8장 직접 화일

❖ 직접 화일의 개념

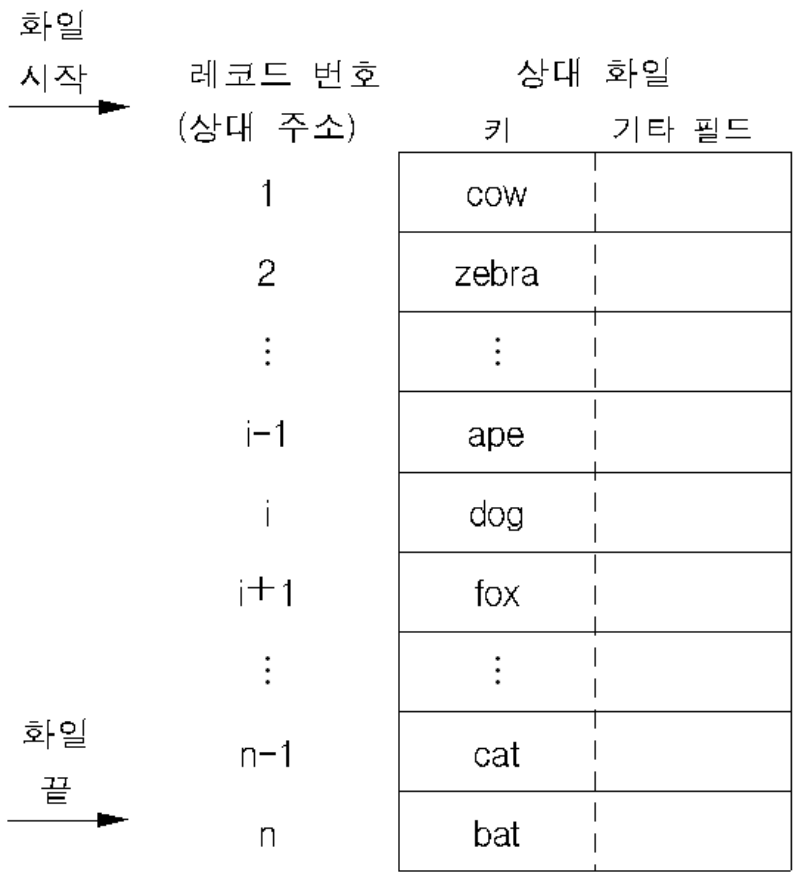
- **임의 접근 화일(random access file) 혹은 직접 화일(direct file)**
  - 임의의 레코드에 그 레코드의 키 값을 통해 접근할 수 있는 화일
  - 다른 레코드를 참조하지 않고도 개개 레코드에 접근 가능 (↔ 순차 접근 파일의 경우 전위 레코드 먼저 접근)

- **임의 접근 파일 (직접 파일)의 종류**
  - 인덱스된 화일(indexed file)
    - 인덱스를 이용해 레코드 접근
  - 인덱스된 순차 화일(indexed sequential file)
    - 인덱스를 이용한 임의 접근/순차 접근 모두 지원
  - 상대 화일(relative file)
    - 레코드 키와 주소 사이의 설정된 관계를 이용
  - 해시 화일(hash file)
    - 키 값을 통해 레코드 주소 생성 (상대 파일의 한 형태)
    - 협의의 직접 화일

▶ 상대 화일 (relative file)

- 레코드의 키와 레코드의 위치(상대 레코드 번호) 사이에 설정된 관계를 통해 레코드에 접근

- 상대 레코드 번호(**relative record number**)
  - 화일이 시작되는 첫 번째 레코드를 0번으로 지정, 이것을 기준으로 다음 레코드들에 1, 2, 3, …, N-1 지정 (= 상대 주소(relarive address))
  - ※ 레코드의 논리적 순서와 물리적 순서는 무관
  - ※ 레코드들이 키 값에 따라 물리적으로 정렬되어있을 필요는 없음



◆ 사상 함수(mapping function)

A : 키 값 → 주소

- 레코드 삽입 시 : 키 값 → 레코드가 저장될 주소
- 레코드 검색 시 : 키 값 → 레코드가 저장되어있는 주소
- 모든 레코드에 직접 접근 가능

  → 메인 메모리, 디스크 등 직접 저장 장치(DASD : Direct Access Storage Device)에 저장하는 것이 효율적

◆ 사상 함수의 구현 방법

- 직접 사상(direct mapping)
- 디렉터리 검사(directory lookup)
- 계산(computation) 을 이용한 방법 등. e.g., 해싱

▶ 직접 사상(direct mapping)

- 절대 주소 **(absolute address)** 이용
  - 키 값은 그 자체가 레코드의 실제 주소임
  - 레코드가 화일에 처음 저장될 때 레코드의 주소, <실린더 번호, 면 번호, 블록 번호>, 즉 절대 주소가 결정

- 장점 : 간단, 처리 시간이 거의 걸리지 않음
- 단점 : 물리적 저장장치에 의존적

  → 잘 사용되지 않음

▶ 디렉터리 검사(directory lookup)

- <키 값, (상대) 주소> 쌍을 엔트리로 하는 테이블 (디렉터리) 유지
- 검색 절차 : 디렉터리에서 키 값 검색 → 키 값에 대응되는 레코드 번호(상대주소) 구함 → 레코드 접근

- 장점 : 빠른 검색
- 단점 : 삽입 비용이 큼, 화일과 디렉터리 재구성 필요

- 구현 예
  - <키 값, 레코드 번호> 쌍을 엔트리로 갖는 배열
  - 디렉터리 엔트리는 키 값으로 정렬
  - 순차 접근은 가능하나 의미는 없음

디렉터리

키	레코드 번호
ape	i-1
bat	n
⋮	⋮
cat	n-1
cow	1
dog	i
⋮	⋮
fox	i+1
zebra	2

상대 화일

레코드 번호	키	기타 필드
1	cow	
2	zebra	
⋮	⋮	
i-1	ape	
i	dog	
i+1	fox	
⋮	⋮	
n-1	cat	
n	bat	

❖ 해싱(hashing)

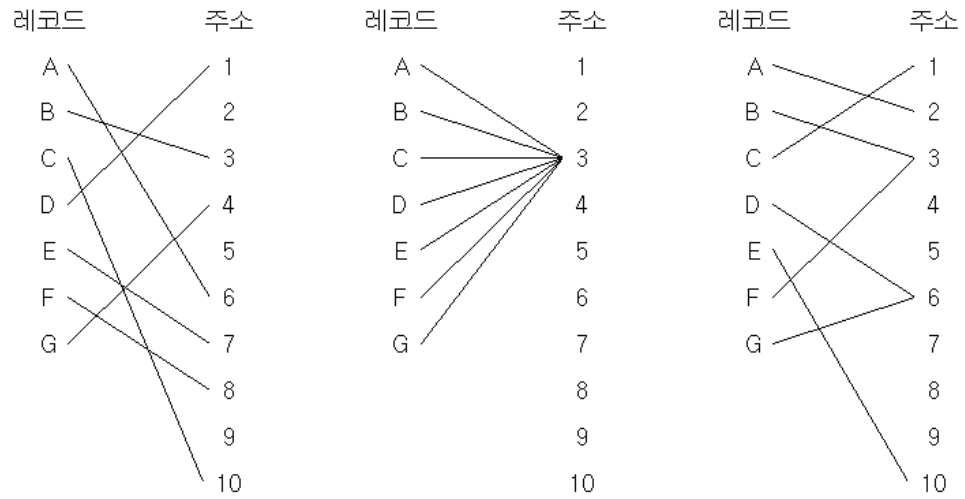
- 계산(computation)을 이용한 상대 화일 구성법

- 일반적으로,
  키 공간(key value space) >> 주소 공간(address space)
  - 주민등록번호(13자리)의 예
    - 가능한 키값의 수 : 10^{13}(개)
    - 실제 필요한 주소의 수 : 약 10^{8} (=1억)(개) 이하
    - 주민등록번호(키) 각각 마다 화일에 빈 레코드 주소를 할당한다면?
      → 막대한 공간 낭비 ∵ $10^{13} >> 10^{8}$
    - 1억개의 레코드 공간을 갖는 화일을 만드는 것이 효율적

- 해싱(hashing)
  - 해싱 함수(hashing function)을 이용하여 키 값을 주소 (hashed address)로 변환하고, 키에서 변환된 주소에 레코드를 저장하는 과정
  - 장점 : 레코드의 주소를 구해 직접 접근 → 빠른 접근 시간
    - cf) 순차 화일 – 레코드 탐색시간 ∝ 레코드 수

▶ 해싱

- **해싱 함수 (hashing function)**
  - 키 공간을 주소 공간으로 사상(mapping)
    ***h*(키 값) →주소, 주소 ⊂ 유효 주소 공간**
  - 키 값들을 한정된 주소 공간으로 균등하게 분산시키는 것이 핵심



(a) 최상의 해싱　(b) 최악의 해싱　(c) 보통의 해싱

▶ 해싱을 이용한 화일 설계시 고려사항

① 버킷(bucket) 크기 : 하나의 주소를 가진 저장 구역에 저장할 수 있는 레코드의 수
- 버킷: 파일에서 같은 주소에 포함될 수 있는 최대 레코드수

② 적재율 : 총 저장 용량에 대한 실제로 저장되는 레코드 수의 대한 비율
- $$\text{적재율} = \frac{\text{저장된 레코드수}}{\text{버킷의 총용량}}$$

③ 해싱 함수 : 레코드 키 값으로부터 주소(버킷 주소)를 생성하는 방법

④ 오버플로 해결 방법 : 주어진 주소 공간이 만원이 된 경우의 해결 방법

▶ 버킷 크기

- **버킷(bucket) : 하나의 주소를 가진 한 저장 구역**
  - 둘 이상의 레코드 저장 가능
  - 하나의 버킷 안의 레코드는 같은 버킷 주소를 가짐
  - 한 화일을 구성하는 구성하는 버킷 수가 그 화일의 주소 공간이 됨
- **버킷 크기**
  - 통상적으로 한 번의 접근으로 버킷 내에 있는 모든 레코드들을 전송할 수 있는 크기로 결정
  - 저장 장치의 물리적 특성과 연관
  - 1000개 레코드를 갖는 화일의 경우
    - 버킷크기=1, 주소공간=1000 (1000개 버킷)
    - 버킷크기=5, 주소공간=200 (200개 버킷)
- **충돌(collision)**
  - 두 개의 상이한 레코드가 똑같은 버킷으로 해싱되는 것
  - **동거자**(synonyms) : 같은 주소로 해싱되어 충돌된 키 값들
  - 버킷이 만원일 때는 충돌이 문제가 됨 → 오버플로(overflow)
- **버킷 크기를 크게하면**
  - 장점 : 오버플로 감소
  - 단점 : 저장 공간 효율 감소, 버킷 내 레코드 탐색 시간 증가

- 예
  - 화일
    - 레코드 크기 = 80 byte
  - 보조기억창치
    - 블록 크기 = 1024 byte (한번의 판독연산으로 1024 byte 채취)
  - 버킷 크기의 결정
    - 버킷 크기 =12개로하면
      한번의 판독으로 채취할 수 있는 버킷 크기 중 제일 큰 것이 됨
    - 버킷에 12개의 논리 레코드가 저장되고 나머지 64 byte는 예비

