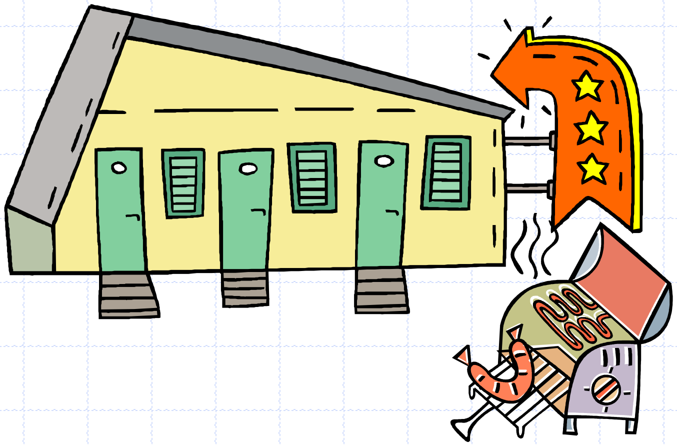
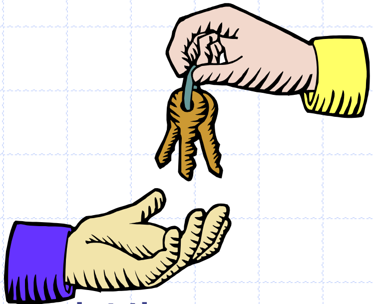
해시테이블



Outline

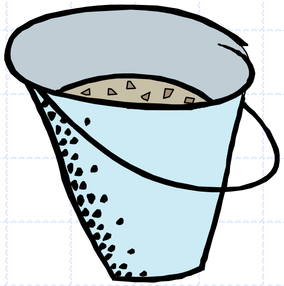
- 12.1 해시테이블
- 12.2 버켓 배열
- 12.3 해시함수
- 12.4 충돌 해결
- 12.5 해시테이블 성능
- 12.6 응용문제

해시테이블

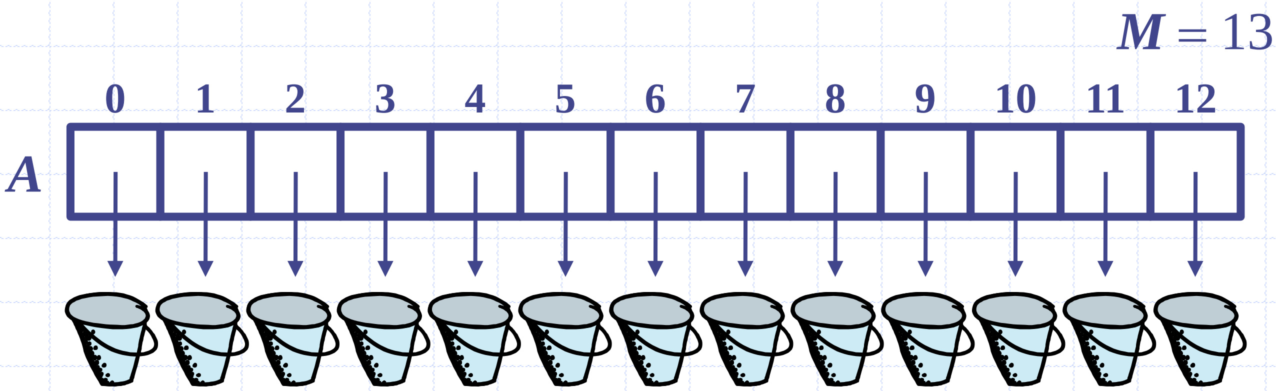


- **해시테이블**(hash table): 은 **키-주소** 매핑에 의해 구현된 사전 ADT
  - **예:** 컴파일러의 심볼 테이블, 환경변수들의 레지스트리
- **해시테이블 = 버켓 배열 + 해시함수**
  - 항목들의 **키**를 **주소**(즉, 배열 첨자)로 매핑함으로서 1차원 배열에 사전 항목들을 저장
- 성능
  - 탐색, 삽입, 삭제: **O(n)** 최악시간, 그러나 **O(1)** **기대시간**

버켓 배열



- 해시테이블을 위한 **버켓 배열**(bucket array)은 크기 $\boldsymbol{M}$의 배열 $\boldsymbol{A}$로서:
  - $\boldsymbol{A}$의 각 셀을 **버켓**(즉, 키-원소 쌍을 담는 그릇)으로 본다 - **슬롯**(slot)이라고도 함
  - 정수 $\boldsymbol{M}$은 배열의 **용량**을 정의
  - 키 $\boldsymbol{k}$를 가진 원소 $\boldsymbol{e}$는 버켓 $\boldsymbol{A[k]}$에 삽입
  - 사전에 존재하지 않는 키에 속하는 버켓 셀들은 ***NoSuchKey***라는 특별한 개체를 담는 것으로 가정



