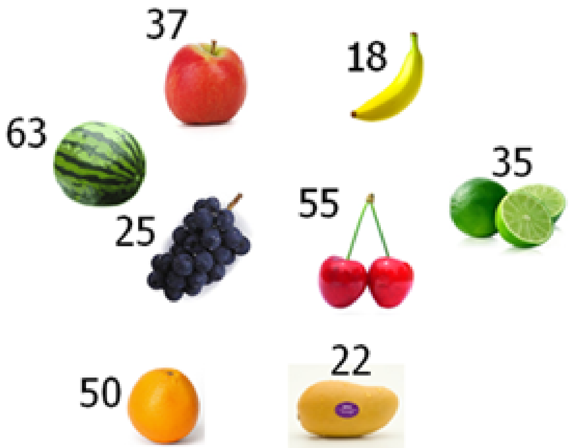
# 이중해싱의 장점

- 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.

# 4. 폐쇄주소방식 (Closed Addressing)

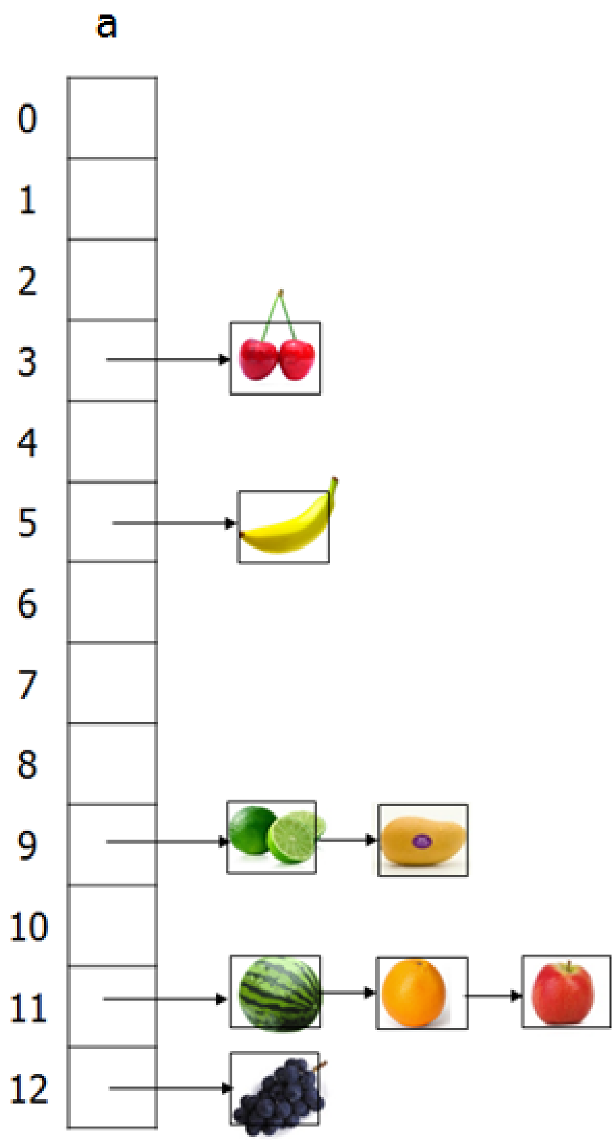
- 폐쇄주소방식(Closed Addressing)의 충돌해결 방법은 키에 대한 해시값에 대응되는 곳에만 키를 저장
- 충돌이 발생한 키들은 한 위치에 모여 저장
- 이를 구현하는 가장 대표적인 방법: 체이닝(Chaining)