▶ 적재 밀도(loading density)

◆ 적재 밀도(loading density) (=패킹 밀도(packing density))

$$\text{적재 밀도} = \frac{\text{저장된 레코드 수}}{\text{화일 저장 공간의 총 용량}}$$

$$= \frac{\mathbf{K}}{\mathbf{c} * \mathbf{N}} < 1$$

- **N :** 버킷 수
- **c :** 버킷 크기 **(**한 버킷에 저장 가능한 레코드의 최대수**)**
- **K :** 화일에 저장된 레코드 수

- 적재 밀도 ↑
  → 삽입시 접근 수 ↑(∵ 이미 레코드가 저장된 주소에 해싱될 경우가 많음)
  → 검색시 접근 수 ↑(∵ 원하지 않는 레코드가 저장된 주소에 해싱될 경우가 많음)
- 적재 밀도 ↓
  → 공간 효율 ↓

- 실험적으로, (적재밀도 > 70%)일 때 충돌이 너무 잦음
  → 30% 정도의 예비 공간 필요

- 예) 학생 레코드 검색 시스템
  - 학생 수 최대 60000명, 예비 공간 30% (적재밀도=0.7), 버킷 크기 12
  - 버킷 수 = 60000 /12 x 10/7 = **7143**

▶ 해싱 함수(hasing function)

- **해싱함수(변환 함수) : 키 → 버킷 주소**
  - 해싱함수 계산시간 << 보조기억장치의 버킷 접근 시간
  - 모든 주소에 대한 균일한 분포를 가지는 것이 좋음

- **주소 변환 과정**

① 키가 숫자가 아닌 경우, 키를 정수 값(A)으로 변환

키 → A

② 변환된 정수 A를 주소공간의 자리 수 만큼의 다른 정수 B로 변환 (→ 해싱 함수)

A → B (B : 균일 분포로 변환)

③ 얻어진 정수 B를 주소공간의 실제범위에 맞게 조정

B × 조정상수 → 주소 (실제 목표 주소로 변환)

(1) 제산 잔여 해싱

① 주소 = 키 값 **mod** 제수 **(**몫은 제외하고 나머지만**)**

→ **0** ~ 제수**-1**

② 주소 범위에 맞도록 조정 **(**조정 상수 사용**)**

◆ 제수

– 직접 주소공간의 크기를 결정

◆ 주소공간: 0 ~ 제수-1

◆ 제수 > 화일의 크기

– 충돌 가능성이 가장 적은 것으로 선택

◆ 버킷수 보다 작으면서 제일 큰 소수(prime number)

◆ 20보다 작은 소수를 인수로 갖지 않는 비소수

▶ 제산 잔여 해싱의 성능

- 적당한 성능을 위한 적재율 최대 허용치는 **0.7 ~ 0.8**
  - n 개의 레코드 → 1.25n 주소 공간(80%의 경우)

- **(예) 레코드 4000개, 적재율 80%**
  - 주소 공간 = 4,000 / 0.8 = 5,000 개 주소공간 필요
  - 제수 : 5003
    - 20이하의 수를 인자로 갖지 않는 수 중 5000에 가장 가까운 수를 선택한 경우

제수 **5003**인 제산 잔여 해싱 예

키 값	주소	
123456789	2761	
987654321	2085	
123456790	2762	
555555555	2423	
000000472	0472	
100064183	4813	
200120472	0472	** 충돌 **
200120473	0473	
117400000	4605	
027000400	4212	

(2) 중간 제곱 (Mid-square) 해싱

① 키 값을 제곱하고
② 중간에서 **n**개의 수를 취함**:** 주소공간은 10^n
③ 주소 범위에 맞도록 조정 **(**조정 상수 사용**)**

◆ 예**)** 레코드가 **4000**개**,** 적재율 **80%**인 경우
- 최소 네 자리의 주소 공간 필요
- 키를 제곱한 수에서 4자리 수를 취함
- 주소공간 조정후, 주소공간의 실제 범위로 매핑 : (4000 / 0.8) * 0.5

중간 제곱 잔여 해싱 예 **(**뒤에서부터 **7~10**자리 수 취함**)**

키 값	키 제곱	주소	
123456789	15241578750190521	8750	
987654321	975461055789971041	5789	
123456790	15241578997104100	8997	
555555555	308641974691358025	4691	
000000472	000000000222784	0000	
100064183	10012860719457489	0719	
200120472	40048203313502784	3313	
200120473	40048203713743729	3713	
117400000	13782760000000000	0000	** 충돌 **
027000400	02430021600160000	1600	

(3) 중첩(Folding) 해싱

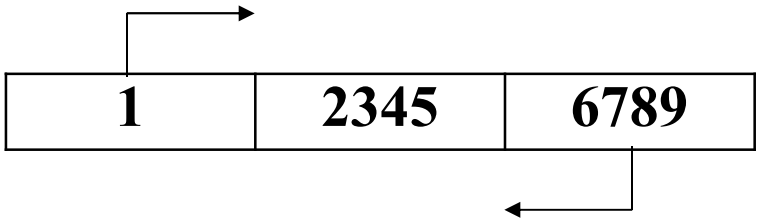
① 키 값을 주소공간과 같은 자리수(n자리수)를 가지는 몇 개의 부분으로 나눔

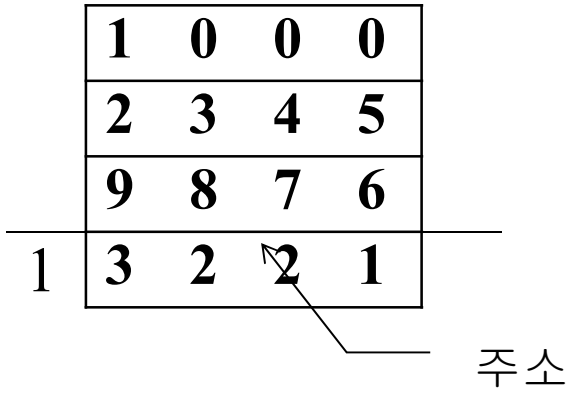
② 접어서 합을 구함 : 주소공간은 10^n

③ 주소 범위에 맞도록 조정 (조정 상수 사용)

키 값	주소	
123456789	2761	
987654321	2085	
123456790	2762	
555555555	2423	
000000472	0472	
100064183	4813	
200120472	0472	** 충돌 **
200120473	0473	
117400000	4605	
027000400	4212	

예) 주소 크기(4자리),
키 값(123456789)





(4) 숫자 분석 방법

- 키 값이 되는 숫자의 분포 이용

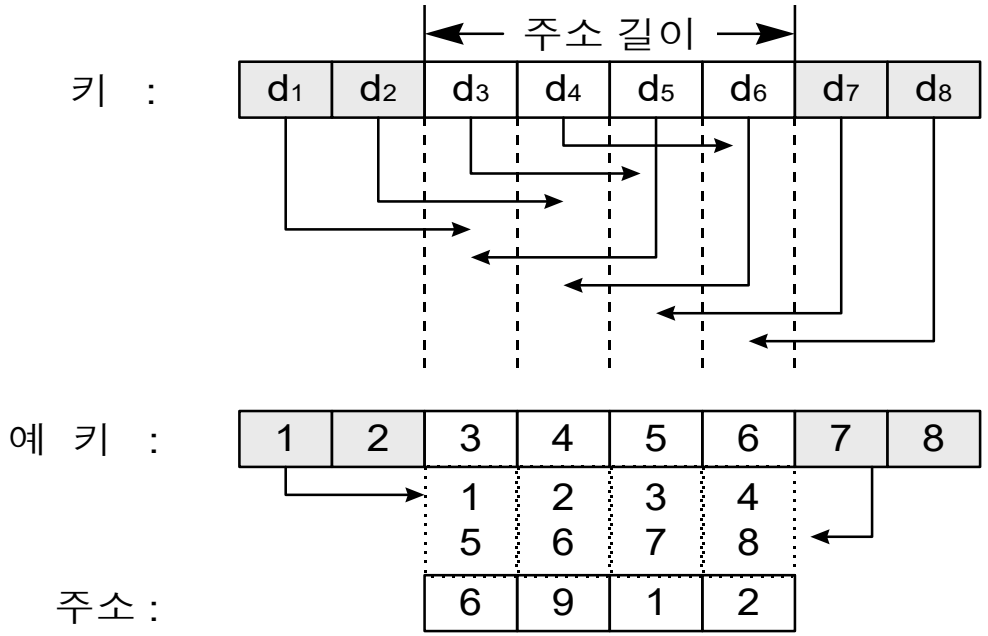
- 키들의 모든 자릿수에 대한 빈도 테이블을 만들고
- 어느 정도 균등한 분포를 갖는 자릿수를 주소로 사용

↖ 같은 빈도수

- 단, 키 값들을 미리 알고 있어야 함

(5) 숫자 이동 변환

① 키를 중앙을 중심으로 양분한 뒤
② 주소 길이(n자리수)만큼 겹치도록 안쪽으로 각각 **shift**
③ 주소 범위에 맞도록 조정 (조정 상수 사용)



◆ 주소 공간 : **5000**라면 주소 범위 조정 필요

6912 * 0.5 = 3456 : 실제 주소

(6) 진수 변환

① 키 값의 진수를 다른 진수로 변환
② 초과하는 높은 자리 수를 절단
③ 주소 범위에 맞도록 조정 (조정 상수 사용)

◆ 예)

172148 : 키 값
주소공간 : **7000**
진수 : **11**

$$1\times 11^5 + 7\times 11^4 + 2\times 11^3 + 1\times 11^2 + 4\times 11^1 + 8\times 11^0 = 266373$$

보정 : **6373 * 0.7 = 4461**

❖ 충돌과 오버플로

- (키 값 공간 >> 주소 공간) → 충돌 불가피
- 오버플로(overflow)
  - 하나의 홈 주소(home address)로 충돌된 동거자들을 한 버킷에 모두 저장할 수 없는 경우

- 해결 방법
  - 개방 주소법(open addressing) : 오버플로된 동거자를 저장할 공간을 상대 화일 내에서 찾아 해결
  - 체인법(chaining) : 오버플로된 동거자를 위한 저장 공간을 상대 화일 밖에서 찾아 해결, 즉 독립된 오버플로 구역을 할당

- 기법
  - 선형 조사(linear probing)
  - 독립 오버플로 구역(separate overflow area)
  - 이중 해싱(double hashing)
  - 동거자 체인(synonym chaining)
  - 버킷 주소법(bucket addressing)

※ 이하, 버킷 크기 = 1로 가정함

(1) 선형 조사

- 오버플로 발생시, 홈 주소에서부터 차례로 조사해서 가장 가까운 빈 공간을 찾는 방법
- 해당 주소가 공백인지 아닌지를 판별할 수 있어야 함 → 플래그(**flag**) 이용

```
insertLinear(key)
    addr ← h(key);
    home-addr ← addr;
    while (addr is full) do {
      addr ← (addr + 1) mod N;
      if (addr = home-addr ) {
        print ("file is completely full");
        return;
      }
    }
    insert the key at addr;
    set the addr is full;
end insertLinear()
```

