# 8장 직접 화일

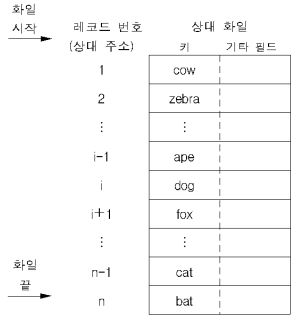
## 직접 화일의 개념

* 임의 접근 화일 = 직접 화일 = 직접 접근 화일
  + 다른 레코드의 참조 없이도 개개 레코드를 접근 할 수 있는 것
  + 레코드의 키 값을 통해 접근할 수 있는 파일
* 임의 접근 파일의 종류
  + Indexed File
    - 인덱스를 이용하여 레코드에 접근한다.
  + Indexed Sequential File
    - 인덱스를 이용한 임의 접근과 순차 접근을 모두 지원한다.
  + Realtive File
    - 레코드 키와 주소 사이의 설정된 관계를 이용한다.
  + Hash File
    - 키 값을 통해 레코드 주소를 생성한다. ( 상태 파일의 한 형태 )
    - 협의의 직접 화일 (?)

## 상대파일 (Relative File)

레코드의 키와 레코드의 위치(상대 레코드 번호) 사이에 설정된 관계를 통해 레코드에 접근한다.

**상대 파일을 정립 한다** == 키 값과 물리적 주소 사이를 서로 변환할 수 있는 어떤 관계를 정의하는 것



**상대 레코드 번호 (Relative Record Number)**

* 화일이 시작되는 첫번째 레코드를 0번으로 지정하고, 이것을 기준으로 다음 레코드에 0, 1, 2, 3, ..., N-1를 지정해준다. ( 시작 지점과 **상대**적인 **레코드 번호** )
* 레코드의 논리적 순서와 물리적 순서는 무관
* 레코드들이 키 값에 따라 물리적으로 정렬되어 있을 필요가 없다.
* i 번째 레코드의 상대 레코드 번호 = i - 1

#### Mapping Function ... 사상 함수

**상대 파일을 정립 한다** == 키 값과 물리적 주소 사이를 서로 변환할 수 있는 어떤 관계를 정의하는 것

이러한 관계를 A라고 하면 <u>A는 키 값을 화일의 주소로 사상시키는 하나의 사상 함수</u>가 된다.

A : 키 값 -> 주소

* 레코드 삽입 시 : 키 값 = 레코드가 저장 될 주소
* 레코드 검색 시 : 키 값 = 레코드가 저장되어있는 주소
* 모든 레코드에 직접 접근이 가능하다.
  + 메인 메모리, 디스크 등 직접 저장 장치에 저장하는 것이 효율적이다.
* 사상 함수의 구현 방법
  + 직접 사상
  + 디렉터리 검사
  + 계산을 이용한 방법 ... LIKE Hashing

#### 직접 사상

키 값 자체가 레코드의 실제 주소로, 절대 주소를 이용한다.

레코드가 파일에 처음 저장될 때 레코드의 주소( 실린더 번호 / 면 번호 / 섹터 번호 , = 절대 주소 ) 가 결정된다.

**장점** : 간단, 처리 시간이 거의 걸리지 않는다.

**단점** : 물리적 저장장치에 의존적이다 .... 잘 사용되지 않는 방식이다.

#### 디렉터리 검사

<키 값, (상대) 주소> 쌍을 엔트리로 하는 테이블 유지

**검색 절차**

* 디렉터리에서 키 값을 검색한다
* 키 값에 대응되는 레코드 번호(상대주소) 구함
* 레코드에 접근한다

장점 : 빠른 검색

단점 : 삽입 비용이 크다. 화일과 디렉터리 재구성이 필요하다.

**구현 예시**

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



#### 해싱 (hashing)

Hash Function 을 통해 **키 값을 주소로 변환**하고 키에서 **변환된 주소에 레코드를 저장**한다.