버켓 배열 분석

- 키가 유일한 정수며 $[0, M-1]$ 범위에 잘 분포되어 있다면, 해시테이블에서의 탐색, 삽입, 삭제에 $\mathbf{O}(1)$ 최악의 시간 소요
- 두 가지 중요한 결함
  - $\Theta(n)$ 공간을 사용하므로, M이 n에 비해 매우 크다면 공간 낭비
  - 키들이 $[0, M-1]$ 범위내의 유일한 정수여야 하지만, 이는 종종 비현실적
- 설계 목표
  - 그러므로 해시테이블 데이터구조를 정의할 때는, 키를 $[0, M-1]$ 범위내의 정수로 매핑하는 **좋은** 방식과 함께 버켓 배열을 구성해야 한다

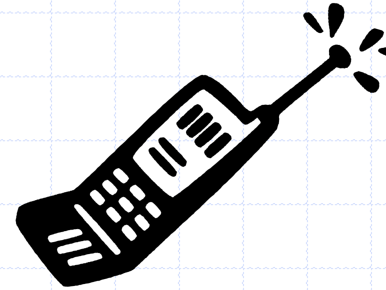
해시함수 및 해시테이블

- **해시함수**(hash function) $\boldsymbol{h}$: 주어진 형의 키를 고정 범위 $[0, M-1]$로 매핑
- **예**

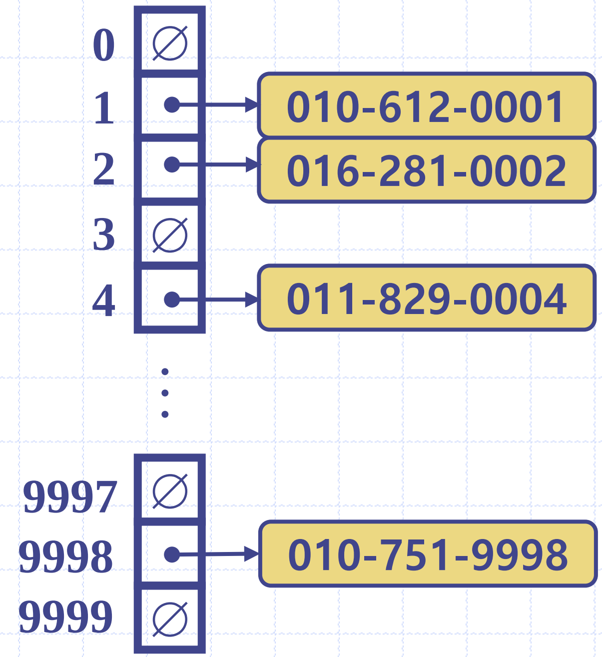
$$h(x) = x \ \% \ M$$

  - h는 정수 키 x에 대한 해시함수
- 정수 $h(x)$: 키 x의 **해시값**(hash value)
- 주어진 키 형의 해시테이블은 다음 두 가지로 구성
  - **해시함수** $\boldsymbol{h}$
  - 크기 M의 **배열**(**테이블**이라 불림)
- 사전을 해시테이블로 구현할 때, 목표는 항목 (k, e)를 첨자 $i = h(k)$에 저장하는 것

간단한 예



- (전화번호, 이름) 항목들을 저장하는 사전을 위한 해시테이블을 설계하자 – 여기서 전화번호는 10자리수의 양의 정수로 가정
- 설계된 해시테이블은 크기 $M = 10{,}000$의 배열과 아래 해시함수를 사용 $h(x) = x$의 마지막 네 자리