▶ 선형 조사 이용시의 저장/검색/삭제

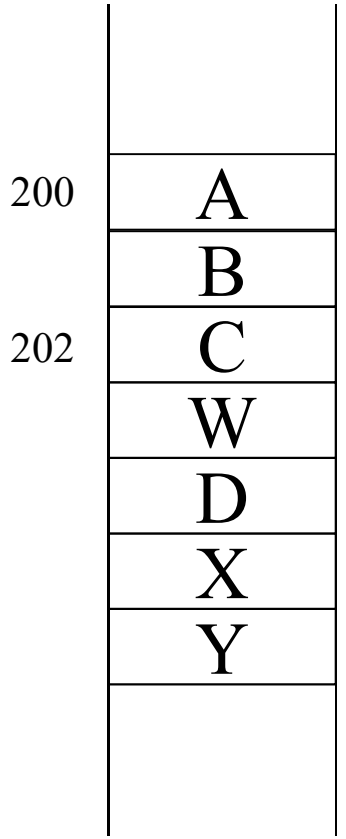
- 저장
  - 원형 탐색 : 빈 주소를 조사하는 과정은 홈 주소에서 시작하여 화일 끝에서 끝나는 것이 아니라 다시 화일 시작으로 돌아가 홈 주소에 다다를 때 끝남 (화일이 만원)

- 검색
  - 레코드 저장 시에 선형 조사를 사용했다면 검색에서도 선형 조사를 사용해야 함
  - 버킷에서 빈 공간이 발견되면 검색이 중단됨.
  - 탐색 키 값을 가진 레코드가 없거나 홈 주소에서 멀 경우, 많은 조사 필요

- 삭제
  - 삭제로 인해 만들어진 빈 공간으로 검색시 선형 조사가 단절될 수 있음 → 삭제 표시(tombstone) 이용 :
    - 삭제된 자리에 삭제 표시를 해서 선형 조사가 단절되지 않게 함

◆ 삭제 예

- 가정
  - A, B, C, D가 200에 해싱됨
  - W, X, Y가 202에 해싱됨
- 삽입
  - A, B, C, W, D, X, Y 순서로 삽입됨
- 삭제
  - 레코드 W가 삭제되면, 이후 오버플로우 레코드 D, X, Y에 대한 검색은 항상 실패하게 됨
- 해결책
  - 논리적 삭제 표시를 사용



▶ 선형 조사의 단점

(1) 어떤 레코드가 화일에 없다는 것을 판단하기 위해 조사해야 할 주소의 수 → 적재율이 커질수록 많아짐

- 적당한 적재율 유지 필요

(2) **환치(displacement)**

- 한 레코드가 자기 홈 주소를 동거자가 아닌 레코드가 차지함으로 인해 다른 레코드의 홈 주소에 저장되는 것
- 다른 환치를 유발
- 탐색할 주소 수 증가 → 삽입/검색 시간 증가를 야기
- 대응책 : 2-패스 해시 화일 생성(two-pass hash file creation)

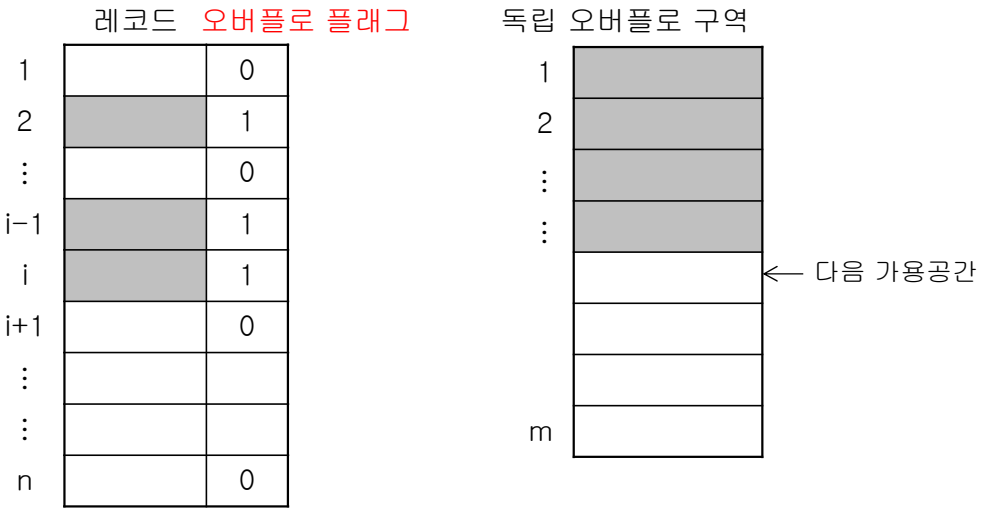
2-패스 해시 화일 생성(two-pass hash file creation)

- 첫 번째 패스
  - 모든 레코드를 해시 함수를 통해 홈 주소에 저장
  - 충돌이 일어나는 동거자들은 바로 저장하지 않고 별도로 임시 화일에 저장
- 두 번째 패스
  - 임시 화일에 저장해 둔 동거자들을 선형 조사를 이용해 화일에 모두 적재

- 1-패스 생성에 비해 훨씬 많은 레코드들이 원래의 자기 홈 주소에 저장됨
- 화일 생성 이전에 레코드 키 값들을 미리 알 때 효율적
- 화일이 생성된 후에 레코드들이 추가될 때는 환치 발생

(2) 독립 오버플로 구역(separate overflow area)

- 별개의 하나의 오버플로 구역을 할당하여 홈 주소에서 오버플로된 "모든" 동거자들을 순차로 저장하는 방법
- 장점
  - 동거자가 없는 레코드에 대해서는 한번의 홈 주소 접근만으로 레코드를 검색
  - 1-패스로 상대 화일을 생성
  - 환치 문제를 제거
- 단점
  - 오버플로된 동거자를 접근하기 위해서는 오버플로 구역에 있는 모든 레코드들을 순차적으로 검색



(3) 이중 해싱(double hashing)

- 오버플로된 동거자들을 오버플로 구역으로 직접 해시
  - 오버플로 구역에서의 순차 탐색을 피할 수 있음
  - 2차 해시 함수(second hash function)
    - 오버플로된 동거자를 해시하는 함수

- 해싱 과정
  - 1차 해시 함수에 의해 상대 화일로 해시
  - 오버플로가 발생하면, 오버플로 구역으로 해시
    - 오버플로 구역 주소
      = (1차 해시 함수값 + 2차 해시 함수값) mod (오버플로 구역 크기)
  - 또 충돌이 일어나면, 선형 탐색 이용

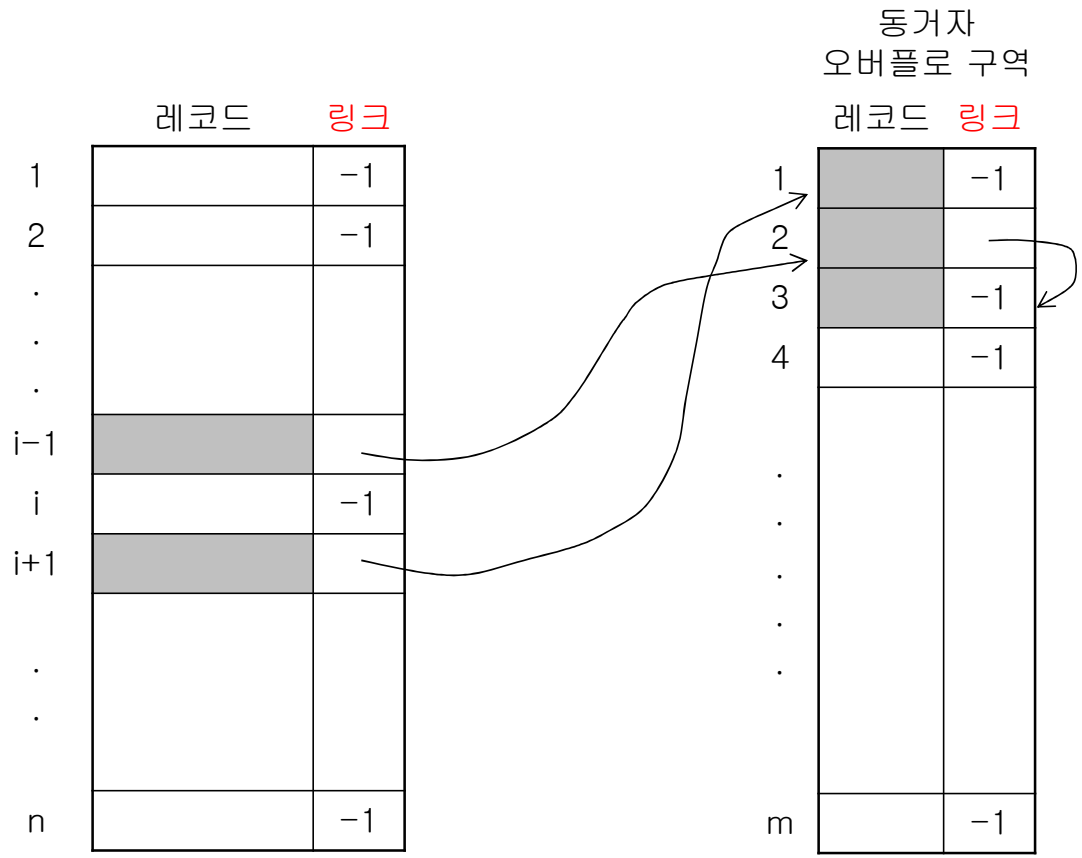
(4) 동거자 체인(synonym chaining)

- 각 주소마다 링크를 두어 오버플로된 레코드들을 연결하는 방법
  - 오버플로가 일어나면 선형 조사나 오버플로 구역을 이용해서 저장후, 처음 해시 주소에 동거자를 포함하고 있는 주소에 대한 링크를 둠 (record chaining)
  - 동거자에 대한 접근은 연결된 동거자들만 조사해 보면 됨

- 독립 오버플로 구역에 사용할 수도 있고 원래의 상대 화일에 적용할 수도 있음
- 장점 :
  - 홈 주소에서의 충돌 감소
  - 환치 제거(독립 오버플로 구역 사용시)
- 단점 :
  - 각 주소가 링크 필드를 포함하도록 확장해야 함

▶ 동거자 체인 예

◆ 동거자 체인 + 독립 오버플로 구역 예



▶ 오버플로 처리 기법들의 성능 비교

- 선형 조사, 이중 해싱, 동거자 체인 비교

- 성공적 탐색 성능(평균 조사 수가 적은 순)
  - 동거자 체인 > 이중 해싱 > 선형 조사

- 실패 탐색
  - 동거자 체인이 월등

(5) 버킷 주소법

- 해싱 함수가 주어진 키 값으로부터 생성한 주소가 어느 특정 버킷이 되도록 사상하는 것
  - 하나의 해시 주소에 가능한 최대 수의 동거자를 저장할 수 있는 공간을 할당
  - 특정 해시 주소에 대한 모든 동거자들은 그 특정 해시 주소의 버킷에 순차로 저장

  → 한 레코드를 검색하기 위하여 조사해야 될 레코드 수는 최대로 버킷 사이즈에 한정 (오버플로 구역 탐색, 화일 전체 탐색 불필요)

