

## **5. 화일의 정렬/합병**

## ❖ 정렬/합병의 개요

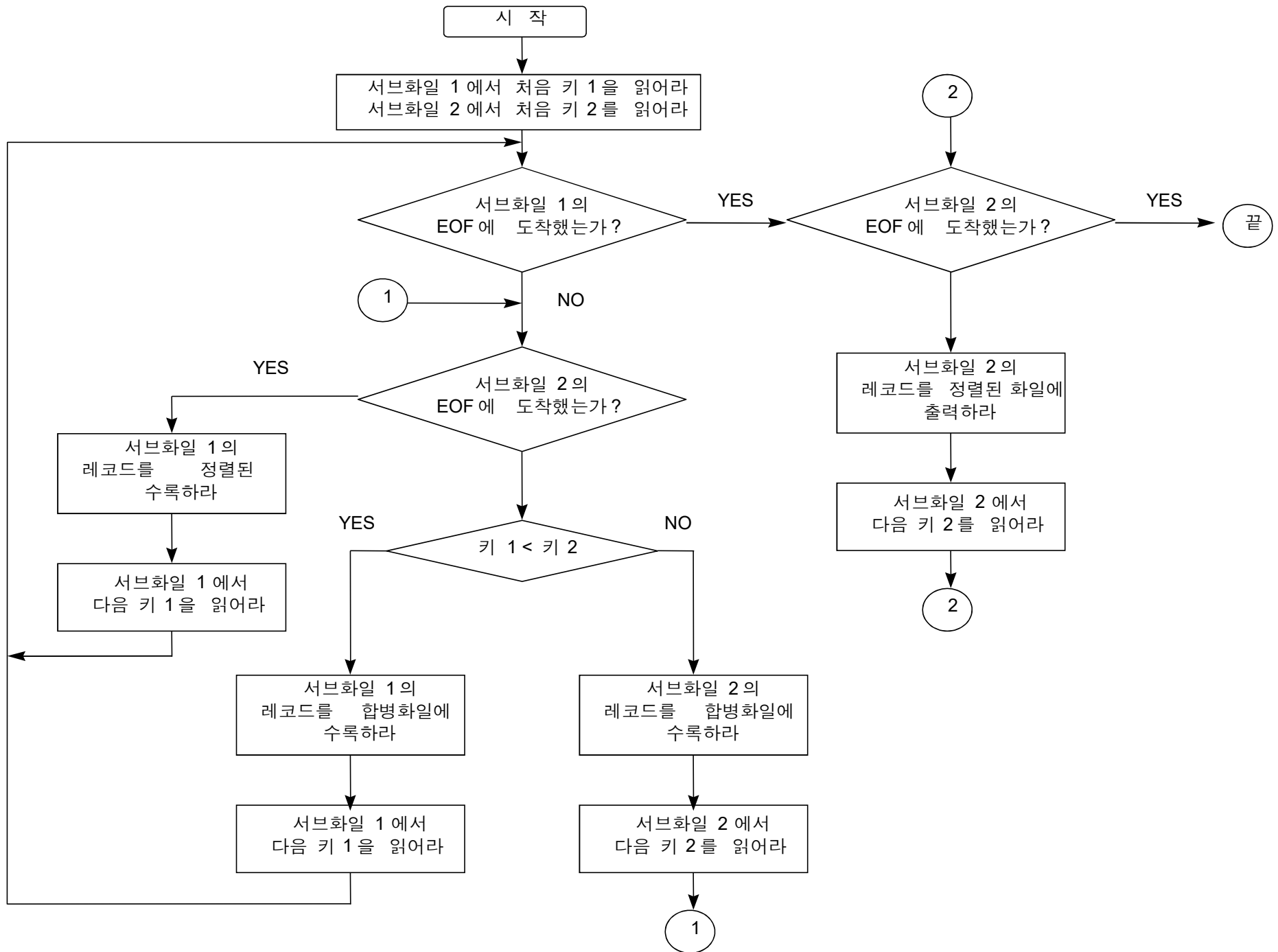
### ◆ 정렬(sorting)

#### – 내부 정렬(internal sorting)

- ◆ 데이터가 적어서 메인 메모리 내에 모두 저장시켜 정렬 가능할 때
- ◆ 레코드 판독, 기록에 걸리는 시간이 문제되지 않는다.

#### – 외부 정렬(external sorting)

- ◆ 데이터가 많아서 메인 메모리의 용량을 초과하여 보조 저장 장치에 저장된 파일을 정렬할 때
- ◆ 레코드 판독, 기록에 걸리는 시간이 중요
- ◆ 정렬/합병(sort/merge)
  - 런(run) : 하나의 파일을 여러 개의 서브파일(subfile)로 나누어 내부 정렬 기법을 사용하여 정렬시킨 파일
  - 런을 합병하여 원하는 하나의 정렬된 파일을 만든다.



