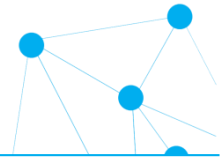
# 7.6 고급 탐색 구조: 해싱



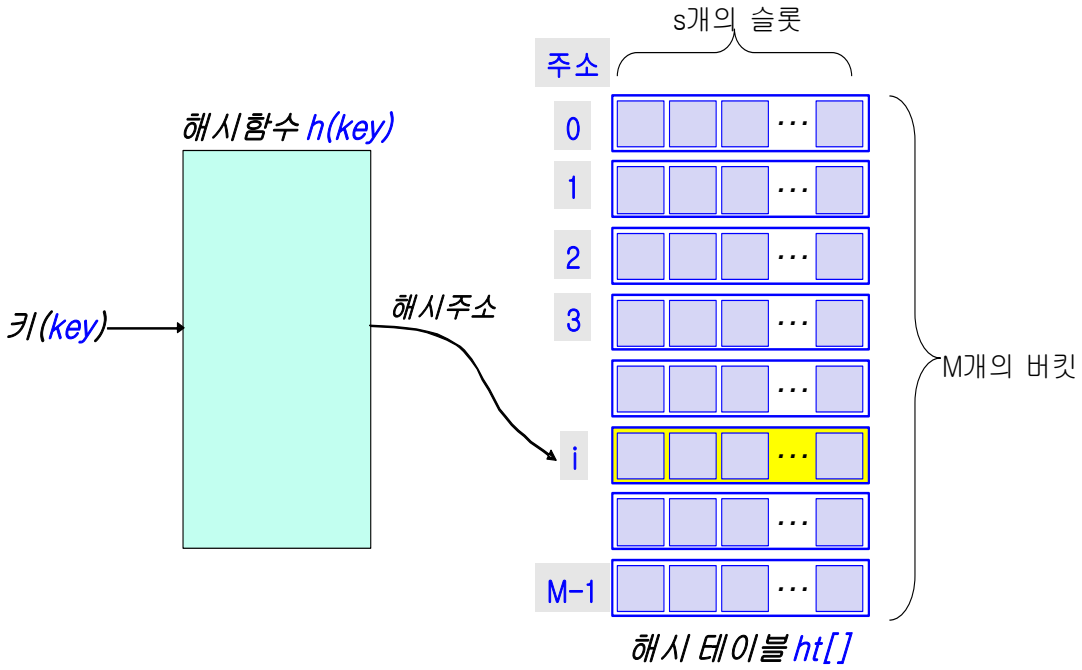
- 해싱이란?
- 선형 조사에 의한 오버플로 처리
- 체이닝(chaining)에 의한 오버플로 처리
- 해시 함수
- 탐색 방법들의 성능 비교

# 해싱이란?

- 해싱(hashing)
  - 키 값에 대한 산술적 연산에 의해 테이블의 주소를 계산
  - 해시 테이블(hash table)
    - 키 값의 연산에 의해 직접 접근이 가능한 구조