- 문제점
  - 공간의 낭비 : 각 해시 주소에 대한 동거자의 수가 다양하고 그 차이가 아주 클 때
    - 버킷 사이즈는 해시 주소에 대한 최대 동거자 수로 정하는 것이 보통 → 낭비된 공간 ↑
  - 버킷 사이즈 설정
    - 화일 생성 전에 데이타를 분석할 수 없을 때, 설정이 어려움
    - 사이즈가 충분치 않으면 오버플로 발생 → 충돌 해결 기법 필요

▶ 버킷 주소법에서의 충돌 해결

- 여유 공간을 가진 가장 가까운 버킷을 사용하는 방법
  - 선형 조사법과 똑같은 단점

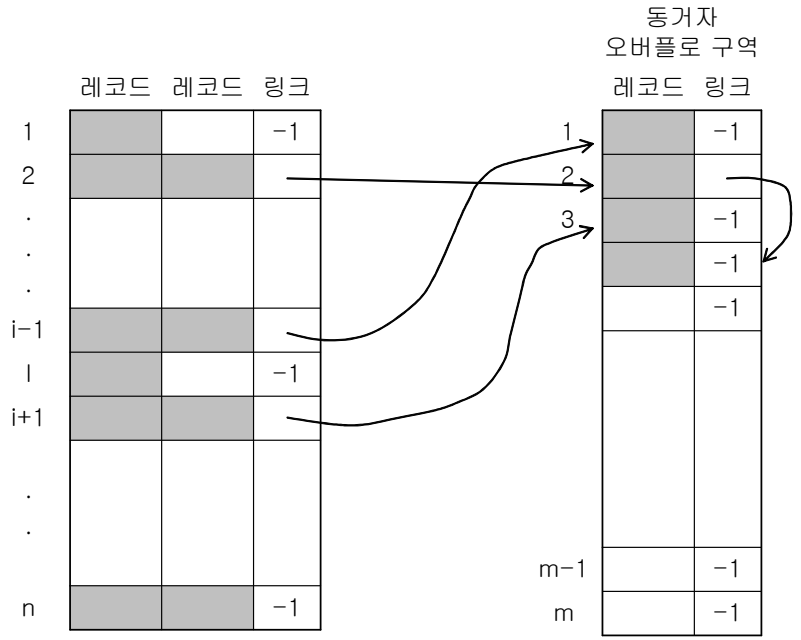
- 버킷 체인(**bucket chaining**)
  - 홈 버킷에서 오버플로가 일어나면 별개의 버킷을 할당받아 오버플로된 동거자를 저장하고 홈 버킷에 이 버킷을 링크로 연결

  - 장점 : 재해싱 불필요
  - 단점 : 한 레코드를 탐색하기 위해서는 최악의 경우 그 홈 버킷에 연결된 모든 오버플로 버킷을 조사해야 함

▶ 버킷 주소법의 성능

- ◆ 성공적 탐색
  - 어떤 다른 방법보다도 평균 조사 수가 더 작음
- ◆ 실패 탐색
  - 성공적 탐색과 비슷하거나 더 작은 조사 수

  (독립 오버플로 구역과 버킷 주소법을 사용하는 경우)



버킷 체인을 사용하는 버킷 주소법의 예

❖ 테이블 이용 해싱

- 보조기억장치에 한번의 접근으로 검색을 보장하는 방법
  - 해싱 테이블 : 키 → <버킷 주소, k-비트 시그너쳐(signature)> 생성
  - 디렉터리 테이블 : <버킷 주소, k-비트 시그너쳐>
  - 해싱 테이블과 디렉터리 테이블 : 주기억장치에 유지
  - 레코드 삽입은 시간이 많이 소요되나, 검색은 매우 빠름

(예) k=5일 때 버킷 주소와 시그너쳐의 순열

키	버킷 주소	5 비트 시그너쳐
White	85 87 89 91 93 …	00101 01001 10100 10111 …
Blue	85 86 87 88 89 …	00110 00011 00110 10000 …
Lilac	85 90 95 0 5 …	01000 10100 11000 10100 …
Red	85 92 99 6 13 …	00010 11000 11110 10010 …
Green	85 86 87 88 89 …	00011 00100 10001 00111 …

▶ 테이블 이용 해싱을 이용한 삽입

- (예) 버킷 크기 = **3**, 각 레코드의 홈 버킷 = **85**, 레코드 삽입 순서 = **White, Blue, Lilac, Red, Green.**

- 네번째 **RED** 삽입할 때 : 오버플로가 생김
  - RED (시그너쳐 00010 = 2)
  - WHITE (시그너쳐 00101 = 5)
  - BLUE (시그너쳐 00110 = 6)
  - LILAC (시그너쳐 01000 = 8) → 시그너쳐값
    - RED, WHITE, BLUE는 85번 버킷에 저장
    - LILAC은 90번 버킷에 저장

- 레코드를 버킷에 저장할 때마다 디렉터리 테이블 유지
  - 엔트리 = (버킷 번호, 시그너쳐 값)
    - 시그너쳐 값은 초기치(11111)에서 버킷 오버플로 경우만 변경
  - RED, WHITE, BLUE는 버킷 85에 저장 = (85, 01000)
  - LILAC 은 버킷 90에 저장 = (90, 11111)

▶ 테이블 이용 해싱을 이용한 검색

◆ **검색 : 아주 효율적임, log-in용 ID 화일조직에 효율적임**

– 검색할 키로부터 다음 정보를 생성함
  버킷1(85), 시그너쳐1(101)
  버킷2(87), 시그너쳐2(1001)
  버킷3, ... 시그너쳐3, ...

– 다음을 만족하는 가장 작은 i
  시그너쳐i < 버킷 i의 분리값, i = 1, 2, 3, …

– 검색 레코드는 버킷 i에 존재

(예) White 검색 시, 버킷 주소와 시그너쳐 순열 테이블 참조함

85,	87,	89,
00101,	01001,	10100,

시그너쳐 i(101) < 테이블 버킷 85(01000)
따라서, 검색할 버킷 번호 : 85

❖ 확장성 직접 화일

- **Span**
  - 어느 한 시점에서 파일에 저장된 레코드수, K
    $$K_{min} < K < K_{max}$$
  - $Span = K_{max} / K_{min}$

- **Span**이 해싱에 미치는 영향
  - Span이 작을 경우: 레코드수가 고정된 고정길이 파일에서의 해싱
    - 설계시 버켓 크기와 적재밀도의 선택이 가능
  - Span이 클 경우
    - $K \rightarrow K_{min}$: 적재밀도가 낮아짐 (저장 공간의 이용률이 낮아짐)
    - $K \rightarrow K_{max}$: 적재밀도가 높아 저장 및 검색 시간이 길어짐.
  - Span이 클 경우 재해싱이 가능
    - 재해싱에 시간이 걸리고, 재해싱 동안 접근 곤란

- 확장성 파일(**extensible file**): **Span**이 큰 경우 해결방안
  - 가상 해싱
  - 동적 해싱
  - 확장 해싱

- 가상 해싱, 동적 해싱, 확장 해싱의 공통점
  - 버켓의 수가 레코드 수에 따라 동적으로 변함.
  - 오버플로우 버켓이 없음.
  - 삭제가 간단함.
  - 검색은 한두번의 디스크 접근으로 가능함.
  - 버켓 크기 C는 일정.

- 확장성 직접 파일에서의 문제
  - 버켓이 다 차면 분할하는 것
  - 그전 버켓과 새 버켓에 레코드를 분산하는 것 (rehashing)
  - 검색시에 버켓 접근을 최소화하는 것
  - 파일로부터 레코드를 물리적으로 제거하는 것

(1) 가상 해싱(Virtual hashing)

- 여러 개의 관련된 해쉬함수 사용
- 제산 잔여기법 사용
  - N : 버켓의 수, C : 버켓 크기

$$h_0: \text{주소} = \text{키 mod } N$$

- 오버플로우 발생시 재해쉬 (버킷 분할)
  - 관련된 버켓을 다음 해쉬함수로 재해쉬하여 버킷을 분할

$$h_j: \text{주소} = \text{키 mod } (2^j * N),\ j = 0,1,2,....$$

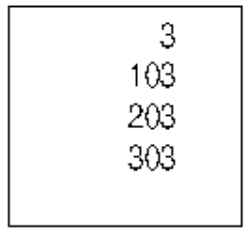
즉

$$h_1: \text{주소} = \text{키 mod } 2N$$
$$h_2: \text{주소} = \text{키 mod } 4N$$

  - 재해쉬는 C+1개의 레코드에 대해 이루어짐
    - 오버플로우된 현재 버켓의 C개 레코드
    - 삽입하려는 레코드

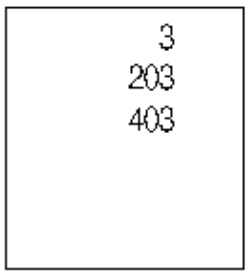
▶ 가상 해싱(Virtual hashing) 예

- **(예) 버킷 수 N = 100, 버킷 크기 C = 4**
  - 버킷 3 = [3, 103, 203, 303]으로 만원
  - **첫 번째 해싱 함수 h_0 : 주소 = 키 mod 100**

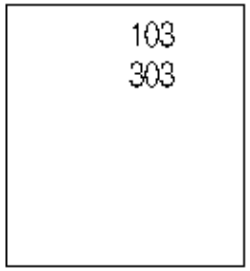


버킷 3

→ 새 레코드 403 삽입 시 오버플로 발생
다음 해싱 함수 h_1 : 주소 = 키 mod 200 를 이용해 버킷 분할

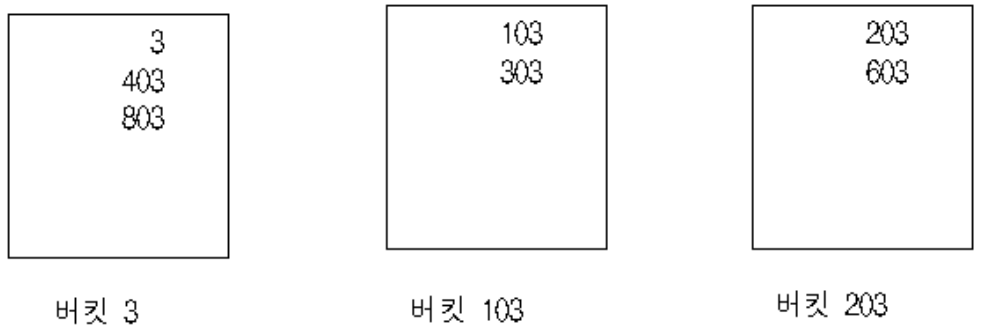


버킷 3



버킷 103

→ 새 레코드 603, 803 삽입시, 버킷 3 오버플로 발생
다음 해싱 함수 h_2 : 주소 = 키 mod 400



※ 각 키값에 적용된 해싱 함수를 유지해야 함
→ 레코드 삽입시 사용한 해쉬 함수 h_k 를 기록
→ 레코드 검색시 그 기록을 사용: structure graph 유지

(2) 동적 해싱(Dynamic hashing)

