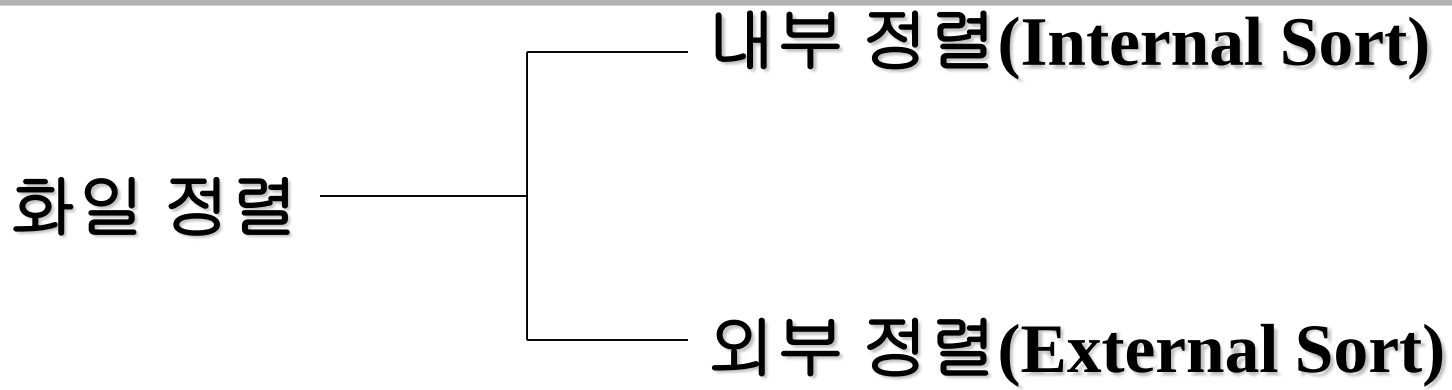
5. 화일의 정렬 합병

❖ 화일 정렬 합병의 개요

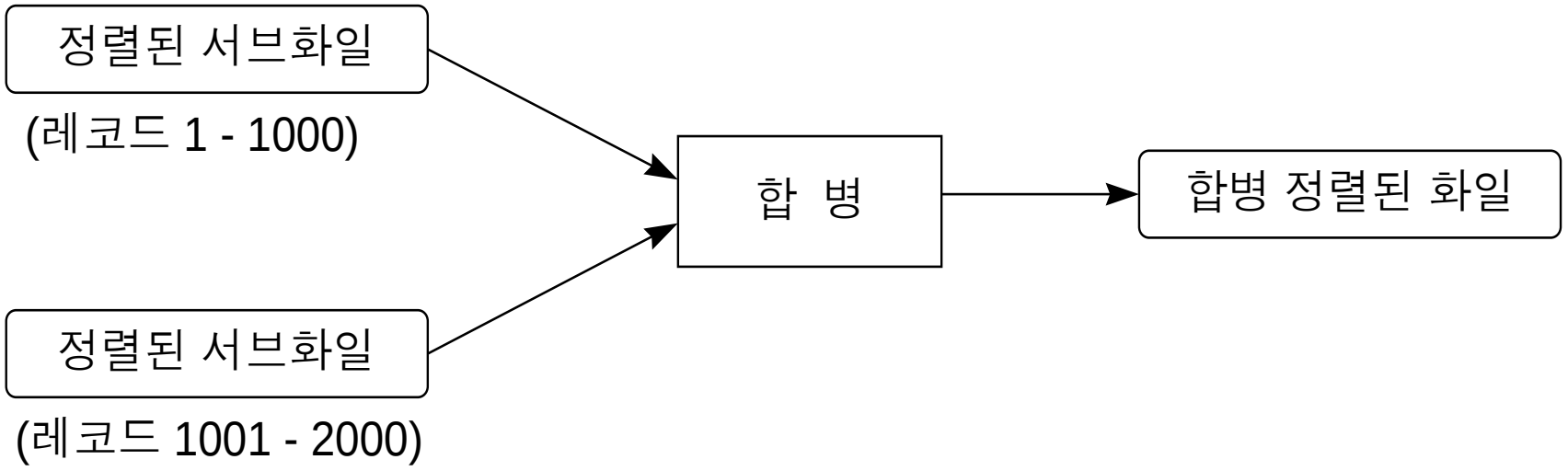


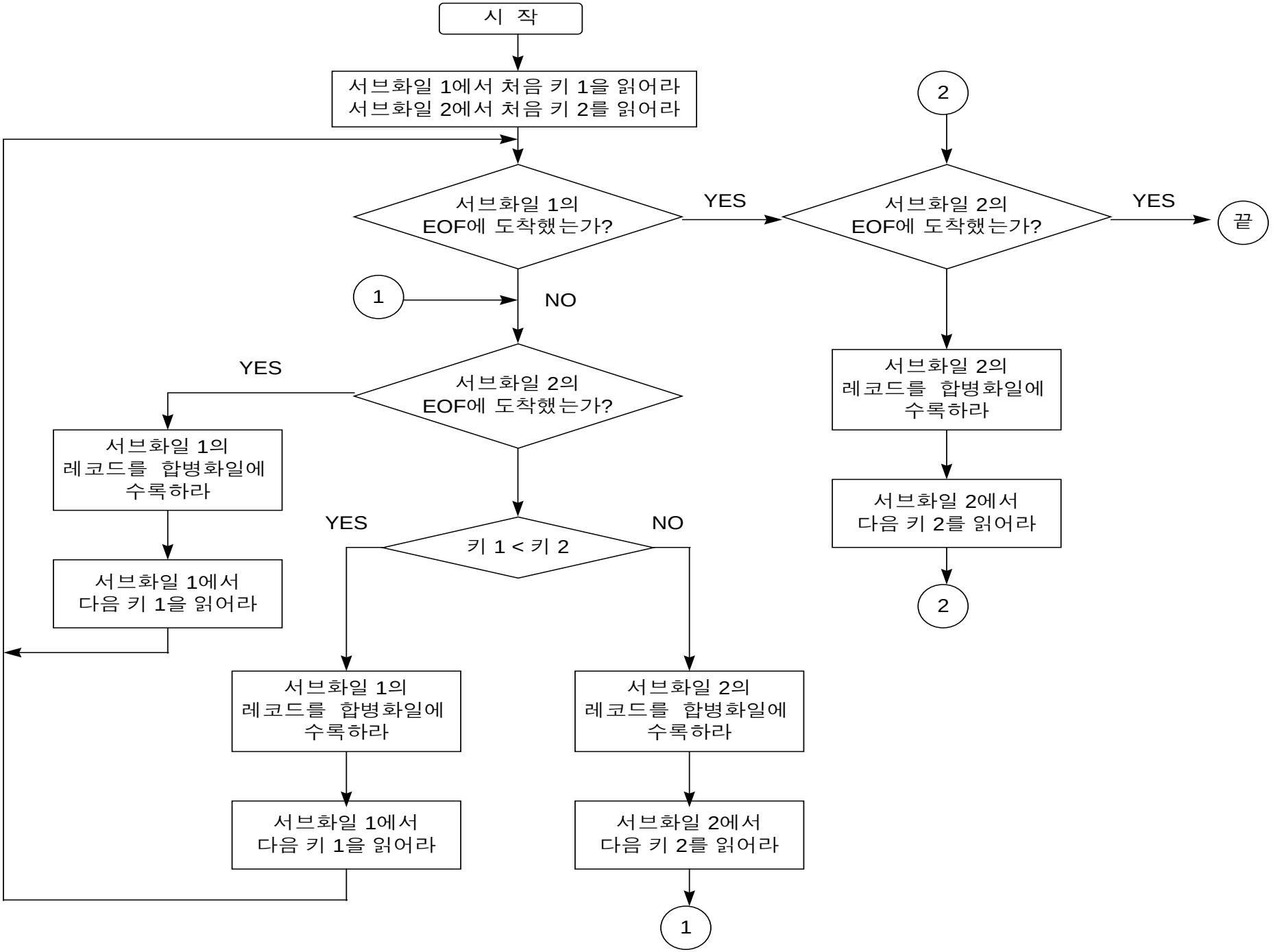
- **내부 정렬**
  - 주기억장치 내에서 정렬
- **외부 정렬**
  - 테입 또는 디스크 내의 자료 정렬
  - 레코드 판독, 기록 시간 중요

▶ 화일 외부 정렬 방법

- **예) 2000 개의 레코드 정렬**

  주기억 용량 : 1000 레코드




시 작
서브화일 1에서 처음 키 1을 읽어라
서브화일 2에서 처음 키 2를 읽어라
서브화일 1의 EOF에 도착했는가?
YES
서브화일 2의 EOF에 도착했는가?
YES
끝
2
1
NO
서브화일 2의 레코드를 합병화일에 수록하라
서브화일 2에서 다음 키 2를 읽어라
2
YES
서브화일 2의 EOF에 도착했는가?
서브화일 1의 레코드를 합병화일에 수록하라
서브화일 1에서 다음 키 1을 읽어라
YES
키 1 < 키 2
NO
서브화일 1의 레코드를 합병화일에 수록하라
서브화일 1에서 다음 키 1을 읽어라
서브화일 2의 레코드를 합병화일에 수록하라
서브화일 2에서 다음 키 2를 읽어라
1

▶ 화일 정렬 합병 단계

(1) 런 생성 단계
- 서브 화일로 분할
- 내부 정렬

(2) 합병 단계

◆ 분할 방법

– 내부 정렬 (internal sort)
– 대체 선택 (replacement selection)
– 자연 선택 (natural selection)

★ 입력 화일의 예

109 49 34 68 45 2 60 38 28 47 16 19 34 55
98 78 76 40 35 86 10 27 61 92 99 72 11 2
29 16 80 73 18 12 89 50 46 36 67 93 22 14
83 44 52 59 10 38 76 16 24 85

1) 내부 정렬 (internal sort)

[1] 화일을 n 레코드씩 분할
[2] 내부 정렬 기법으로 정렬
[3] 출력화일(런) 생성

★ 런의 길이가 동일 (마지막 런은 제외)

– 내부정렬로 생성된 런

런 1 : 34 45 49 68 109
런 2 : 2 28 38 47 60
런 3 : 16 19 34 55 98
런 4 : 35 40 76 78 86
런 5 : 10 27 61 92 99
런 6 : 2 11 16 29 72
런 7 : 12 18 73 80 89
런 8 : 36 46 50 67 93
런 9 : 14 22 44 52 83
런 10 : 10 16 38 59 76
런 11 : 24 85

2) 대체 선택 (replacement selection)

◆ **입력화일의 부분 순서(partial ordering) 이용**

[1] 입력화일에서 m 레코드 판독, 첫번째 런 생성 시작

[2] 최소 키값의 레코드 선택 출력

[3] 입력화일에서 다음 레코드 판독, 출력된 레코드와 대체
- ◆ 입력 레코드 키 < 출력 레코드 키
  - 입력 레코드에 동결 표시
- ◆ 동결된 레코드는 [2]의 선택시 제외
- ◆ 동결되지 않은 레코드가 있으면 [2]로 되돌아감