# 해싱의 구조

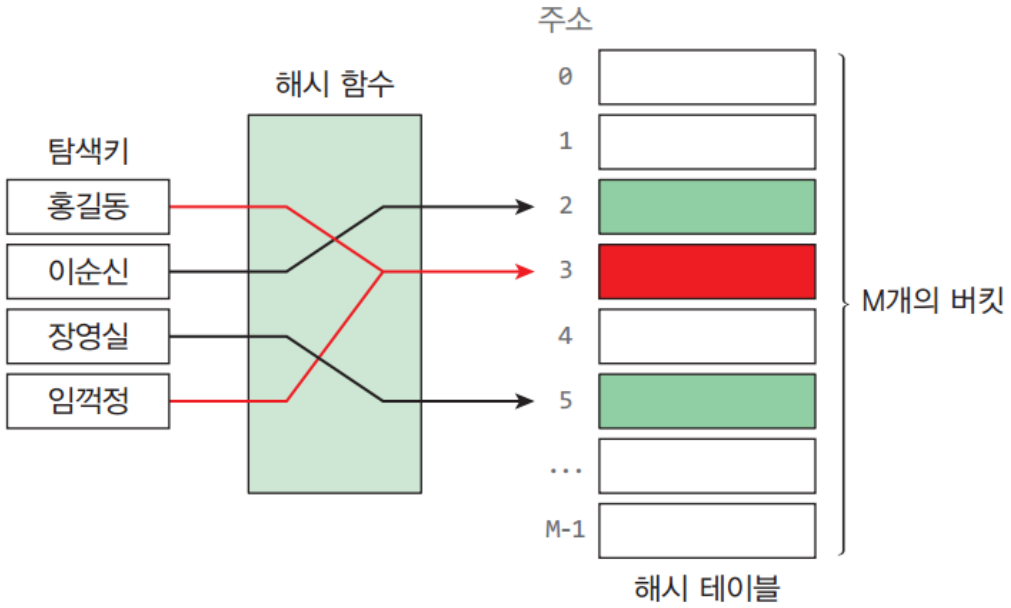
- 해시 테이블, 버킷, 슬롯
- 해시 함수(hash function)
  - 탐색키를 입력받아 해시 주소(hash address) 생성



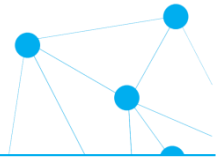
# 충돌과 오버플로

- 충돌
  - 서로 다른 키 가 해시 함수에 의해 같은 주소로 계산되는 상황
- 오버플로우
  - 충돌이 슬롯 수보다 많이 발생하는 것

h(홍길동)⇒3, h(이순신)⇒2, h(장영실)⇒5, h(임꺽정)⇒3



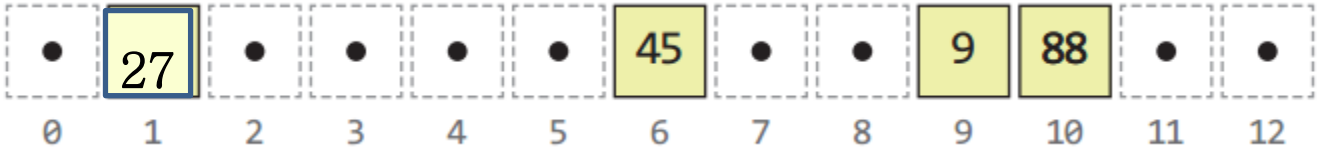
# 선형 조사에 의한 오버플로 처리



- 충돌이 일어나면 해시 테이블의 다음 위치에 저장
  - 다음 항목을 순서대로 조사: h(k), h(k)+1, h(k)+2,...
  - 빈 곳이 있으면 저장.
  - 테이블의 크기 *M*=13
  - h(k) = k%M
- 예) 45, 27, 88, 9, 71, 60, 46, 38, 24 저장 과정

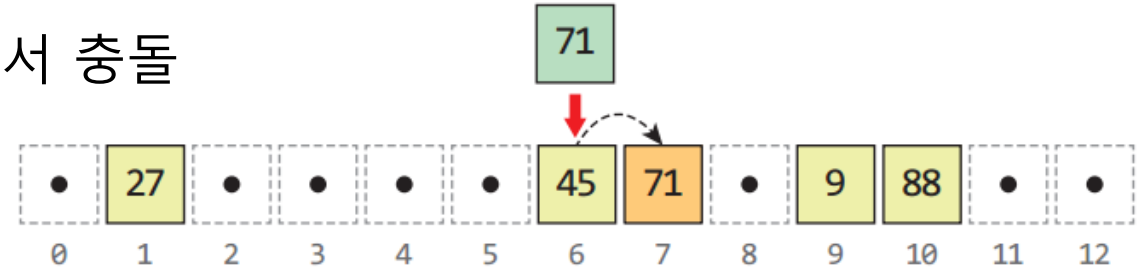
| *key* | 45 | 27 | 88 | 9 | 71 | 60 | 46 | 38 | 24 |
|---|---|---|---|---|---|---|---|---|---|
| *h(key)* | 6 | 1 | 10 | 9 | 6 | 8 | 7 | 12 | 11 |