## ▶ 화일 정렬/합병

### ◆ 정렬 단계(sort phase)

- 정렬할 화일의 레코드들을 지정된 길이의 서브화일로 분할
- 내부 정렬(internal sort)하여 런(run)을 만들어 입력 파일로 분배하는 단계

### ◆ 합병 단계(merge phase)

- 정렬된 런들을 합병해서 보다 큰 런으로 만들고, 이것들을 다시 입력 파일로 재분배하여 합병하는 방식으로 모든 레코드들이 하나의 런에 포함되도록 만드는 단계

## ▶ 정렬 단계

### ◆ 런 생성 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)

## ◆ 런 생성 방법

1. 파일을 m개 레코드씩 분할
2. 분할된 레코드들을 내부 정렬 기법으로 정렬

## ◆ 런 생성 결과

- 제일 마지막 런을 제외하고 모두 길이가 같다.
- 가정 : 메인 메모리는 5개의 레코드를 유지

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)

### ◆ 런 생성 방법

1. 입력 파일에서 버퍼에  $m$ 개 레코드 판독, 첫 번째 런 생성
2. 버퍼에서 최소 키 값의 레코드를 선택해 출력
3. 입력 파일에서 다음 레코드 판독, 출력된 레코드와 대체
  - ◆ if (입력 레코드의 키 값 < 출력된 레코드의 키 값)  
then 입력 레코드에 “동결(frozen)” 표시하고, 출력을 동결시킴
  - ◆ 동결된 레코드는 단계 2의 선택에서 제외
  - ◆ 동결되지 않은 레코드는 단계 2로 돌아간다.
4. 동결된 레코드를 모두 해제, 단계 2로 돌아가 새로운 런  
선택

### ◆ 특징

- 내부 정렬과 달리 입력 파일의 일부 정렬(partial ordering)된 레코드들의 순서를 이용
- 내부 정렬을 이용한 경우보다 런의 길이가 길다.
- 런의 평균 예상 길이 :  $2m$

## ◆ 예

### – 입력

- ◆ 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

### – 파라미터 설정

- ◆  $m = 5$
- ◆ 주기억장치는  $m$ 개의 레코드를 저장할 수 있어야 함

### – 리스트의 저장

- ◆ Unfrozen list : 주기억장치
- ◆ Frozen list : 주기억장치

### – 특징

- ◆ Unfrozen list와 frozen list 를 합하여 항상  $m$ 개 레코드가 주기억장치에 존재



## – 런 1의 생성

### ◆ Initial state

109(F,F) 49(F,F) 34(F,F) 68(F,F) 45(F,F)

2(F,F) 60(F,F) 38(F,F) 28(F,F) 47(F,F) 16(F,F) 19(F,F) . . .

### ◆ Last key : input

		unfrozen={109,49,34,68,45}	frozen={}
34 (T, <b>T</b> ) > 2 ( <b>T</b> ,F)		unfrozen={109,49,68,45}	frozen={ <b>2</b> }
45 (T, <b>T</b> ) < 60 ( <b>F</b> ,F)		unfrozen={109,49,68, <b>60</b> }	frozen={2}
49 (T, <b>T</b> ) > 38 ( <b>T</b> ,F)		unfrozen={109,68,60}	frozen={2, <b>38</b> }
60 (T, <b>T</b> ) > 28 ( <b>T</b> ,F)		unfrozen={109,68}	frozen={2,38, <b>28</b> }
68 (T, <b>T</b> ) > 47 ( <b>T</b> ,F)		unfrozen={109}	frozen={2,38,28, <b>47</b> }
109 (T, <b>T</b> ) > 16 ( <b>T</b> ,F)		unfrozen={}	frozen={2,38,28,47, <b>16</b> }

### ◆ 참고

- 데이타구조 : key (frozen, written)

## – 런 2의 생성

### ◆ Initial state


2(F,F) 38(F,F) 28(F,F) 47(F,F) 16(F,F)

19(F,F) 34(F,F) 55(F,F) 98(F,F) 78(F,F) 76(F,F) 40(F,F)  
35(F,F) 86(F,F) 10(F,F) 27(F,F) 61(F,F) 92(F,F) . . .

### ◆ Last key : input

2 (T,T) < 19 (F,F)  
16 (T,T) < 34 (F,F)  
19 (T,T) < 55 (F,F)  
28 (T,T) < 98 (F,F)  
34 (T,T) < 78 (F,F)  
38 (T,T) < 76 (F,F)  
47 (T,T) > 40 (T,F)  
55 (T,T) > 35 (T,F)  
76 (T,T) < 86 (F,F)  
78 (T,T) > 10 (T,F)  
86 (T,T) > 27 (T,F)  
98 (T,T) > 61 (T,F)

unfrozen={2,38,28,47,16}	frozen={}
unfrozen={38,28,47,16,19}	frozen={}
unfrozen={38,28,47,19,34}	frozen={}
unfrozen={38,28,47,34,55}	frozen={}
unfrozen={38,47,34,55,98}	frozen={}
unfrozen={38,47,55,98,78}	frozen={}
unfrozen={47,55,98,78,76}	frozen={40}
unfrozen={55,98,78,76}	frozen={40,35}
unfrozen={98,78,76}	frozen={40,35,10}
unfrozen={98,78,86}	frozen={40,35,10,27}
unfrozen={98,86}	frozen={40,35,10,27,61}
unfrozen={98}	
unfrozen={}	



- 런3의 생성

- ◆ ...