[4] 동결된 레코드 해제,
- ◆ 새로운 런 생성을 위해 [2]로 감

★ 최종 출력화일에 가까울 수록 긴 런 생성
★ 런의 평균 예상 길이 : 2m

▶ 대체 선택으로 생성된 런

런 1 : 34 45 49 60 68 109
런 2 : 2 16 19 28 34 38 47 55 76 78 86 98
런 3 : 10 27 35 40 61 72 92 99
런 4 : 2 11 16 18 29 50 73 80 89 93
런 5 : 12 14 22 36 44 46 52 59 67 76 83 85
런 6 : 10 16 24 38

▶ 대체 선택 알고리즘

```
/* m       : 버퍼에 들어가는 레코드수
   Buffer  : 버퍼-레코드 배열
   written : 해당 버퍼 레코드가 출력되었는지를 나타내는
   플래그 배열
   frozen  : 해당 버퍼 레코드가 동결되었는지를 나타내는
   플래그 배열
   last-key : 제일 나중에 출력된 레코드의 키값  */

for (i=1; i<=m; i++)
   written[i] = true;
i = 0;
do {
   i += 1;
   read Buffer[i] from input;
} while (!end-of-file(input) && i != m);
```

```
while (!end-of-file(input)) {                    /* 런의 생성 */
    /* 런 하나의 생성 */
    /* 동결 플래그 초기화 */
    for (i=1; i<=m; i++)
        if (!written[i]) frozen[i] = false;
        while (any unfrozen records remain) {
     /* 레코드 하나를 런에 출력 */
     find smallest unfrozen record(assume Buffer[s]);
     append Buffer[s] to sorted partition;
     last-key = Buffer[s].key;
     written[s] = true; frozen[s] = true;
     if (!end-of-file(input)) {
         read new Buffer[s] from input;
         written[s] = false;
         if (Buffer[s].key > last-key)   frozen[s] = false;
     }   }
   /* 버퍼에 있는 나머지 레코드들을 출력 */
 output unwritten records in ascending key order;
```

3) 자연 선택 (natural selection)

- 대체 선택
  - 런 생성 직전은 주기억장치내 대부분 레코드가 동결
- 동결 레코드를 보조기억장치의 예비장소(reservoir)에 저장

◆ **자연 선택으로 생성된 런**

런 1 : 34 45 47 49 60 68 109
런 2 : 2 16 19 28 34 38 40 55 61 76 78 86 92 98 99
런 3 : 10 11 16 27 29 35 50 67 72 73 80 89 93
런 4 : 2 12 14 18 22 36 44 46 52 59 76 83 85
런 5 : 10 16 24 38

★ 하나의 런은 예비 장소가 **full** 또는 입력화일이 **empty**가 될때까지 생성

▶ 자연 선택 알고리즘

```
/* m, m'     : 버퍼와 외부 예비 장소의 레코드수
   Buffer    : 버퍼-레코드 배열
   written   : 해당 버퍼 레코드가 출력되었는지를 나타내는 플래그 배열
   frozen    : 해당 버퍼 레코드가 동결되었는지를 나타내는 플래그 배열
   reservoir-count : 예비 장소에 들어있는 레코드수
   no-space  : 예비 장소 오버플로를 나타내는 플래그
   last-key  : 제일 나중에 출력된 레코드의 키값   */
  for (i=1; i<=m; i++) written[i] = true;
  i = 0;
  do {
    i += 1;
    read Buffer[i] from input;
    written[i] = false;
  } while (!end-of-file(input) && i != m);
   /* 화일로부터 읽어 런에 출력 */
  do { /* 런 하나의 생성 */
    reservoir-count = 0;
    no-space = false;
    do { /* 레코드 하나를 출력 */
      output smallest unwritten record(assume Buffer[s]);
      last-key = Buffer[s].key;
      written[s] = true;
```

```
if (!end-of-file(input)) {
        done = false;
        do {
           if (key of next record in input file > last-key) {
              read next input record into Buffer[s];
              written[s] = false;            done = true;
           } else if(reservoir-count < m') {
              read next input record into reservoir;
              reservoir-count += 1;
           } else no-space = true;
        } while (!done && !no-space);
     }
  } while (!end-of-file(input) && !no-space);
  output unwritten buffer records in ascending key order;
  set corresponding elements of written to true;
    /* 다음 런을 위해 버퍼 정리 */
  if (reservoir-count > 0) {
    move min(reservoir-count, m) records from reservoir to buffer;
    set corresponding elements of written to false;
    decrease reservoir-count;
  }
  if (buffer not full && !end-of-file(input)) {
    fill buffer as much as possible from input;
    set corresponding elements of written to false;
  }
} while (unwritten records exist in buffer);
```

▶ 화일 정렬 합병의 매개 변수

- 적용 내부정렬 방식
- 내부정렬을 위한 주기억장치 공간의 양
- 정렬된 런들의 보조기억장치에서의 분포
- 한 합병과정에서 합병될 수 있는 런의 수

◆ **정렬 합병 기법의 성능**

- 보조기억장치에 대한 상대적 참조 빈도수
  - ◆ 합병 단계수, 레코드의 판독/기록 회수

1) 합병될 런수의 최소화 : 서브화일의 길이를 최대화
2) 한번에 합병가능한 서브화일 수의 최대화
3) 합병을 위한 입력 서브화일을 보조기억장치에 분산 저장

▶ 합병 단계를 수행하는 방법

- 자연 합병 (natural merge)
- 균형 합병 (balanced merge)
- 다단계 합병 (polyphase merge)
- 계단식 합병 (cascade merge)

❖ 자연 합병 (natural merge)

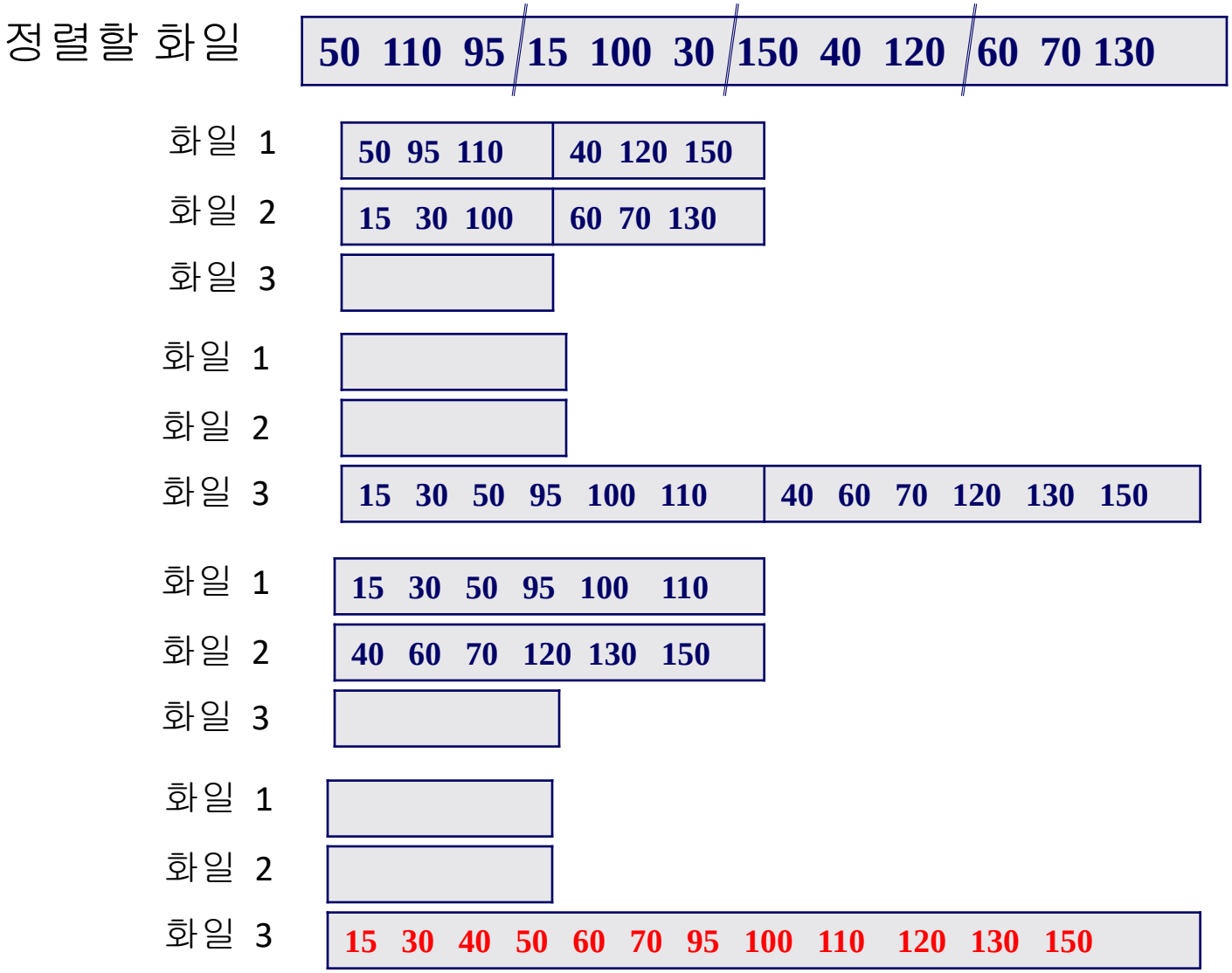
- 2-원 합병
- m-원 합병 (m-way merge)
  - m : 합병의 차수