① 45, 27, 88, 9 까지의 삽입



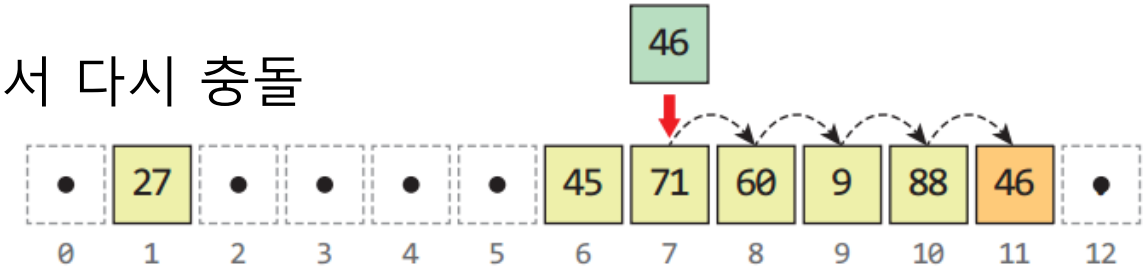
# 선형 조사: 삽입 연산

② 71의 삽입에서 충돌

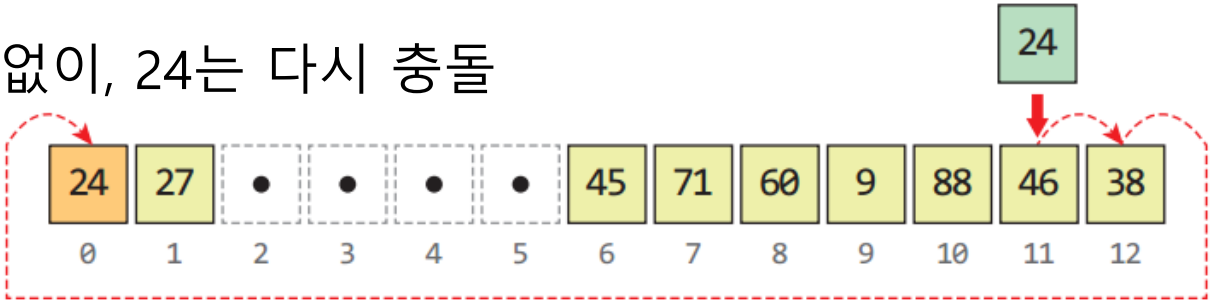



③ 60의 삽입

④ 46의 삽입에서 다시 충돌



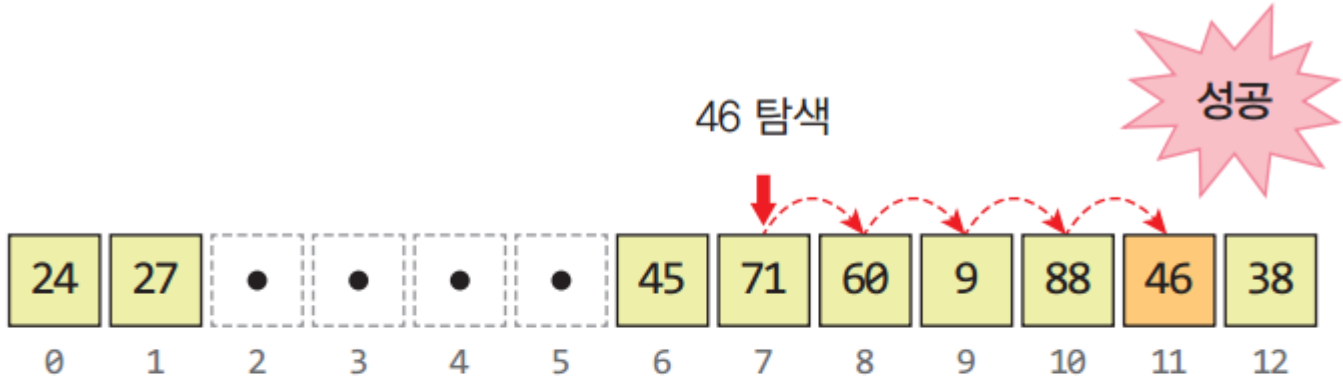