- 출력

- ◆ 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

```
replacementSelection()           // 대체 선택 알고리즘
// m                             : 버퍼에 들어가는 레코드 수
// Buffer[]                       : 버퍼-레코드 배열
// written[]                      : 해당 버퍼 레코드가 출력되었는지를 나타내는 플래그 배열
// frozen[]                       : 해당 버퍼 레코드가 동결되었는지를 나타내는 플래그 배열
// lastKey                        : 마지막으로 런에 출력된 레코드 키 값
// input                          : 입력 화일

for (i ← 0; i < m; i++)
    written[i] ← true;
i ← 0;
do {
    readFrom(Buffer[i], input);
    written[i] ← false;
    i ← i + 1;
} while (!end-of-file(input) && i ≠ m);

// 런의 생성
while (!end-of-file(input)) {
    // 새로운 런 하나를 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  
    Buffer[s] ← smallest unfrozen record;  
    appendToRun(Buffer[s]);  
    lastKey ← Buffer[s].key;  
    written[s] ← true;  
    frozen[s] ← true;  
    if (!end-of-file(input)) {  
        readFrom(new Buffer[s], input);  
        written[s] ← false;  
        if (Buffer[s].key > lastKey)  
            frozen[s] ← false;  
    }  
}
```

```
// 버퍼에 있는 나머지 레코드들을 출력  
appendToRun(unwritten Buffer[] records, ascending key order);  
end replacementSelection()
```

### 3) 자연 선택 (natural selection)

#### ◆ 대체 선택의 단점 보완

- 대체 선택의 단점
  - ◆ 런 생성 직전에 주기억장치내 대부분 레코드가 동결 상태
  - ◆ 동결된 레코드는 런 생성에 전혀 도움을 주지 못함
- 동결된 레코드를 보조기억장치의 예비장소(reservoir)에 저장
  - ◆ Unfrozen list : 주기억장치
  - ◆ Frozen list : 보조기억장치의 예비장소
- 주기억장치에 가능한 많은 레코드를 유지하여, 런을 길게 함
- 런을 길게 하여, 런 수를 줄임 → 합병 과정 비용을 줄임

#### ◆ 특징

- 하나의 런은
  - ◆ 예비장소가 꽉 차거나,
  - ◆ 입력 화일이 빌 때까지 생성이 계속됨
- 주기억장치에 저장 가능한 레코드수 :  $m$   
예비 장소에 저장 가능한 레코드수 :  $m'$ 
  - ◆ 일반적으로  $m \ll m'$

## ◆ 예

### – 입력

◆ 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

### – 파라미터 설정

◆  $m = m' = 5$

## – 런 1의 생성

### ◆ Initial state

109(F,F) 49(F,F) 34(F,F) 68(F,F) 45(F,F)

2(F,F) 60(F,F) 38(F,F) 28(F,F) 47(F,F) 16(F,F) 19(F,F) 34(F,F) . .

### ◆ Last key : input

	unfrozen={109,49,34,68,45}	frozen={}
34 (T,T) > 2 (T,F)	unfrozen={109,49,68,45}	frozen={2}
34 (T,T) < 60 (F,F)	unfrozen={109,49,68,45,60}	frozen={2}
45 (T,T) > 38 (T,F)	unfrozen={109,49,68,60}	frozen={2,38}
45 (T,T) > 28 (T,F)	unfrozen={109,49,68,60}	frozen={2,38,28}
45 (T,T) > 47 (F,F)	unfrozen={109,49,68,60,47}	frozen={2,38,28}
47 (T,T) > 16 (T,F)	unfrozen={109,49,68,60}	frozen={2,38,28,16}
47 (T,T) > 19 (T,F)	unfrozen={109,49,68,60}	frozen={2,38,28,16,19}
(47 > 34)		
49 (T,T)	unfrozen={109,68,60}	frozen={2,38,28,16,19}
60 (T,T)	unfrozen={109,68}	frozen={2,38,28,16,19}
68 (T,T)	unfrozen={109}	frozen={2,38,28,16,19}
109 (T,T)	unfrozen={}	frozen={2,38,28,16,19}



## – 런 2의 생성

### ◆ Initial state

2(F,F) 38(F,F) 28(F,F) 16(F,F) 19(F,F)

34(F,F) 55(F,F) 98(F,F) 78(F,F) 76(F,F) 40(F,F) 35(F,F) 86(F,F)  
10(F,F) 27(F,F) 61(F,F) 92(F,F) 99(F,F) 72(F,F) 11(F,F) 2(F,F) . .