◆ m-원 자연 합병

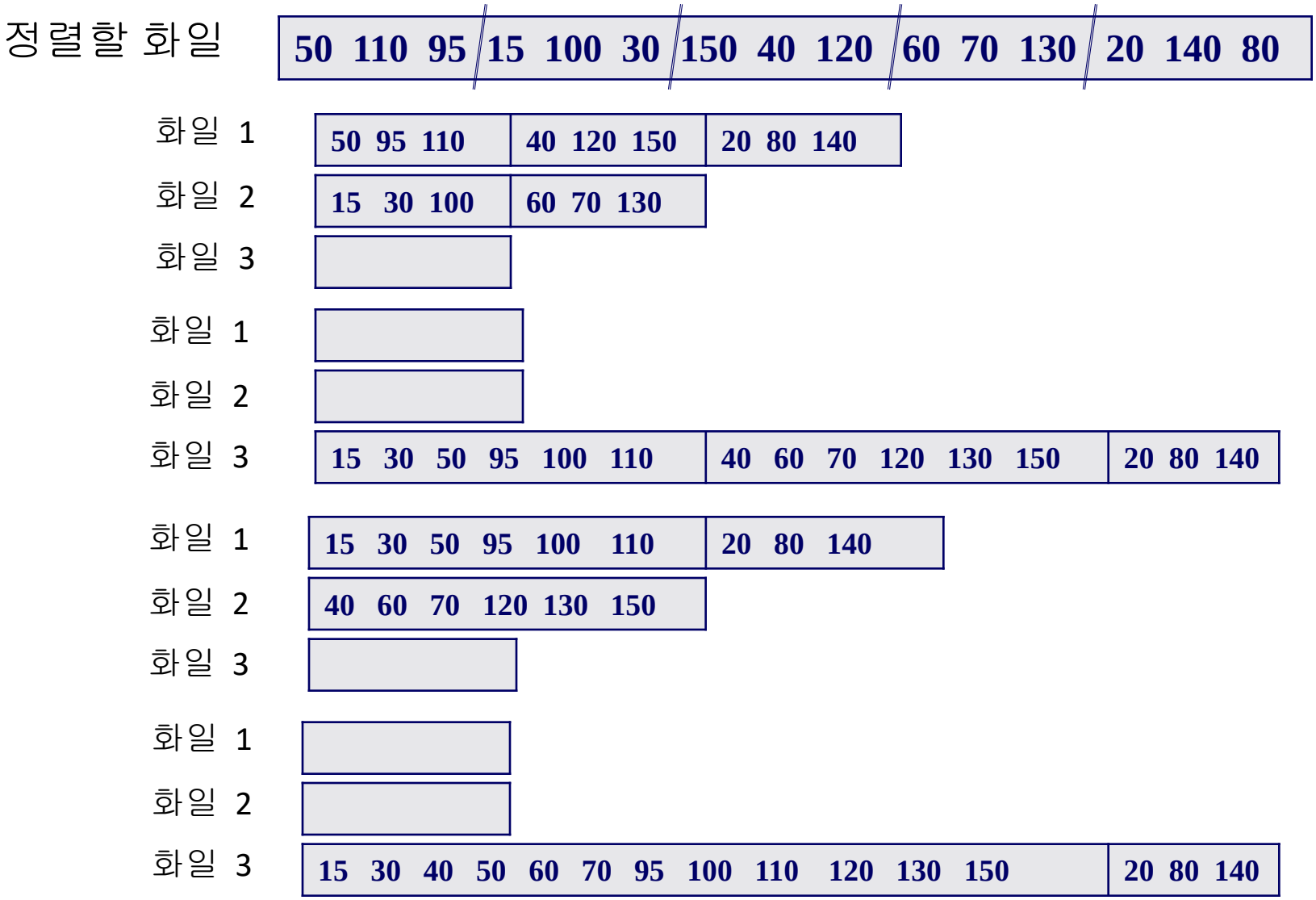
- m개의 입력화일

[1] m개의 입력화일을 합병
[2] 1의 출력결과 서브화일을 m개의 입력화일로 재분배

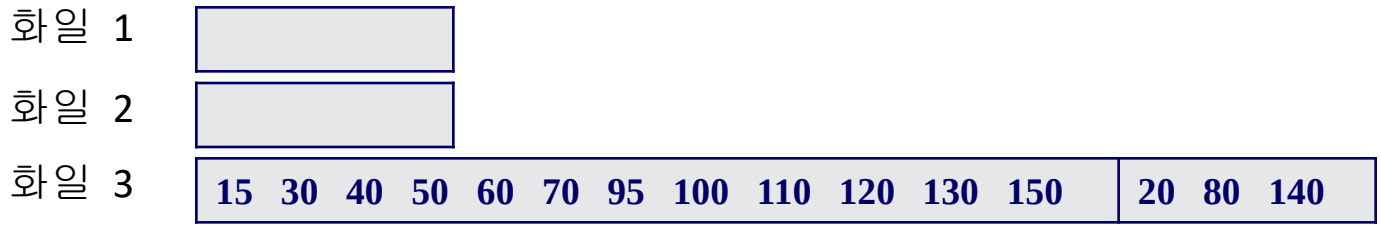
▶ 4 개의 런에 대한 2-원 합병 예제

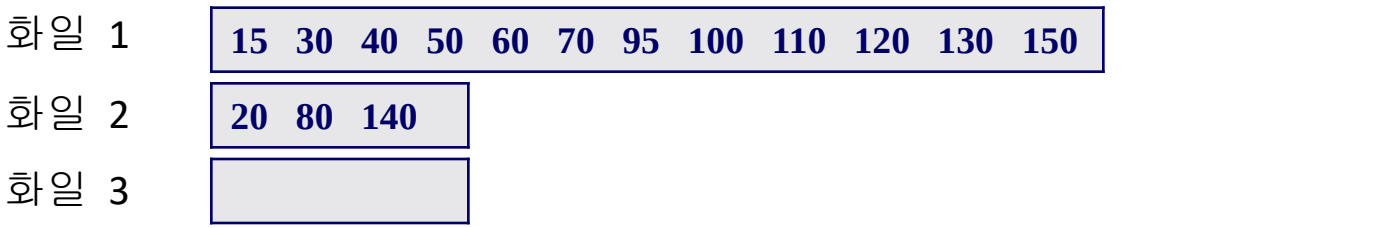


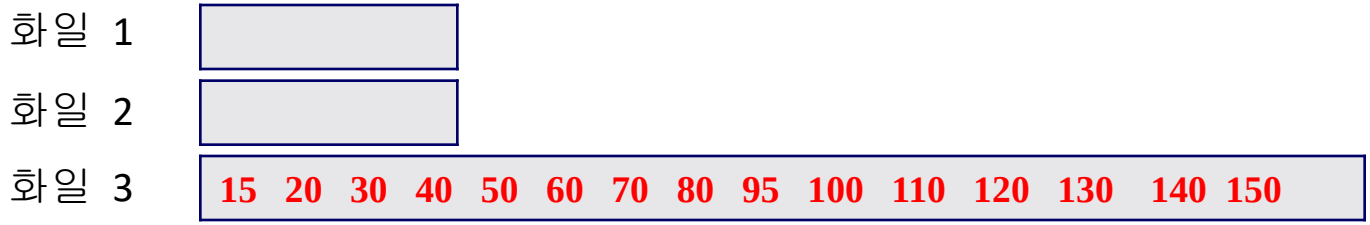
▶ 5 개의 런에 대한 2-원 합병 예제



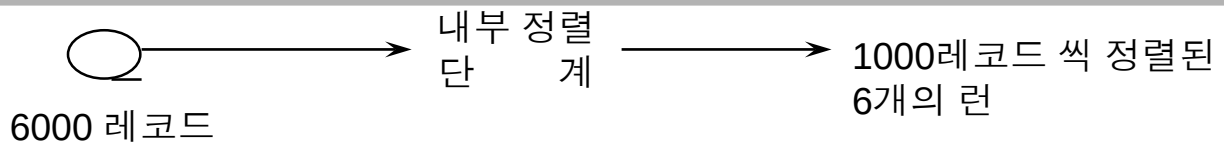
▶ 2-원 합병 예제(계속)


화일 1
화일 2
화일 3
15 30 40 50 60 70 95 100 110 120 130 150
20 80 140


화일 1
15 30 40 50 60 70 95 100 110 120 130 150
화일 2
20 80 140
화일 3

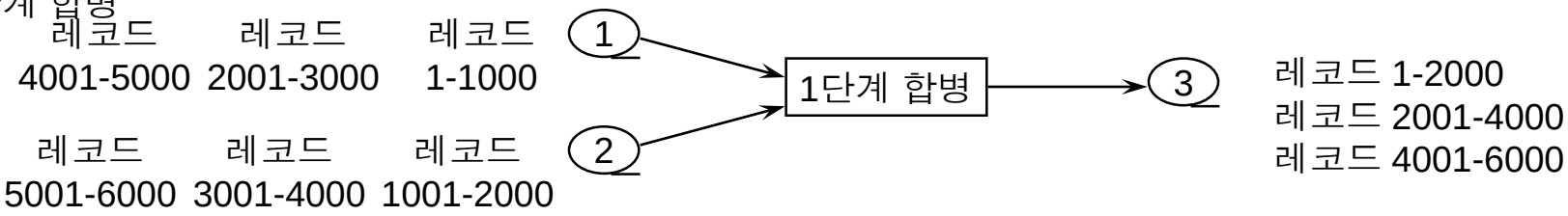

화일 1
화일 2
화일 3
15 20 30 40 50 60 70 80 95 100 110 120 130 140 150

▶ 6 개의 런에 대한 2-원 합병



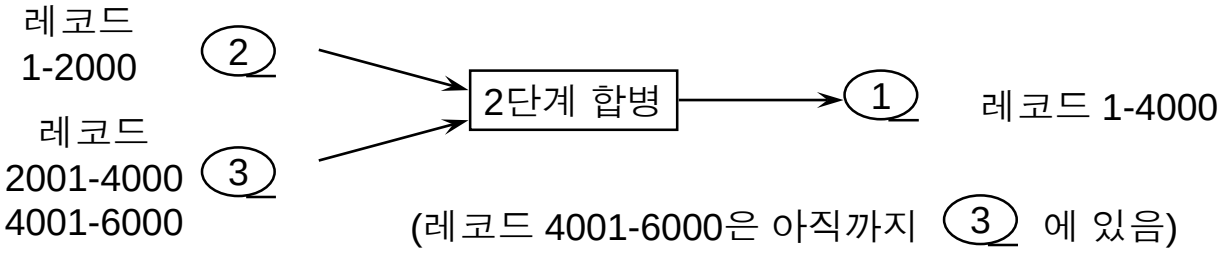
(1) 정렬된 6개의 런을 2개의 화일에 분산 시킨다.

(2) 1단계 합병

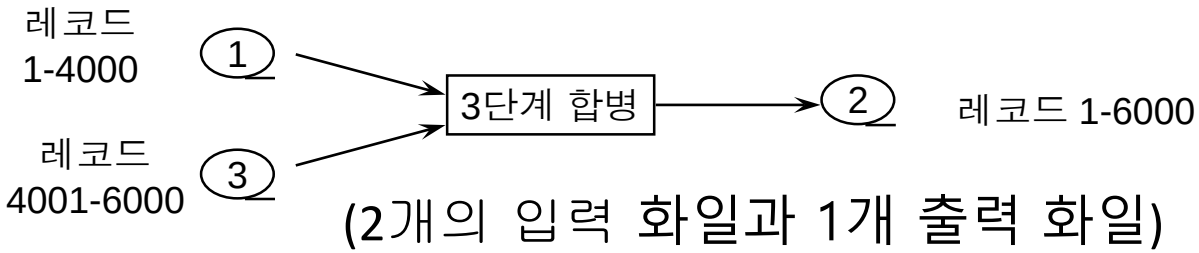


(3) 3 에 있는 정렬된 런을 2개의 화일에 분산시킨다.