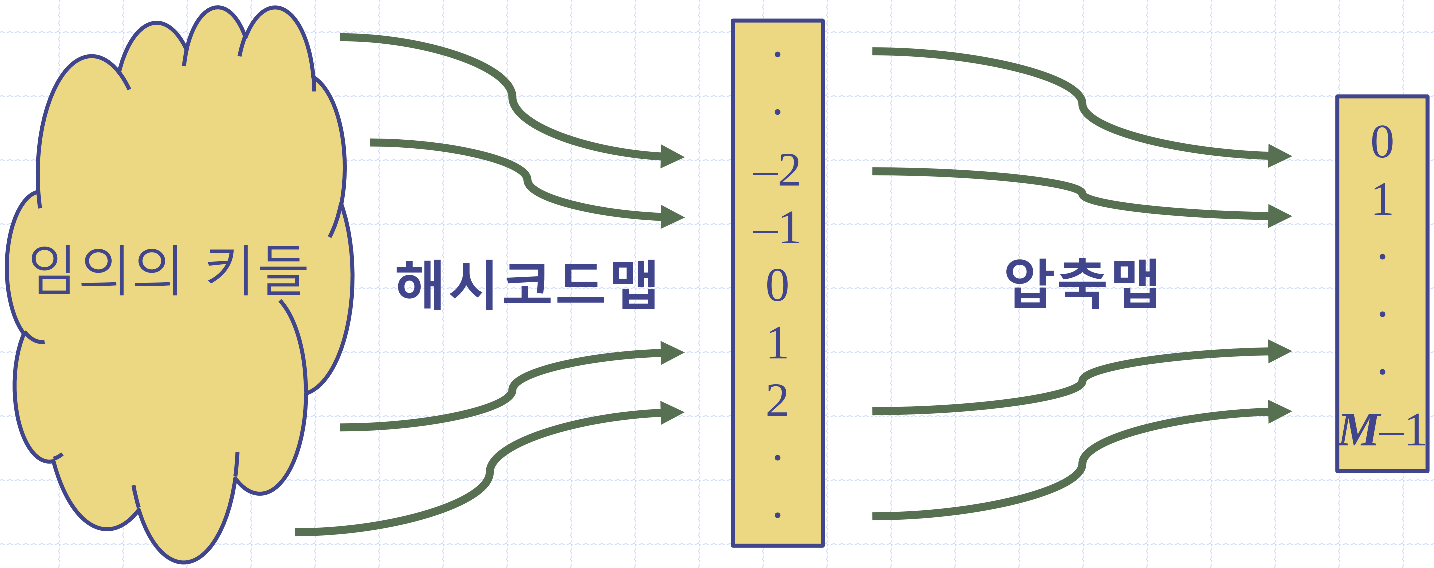
해시함수

- **해시함수**(hash function)는 보통 두 함수의 복합체로 명세
  - **해시코드맵**(hash code map) h_1: keys → integers
  - **압축맵**(compression map) h_2: integers → $[0, M-1]$
- 먼저 **해시코드맵**을 적용하고 그 결과에 **압축맵**을 적용 – 즉,

$$h(k) = h_2(h_1(k))$$

- **좋은** 해시함수가 되기 위한 조건
  - 키들을 외견상 **무작위**하게(random) 분산시켜야 한다
  - 계산이 빠르고 쉬워야 한다(가능하면 **상수시간**)

해시함수 예



- **예**
  - 학번 → 마지막 4 자리 수 → 방번호 [0, 2]
  - 식별자 → 문자합 → 심볼 테이블 행번호 [0, 27]

해시코드맵

- **메모리 주소**(memory address)
  - 키 개체의 메모리 주소를 정수로 재해석(모든 Java 객체들의 기본 해시코드)
  - 일반적으로 만족스러우나 **수치** 또는 **문자열** 키에는 적용 곤란
- **정수 캐스트**(integer cast)
  - 키의 비트값을 정수로 재해석
  - 정수형에 할당된 비트 수를 초과하지 않는 길이의 키에는 적당
    - **예:** Java의 byte, short, int, float

해시코드맵 (conti.)

- **요소합**(component sum)
  - 키의 비트들을 고정길이(**예:** 16 또는 32bits)의 요소들로 분할한 후 각 요소를 합한다(overflow는 무시)
  - 정수형에 할당된 비트 수 이상의 고정길이의 수치 키에 적당
    - **예:** Java의 long, double
  - 문자의 순서에 의미가 있는 문자열 키에는 부적당
    - **예:** temp01-temp10, stop-tops-spot-pots

해시코드맵 (conti.)