### ◆ Last key : input

2 (T,T) < 34 (F,F)	unfrozen={2,38,28,16,19}	frozen={}
16 (T,T) < 55 (F,F)	unfrozen={38,28,16,19,34}	frozen={}
19 (T,T) < 98 (F,F)	unfrozen={38,28,19,34,55}	frozen={}
28 (T,T) < 78 (F,F)	unfrozen={38,28,34,55,98}	frozen={}
34 (T,T) < 76 (F,F)	unfrozen={38,34,55,98,78}	frozen={}
38 (T,T) < 40 (F,F)	unfrozen={38,55,98,78,76}	frozen={}
40 (T,T) > 35 (T,F)	unfrozen={55,98,78,76,40}	frozen={35}
40 (T,T) < 86 (F,F)	unfrozen={55,98,78,76,86}	frozen={35}
55 (T,T) > 10 (T,F)	unfrozen={98,78,76,86}	frozen={35,10}
55 (T,T) > 27 (T,F)	unfrozen={98,78,76,86}	frozen={35,10,27}
55 (T,T) < 61 (F,F)	unfrozen={98,78,76,86,61}	frozen={35,10,27}
61 (T,T) < 92 (F,F)	unfrozen={98,78,76,86,92}	frozen={35,10,27}
76 (T,T) < 99 (F,F)	unfrozen={98,78,86,92,99}	frozen={35,10,27}
78 (T,T) > 72 (T,F)	unfrozen={98,86,92,99}	frozen={35,10,27,72}
78 (T,T) > 11 (T,F)	unfrozen={98,86,92,99}	frozen={35,10,27,72,11}
(78 > 2)		
86 (T,T)	unfrozen={98,92,99}	frozen={35,10,27,72,11}
92 (T,T)	unfrozen={98,99}	frozen={35,10,27,72,11}
98 (T,T)	unfrozen={99}	frozen={35,10,27,72,11}
99 (T,T)	unfrozen={}	frozen={35,10,27,72,11}

## – 런 3의 생성

### ◆ Initial state

35(F,F) 10(F,F) 27(F,F) 72(F,F) 11(F,F)

2(F,F) 29(F,F) 16(F,F) 80(F,F) 73(F,F) 18(F,F) 12(F,F) 89(F,F)  
50(F,F) 46(F,F) 36(F,F) 67(F,F) 93(F,F) 22(F,F) . . .

### ◆ Last key : input

	unfrozen={35,10,27,72,11}	frozen={}
10 (T, <b>T</b> ) > 2 ( <b>T</b> ,F)	unfrozen={35,27,72,11}	frozen={ <b>2</b> }
10 (T, <b>T</b> ) < 29 ( <b>F</b> ,F)	unfrozen={35,27,72,11, <b>29</b> }	frozen={2}
11 (T, <b>T</b> ) < 16 ( <b>F</b> ,F)	unfrozen={35,27,72,29, <b>16</b> }	frozen={2}
16 (T, <b>T</b> ) < 80 ( <b>F</b> ,F)	unfrozen={35,27,72,29, <b>80</b> }	frozen={2}
27 (T, <b>T</b> ) < 73 ( <b>F</b> ,F)	unfrozen={35,72,29,80, <b>73</b> }	frozen={2}
29 (T, <b>T</b> ) > 18 ( <b>T</b> ,F)	unfrozen={35,72,80,73}	frozen={2,18}
29 (T, <b>T</b> ) > 12 ( <b>T</b> ,F)	unfrozen={35,72,80,73}	frozen={2,18, <b>12</b> }
29 (T, <b>T</b> ) < 89 ( <b>F</b> ,F)	unfrozen={35,72,80,73, <b>89</b> }	frozen={2,18,12}
35 (T, <b>T</b> ) < 50 ( <b>F</b> ,F)	unfrozen={72,80,73,89, <b>50</b> }	frozen={2,18,12}
50 (T, <b>T</b> ) > 46 ( <b>T</b> ,F)	unfrozen={72,80,73,89}	frozen={2,18,12, <b>46</b> }
50 (T, <b>T</b> ) > 36 ( <b>T</b> ,F)	unfrozen={72,80,73,89}	frozen={2,18,12,46, <b>36</b> }
50 (T, <b>T</b> ) < <b>67</b> ( <b>F</b> ,F)	unfrozen={72,80,73,89, <b>67</b> }	frozen={2,18,12,46,36}
67 (T, <b>T</b> ) < <b>93</b> ( <b>F</b> ,F)	unfrozen={72,80,73,89, <b>93</b> }	frozen={2,18,12,46,36}
(72 > 22)		
72 (T, <b>T</b> )	unfrozen={80,73,89,93}	frozen={2,18,12,46,36}
73 (T, <b>T</b> )	unfrozen={80,89,93}	frozen={2,18,12,46,36}
80 (T, <b>T</b> )	unfrozen={89,93}	frozen={2,18,12,46,36}
89 (T, <b>T</b> )	unfrozen={93}	frozen={2,18,12,46,36}
93 (T, <b>T</b> )	unfrozen={}	frozen={2,18,12,46,36} <sub>18</sub>

## – 런 4의 생성

### ◆ Initial state

2(F,F) 18(F,F) 12(F,F) 46(F,F) 36(F,F)

22(F,F) 14(F,F) 83(F,F) 44(F,F) 52(F,F) 59(F,F) 10(F,F) 38(F,F)  
76(F,F) 16(F,F) 24(F,F) 85(F,F)