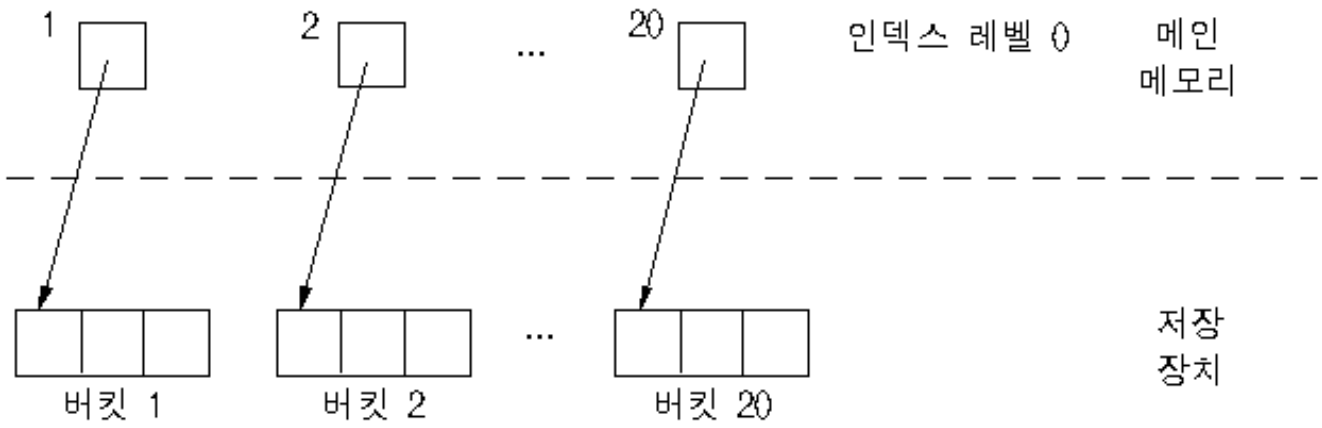
- 두 단계의 조직 사용
  - 인덱스: 주 기억 장치
  - 버켓: 보조 저장 장치
- 기본 개념
  - 해쉬된 주소로서 인덱스를 사용
    - N개의 인덱스 (초기 N개 홈버켓에 대응)
  - 각 인덱스는 이진 트리를 가리킴
  - 각 버켓은 이진 트리의 리프 노드에 의해 지시됨
- 두 개의 해쉬 함수 사용
  - H_0(키)
    - 인덱스 레벨 = 0인 경우
    - 적절한 인덱스 트리를 식별하는데 사용 ($1 \leq H_0$(키) $\leq N$)
  - B(키)
    - 인덱스 레벨 > 0인 경우
    - 선택된 인덱스 트리의 분기를 결정하는데 사용
- 인덱스
  - N개의 이진트리의 포리스트(forest)
  - 인덱스 노드
    - 내부노드: (0, 부모, 왼쪽 자식, 오른쪽 자식)
    - 외부(리프)노드: (1, 부모, 레코드 수, 버켓 포인터)

▶ 동적 해싱에서의 삽입

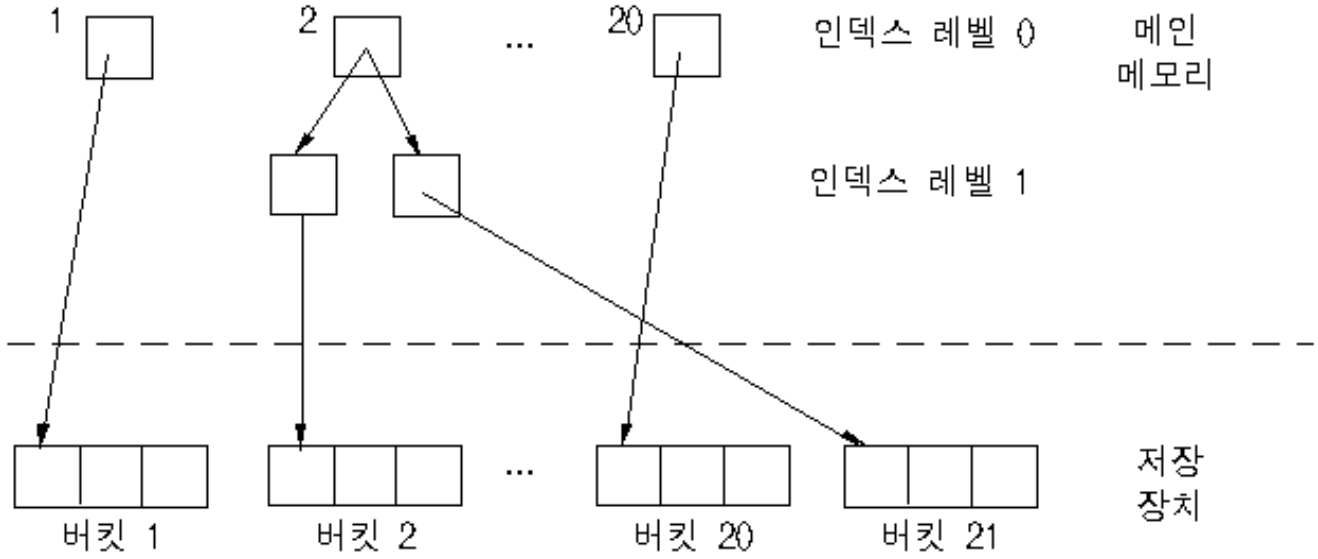
- 저장 절차
  - 키를 레벨 0 인덱스 엔트리의 한 주소로 변환
    - 해싱함수 H_0 이용
  - 인덱스 엔트리의 포인터를 통해 버킷에 접근한 후 레코드 저장
    - 1 이상의 인덱스 레벨이 있을 경우 함수 B를 이용해 저장할 버킷을 찾음
    - 분할 되는 버켓이 레벨 i이면 B(키)의 i+1째 비트를 사용
      - 값이 0이면, 왼쪽(이전) 버켓
      - 값이 1이면, 오른쪽(새로운) 버켓
  - 버킷이 오버플로우인 경우
    - 버킷을 새로 할당하여 “이미 저장된 레코드들 + 새로 저장할 레코드”를 재해싱하여 분할 저장
  - 해당 인덱스를 이진 트리로 바꾸어 두 버킷을 지시하게 함

▶ 오버플로 처리: 버켓 분할

- 가정
  - N=20, C=3
- 기본 개념
  - 인덱스로 사용되는 이진 트리에서의 분할 (버켓 분할)
  - 버켓은 언제나 새로 생성되어 추가 가능
- 버킷 분할 전