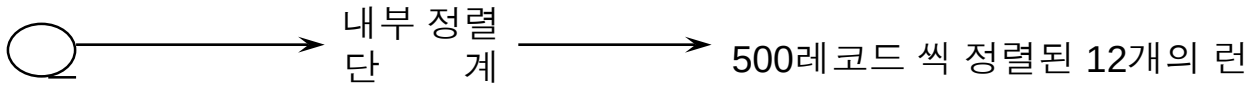
(4) 2단계 합병



(5) 3단계 합병

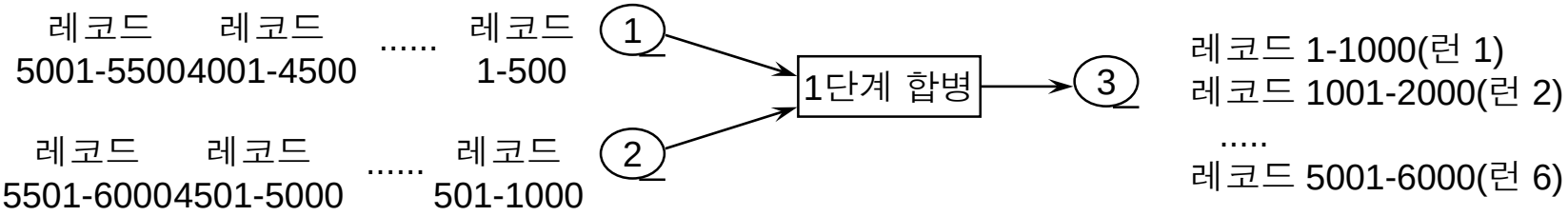


▶12개의 런에 대한 2-원 자연 합병



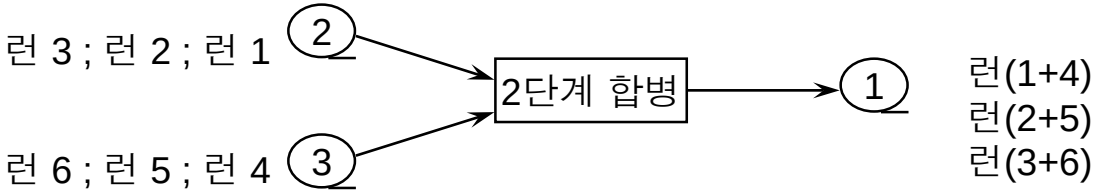
(1) 정렬된 12개의 런을 2개의 화일에 분산 시킨다.

(2) 1단계 합병



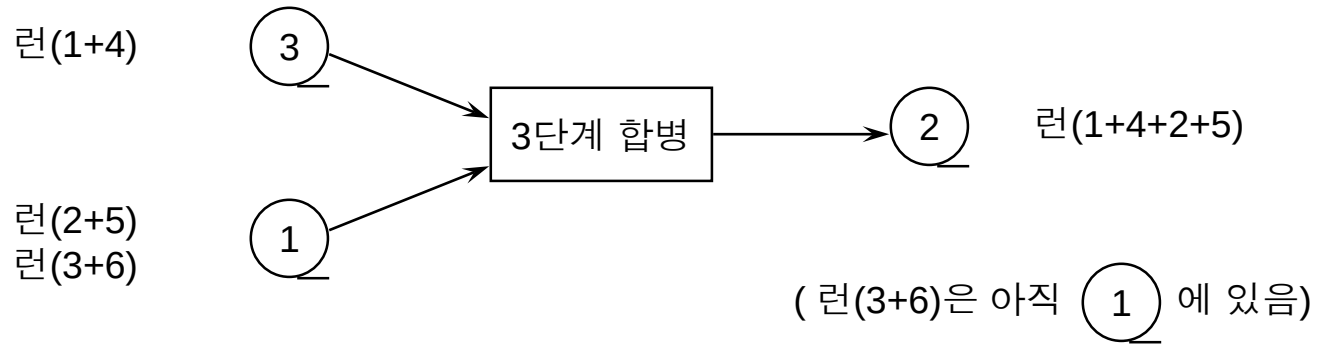
(3) ③ 에 있는 정렬된 런을 2개의 화일에 분산시킨다.

(4) 2단계 합병

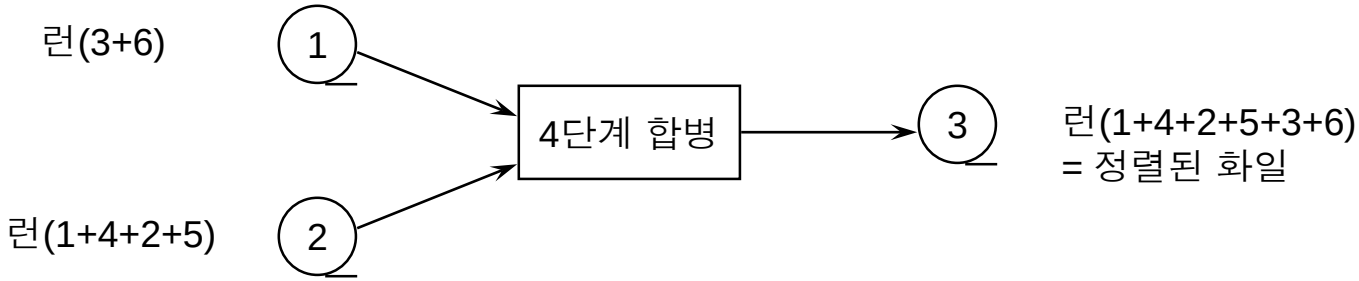


(5) ① 에 있는 정렬된 런을 2개의 화일에 분산시킨다.

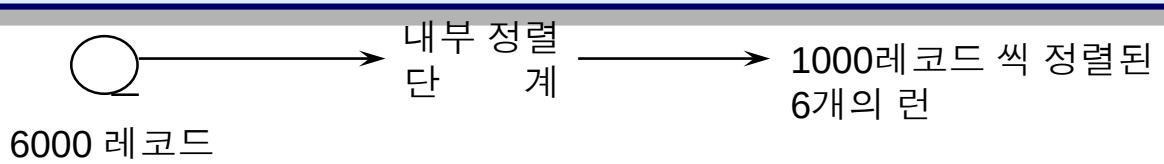
(6) 3단계 합병



(7) 4단계 합병

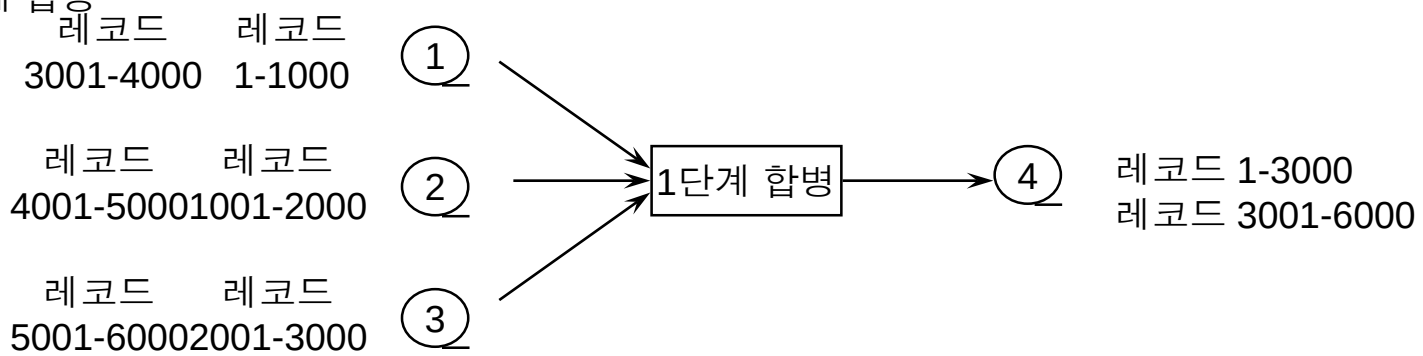


▶ 6개의 런에 대한 3-원 자연 합병



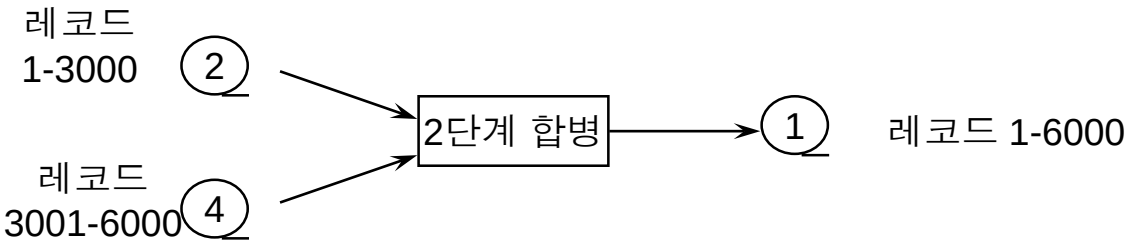
(1) 정렬된 6개의 런을 3개의 화일에 분산 시킨다.

(2) 1단계 합병



(3) (4) 에 있는 정렬된 런을 3개의 화일에 분산시킨다. (이 경우에는 2개의 화일만 사용됨)

(4) 2단계 합병



(3개의 입력 화일과 1개의 출력 화일)

▶ 5 개의 런에 대한 5-원 합병 예제

정렬할 화일

50 110 95	15 100 30	150 40 120	60 70 130	20 140 80

화일 1	50 95 110
화일 2	15 30 100
화일 3	40 120 150
화일 4	60 70 130
화일 5	20 80 140
화일 6	

화일 1	
화일 2	
화일 3	
화일 4	
화일 5	
화일 6	15 20 30 40 50 60 70 80 95 100 110 120 130 140 150

m-원 자연합병 알고리즘

```
/*  m : 입력 화일의 수
    FILE : 화일 변수 또는 채널들의 배열
    outfilenum  : 출력 화일을 포함하는 채널 배열의 인덱스
    r  : 현 단계에서 합병되지 않고 입력 화일에 남아있는 런수
    runcount          : 현 단계에서 생성된 런의 수     */
/* 합병 단계 */
do {
   open files on FILE[1], ..., FILE[m] for reading;
   open file on FILE[m+1] for writing;
   runcount = 0;
   do {
        merge run from each file on FILE[1], ..., FILE[m]
           onto the file on FILE[m+1];
        runcount += 1;
   }while (!end-of-file on any one input file);
/* 합병되지 않고 입력 화일에 남아 있는 런수의 계산 및 다음 단계의
   출력 화일 선택 */
   r = 0;
   for (i=m; i>=1; i--) {
      if (!end-of-file(FILE[i])) r += 1;
      else outfilenum = i;
   }
   runcount += r;
```

```
/* 화일을 채널에 재할당 */
  FILE[1], …, FILE[m+1] = FILE[1], …, FILE[outfilenum-1],
    FILE[outfilenum+1], ..., FILE[m+1], FILE[outfilenum];
  close files on FILE[1], …, FILE[m];
 /* 런들의 분산 단계 */
 open file on FILE[m] for reading;
 open files on FILE[1], ... FILE[m-1];
/* FILE[1],...,FILE[m] 내의 런수 차가 1 이하가 되도록 ⌈runcount / m⌉,
⌈runcount / m⌉ - 1, ⌈runcount / m⌉ 2개씩의 런들을 배분 */
 i = 0;
 k = m ⌈runcount / m⌉ runcount;
 do {
    i += 1;
    if (k = = 0) {
       if (end-of-file(FILE[i]))
          move ⌈runcount / m⌉ runs from FILE[m] to FILE[i];
       else move ( ⌈runcount / m⌉ - 1) runs from FILE[m] to FILE[i];
    } else {
       k -= 1;
       if (end-of-file(FILE[i]))
          move ( ⌈runcount / m⌉ - 1) runs from FILE[m] to FILE[i];
       else move ( ⌈runcount / m⌉ - 2) runs from FILE[m] to FILE[i];
    }
  } while (i != m-1);
} while (runcount !=1);
  /* 정렬된 최종 화일은 FILE[m] */
```