### ◆ Last key : input

2 (T, <b>T</b> ) < 22 ( <b>F</b> ,F)	unfrozen={2,18,12,46,36}	frozen={}
12 (T, <b>T</b> ) < 14 ( <b>F</b> ,F)	unfrozen={18,12,46,36, <b>22</b> }	frozen={}
14 (T, <b>T</b> ) < 83 ( <b>F</b> ,F)	unfrozen={18,46,36,22, <b>14</b> }	frozen={}
18 (T, <b>T</b> ) < 44 ( <b>F</b> ,F)	unfrozen={18,46,36,22, <b>83</b> }	frozen={}
22 (T, <b>T</b> ) < 52 ( <b>F</b> ,F)	unfrozen={46,36,22,83, <b>44</b> }	frozen={}
36 (T, <b>T</b> ) < 59 ( <b>F</b> ,F)	unfrozen={46,36,83,44, <b>52</b> }	frozen={}
44 (T, <b>T</b> ) > 10 ( <b>T</b> ,F)	unfrozen={46,83,44,52, <b>59</b> }	frozen={}
44 (T, <b>T</b> ) > 38 ( <b>T</b> ,F)	unfrozen={46,83,52,59}	frozen={10}
44 (T, <b>T</b> ) < 76 ( <b>F</b> ,F)	unfrozen={46,83,52,59}	frozen={10,38}
46 (T, <b>T</b> ) > 16 ( <b>T</b> ,F)	unfrozen={46,83,52,59, <b>76</b> }	frozen={10,38}
46 (T, <b>T</b> ) > 24 ( <b>T</b> ,F)	unfrozen={83,52,59,76}	frozen={10,38,16}
46 (T, <b>T</b> ) < <b>85</b> ( <b>F</b> ,F)	unfrozen={83,52,59,76}	frozen={10,38,16,24}
(EOF)	unfrozen={83,52,59,76, <b>85</b> }	frozen={10,38,16,24}
52 (T, <b>T</b> )	unfrozen={83,59,76,85}	frozen={10,38,16,24}
59 (T, <b>T</b> )	unfrozen={83,76,85}	frozen={10,38,16,24}
76 (T, <b>T</b> )	unfrozen={83,85}	frozen={10,38,16,24}
83 (T, <b>T</b> )	unfrozen={85}	frozen={10,38,16,24}
85 (T, <b>T</b> )	unfrozen={}	frozen={10,38,16,24}

– 런5의 생성

◆ Initial state

10(F,F) 38(F,F) 16(F,F) 24(F,F)

◆ Last key : input

10 (T,T)	unfrozen={10,38,16,24}	frozen={}
16 (T,T)	unfrozen={38,16,24}	frozen={}
24 (T,T)	unfrozen={38,24}	frozen={}
38 (T,T)	unfrozen={38}	frozen={}
	unfrozen={}	frozen={}

– 출력

◆ 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

## ▶ 자연 선택 알고리즘

```
naturalSelection()    // 자연 선택 알고리즘
// m, m'              : 버퍼와 외부 예비 장소의 레코드 수
// Buffer[]           : 버퍼-레코드 배열
// written[]          : 해당 버퍼 레코드가 출력되었는지를 나타내는 플래그 배열
// reservoir-count     : 예비 장소에 저장된 레코드 수
// spaceFull           : 예비 장소 오버플로를 나타내는 플래그
// lastKey             : 마지막으로 출력된 레코드 키 값
// input              : 입력 화일

for ( i ← 0; i < m; i++)      written[i] ← true;
i ← 0;
do {
    readFrom(Buffer[i], input);
    written[i] ← false;
    i ← i + 1;
} while (!end-of-file(input) && i ≠ m);
reservoir-count ← 0;

// 화일로로부터 읽어 런에 출력
do {                          // 런 하나를 생성
    spaceFull ← false;
    do { // 레코드 하나를 출력
        Buffer[s] ← smallest unwritten record;
        appendToRun(Buffer[s]);
        lastKey ← Buffer[s].key;
        written[s] ← true;
```

```

do {
  if (!end-of-file(input)) {
    read(new Buffer[s], input);
    if (Buffer[s].key ≥ lastKey) {
      written[s] ← false;
    } else {
      move Buffer[s] to reservoir;
      reservoir-count ← reservoir-count + 1;
      if (reservoir-count = m') spaceFull ← true;
    }
  }
} while (written[s] && !spaceFull && !end-of-file(input));
} while (!end-of-file(input) && !spaceFull);
appendToRun(unwritten Buffer[] records, ascending key order);
setTrue(corresponding elements of written[]);

// 다음 런을 생성하기 위해 버퍼 정리
for (i ← 0; i < min(reservoir-count, m); i++) {
  moveTo(a record from reservoir, Buffer[i]);
  written[i] ← false;
  reservoir-count ← reservoir-count - 1;
}
while (Buffer[] not full && !end-of-file(input)) {
  moveTo(a record from input, Buffer[i]);
  written[i] ← false;
}
} while (unwritten records exist in Buffer[]);
end naturalSelection()

```

## ▶ 런 생성 방법의 비교

### ◆ 내부 정렬

- 마지막 런을 제외하고 모든 런들의 길이가 같음
- 런의 길이를 예측할 수 있으므로 합병 알고리즘이 간단

### ◆ 대체 선택

- 내부 정렬보다 평균적으로 훨씬 긴 런을 생성
- 런의 길이가 길수록 합병 비용이 적음
- 런의 길이가 일정치 않아 정렬/합병 알고리즘이 복잡

### ◆ 자연 선택

- 앞의 두 방법보다 더 긴 런을 생성할 수 있음
- 예비 장소로의 입출력이 문제가 됨
- 긴 런 생성에 따른 효율화가 예비 장소 전송 비용보다 이익이 될 수도 있음

## ▶ 화일 분할 방법의 평가

### ◆ 성능 평가 요인

- 1. 런이 길면 합병 비용이 적게 든다
  - ◆ 자연 선택 < 대체 선택 < 내부 정렬
- 2. 런의 길이를 예측하면 합병 알고리즘이 간단해 진다.
  - ◆ 내부 정렬 < 대체 선택 < 자연 선택
- 3. 런 생산 과정에서의 보조기억장치로의 입출력은 성능을 저하시킨다.
  - ◆ 내부 정렬, 대체 선택 < 자연 선택
- 평가
  - ◆ 자연 선택의 경우 초기에 긴 런을 생성하여 합병 단계의 효율화를 꾀하는 것이, 예비 장소로의 전송에 드는 비용보다 이익이 될 수 있다.