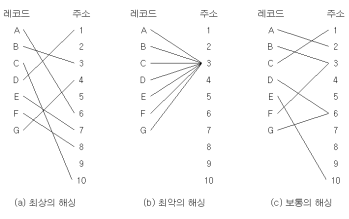
**장점** : 레코드의 주소를 구해 직접 접근이 가능하다. ( 접근시간이 빠르다 )

##### 키 공간 >> 주소 공간 : 일반적으로 키로 표현 할 수 있는 크기가 실제 사용 하는 주소 공간보다 크기가 크다

* 주민등록번호로 예시
  + 가능한 키 값의 수 1013개 ( 키 공간 = 1013 )
  + 실제 필요한 주소 수 : 108 (=1억) 개 이하
  + 주민등록번호(키) 각각 마다 화일에 빈 레코드 주소를 할당한다면 막대한 공간이 낭비된다.
  + 1억개의 레코드 공간을 갖는 파일을 만드는것이 매우 효율적이다.

#### Hashing Function

* 키 공간을 주소 공간으로 Mapping 해준다.
* H(Key) -> Address , Address 는 유효한 주소 공간이다.
* 키 값들을 한정된 주소 공간으로 <u>\*\*균등하게\*\*</u> 분산시켜야한다.



#### 해싱을 이용한 화일 설계시 고려 사항

1. 버킷의 크기
   * 하나의 주소를 가진 저장 구역에 저장할 수 있는 레코드의 수
   * 통상적으로 한 번의 접근으로 버킷 내에 있는 모든 레코드들을 전송할 수 있는 크기로 결정한다
   * 저장 장치의 물리적 특성과 연관
   * 1,000개 레코드를 갖는 파일
     + 버킷의 크기가 1인 경우 주소 공간은 1,000 ( 버킷 1,000개 )을 갖는다
     + 버킷의 크기가 5인 경우 주소 공간은 200 ( 버킷 200개 ) 이다.
   * 버킷 = 파일에서 같은 주소에 포함될 수 있는 최대 레코드 수
     + 둘 이상의 레코드 저장 가능
     + 같은 버킷 안의 레코드들은 같은 버킷 주소를 가짐
     + 버킷 수 = 파일의 주소 공간
   * 충돌(Collision)
     + 서로 다른 레코드가 똑같은 버킷으로 해싱되는 것
     + 동거자(Synonyms) : 같은 주소로 해싱되어 충돌된 키 값들
     + 버킷이 만원일 때는 충돌이 문제가 된다 ... Overflow 발생
   * 버킷의 크기를 크게 하면 충돌을 해결 할 수 있지 않은가?
     + 오버 플로우는 덜 발생한다.
     + 하지만 **저장 공간의 효율 감소**하며, 버킷 내에서 레코드를 **탐색하는데 시간이 증가**한다.
2. 적재율 (적재 밀도)
   * 총 저장 용량에 대한 실제로 저장되는 레코드 수의 대한 비율
   * 적재율 = <저장된 레코드 수> / < 버킷의 총 용량 > = K / (c \* N) < 1
   * 적재율 = <화일에 저장된 레코드 수> / ( <버킷의 크기> \* <버킷 수> )\
   * 적재 밀도가 큰 경우 ( 동거자가 많이 생긴다는 뜻 )
     + 삽입시 접근 수가 많아진다.
     + 검색시 접근 수가 많아진다.
   * 적재 밀도가 작은 경우 ( 쓸대없이 방이 크다 )
     + 공간 효율이 떨어진다.
   * <u>\*\*실험적으로, 적재 밀도가 70% 보다 클 때, 충돌이 잦으므로, 30% 정도의 예비 공간이 필요하다\*\*</u>
3. 해싱 함수 : 키를 버킷 주소로 변환하는 함수
   * 레코드 키 값으로부터 주소(버킷 주소)를 생성하는 방법
   * 일종의 변환 함수
   * 해싱함수의 계산 시간 << 보조기억장치의 버킷 접근 시간
   * 모든 주소에 대해 균일한 분포를 가져야함 ( 골고루 )
   * 주소 변환 과정
     1. 키가 숫자가 아닌 경우, 키를 정수 값으로 변환 ( output : A )
     2. 변환된 정수(A)를 주소 공간의 자리 수 만큼 다른 정수로 변환 ( output : B )
     3. 얻어진 정수 B를 주소공간의 실제범위에 맞게 조정 ( output : 주소 = B \* 조정상수 )
4. 오버플로 해결 방법
   * 버킷이 꽉 찬 경우 어떻게 해결 할 것인가