- **다항 누적**(polynomial accumulation)
  - 요소합과 마찬가지로, 키의 비트들을 고정길이(**예:** 8, 16, 32bits)의 요소들로 분할

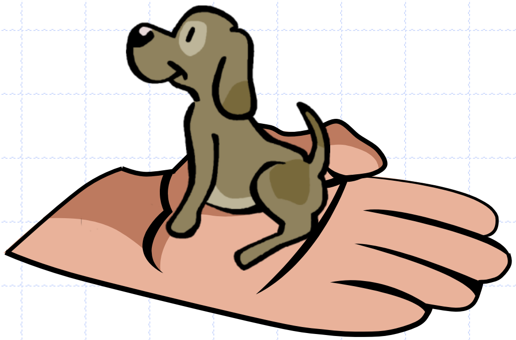
    $$a_0\, a_1\, \ldots\, a_{n-1}$$

  - 고정값 z를 사용하여 각 요소의 위치에 따른 별도 계산을 부과한 다항식 $p(z)$를 계산(overflow는 무시)

    $$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$$

  - 문자열에 특히 적당
    - **예:** 고정값 $z = 33$을 선택할 경우, 50,000개의 영단어에 대해 단지 6회의 충돌 발생

압축맵

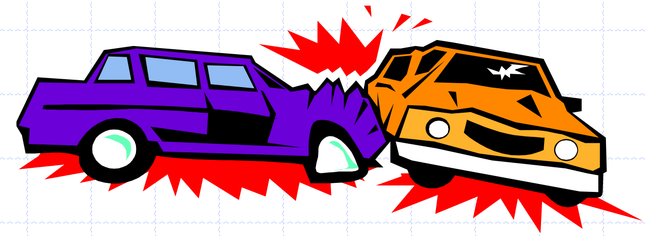


- **나누기**(division)
  - $h_2(k) = |k| \% M$
  - 해시테이블의 크기 M은 일반적으로 소수(prime)로 선택
- **승합제**(multiply, add and divide, **MAD**)
  - $h_2(k) = |ak + b| \% M$
  - a와 b는 음이 아닌 정수로서