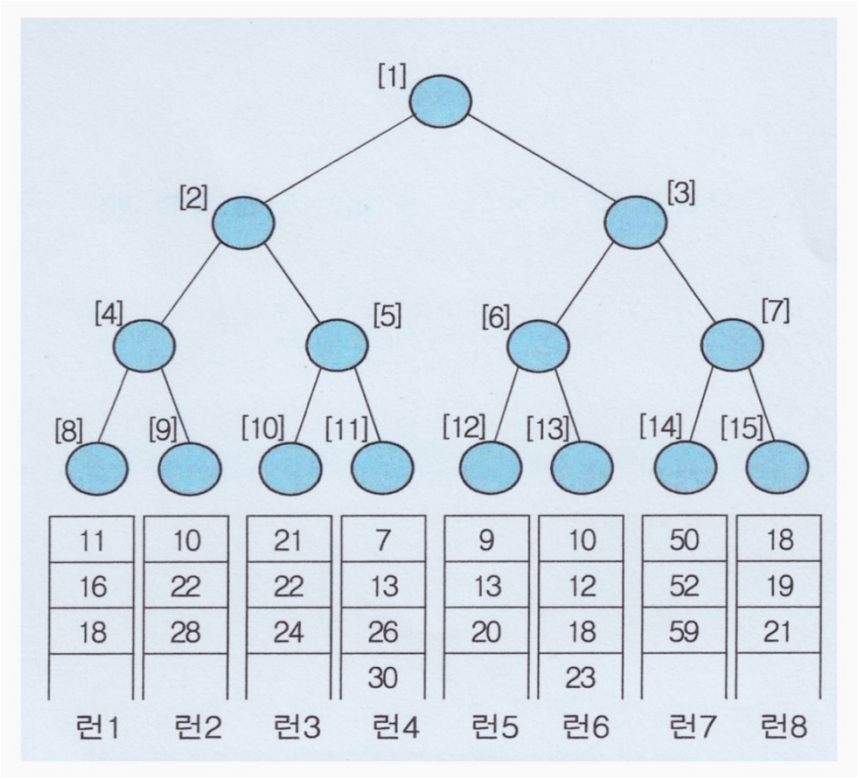
▶ 선택 트리(Selection Tree) (1)

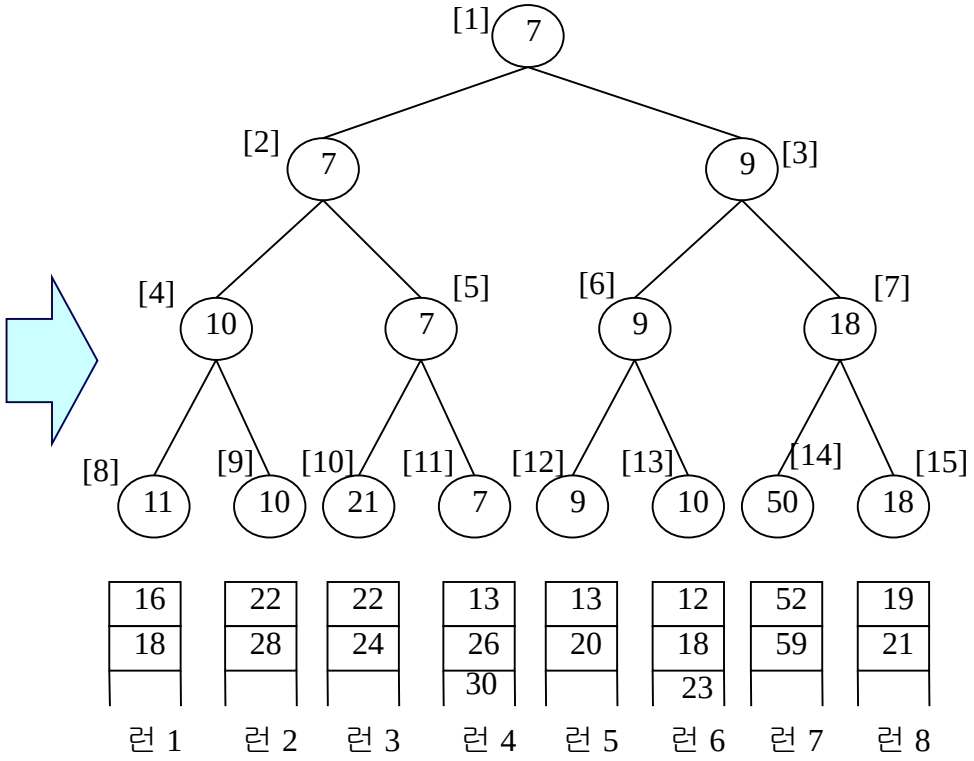
- **m개의 런을 하나의 큰 런으로 정렬**
  - m개의 런 중 가장 작은 키 값의 레코드를 계속 선택, 출력
  - 간단한 방법 : m-1번 비교
  - 선택 트리 : 비교 횟수를 줄일 수 있음

- **선택 트리의 종류**
  - 승자 트리(winner tree)
  - 패자 트리(loser tree)

선택 트리 (2)

- **승자 트리(winner tree)**
  - 특징
    - 완전 이진 트리
    - 각 단말 노드는 각 런의 최소 키 값 원소를 나타냄
    - 내부 노드는 그의 두 자식 중에서 가장 작은 키 값을 가진 원소를 나타냄
  - 런이 8개(k=8)인 경우 승자 트리 예



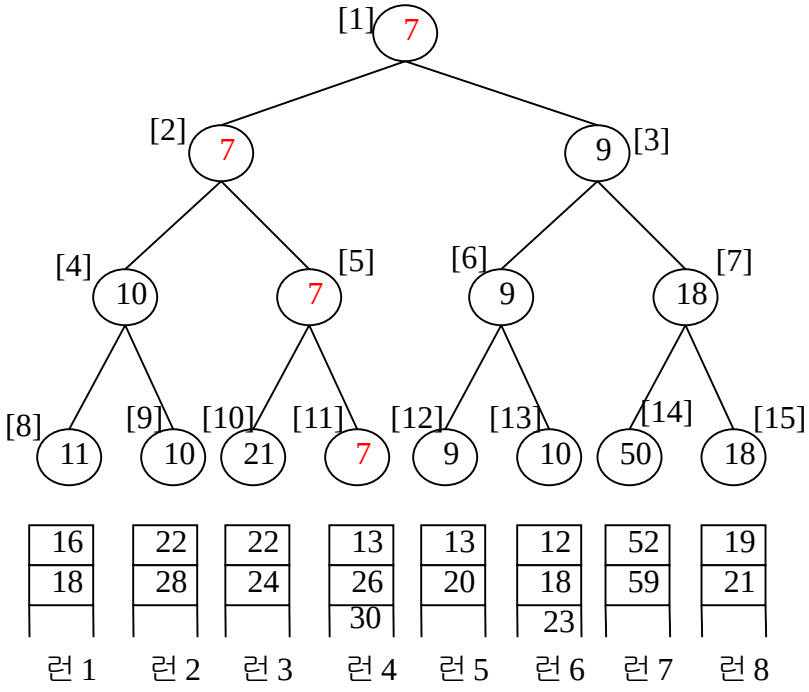


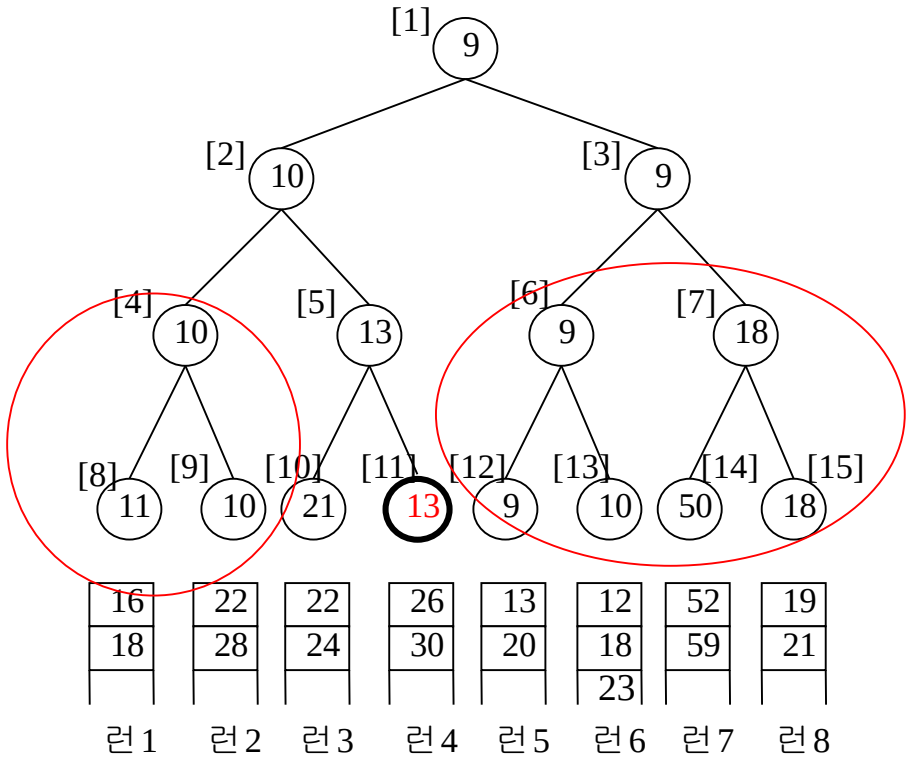
선택 트리 (3)

- 승자 트리 구축 과정
  - 가장 작은 키 값을 가진 원소가 승자로 올라가는 토너먼트 경기로 표현
  - 트리의 각 내부 노드: 두 자식 노드 원소의 토너먼트 승자
  - 루트 노드: 전체 토너먼트 승자, 즉 트리에서 가장 작은 키 값 가진 원소
- 승자 트리의 표현
  - 승자트리는 완전 이원트리이기 때문에 순차 표현이 유리
  - 인덱스 값이 i인 노드의 두 자식 인덱스는 2i와 2i+1
- 합병의 진행
  - 루트가 결정되는 대로 순서순차에 출력 (여기선 7)
  - 다음 원소 즉 키값이 13인 원소가 승자트리로 들어감
  - 승자 트리를 다시 재구성
    - 노드 11에서부터 루트까지의 경로를 따라가면서 형제 노드간 토너먼트 진행