#### 해싱 방법

* 제산 잔여 해싱
  + 제수
    - 직접 주소공간의 크기를 결정
      * 주소공간 : 0 ~ (제수 - 1)
      * 제수 > 화일의 크기
    - 충돌 가능성이 가장 적은 것으로 선택한다.
      * <u>\*\*버킷수보다 작으면서 가장 큰 소수\*\*</u>
      * <u>버킷수와 가장 가까운 소수</u>
      * 20보다 작은 소수를 인수로 갖지 않는 비소수를 사용한다.
  + 성능
    - 적당한 성능을 위한 적재율 최대 허용치는 0.7 ~ 0.8
    - n개의 레코드를 갖고 있다면 1.25n 주소 공간을 할당한다.
  + 작동 방식
    1. 주소 = 키 값 % 제수 -> 0 ~ (제수 - 1)
    2. 주소 범위에 맞도록 조정 (조정 상수 사용)
  + 예시)
    - 주소 공간 = 4,000 / 0.8 = 5,000
    - 제수 : 5003 ( 20이하의 수를 인자로 가장 가까운 소수 )
* 중간 제곱(Mid-square) 해싱
  + 작동 방식
    1. 키 값을 제곱한다
    2. 중간에서 n개의 수를 취한다 : 주소공간은 10n
    3. 조정 상수를 사용하여 주소 범위에 맞도록 조정한다.
  + 예시)
    - 레코드 4,000개, 적재율 80%
    - 최소 104 이므로 최소 네 자리의주소 공간이 필요하다.
    - 키를 제곱한 수에서 4자리 수를 취한다
    - 주소공간 조정후, 주소공간의 실제 범위로 매핑한다. (4000/0.8)\*0.5
* 중첩(Folding) 해싱
  + 작동 방식
    1. 키 값을 주소공간과 같은 자리수(n자리수)를 가지는 몇 개의 부분으로 나눈다
    2. 접어서 합을 구한다. 이때 주소 공간은 10n이다.
    3. 조정 상수를 사용하여 주소 범위에 맞도록 조정한다.
* 숫자 분석 방법
  + 작동 방식
    - 키 값이 되는 숫자의 분포 이용
    - 키들의 모든 자릿수에 대한 빈도 테이블을 만들고, 어느 정도 **균등한 분포를 갖는 자릿수를 주소로 사용**한다.
    - 단, **키 값들을 미리 알고 있어야 한다**는 단점이 있다.
* 숫자 이동 변환
  + 작동 방식
    - 중앙을 중심으로 키를 양쪽으로 나눈다.
    - 주소 길이(n자리수)만큼 겹치도록 안쪽으로 각각 shift한다.
    - 주소 범위에 맞도록 조정한다
* 진수 변환
  + 작동 방식
    1. 키 값의 진수를 다른 진수로 변환한다
    2. 초과하는 높은 자리 수를 절단한다
    3. 주소 범위에 맞도록 조정한다.
  + 예시
    - 키 값 = 172,148, 주소 공간 = 7,000, 진수 : 11
    - 변환된 값 : 266373(11)
    - 뒤에 4자리 \* 0.7 = 4461