◆ 버킷 분할 후



▶ 동적 해싱 화일의 삽입 예

◆ 버킷 분할 방법

– 분할되는 버킷이 레벨 i이면 B(키)의 i+1째 비트를 사용

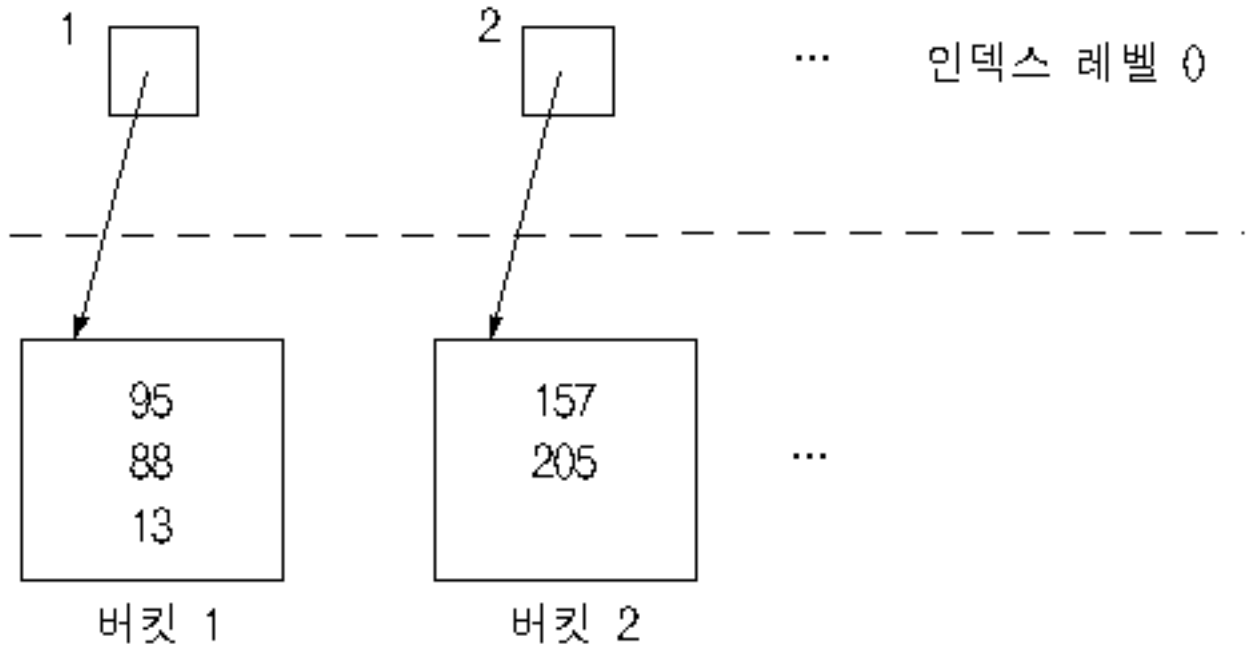
∴ 0 왼쪽(이전) 버킷

1 오른쪽(새로운) 버킷

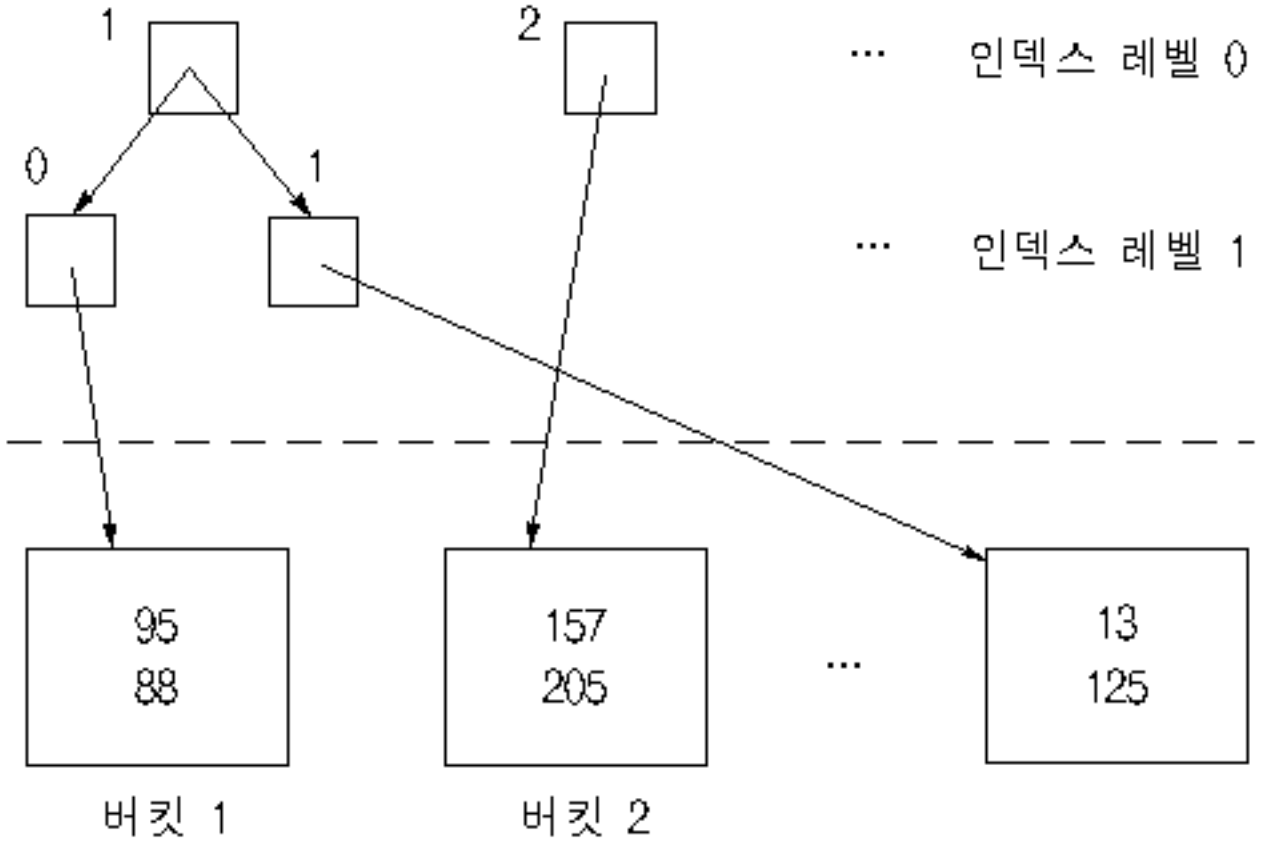
- 가정 : N=20, C=3

키	H_0(키)	B(키)
157	**2**	**10100 …**
95	**1**	**00011 …**
88	**1**	**01100 …**
205	**2**	**10010 …**
13	**1**	**10111 …**
125	**1**	**10001 …**
6	**1**	**01000 …**
301	**1**	**00110 …**

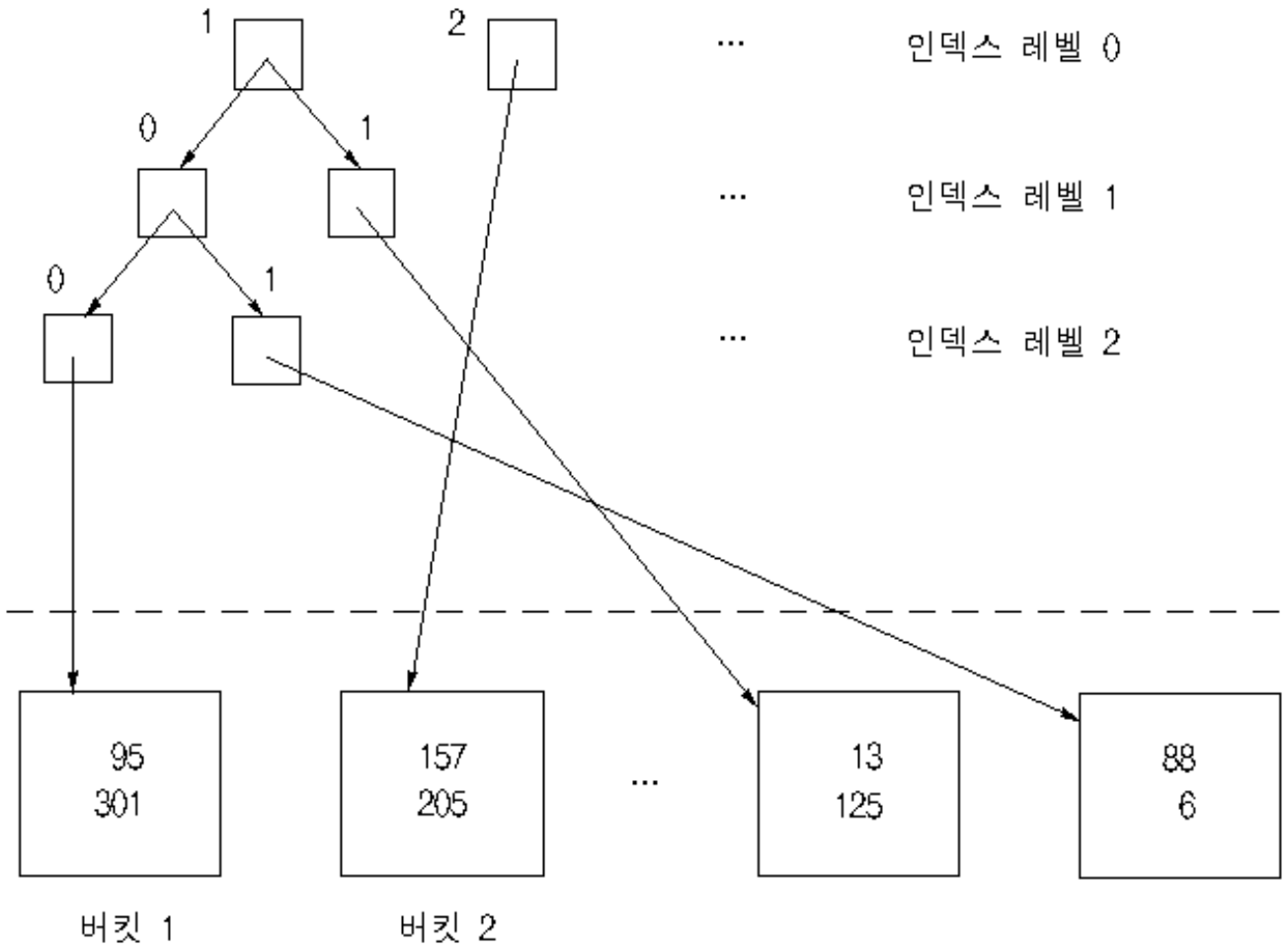
(예) 레코드 **157, 95, 88, 205, 13**을 삽입 후의 초기 화일



(예) 레코드 **125**를 추가 삽입하여 버킷 분할 후의 화일

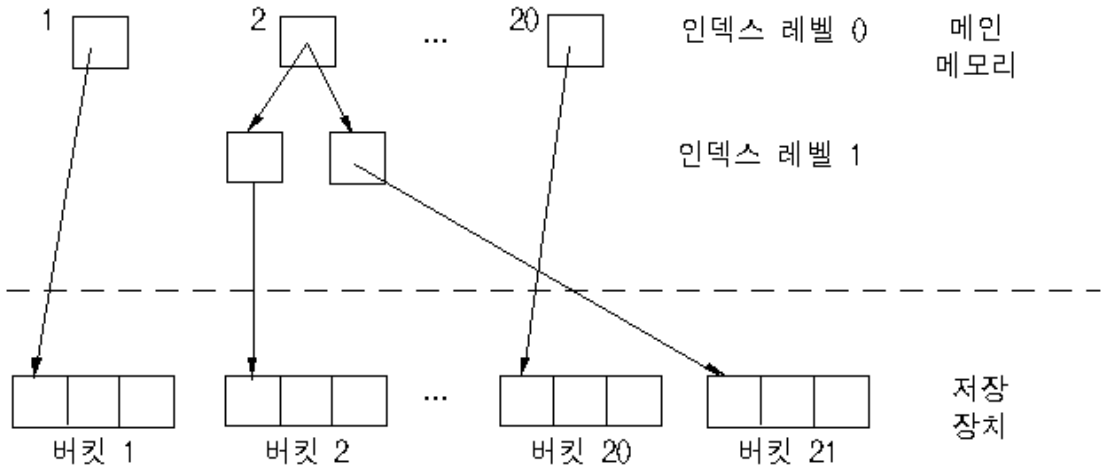


(예) 레코드 **6, 301**을 추가 삽입한 후의 화일



▶ 동적 해싱에서의 검색

- 검색 절차
  - H_0(키) 함수에 의한 값 (1~N)으로 인덱스 레벨 0 의 forest에서 해당 트리를 식별하고
  - B(키) 함수에 의한 비트 스트링 값으로 인덱스 레벨 1부터 분기를 결정



▶ 동적 해싱의 고려사항

- 함수 **B**의 설계
  - ① 레코드의 키(또는 키에 대한 어떤 함수 H_1)를 모조 난수 생성기(**pseudo-random number generator**)의 초기 값으로 사용하는 방법
    - 생성된 각 정수를 하나의 비트로 변환
  - ② 이 정수의 이진 표현에 대한 패리티 비트를 사용하는 것

- 인덱스 노드들이 메인 메모리에 광범위하게 되는 경우 → 트리 순회 과정에서 페이지 부재(page fault) 발생 – 추가적인 저장 장치 접근
- 인덱스 포리스트가 커져서 하위 레벨 인덱스가 저장장치에 저장되어야 하는 경우 → 레코드 접근 시간이 오히려 증가

※ 개선책 : 오버플로 버킷을 두어 버킷 분할 지연
– 저장 공간 효율 ↑

(3) 확장성 해싱(Extendible hashng)

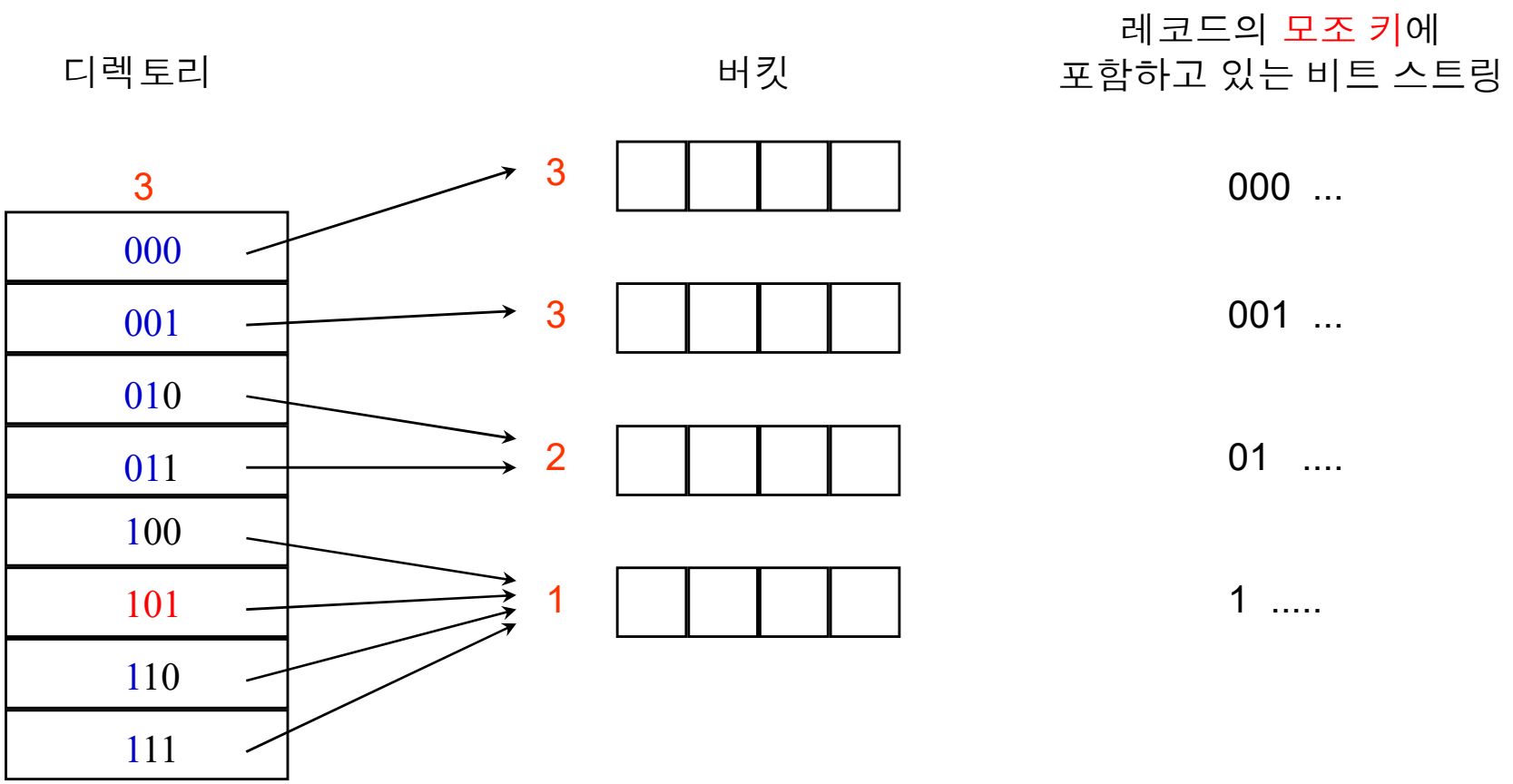
- 두 단계의 조직 사용
  - 디렉토리(directory): 인덱스에 해당
    - 전역깊이(global depth)를 의미하는 정수값 d 을 갖는 헤더와,
    - (리프) 버켓에 대한 2^d개의 포인터로 구성됨.
    - $d = \lceil \log_2 (N-1) \rceil + 1$
  - (리프) 버켓의 집합: 레코드들이 저장되는 공간
    - 지역깊이(local depth)를 의미하는 정수값 p를 갖는 헤더를 가지는 N개의 버켓