## ▶ 화일 정렬/합병 방법의 차이점

### ◆ 차이점을 나타내는 매개 변수

- 적용하는 내부 정렬 방식
- 내부정렬을 위해 할당된 메인 메모리의 크기
- 정렬된 런들의 보조 저장 장치에서의 저장 분포
- 정렬/합병 단계에서 동시에 처리할 수 있는 런의 수

### ◆ 정렬/합병 기법의 성능

- 매개 변수에 따른 런의 수와 합병의 패스(pass) 수 결정
- 보조 저장 장치의 상대적 참조 빈도수(reference frequency)
  - ◆ 전체 레코드 집합에 대해 수행되어야 할 정렬/합병의 패스 수 비교
- 성능에 영향을 미치는 요인
  - ◆ 가능한 런의 길이를 길게 만들어 런의 수를 최소화
  - ◆ 동시에 합병할 수 있는 런의 수를 늘리면 합병의 패스 수는 감소
  - ◆ 보조 저장 장치에 분산 저장하면 부수적인 입출력 연산 동반

# 화일 정렬 합병

## ◆ 원리

- 런 (정렬된 서브화일) = 순차화일
- 런의 합병: 순차 화일에서 마스터 화일과 로그 화일의 합병과 유사

## ◆ 합병 방법

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

## ◆ 합병 방식들의 차이점

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

## ❖ m-원 자연 합병(m-way natural merge)

### ◆ 동작 원리

- 1. m개의 입력화일을 1개 출력화일로 합병
  - ◆ 사용 화일의 수 = m+1
- 2. 1의 결과로 생성된 서브화일을 다음 단계(pass)에서의 m개의 입력화일로 재분배

### ◆ 특징

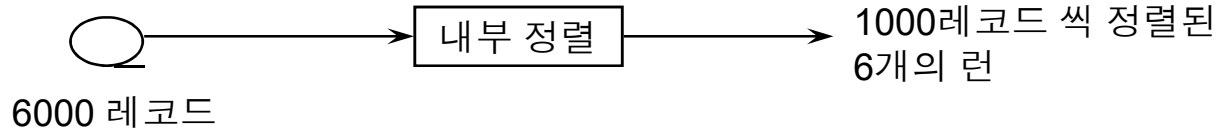
- 많은 입출력 : 한 패스에 합병이 끝나지 않으면 런들을 다시 분배하기 위해 복사, 이동해야 함
- 이상적 정렬/합병 : m개의 런에 m개의 입력 파일 사용하여 한번의 m-원 합병을 적용

### ◆ 2-원 합병의 경우

- 한번 패스 : 합병된 런의 크기는 2배, 런의 수는 반
- N개 런으로 분할된 파일 정렬 위한 단계(pass) 수 :  $\lceil \log_2 N \rceil$

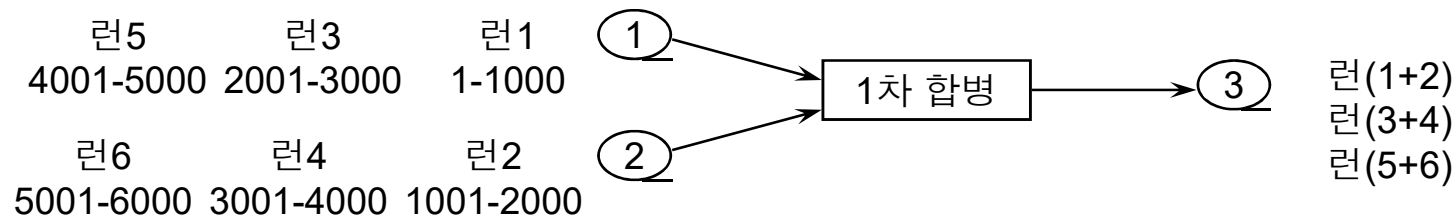
## ▶ 6 개의 런에 대한 2-원 합병

(1) 정렬단계



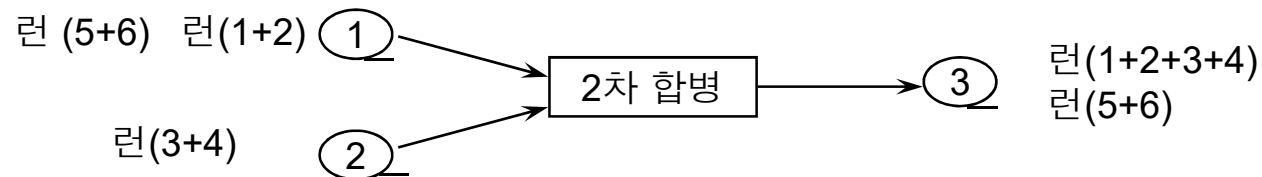
정렬된 6개의 런을 2개의 화일에 분배한다.