#### 충돌과 오버플로

Overflow 해결 방법

* **개방 주소법** : Overflow 된 동거자를 저장할 공간을 **상대 파일 내에서 찾아 해결**.
* **체인법** : 오버플로된 동거자를 위한 저장 공간을 **상대 파일 밖에서 찾아 해결**. ( 독립된 오버플로우 구역 할당 )

**해결 기법**

* 선형 조사
* insertLinear(key);  
   addr ← h(key); // 변환된 주소  
   home-addr ← addr; // 홈 주소  
   while (addr is full) do { // 꽉 찬 경우 반복  
   addr ← (addr + 1) mod N; // 주소 = (주소 + 1) % N  
   if (addr == home - addr ) { // 한 바퀴 다 돈 경우  
   print ("file is completely full"); // 화일이 꽉참  
   return;  
   }  
   }  
   insert the key at addr;  
   set the addr is full;  
  end insertLinear()
  + 오버플로 발생시, 홈 주소에서부터 차례대로 조사해서 가장 가까운 빈 공간을 찾는 방법
  + 해당 주소가 공백인지 아닌지 판별을 할 수 있어야 한다. ( flag를 이용해서 판별하자 )
  + **단점**
    - 어떤 레코드가 파일에 없다는 것을 판단하기 위해 조사해야하는 주소의 수는 적재율이 커질수록 많아진다.
    - **환치** : 한 레코드가 자기 홈 주소를 동거자가 아닌 레코드가 차지함으로 인해 다른 레코드의 홈 주소에 저장되는 것 -> <u>2-패스 해시 파일 생성</u>으로 대응
      * **2-패스 해시 파일**
        + 첫 번째 패스

모든 레코드를 해시 함수를 통해 홈 주소에 저장한다.

충돌이 일어나는 동거자들은 바로 저장하지 않고 별도로 임시 화일에 저장한다

* + - * + 두 번째 패스

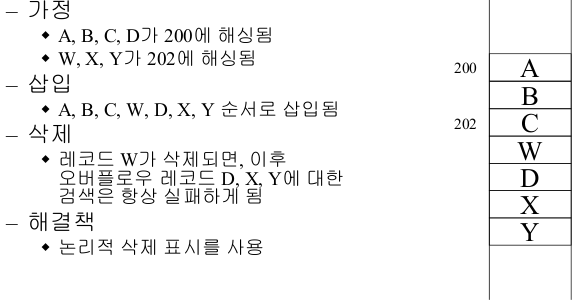
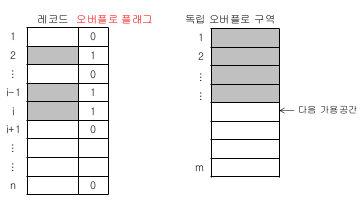
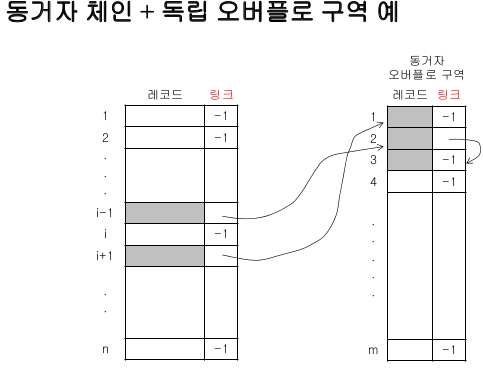
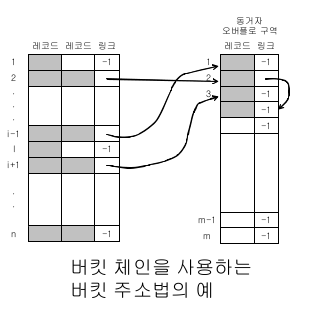
임시 화일에 저장해 둔 동거자들을 선형 조사를 이용해 화일에 모두 적재한다.