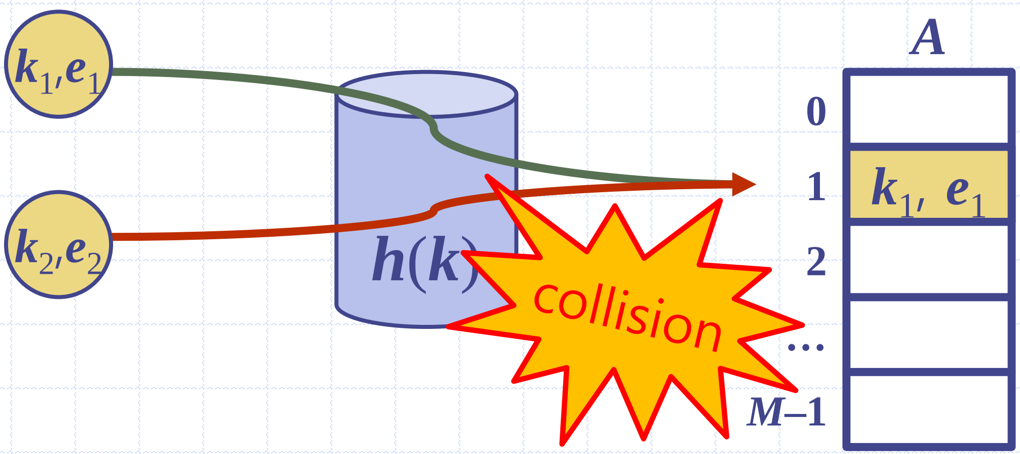
    $a \% M \neq 0$

    그렇지 않으면, 모든 정수가 동일한 값 b로 매핑됨

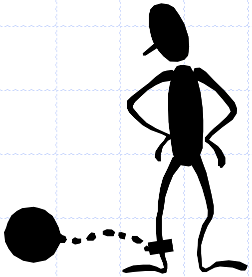
충돌 해결



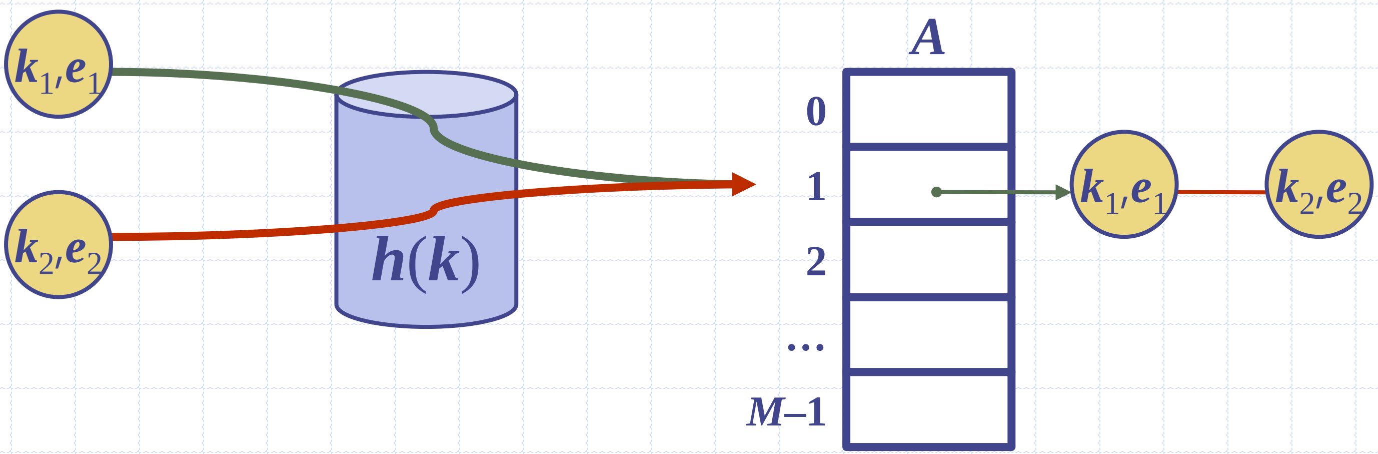
- **충돌**(collision): 두 개 이상의 원소들이 동일한 셀로 매핑
- 즉, 상이한 키, k_1과 k_2에 대해 $h(k_1) = h(k_2)$면 "충돌이 일어났다"고 말한다
- **충돌 해결**(collision resolution)을 위한 일관된 전략 필요



분리연쇄법



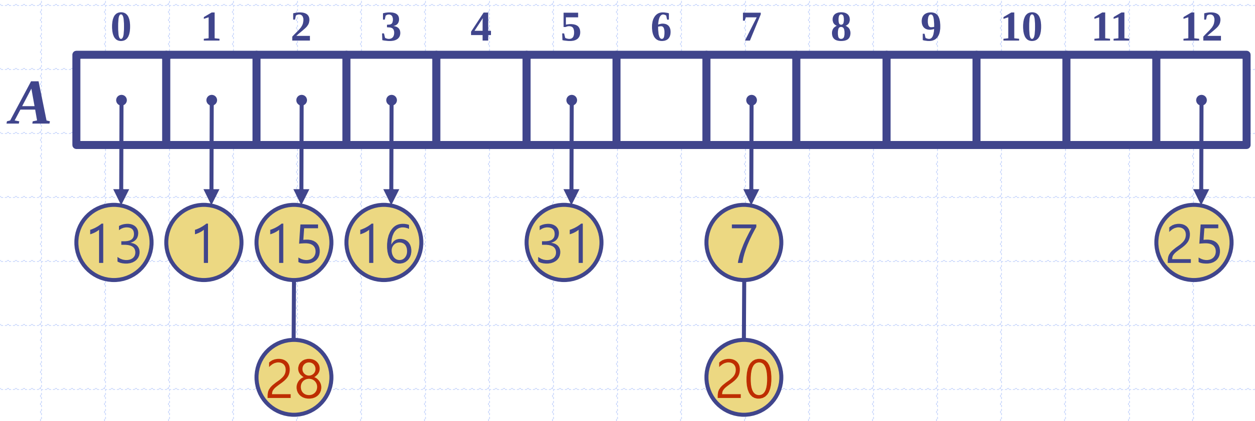
- **분리연쇄법**(separate chaining) 또는 **연쇄법**에서는 각 버켓 $A[i]$는 리스트 L_i에 대한 참조를 저장 – 여기서 L_i는:
  - 해시함수가 버켓 $A[i]$로 매핑한 모든 항목들을 저장
  - 무순리스트 또는 기록파일 방식을 사용하여 구현된 **미니 사전**이라 볼 수 있다
- **장단점:** 단순하고 빠르다는 장점이 있으나 테이블 외부에 추가적인 저장공간을 요구



분리연쇄법 예

◆ 예

- $h(k) = k \% M$
- 키(주어진 순서대로 삽입): 25, 13, 16, 15, 7, 28, 31, 20, 1



참고: 리스트에 추가되는 항목들의 **위치**는, 리스트의 테일 포인터를 별도로 유지하지 않는 경우라면 리스트의 **맨 앞**에 삽입하는 것이 유리

분리연쇄법 알고리즘

Alg ***findElement(k)***
  input bucket array $A[0..M-1]$,
    hash function h, key k
  output element with key k

1. $v \leftarrow h(k)$
2. **return** $A[v]$.*findElement*(k)