(2) 1단계 합병

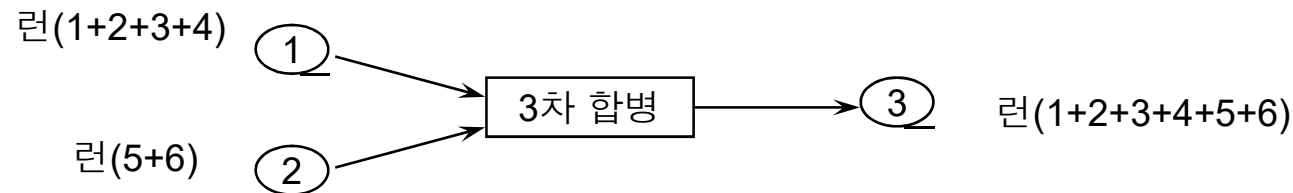


3 에 있는 3개의 런을 2개의 화일에 분배한다.

(4) 2단계 합병



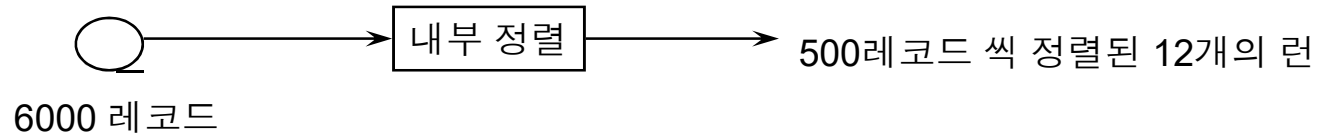
(5) 3단계 합병



(2개의 입력 화일과 1개 출력 화일)

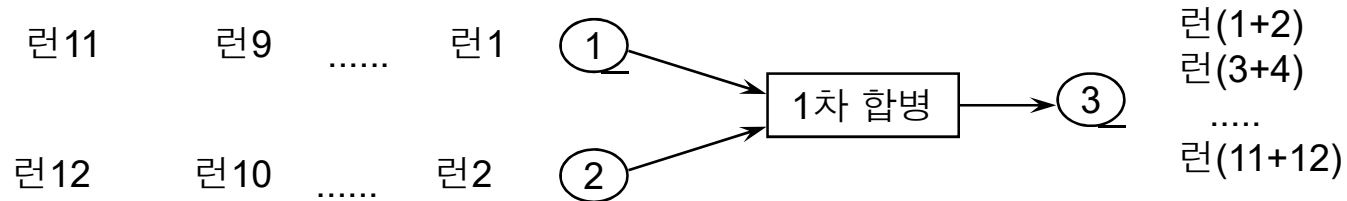
# ▶ 12개의 런에 대한 2-원 합병

(1) 정렬단계



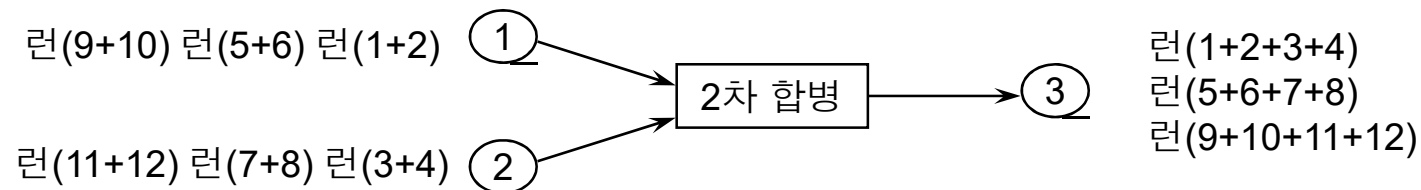
정렬된 12개의 런을 2개의 화일에 분배한다.

(2) 1차 합병



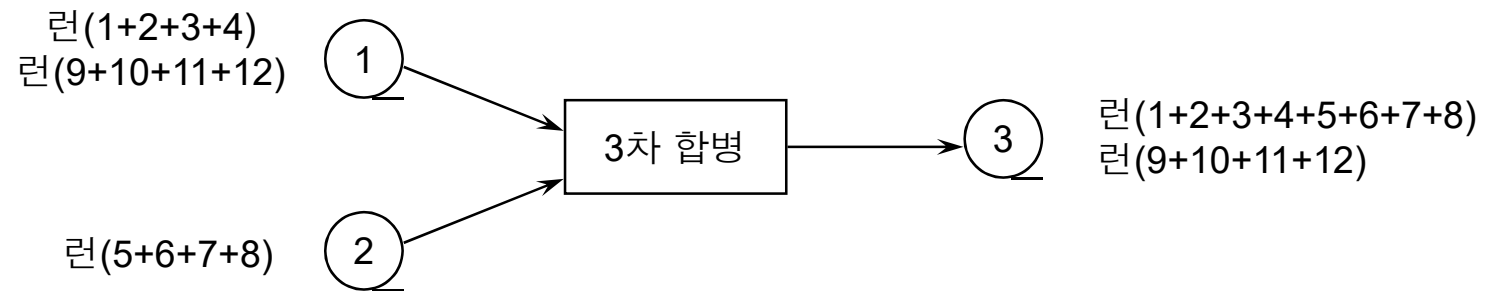
③ 에 있는 6개의된 런을 2개의 화일에 분배한다.