- 하나의 해쉬함수 사용
  - 확장성 해쉬함수
    - h : 키 → 모조키(pseudokey)
    - 모조키: 임의의 고정 길이 비트 스트링
  - 리프 버켓의 헤더
    - 버켓에 있는 레코드들의 모조키들이 공통으로 갖는 전위 비트의 개수

- 초기 화일
  - N개의 버킷일 때,
  - $d = \lceil \log (N-1) \rceil + 1$, 디렉터리 = 2^d개의 엔트리 가짐

▶ 확장성 해싱의 검색

- 검색
  - 모조 키의 처음 d비트를 디렉토리에 대한 인덱스로 사용하여, 디렉토리에서 버킷을 찾아 레코드를 접근
- 예
  - 가정
    - $N=4,\ \ d = \lceil \log_2 (4-1) \rceil + 1 = 3,\ \ 2^d = 2^3 = 8$
    - 키 K의 모조키: $h(K) = 101 0000 0100 0010$
  - d = 3이므로 모조키의 처음 3 비트를 사용하여, 디렉토리의 6번째 엔트리(포인터가 101)에 접근


디렉토리
버킷
레코드의 모조 키에
포함하고 있는 비트 스트링
3
000
001
010
011
100
101
110
111
3
3
2
1
000 ...
001 ...
01 ....
1 .....

▶ 확장성 해싱 화일에서의 삽입

- 삽입
  - 모조키의 처음 d비트 이용, 디렉터리 접근, 버킷 찾아 삽입
  - 만약, 버킷이 오버플로우이면

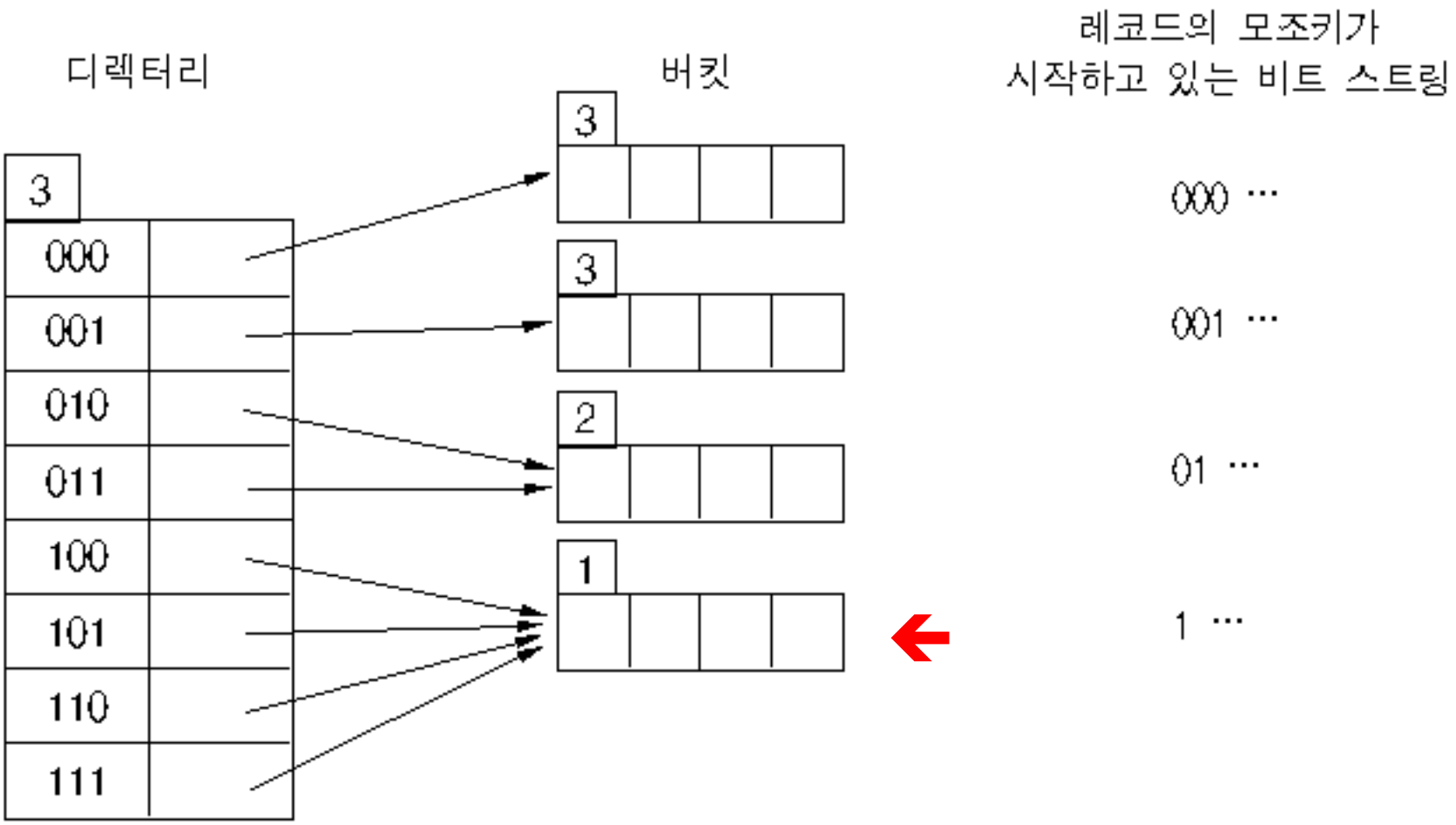
① 새로운 버킷 할당

② 오버프로우된 버킷의 레코드와 삽일할 레코드를 새로 할당된 버킷으로 재해싱
  - 버킷의 지역 깊이가 p이면,
    모조키의 (p+1) 번째 비트의 값에 따라 C+1개의 레코드를 분할

③ 두 버킷의 지역 깊이를 p+1로 조정

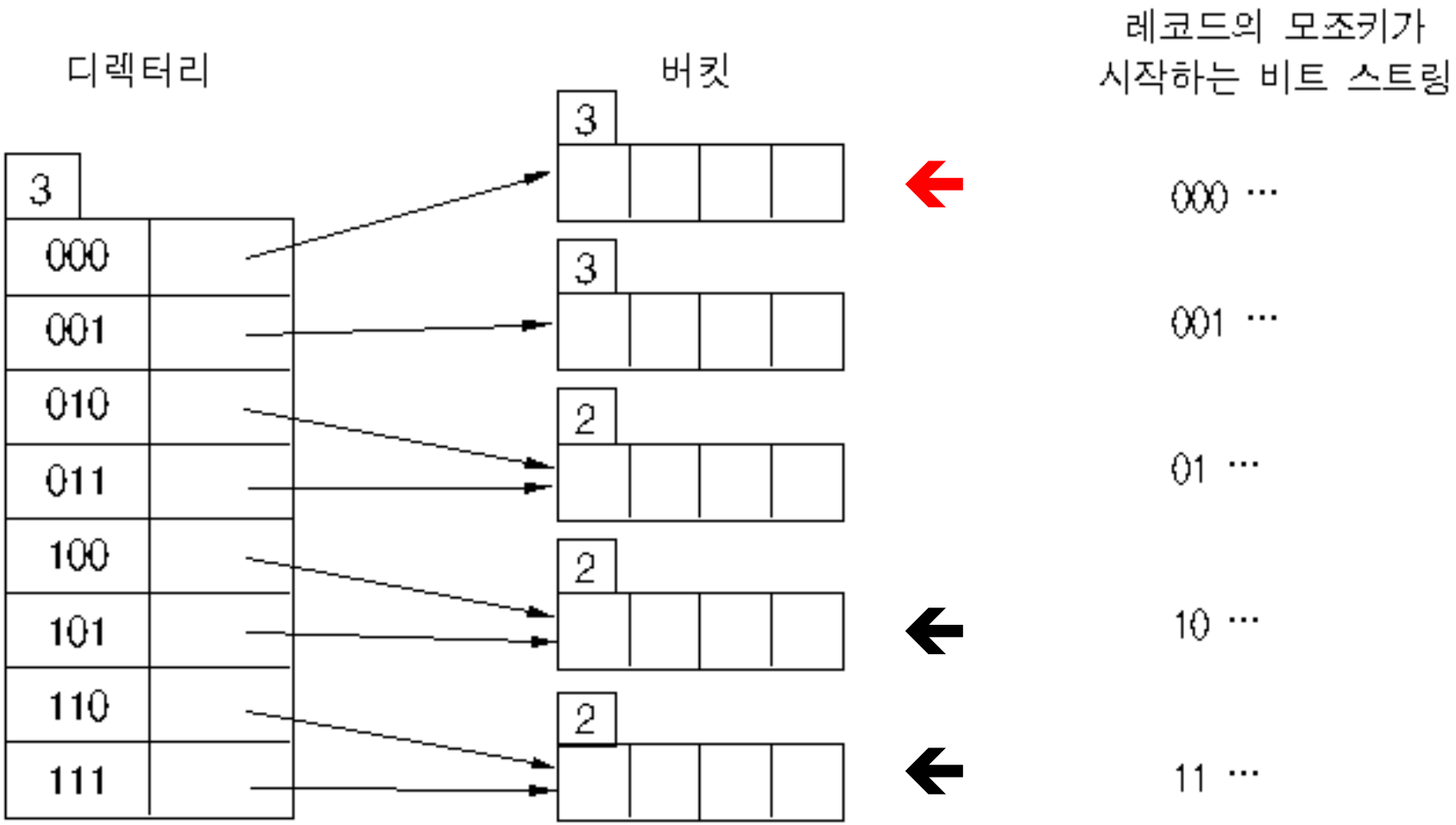
④ d, p+1의 값에 따라
  - $d \geq p+1$
    - 분할된 두 버킷에 대한여만, 해당 디렉토리 엔드리의 포인터들을 분할
  - $d < p+1$
    - 디렉터리의 크기를 두 배로 늘림, 즉 $d \leftarrow d+1$ 로 조정.
      따라서 디렉토리의 모든 포인터를 변경 (d+1개 비트로 만들기 위해 모든 포인터의 뒤에 0과 1을 각각 추가)