Alg ***insertItem(k, e)***
1. $v \leftarrow h(k)$
2. $A[v]$.*insertItem*(k, e)
3. **return**

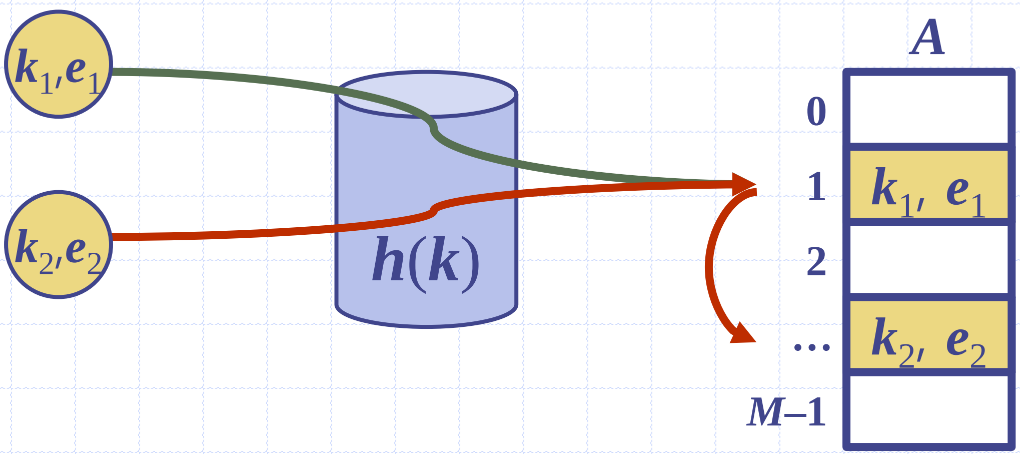
Alg ***removeElement(k)***
1. $v \leftarrow h(k)$
2. **return** $A[v]$.*removeElement*(k)

Alg ***initBucketArray()***
  input bucket array $A[0..M-1]$
  output bucket array $A[0..M-1]$
    initialized with null buckets

1. **for** $i \leftarrow 0$ **to** $M-1$
    $A[i] \leftarrow$ *empty list*
2. **return**

개방주소법

- **개방주소법**(open addressing): 충돌 항목을 테이블의 다른 셀에 저장
- **장단점:** 분리연쇄법에 비해 공간 사용을 절약하지만, 삭제가 어렵다는 것과 사전 항목들이 연이어 **군집화**(clustering)



선형조사법

- **선형조사법**(linear probing): 충돌 항목을 (원형으로) 바로 다음의 비어 있는 테이블 셀에 저장함으로써 충돌을 처리 – 즉, 다음 순서에 의해 버켓을 조사

  $A[(h(k) + f(i)) \% M],\ \ f(i) = i,\ i = 0, 1, 2, \ldots$

  (즉, $A[(h(k)]$, $A[(h(k) + 1]$, $A[(h(k) + 2]$, $A[(h(k) + 3]$, …의 순서)
- 검사되는 각 테이블 셀은 **조사**(probe)라 불린다
- 충돌 항목들은 군집화하며, 이후의 충돌에 의해 더욱 긴 **조사열**(probe sequence)로 군집 – "**1차 군집화**(primary clustering)"
- **예**
  - $h(k) = k \% M$
  - 키(주어진 순서대로 삽입): 25, 13, 16, 15, 7, 28, 31, 20, 1, 38

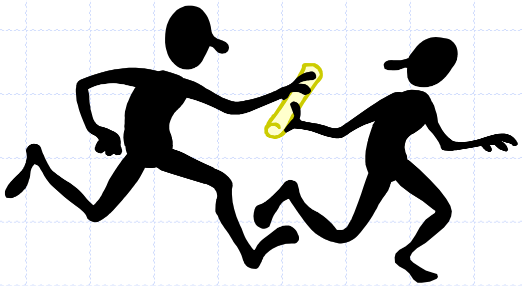
	0	1	2	3	4	5	6	7	8	9	10	11	12
A	13	1	15	16	28	31	38	7	20				25

2차 조사법

- **2차 조사법**(quadratic probing): 다음 순서에 의해 버켓을 조사
  $A[(h(k) + f(i)) \% M]$, $f(i) = i^2$, $i = 0, 1, 2, \ldots$
  (즉, $A[(h(k)]$, $A[(h(k) + 1]$, $A[(h(k) + 4]$, $A[(h(k) + 9]$, ...의 순서)
- 해시값이 동일한 키들은 동일한 조사를 수반
- 1차 군집화를 피하지만, 나름대로의 군집을 형성 – "**2차 군집화**(secondary clustering)"
- M이 소수가 아니거나 버켓 배열이 반 이상 차면, 비어 있는 버켓이 남아 있더라도 찾지 못할 수 있다
- **예**
  - $h(k) = k \% M$, $f(i) = i^2$
  - 키(주어진 순서대로 삽입): 25, 13, 16, 15, 7, 28, 31, 20, 1, 38