⑤ 38은 충돌 없이, 24는 다시 충돌



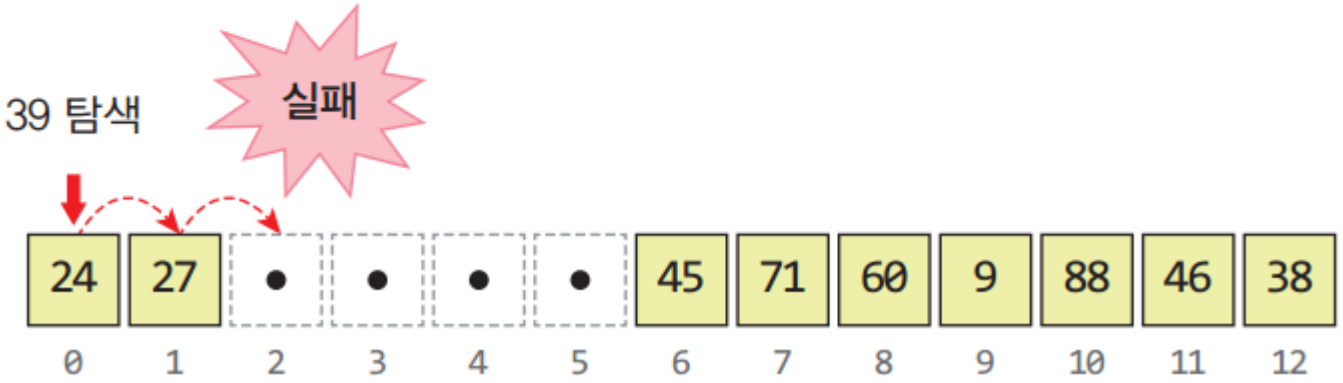

군집화 현상

# 선형 조사: 탐색 연산

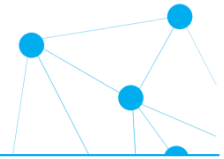
- 46 탐색



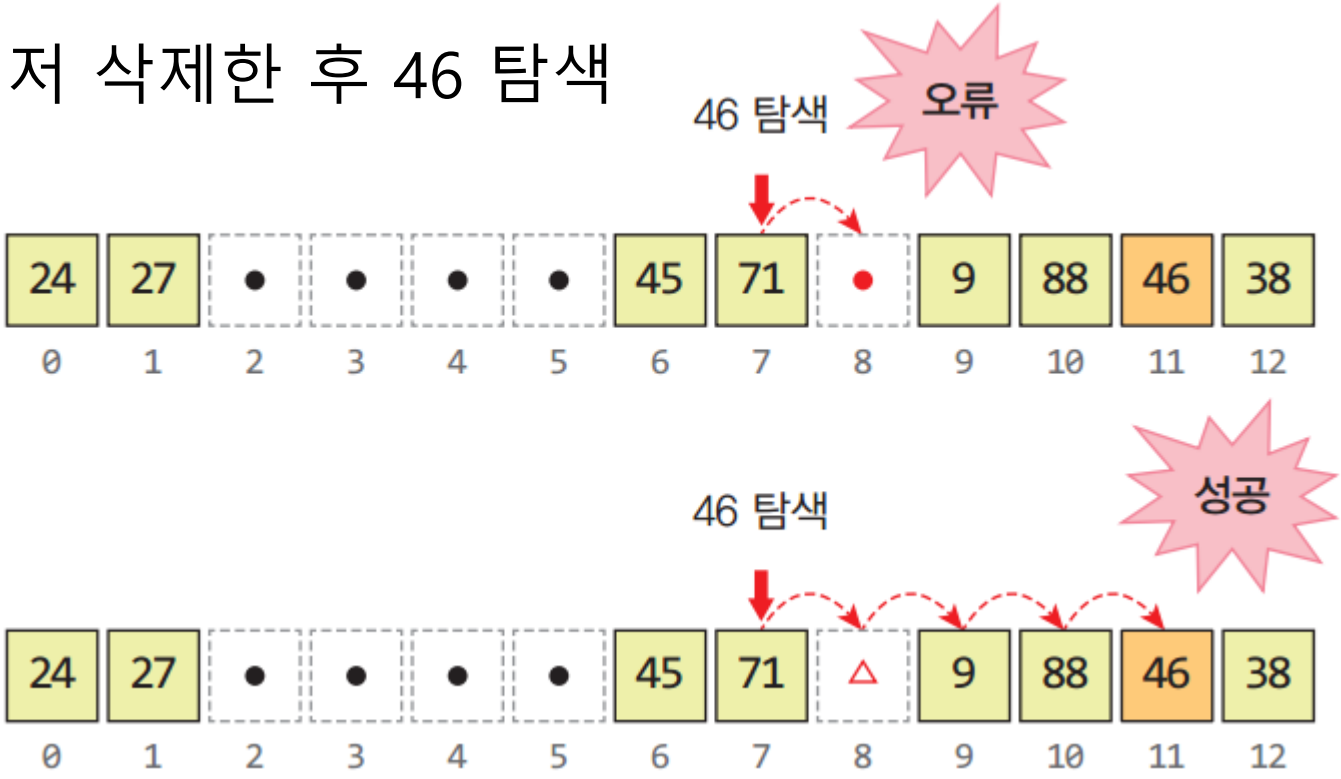

- 39 탐색



# 선형 조사: 삭제 연산

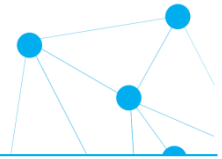


- 60을 먼저 삭제한 후 46 탐색




- 빈 버킷을 두 가지로 분류해야 함.

# 선형 조사 군집화 완화 방법

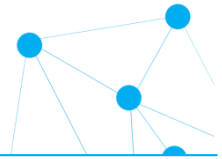


- 이차 조사법(quadratic probing)