* + - * + **특징**

1-패스 생서에 비해 훨씬 많은 레코드들이 원래 자시 홈 주소에 저장된다

화일 생성 이전에 레코드 키 값들을 미리 알 때 효율적이다.

화일이 생성된 이후에 레코드들이 추가되면 환치가 발생한다.

* + 저장
    - 원형 탐색 : 빈 주소를 조사하는 과정은 홈 주소에서 시작하여 화일 끝에서 끝나는 것이 아니라 다시 화일 시작으로 돌아가 홈 주소에 다다를 때 끝난다.
  + 검색
    - 레코드 저장 시에 선형 조사를 사용했다면, 검색에서도 선형 조사를 사용해야한다.
    - 버킷에서 빈 공간이 발견되면 검색을 중단한다.
    - 탐색 키 값을 가진 레코드가 **없거나**, **홈 주소에서 멀 경우**, 많은 조사가 필요하다.
  + 삭제
    - 삭제로 인해 만들어진 빈 공간으로 검색시 선형 조사가 단절될 수 있다 ( 삭제 표시를 하자 )
    - 다음은 삭제에 대한 예시이다.
    - 
* 독립 오버플로 구역
  + 예시
  + 
  + 별개의 하나의 오버플로 구역을 할당하여 홈 주소에서 오버플로된 모든 동거자들을 순차로 저장하는 방법
  + **장점**
    - 동거자가 없는 레코드에 대해서는 한번의 홈 주소 접근만으로도 레코드를 검색
    - 1-패스로 상대 파일을 생성
    - 환치 문제 제거
  + **단점**
    - 오버플로된 동거자를 접근하기 위해서는 오버플로 구역에 있는 모든 레코드들을 순차적으로 검색해야 한다.
* 이중 해싱
  + 오버플로된 동거자들을 오버플로 구역으로 직접 해시
    - 오버플로 구역에서의 순차 탐색을 피할 수 있다
    - 2차 해시 함수
      * 오버플로된 동거자를 해시하는 함수
  + 해싱 과정
    - 1차 해시 함수에 의해 상대 파일로 해시
    - 오버플로가 발생하면, 오버플로 구역으로 해시
      * 오버플로 구역 주소 = (1차 해시 함수값 + 2차 해시 함수값) mod (오버플로 구역 크기)
    - 오버플로 구역에서 또 충돌이 일어난 경우 선형 탐색을 이용한다.
* 동거자 체인
* 
  + 각 주소마다 링크를 두어 오버플로된 레코드들을 연결하는 방법
    - 오버플로가 일어나면 선형 조사나 오버플로 구역을 이용해서 저장 후, 처음 해시 주소에 동거자를 포함하고 있는 주소에 대한 링크를 둔다.
    - 동거자에 대한 접근은 연결된 동거자들만 조사해 보면 된다.
  + 독립 오버플로 구역에 사용할 수도 있고 원래의 상대 파일에 적용할 수도 있다.
  + **장점**
    - 홈 주소에서의 충돌 감소
    - 환치 제거 ( 독립 오버플로 구역 사용시 )
  + **단점**
    - 각 주소가 링크 필드를 포함하도록 확장
* 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, 설 정이 어려움
      * 사이즈가 충분치 않으면 오버플로 발생 → 충돌 해결 기법 필요
  + 버킷 주소법에서의 충돌 해결
    - 여유 공간을 가진 가장 가까운 버킷을 사용하는 방법
    - 버킷 체인
      * 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 작다
    - **실패 탐색** : 성공적 탐색과 비슷하거나 더 작다

#### 오버플로 처리 기법들의 성능 비교 ( 선형 조사 / 이중 해싱 / 동거자 체인 )

1. 성공적 탐색 성능
   * 선형 조사가 월등하게 우수하다.
     + (검색시간 : 동거자 체인 > 이중 해싱 > 선형 조사 )
2. 실패 탐색 성능
   * 동거자 체인이 월등히 우수하다.

...