	0	1	2	3	4	5	6	7	8	9	10	11	12
A	13	1	15	16		31	28	7	20			38	25

이중해싱



- **이중해싱**(double hashing): 두번째의 해시함수 h'를 사용하여 다음 순서에 의해 버켓을 조사

  $A[(h(k) + f(i)) \% M], \; f(i) = i \cdot h'(k), i = 0, 1, 2, \ldots$

  (즉, $A[(h(k)], A[(h(k) + h'(k)], A[(h(k) + 2h'(k)], A[(h(k) + 3h'(k)], \ldots$의 순서)
- 동일한 해시값을 가지는 키들도 상이한 조사를 수반할 수 있기 때문에 군집화를 최소화
- h'는 계산 결과가 0이 되면 안 된다
- 최선의 결과를 위해, $h'(k)$와 M은 **서로소**(relative prime)여야 한다
  - $d_1 \cdot M = d_2 \cdot h'(k)$ 이면, d_2개의 조사만 시도 – 즉, 버켓들 중 d_2/M만 검사
  - $h'(k) = q - (k \% q)$ 또는 $1 + (k \% q)$를 사용 – 단, $q < M$은 소수며 M 역시 소수
- **예**
  - $h(k) = k \% M, h'(k) = 11 - (k \% 11)$
  - 키(주어진 순서대로 삽입): 25, 13, 16, 15, 7, 28, 31, 20, 1, 38

A	0	1	2	3	4	5	6	7	8	9	10	11	12
	13	1	15	16	28	31		7		20		38	25

개방주소법 알고리즘

Alg ***findElement(k)***
input bucket array $A[0..M-1]$,
hash function h, key k
output element with key k

```
1. v ← h(k)
2. i ← 0
3. while (i < M)
       b ← getNextBucket(v, i)
       if (isEmpty(A[b]))
           return NoSuchKey
       elseif (k = key(A[b]))
           return element(A[b])
       else
           i ← i + 1
4. return NoSuchKey
```

Alg ***insertItem(k, e)***

```
1. v ← h(k)
2. i ← 0
3. while (i < M)
       b ← getNextBucket(v, i)
       if (isEmpty(A[b]))
           Set bucket A[b] to (k, e)
           return
       else
           i ← i + 1
4. overflowException()
5. return
```

개방주소법 알고리즘 (conti.)

Alg ***getNextBucket*(*v*, *i*)**
{linear probing}
1. **return** $(v + i) \% M$

Alg ***getNextBucket*(*v*, *i*)**
{quadratic probing}
1. **return** $(v + i^2) \% M$

Alg ***getNextBucket*(*v*, *i*)**
{double hashing}
1. **return** $(v + i \cdot h'(k)) \% M$

Alg ***initBucketArray*()** {example}
input bucket array $A[0..M-1]$
output bucket array $A[0..M-1]$
initialized with null buckets

1. **for** $i \leftarrow 0$ **to** $M - 1$
   $A[i]$.**empty** $\leftarrow 1$ {set empty}
2. **return**

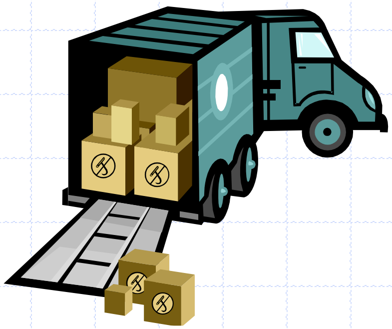
Alg ***isEmpty*(*b*)**
input bucket b
output boolean