$$(h(k)+i*i)\ \%\ M \quad \text{for} \quad i=0,1,\cdots,M-1$$

- 이중 해싱법(double hashing)
  - 재해싱(rehashing)
  - 충돌이 발생하면, 다른 해시 함수를 이용해 다음 위치 계산

# 체이닝에 의한 오버플로 처리



- 하나의 버킷에 여러 개의 레코드를 저장할 수 있도록 하는 방법

- 예) h(k)=k%7 을 이용해 8, 1, 9, 6, 13 을 삽입

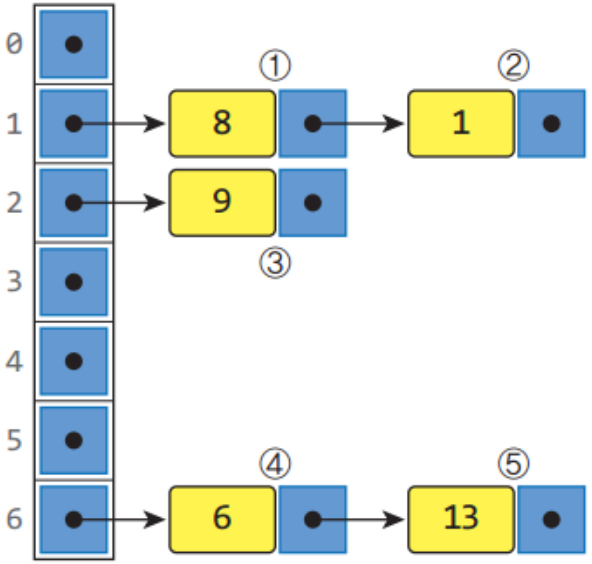
① 8 저장 : h(8) = 8 % 7 = 1 ⇒ 저장

② 1 저장 : h(1) = 1 % 7 = 1 ⇒ 충돌 ⇒ 새로운 노드 생성 및 저장

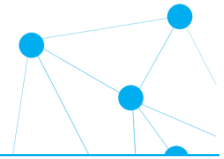
③ 9 저장 : h(9) = 9 % 7 = 2 ⇒ 저장

④ 6 저장 : h(6) = 6 % 7 = 6 ⇒ 저장

⑤ 13저장 : h(13) = 13 % 7 = 6 ⇒ 충돌 ⇒ 새로운 노드 생성 및 저장



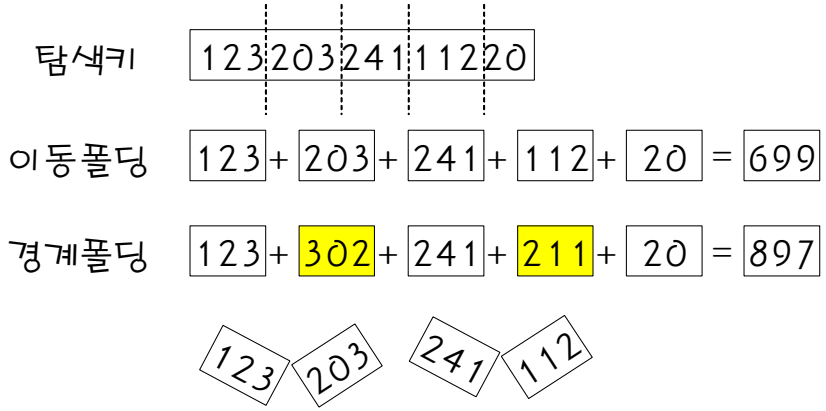
# 해시 함수



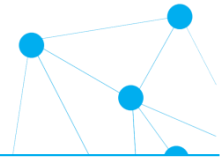
- 좋은 해시 함수의 조건
  - 충돌이 적어야 한다
  - 함수 값이 테이블의 주소 영역 내에서 고르게 분포되어야 한다
  - 계산이 빨라야 한다

- 제산 함수
  - h(k)=k mod M
  - 해시 테이블의 크기 M은 소수(prime number) 선택

- 폴딩 함수



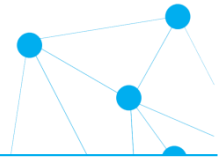
# 해시 함수



- 중간 제곱 함수
  - 탐색키를 제곱한 다음, 중간의 몇 비트를 취해서 해시 주소 생성
- 비트 추출 함수
  - 키를 이진수로 간주. 임의의 위치의 k개의 비트를 사용
- 숫자 분석 방법
  - 키에서 편중되지 않는 수들을 테이블의 크기에 적합하게 조합
- 탐색키가 문자열인 경우

```
def hashFn(key) :
    sum = 0
    for c in key :                  # 문자열의 모든 문자에 대해
        sum = sum +  ord(c)         # 그 문자의 아스키 코드 값을 sum에 더함
    return sum % M
```

# 탐색 방법들의 성능 비교



- 해싱의 적재 밀도(loading density) 또는 적재 비율
  - 저장되는 항목의 개수 n과 해시 테이블의 크기 M의 비율

$$\alpha = \frac{\text{저장된 항목의 개수}}{\text{해싱테이블의 버킷의 개수}} = \frac{n}{M}$$

- 다양한 탐색 방법의 성능 비교

| 탐색방법 | | 탐색 | 삽입 | 삭제 |
|---|---|---|---|---|
| 순차탐색 | | $O(n)$ | $O(1)$ | $O(n)$ |
| 이진탐색 | | $O(\log_2 n)$ | $O(n)$ | $O(n)$ |
| 이진탐색트리 | 균형트리 | $O(\log_2 n)$ | $O(\log_2 n)$ | $O(\log_2 n)$ |
| | 경사트리 | $O(n)$ | $O(n)$ | $O(n)$ |
| 해싱 | 최선의 경우 | $O(1)$ | $O(1)$ | $O(1)$ |
| | 최악의 경우 | $O(n)$ | $O(n)$ | $O(n)$ |