▶ 삽입 예

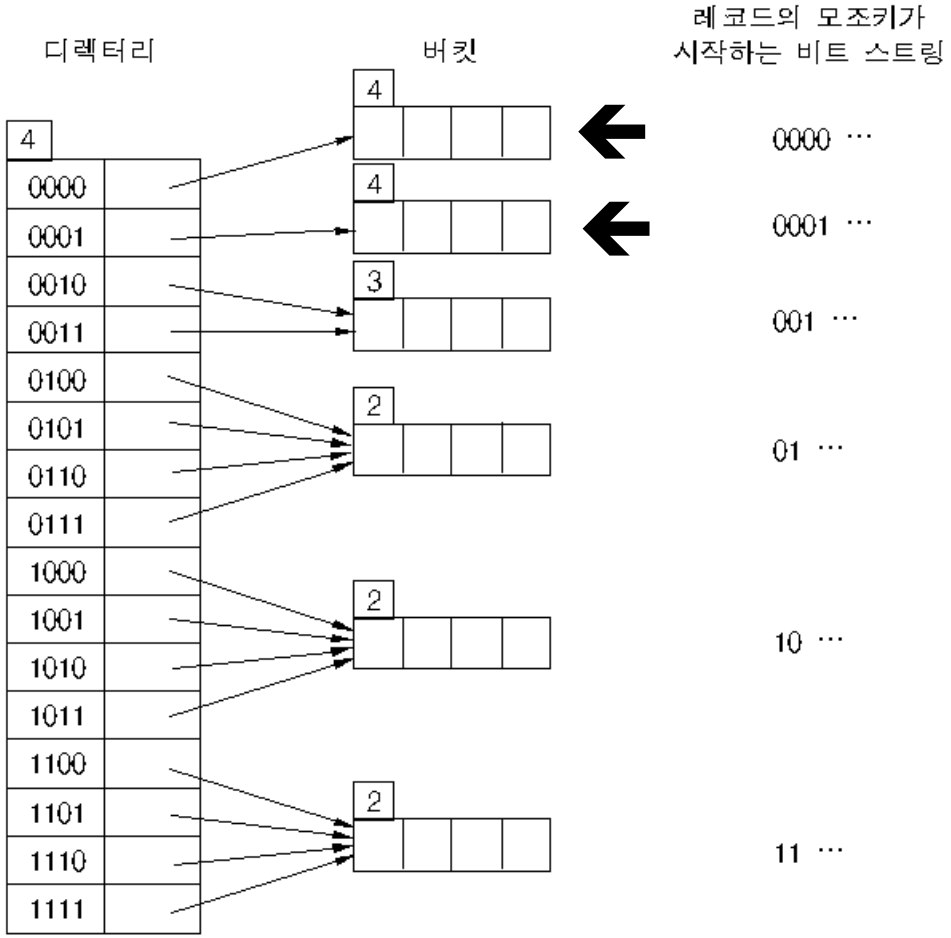

디렉터리
버킷
레코드의 모조키가
시작하고 있는 비트 스트링
3
000
001
010
011
100
101
110
111
3
3
2
1
000 …
001 …
01 …
1 …

◆ 버킷 4에서 오버플오우가 발생: 버킷 분할

– d ≥ p+1 인 경우 (즉 d=3, p=1)



- 버킷 1에서 오버플오우가 발생: 버킷 분할 및 디렉토리 확장
  - d < p+1 인 경우 (즉 d=3, p=3)



▶ 확장성 해싱 화일에서의 삭제

- **Buddy bucket**
  - 똑같은 지역깊이 p를 가지며,
  - 모조키의 처음 (p-1) 비트들이 모두 같은 버킷

- 삭제
  ① 삭제할 레코드를 제거하고, 나머지 레코드를 정돈
  ② 삭제된 레코드가 저장되었던 버킷이 버디 버킷을 가지고 있고, 그 두 버킷의 레코드들이 한 버킷 안에 들어갈 수 있다면
    - 하나의 버킷으로 병합 (빈 버킷 회수)하고, 병합된 버킷의 지역깊이를 p-1로 조정
    - 이때 모든 버킷의 지역깊이가 디렉토리의 전역깊이 보다 작으면, 디렉토리의 크기를 반으로 줄이고(d=d-1), 디렉토리 엔트리의 포인터들을 재조정함.(포인터의 뒤쪽에서 한 비트를 삭제)

▶ 선형 해싱(linear hashing)

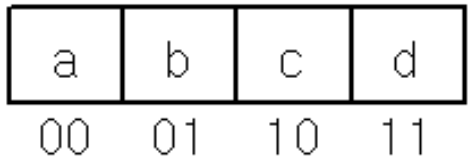
- 디렉터리를 사용하지 않는 확장성 해싱 방식

- 주소 공간이 확장될 때 해시 값의 비트(bits)를 사용
  - e.g. 주소 공간이 4 → 2비트 주소를 생성하는 해싱 함수 이용

  ※ 주소 공간이 4개의 버킷으로 구성된 것이지 4개의 디렉터리 노드가 버킷을 가리키는 것이 아님에 주의

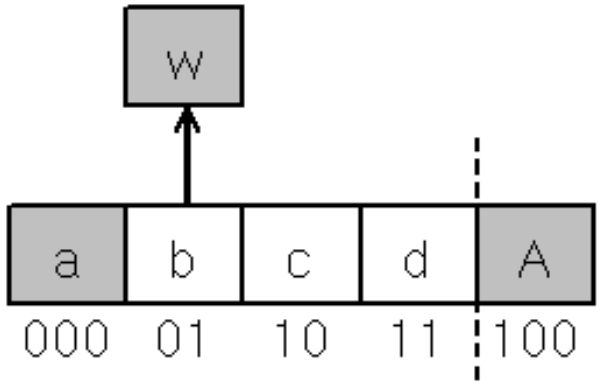
▶ 선형 해싱 예

◆ 초기 상태



◆ 삽입 과정 중 버킷 **b**에 오버플로 발생

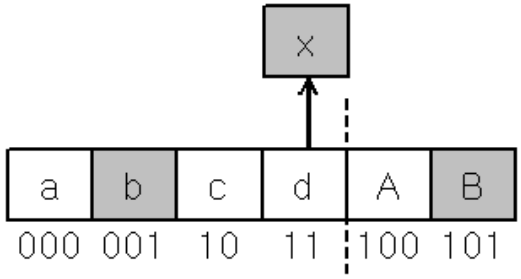
- 첫 버킷인 버킷 **a**를 분할, 새 버킷 **A** 할당
- 오버플로 레코드들은 오버플로 체인 **w**을 할당받아 저장
- 버킷 **a**와 새 버킷 **A**의 주소는 3비트로 바꿈



◆ 버킷 **d**에 오버플로 발생

– 두번째 버킷인 **b**를 분할, 새 버킷 **B**를 할당

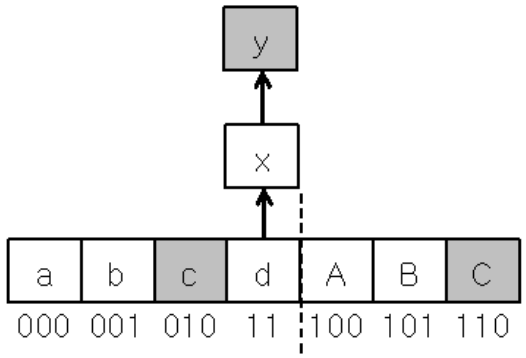
– 오버플로 버킷 **x** 생성



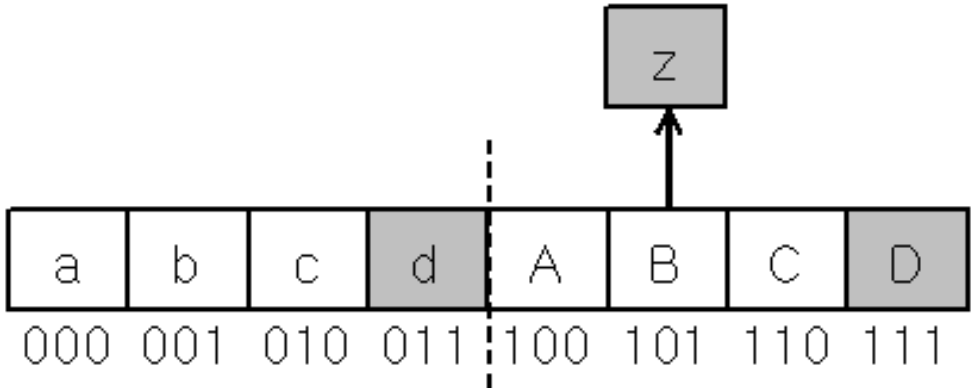
◆ 버킷 **x**에 오버플로 발생

– 세번째 버킷인 버킷 **c**를 분할, 새 버킷 **C**를 할당

– 오버플로 버킷 **y** 생성



- 버킷 **B**에 오버플로 발생
  - 네번째 버킷인 버킷 **d**를 분할, 새 버킷 **D**를 할당
  - 오버플로 버킷 **z**를 생성
  - 다음에 분할될 버킷을 가리키는 포인터는 다시 버킷 **a**를 가리킴
  - 모든 버킷이 3비트 해싱 함수를 사용하게 됨
    - 새로운 버킷을 생성하여 접근하기 위해서는 4-비트 해싱 함수 $h_4(k)$를 이용하는 새로운 사이클에 들어갈 준비가 된 것



▶ 선형 해싱에서의 검색

- 두 개의 해싱 함수 사용
  - 현재 주소 길이를 가진 버킷에 대해서는 $h_d(k)$ 사용
  - 확장 버킷에 대해서는 $h_{d+1}(k)$를 사용

  → 어떤 레코드의 검색을 위해 어떤 해싱 함수가 사용되는지 알아야 함

- **키 값 k를 가지 레코드를 포함하고 있는 버킷의 주소(address)를 찾기 위한 프로시저**

```
if (h_d(k) ≥ p)
    address = h_d(k);
else
    address = h_{d+1}(k);
```

p : 다음에 분할될 버킷을 가리키는 포인터

▶ 선형 해싱의 특징

- 접근하고 유지해야 될 디렉터리가 없음
- 오버플로 체인은 많이 길어지지 않음
  - 오버플로가 일어날 때 마다 분할을 통해 주소 공간을 확장하기 때문
- 버킷 크기를 **50**이라고 할 때 한번 검색 연산에 필요한 평균 디스크 접근 수는 거의 **1 (**실험 결과**)**
- 저장 공간 활용도는 확장성 해싱이나 동적 해싱보다 낮음
  - 평균 60%