선택 트리 (4)

- 다시 만들어진 승자트리의 예 (붉은 원 부분은 재계산 필요없음)





- 이런 방식으로 순서 순차구축을 계속함

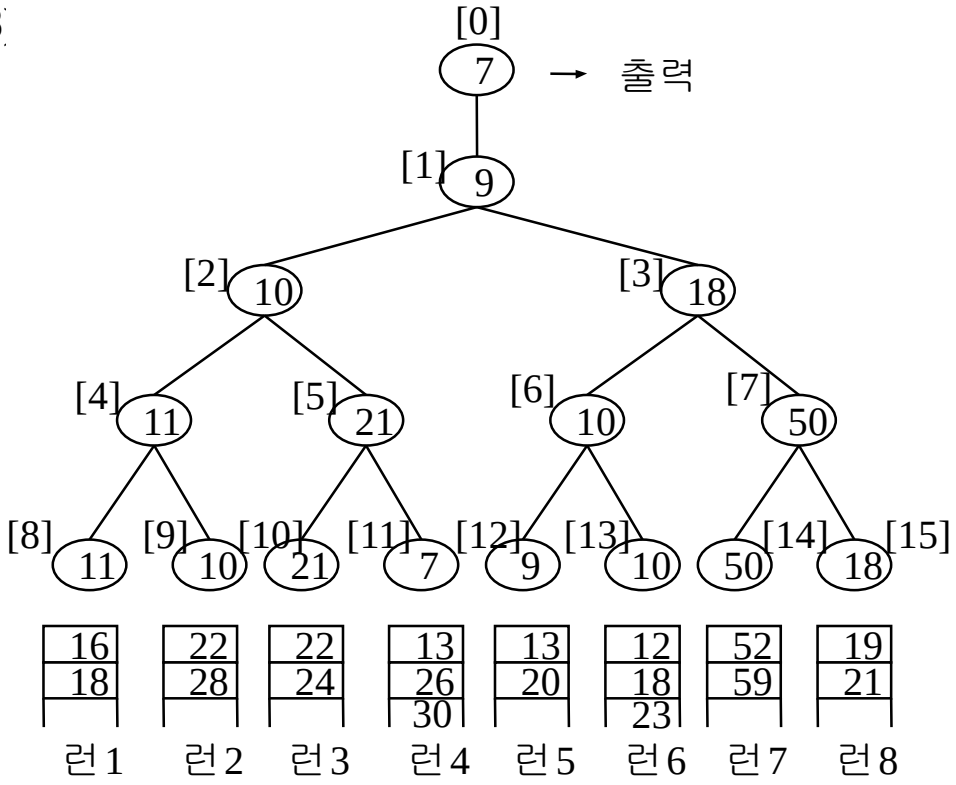
선택 트리 (5)

- **패자 트리(loser tree)**
  - 루트 위에 0번 노드가 추가된 완전 이원트리
    - 성질

      (1) 단말노드 :각 런의 최소 키값을 가진 원소
      (2) 내부 노드 : 토너먼트의 승자대신 패자 원소
      (3) 루트(1번 노드) : 결승 토너먼트의 패자
      (4) 0번 노드 : 전체 승자(루트 위에 별도로 위치)

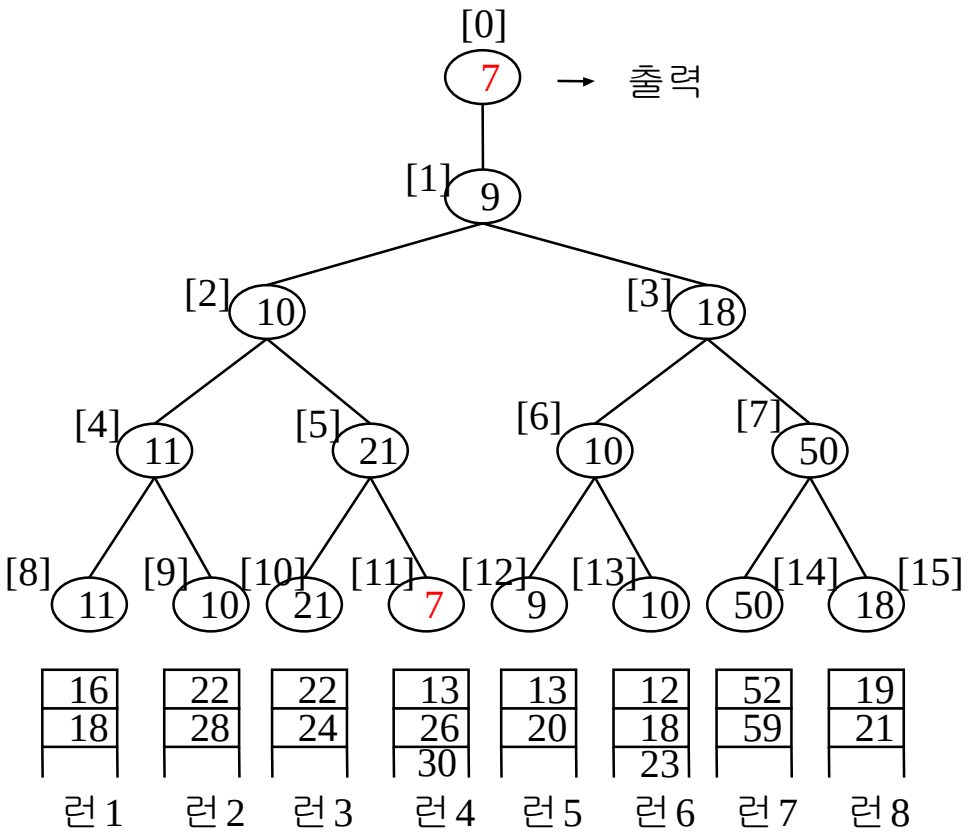
선택 트리 (6) 패자 트리 구축 과정

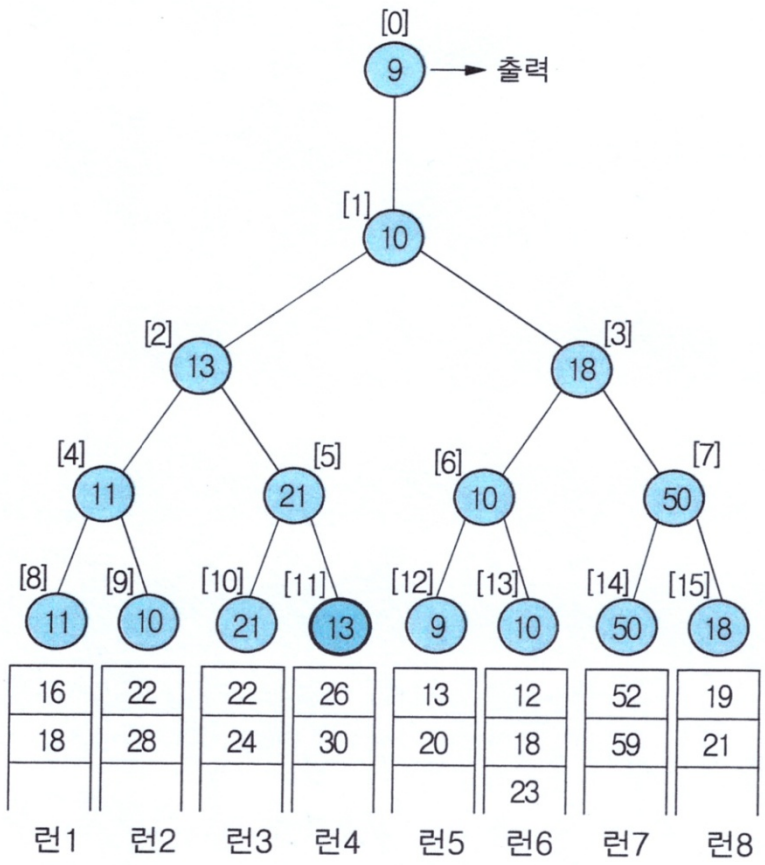
- 단말 노드: 각 런의 최소 키 값 원소
  - 내부 노드
    - 두 자식 노드들이 부모노드에서 토너먼트 경기를 수행
    - 패자는 부모 노드에 남음
    - 승자는 그 위 부모 노드로 올라가서 다시 토너먼트 경기를 계속
  - 1번 루트 노드
    - 마찬가지로 패자는 1번 루트 노드에 남음
    - 승자는 전체 토너먼트의 승자로서 0번 노드로 올라가 순서순차에 출력됨
  - 런이 8개(k=8)



선택 트리 (7) - 합병의 진행

- 출력된 원소가 속한 런 4의 다음 원소, 즉 키값이 13인 원소를 패자트리 노드 11에 삽입
- 패자 트리를 다시 재구성
  - 토너먼트는 노드 11에서부터 루트 노드 1까지의 경로를 따라 경기를 진행
  - 다만 경기는 형제 노드 대신 형식 상 부모 노드와 경기를 함





선택 트리 (8)

- 패자 트리 구축 과정
  - 단말 노드: 각 런의 최소 키값 원소
  - 내부 노드
    - 두 자식 노드들이 부모노드에서 토너먼트 경기를 수행
    - 패자는 부모 노드에 남음
    - 승자는 그 위 부모 노드로 올라가서 다시 토너먼트 경기를 계속
  - 1번 루트 노드
    - 마찬가지로 패자는 1번 루트 노드에 남음
    - 승자는 전체 토너먼트의 승자로서 0번 노드로 올라가 순서순차에 출력됨
- 합병의 진행
  - 출력된 원소가 속한 런 4의 다음 원소, 즉 키값이 13인 원소를 패자트리
  - 노드 11에 삽입
  - 패자 트리를 다시 재구성
    - 토너먼트는 노드 11에서부터 루트 노드 1까지의 경로를 따라 경기를 진행
    - 다만 경기는 형제 노드 대신 형식상 부모 노드와 경기를 함

❖ 균형 합병 (balanced merge)

- 자연합병 : 화일의 재분배 ⟶ 많은 I/O

◆ 균형합병

- 출력을 미리 다음 단계의 입력화일로 재분배
  - m-원 자연합병 : m + 1 개의 화일
  - m-원 균형합병 : 2m 개의 화일 (m 입력화일, m 출력화일)

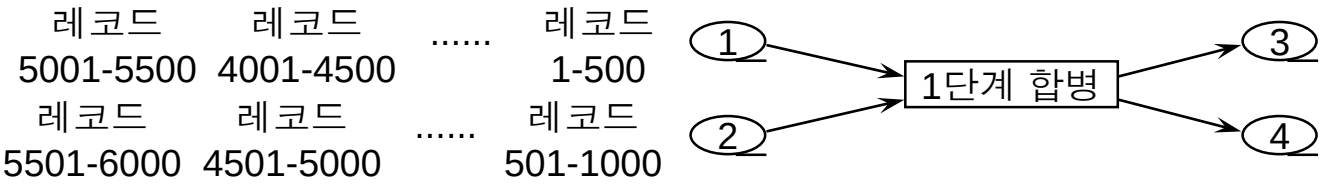
◆ 각 합병 단계 후

- 런의 총수는 합병 차수로 나눈 만큼 감소
- 런의 길이는 합병 차수배씩 증가
- $O(\log_m N)$, N : 초기 런의 수

▶ 12개의 런에 대한 2-원 균형 합병

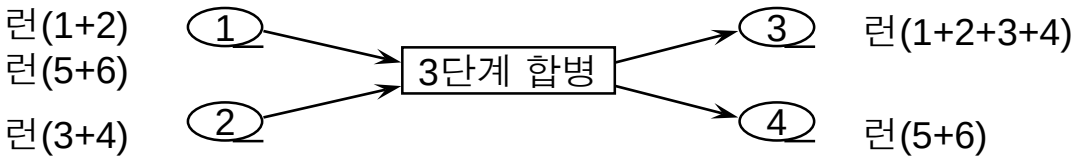
(1) 정렬된 12개의 런을 2개의 화일에 분산 시킨다.