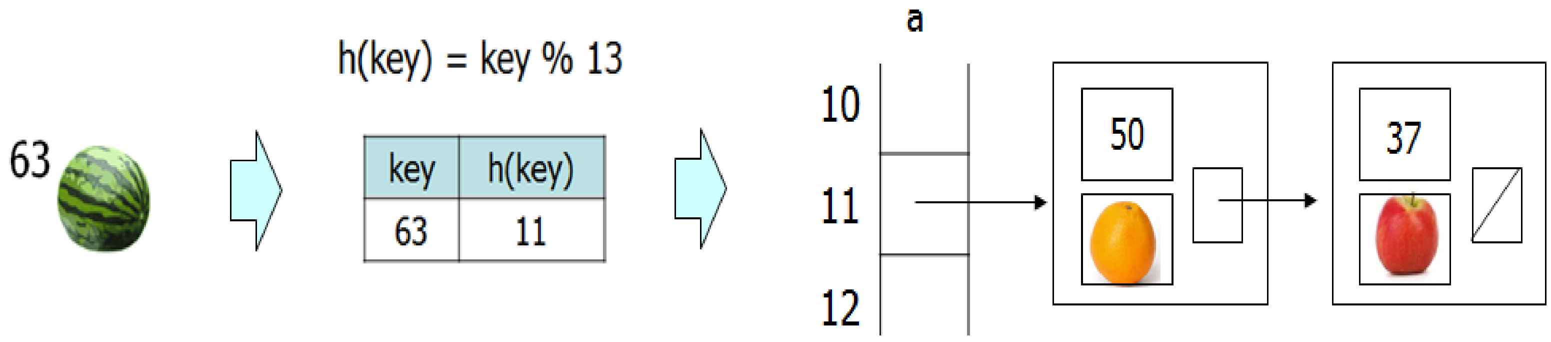
Liked-List 이용



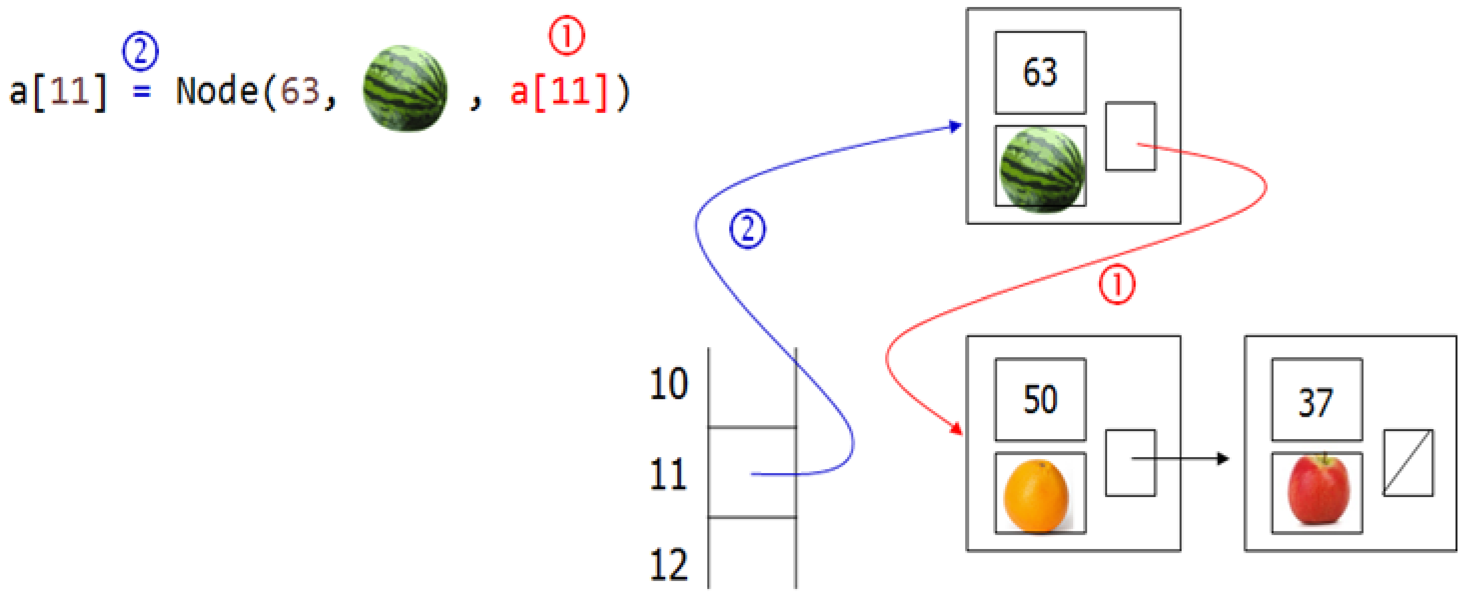

$h(key) = key \% 13$

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






63을 삽입하기 전




63을 삽입한 후

# 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 $\alpha$ = (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

# 5. 해시방법의 성능 비교 및 응용

- 해시방법의 성능은 탐색이나 삽입 연산을 수행할 때 성공과 실패한 경우를 각각 분석하여 측정
- 선형조사는 적재율 $\alpha$가 너무 작으면 해시테이블에 empty 원소가 너무 많고, $\alpha$ 값이 1.0에 근접할수록 군집화가 심화됨
- 개방주소방식의 해싱은 $\alpha \approx 0.5$, 즉, $M \approx 2N$일 때 상수시간 성능 보임

- 체이닝은 $\alpha$가 너무 작으면 대부분의 연결리스트들이 empty 가 되고, $\alpha$가 너무 크면 연결리스트들의 길이가 너무 길어져 해시성능이 매우 저하됨

- 일반적으로 M이 소수이고, $\alpha \approx 10$ 정도이면 O(1) 시간 성능을 보임