1. **return** b.**empty**

개방주소법에서의 갱신

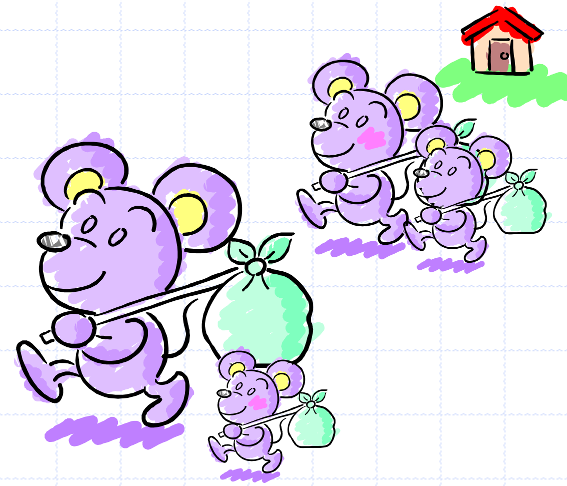
- **비활성화 전략**
  - 기존 태그
    - **empty:** 비어 있는 셀
    - **active:** 사용 중인 셀(활성)
  - 추가 태그
    - **inactive:** 삭제된 셀(비활성)
- findElement(k)
  1. 셀 ***h*(*k*)**에서 출발하여, 다음 가운데 하나일 때까지 조사
     - 비어 있는 셀을 만나면 탐색 실패
     - 활성 셀의 항목 **(*k*, *e*)**를 만나면 ***e***를 반환
     - ***M***개의 셀을 검사
  2. 탐색 실패
- insertItem(k, e)
  1. 셀 ***h*(*k*)**에서 출발하여, 다음 가운데 하나일 때까지 조사
     - 비어 있거나 비활성인 셀을 만나면 항목 **(*k*, *e*)**를 셀에 저장한 후 **활성화**
     - ***M***개의 셀을 검사
  2. 테이블 만원 예외를 발령
- removeElement(k)
  1. 셀 ***h*(*k*)**에서 출발하여, 다음 가운데 하나일 때까지 조사
     - 비어 있는 셀을 만나면 탐색 실패
     - 활성 셀의 항목 **(*k*, *e*)**를 만나면 **비활성화**하고 ***e***를 반환
     - ***M***개의 셀을 검사
  2. 탐색 실패

적재율



- 해시테이블의 **적재율**(load factor), $\alpha = n/M$
  - 즉, 좋은 해시함수를 사용할 경우의 각 버켓의 **기대 크기**
- 적재율은 낮게 유지되어야 한다(가능하면 1 아래로)
- **좋은** 해시함수가 주어졌다면, findElement, insertItem, removeElement 각 작업의 **기대실행시간**(expected running time): $\mathbf{O}(\alpha)$
- 분리연쇄법
  - $\alpha > 1$이면, 작동은 하지만 비효율적
  - $\alpha \le 1$이면(기왕이면 0.75 미만이면), $\mathbf{O}(\alpha) = \mathbf{O}(1)$의 **기대실행시간** 성취 가능
- 개방주소법
  - 항상 $\alpha \le 1$
  - $\alpha > 0.5$면, 선형 및 2차 조사법인 경우 군집화 가능성 농후
  - $\alpha \le 0.5$면, $\mathbf{O}(\alpha) = \mathbf{O}(1)$ **기대실행시간**

재해싱



- 해시테이블의 적재율을 상수(보통 0.75) 이하로 유지하기 위해서는, 원소를 삽입할 때마다 이 한계를 넘기지 않기 위해 추가적인 작업 필요
- 언제 **재해싱**(rehashing) 하는가?
  - 적재율의 최적치를 초과했을 때
  - 삽입이 실패한 경우
  - 너무 많은 비활성 셀들로 포화되어 성능이 저하되었을 때
- 재해싱의 단계
  1. 버켓 배열의 크기를 증가시킨다(원래 배열의 대략 두 배 크기로 – 이때 새 배열의 크기를 **소수**로 설정하는 것에 유의)
  2. 새 크기에 대응하도록 압축맵을 수정
  3. 새 압축맵을 사용하여, 기존 해시테이블의 모든 원소들을 새 테이블에 삽입

해싱의 성능

- 해시테이블에 대한 탐색, 삽입, 삭제: 최악의 경우 $\mathbf{O}(n)$ 시간 소요
- 최악의 경우: 사전에 삽입된 모든 키가 충돌할 경우
- **적재율**(load factor), $\alpha = n/N$은 해시테이블의 성능을 좌우
- 해시값들을 난수(random numbers)와 같다고 가정하면, 개방주소법에 의한 삽입을 위한 기대 조사 횟수는 $1/(1-\alpha)$라고 알려짐
- 해시테이블에서 모든 사전 ADT 작업들의 **기대실행시간:** $\mathbf{O}(1)$
- 실전에서, 적재율이 1(즉, 100%)에 가깝지만 않다면 해싱은 매우 빠르다
- **응용**
  - 소규모 데이터베이스
  - 컴파일러
  - 브라우저 캐시