(2) 1단계 합병



(3) 2단계 합병

(4) 3단계 합병



(5) 4단계 합병

주의 런1은 4번 판독/기록되었음.

▶ m-원 균형합병 알고리즘

```
/* m   : 입력 화일수(모두 2m개의 화일 사용
   input-set-first : 현 단계에서 입력과 출력 세트를 구분하기 위한
   플래그
   base   : 출력 세트의 화일 중 첫번째 화일의 번호
   outfilenum    : 현 단계의 출력 화일 번호
   runcount     : 현 단계에서 생성된 런의 수     */

  input-set-first = false;
  do {
    change value of input-set-first;
    if (input-set-first) {
        open files 1 through m for reading;
        open files m+1 through 2m for writing;
        base = m + 1;
    } else {
        open files 1 through m for writing;
        open files m+1 through 2m for reading;
        base = 1;
    }
```

```
/* 합병 단계의 수행 */
    outfilenum = 0;
    runcount = 0;

    do {
      merge run from each input file onto
       file numbered(base+outfilenum);
      runcount += 1;
      outfilenum = (outfilenum + 1) % m;
    } while (!end-of-file on all input files);
    rewind input files and output files;
  } while (runcount != 1);

  /* input-set-first가 true면 최종 정렬 화일은 m+1,
    false면 최종 정렬 화일은 1 */
```

❖ 다단계 합병 (polyphase merge)

- m-원 균형 합병 : m 입력화일, m 출력화일
  - m 입력화일의 런들이 한 화일로 기록
  - m-1 개의 화일은 항상 유휴 상태
- 화일의 활용도 증가

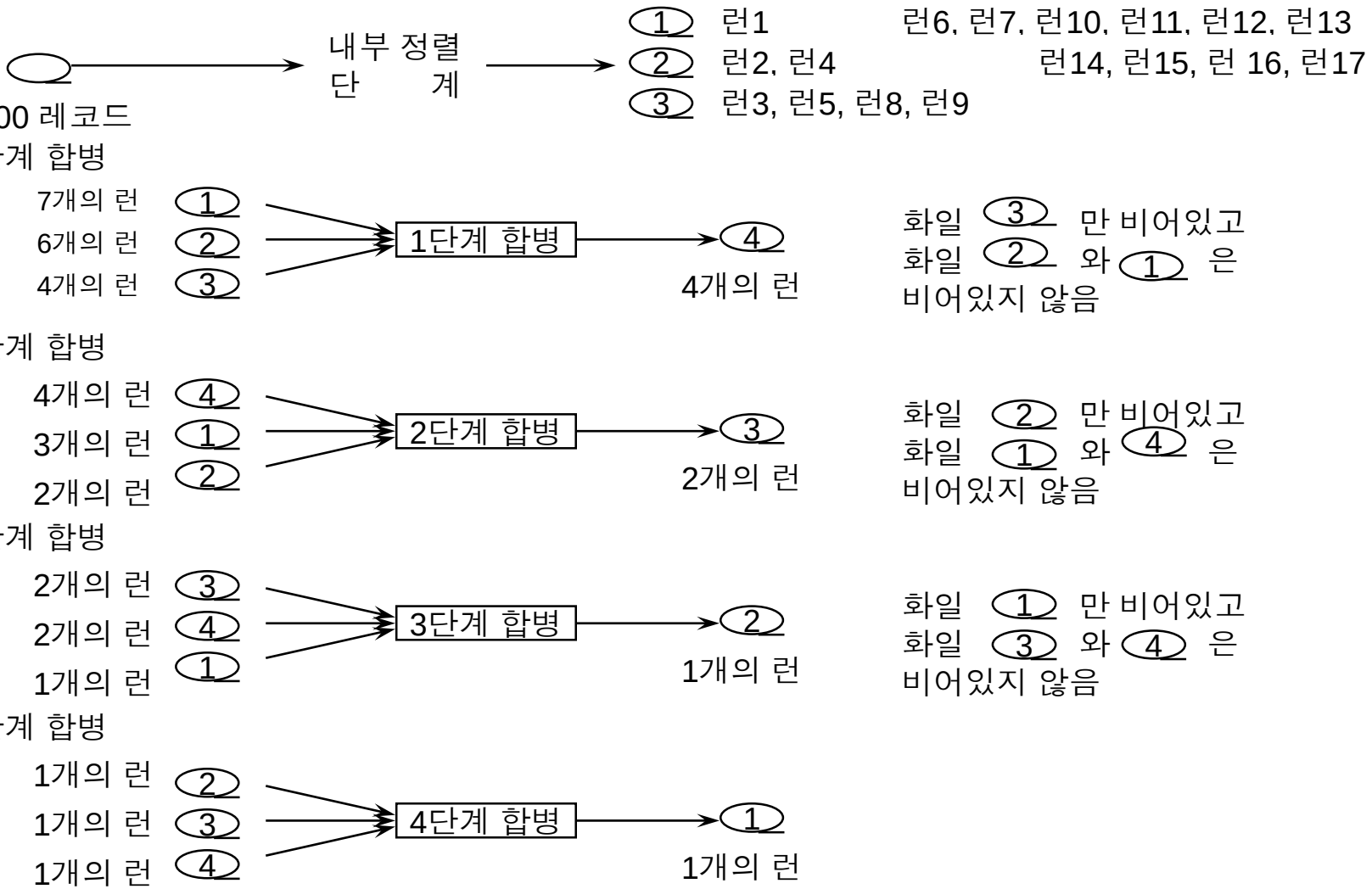
◆ m-원 다단계 합병

- m 개의 입력 화일 + 1 개의 출력 화일
- "불균형" 합병
- 초기 입력 화일에 대한 런의 분배가 복잡

◆ 각 합병단계(pass)

- 입력 화일의 어느 하나가 공백이 될 때까지 런들을 합병
- 공백이 된 입력 화일이 다음 합병 단계의 출력 화일이 됨

▶ 3-원 다단계 합병


6000 레코드
내부 정렬
단 계
1 런1
2 런2, 런4
3 런3, 런5, 런8, 런9
런6, 런7, 런10, 런11, 런12, 런13
런14, 런15, 런 16, 런17
1단계 합병
7개의 런
6개의 런
4개의 런
1
2
3
1단계 합병
4
4개의 런
화일 3 만 비어있고
화일 2 와 1 은
비어있지 않음
2단계 합병
4개의 런
3개의 런
2개의 런
4
1
2
2단계 합병
3
2개의 런
화일 2 만 비어있고
화일 1 와 4 은
비어있지 않음
3단계 합병
2개의 런
2개의 런
1개의 런
3
4
1
3단계 합병
2
1개의 런
화일 1 만 비어있고
화일 3 와 4 은
비어있지 않음
4단계 합병
1개의 런
1개의 런
1개의 런
2
3
4
4단계 합병
1
1개의 런

▶ 초기 런 분배 방법

- 런수의 변화 **(m = 3)**

  1, 1, 1, 3, 5, 9, 17, 31, …

- 피보나치**(Fibonacci)** 수열

  $T_i = T_i\text{-}1 + T_i\text{-}2 + T_i\text{-}3, \quad i > 3$

  $T_i = 1, \ i \leq 3$ 이 된다.

- 일반형

$$T_i = 1 , \ \ i \leq m$$

$$T_i = \sum_{k=i}^{i-1} T_k , i > m$$

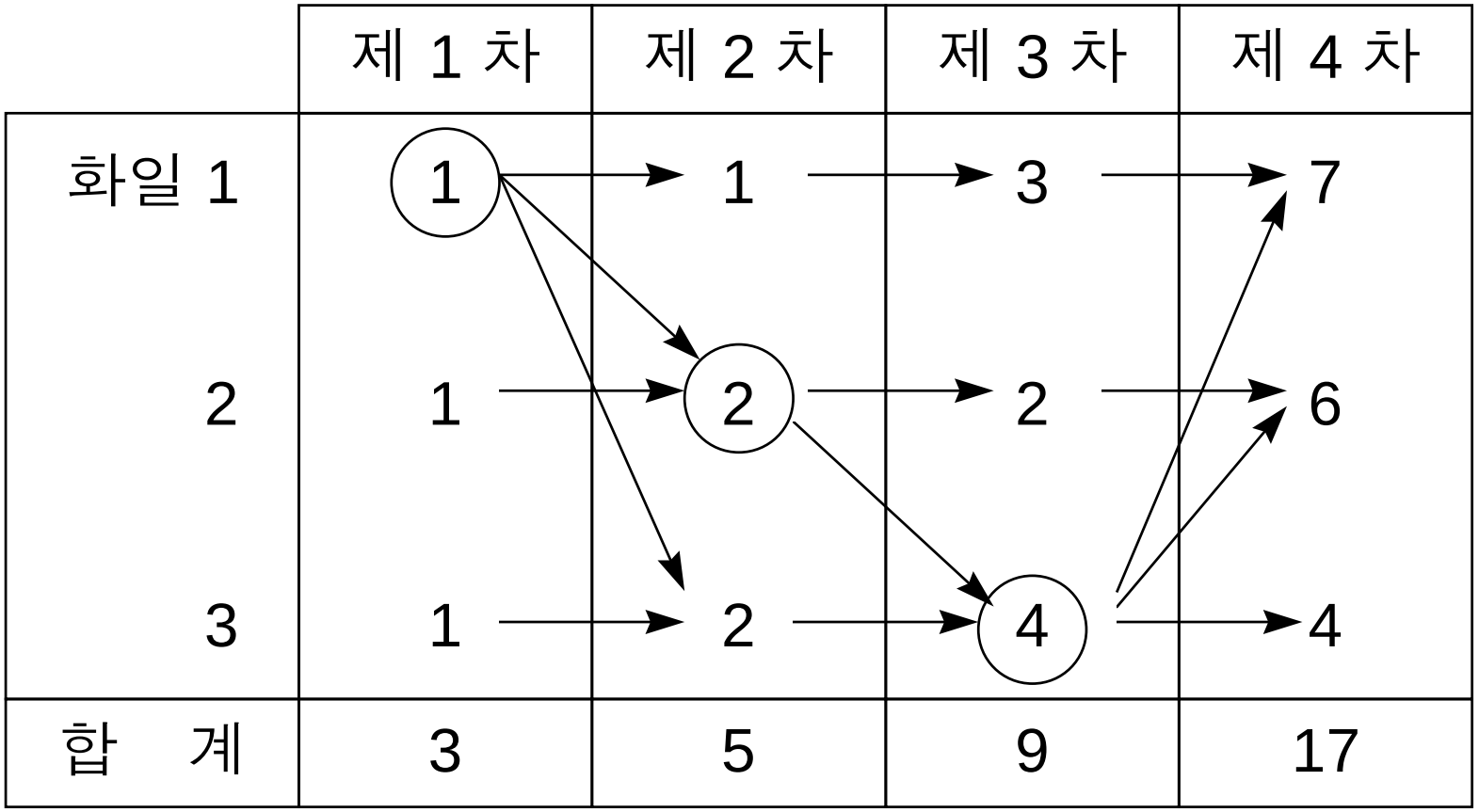
▶ m-원 다단계 합병 알고리즘

```
/*   m   : 입력 화일의 수
     FILE  : 화일 변수 또는 채널들의 배열
(초기 입력 화일에는 피보나치 수열에 따라 런들이 분산되어 있다고 가정)
   */
/* 화일의 준비 */
   for (i=1; i<=m; i++)
      open file on FILE[i] for reading;
   open file on FILE[m+1] for writing;

/* 합병 단계 */
   do {
      do merge run from files on FILE[1], ..., FILE[m]
           onto the file on FILE[m+1];
      while (!end-of-file(FILE[m]));
      rewind and close files on FILE[m] and FILE[m+1];
      open file on FILE[m+1] for reading;
      open file on FILE[m] for writing;

/* 화일을 배열에 재할당 */
      FILE[1], FILE[2], ..., FILE[m+1]
        = FILE[m+1], FILE[1], ..., FILE[m];
   } while (!end-of-file on all files on FILE[2], ..., FILE[m]);
/* 정렬된 최종 화일은 FILE[1] */
```

★ 초기 런 분배 방법 (m = 3, 화일수 = 17)



★ 각 합병 단계에서 화일당 런 수의 변화

	1단계 시작	1단계 끝	2단계 끝	3단계 끝	4단계 끝
화일 1	7	3	1	0	1
화일 2	6	2	0	1	0
화일 3	4	0	2	1	0
화일 4	0	4	2	1	0
합 계	17	9	5	3	1

❖ 계단식 합병 (cascade merge)

- 레코드의 복사작업을 줄이려는 불균형 합병의 또 다른 형태

◆ m-원 계단식 합병

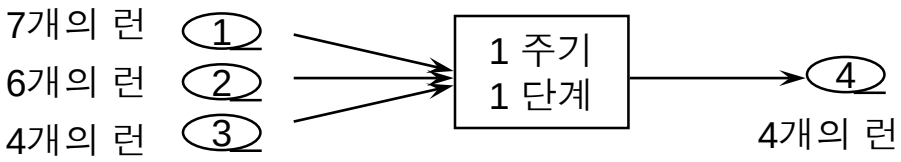
- 주기 : m, m-1, m-2, …, 그리고 마지막에 2개의 입력 화일을 사용
- 런 생성 단계에서 런의 초기 분배가 중요

◆ 합병 단계

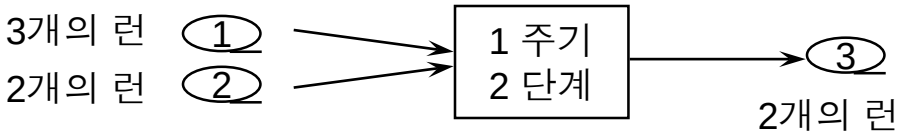
- m 입력화일을 하나의 출력화일로 합병
- 처음 공백이 되는 입력 화일이 새로운 출력 화일이 됨
- m-1 개의 입력 화일이 이 새로운 출력 화일로 합병
- 2개의 입력 화일을 합병하는 단계가 되면 합병의 한 주기가 종료
  - 한 주기에 각 레코드는 한번씩 처리

▶ 3-원 계단식 합병

1 주기 합병

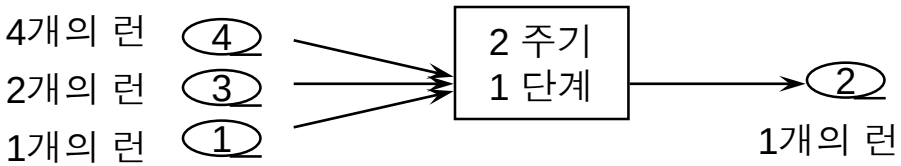


화일 3 은 비었음

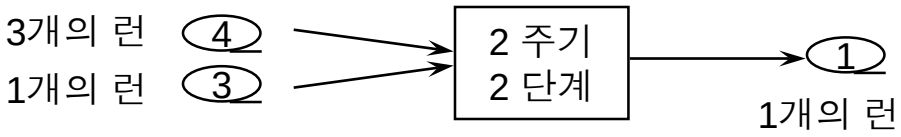


화일 2 은 비었음

2 주기 합병

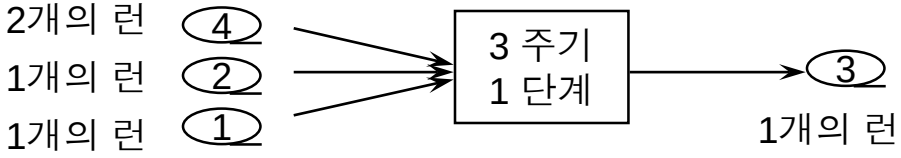


화일 1 은 비었음



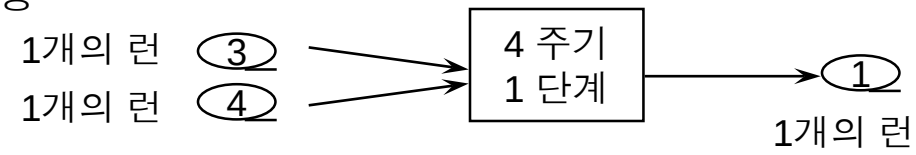
화일 3 은 비었음

3 주기 합병



화일 1 과 2 는 비었음

4 주기 합병



화일 2 3 4 는 비었음

▶ m-원 계단식 합병 알고리즘

```
/*   m      : 입력 화일의 수
     FILE   : 화일 변수 또는 채널들의 배열
   (초기 입력 화일에는 피보나치 수열에 따라 런들이 분산되어
    있다고 가정)   */
/* 화일의 준비 */
   for (i=1; i<=m; i++)
      open file on FILE[i] for reading;
   open file on FILE[m+1] for writing;

/* 합병 단계 */
   do {
      i = 0;
      do {
          do merge runs from files on FILE[1], ..., FILE[m-i+1] onto
                                    the file on FILE[m-i+2];
        while (!end-of-file(FILE[m-i+1]));
        rewind and close files on FILE[m-i+1] and FILE[m-i+2];
        /* 하나 이상의 화일이 비었을 경우 해당 단계를 생략함 */
        empty-file = true;
        k = 1;
```

```
while (empty-file && i < m-1) {
        if (end-of-file(FILE[m-i+1-k])) {
            i += 1;
            k += 1;
            rewind and close file on FILE[m-i+1-k];
        } else empty-file = false;
      }
      /* 다음 단계의 출력 화일 준비 */
      open file on FILE[m-i+1] for writing;
   } while(i != m-1);
 /* 화일을 런수의 내림차순으로 정렬한 후 배열에 재할당 */
   reallocate files to FILE[1], ..., FILE[m+1] such that
      run count  of FILE[1]≥run count  of FILE[2]≥...≥run
                                                count of  FILE[m+1];
   /* 남은 화일수의 재조정 */
   no-more-empty = false;
   do {
       if (end-of-file(FILE[k])) m -= 1;
       else no-more-empty = true;
   } while (m != 1 && !no-more-empty);
 } while(m != 1);
 /* 최종 정렬 화일은 FILE[1] */
```

❖ 유틸리티에 의한 정렬 합병

- **정렬 합병 유틸리티 (utility)**
  - 범용의 화일 정렬 합병을 지원

- **유틸리티의 기능**
  (1) 하나 또는 그 이상의 화일 정렬
  (2) 둘 또는 그 이상의 화일 합병
  (3) 둘 또는 그 이상의 화일 정렬과 합병

- **명세 내용**
  (1) 정렬 합병할 화일의 이름
  (2) 정렬 합병의 키로 사용될 필드의 데이타 타입, 길이, 위치
  (3) 키 필드들의 순서(주에서 보조순으로)
  (4) 각 키 필드에 적용할 배열 순서 (오름차순 또는 내림차순)
  (5) 각 키 필드에 적용될 순서 기준
  (6) 정렬 합병 결과를 수록할 출력화일의 이름

▶ 세밀한 사항의 지시

(1) 사용자가 정의한 정렬 순서 및 기준

(2) 내부 정렬 단계에서 사용할 알고리즘
(예 : 퀵(quick), 힙(heap))

(3) 합병 단계에서 사용할 알고리즘
(예 : 균형, 다단계, 계단식 합병)

(4) 화일 사용 전후에 필요한 동작
(예 : rewind, unload)

(5) 합병 단계에서 입력 레코드가 올바른 순서로 되어 있는가의 검증

(6) 회복(recovery)을 위해서 체크 포인트/덤프 레코드를 사용하는 주기

(7) 예상 입력 레코드 수

▶ 정렬 합병의 예

```
/ / SORTNOW      EXEC  SORTMRG
/ / SORTIN       DD     DSN = name of input file, ....,
/ /                        DISP = (OLD, KEEP)
/ / SORTOUT      DD     DSN = name of output file, ...,
/ /                       DISP = (NEW, KEEP)
/ / SYSIN        DD          *
   SORT FILDS=(1,4,CH,A,20,10,CH,D), FILEZ=E2000
/ *
```

```
SORTMRG(input-filename, output-filename)
...
SORT,   VAR = POLY
FILE,    INPUT = name(CU), OUTPUT = name(R)
FIELD,  DEPT(1,4,ASCII6), SALEDATE(20, 10, ASCII6)
KEY,    DEPT(A,ASCII6), SALEDATE(D, ASCII6)
```

★ 제어 카드 정렬(control card sort)

❖ 정렬 합병의 성능

- 성능 평가의 요소

(1) 정렬 합병되는 레코드의 수
(2) 레코드의 크기
(3) 이용될 저장 장치의 수
(4) 이용가능 I/O 채널에서의 저장 장치 분포
(5) 입력 화일에서 키값의 분포