(3) 2차 합병

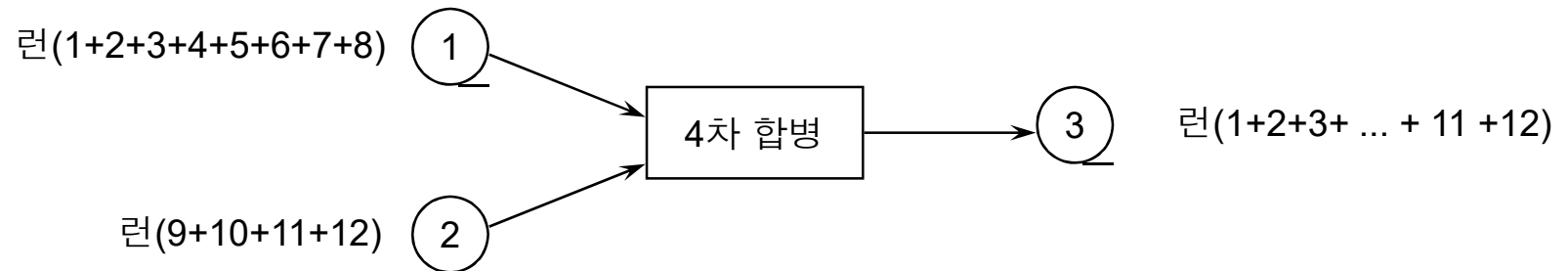


③ 에 있는 3개의된 런을 2개의 화일에 분배한다.

(4) 3차 합병

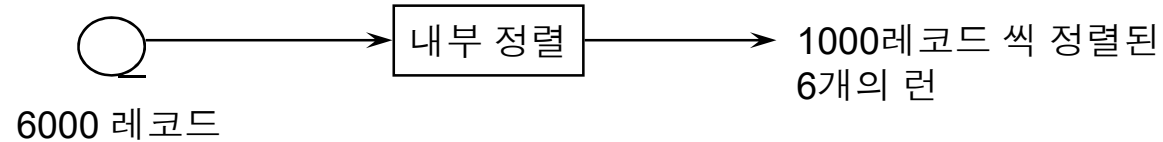


(5) 4차 합병



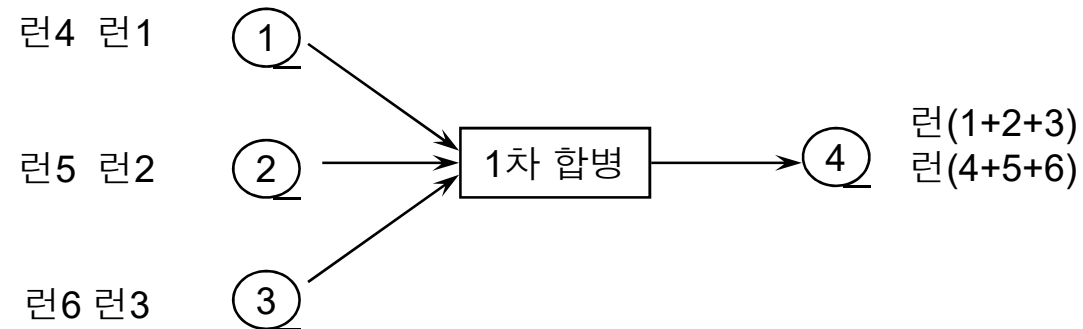
## ▶ 6개의 런에 대한 3-원 합병

(1) 정렬단계



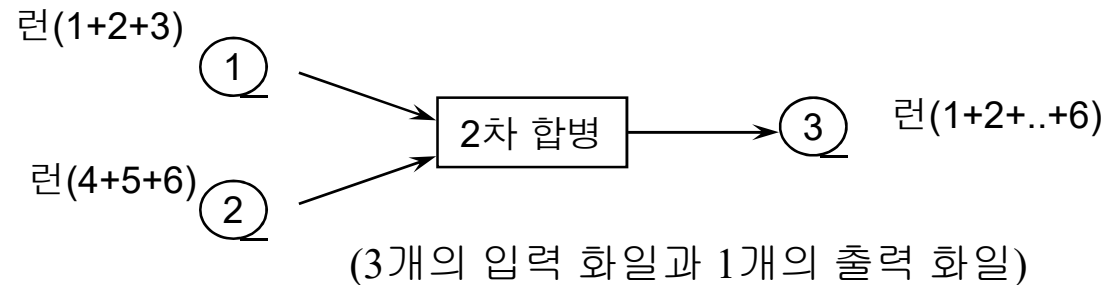
정렬된 6개의 런을 3개의 입력 화일에 분배한다.

(2) 1차 합병



④ 에 있는 런을 3개의 입력 화일에 분배한다.( 이 경우에는 2개의 화일만 사용됨)

(4) 2단계 합병



## ◆ 자연 합병의 성능

- 각 단계(pass) 마다 데이터 재분배를 위한 입출력 요구
  - ◆ 최초에 생성되는 런의 수가 많고,
  - ◆ 자연 합병의 차수가 낮을수록 입출력 요구가 더욱 증가됨
- 자연 합병에서 입출력 활동의 반은 다음 단계를 위한 입력 파일 생성, 즉 재분배에 사용됨



## ▶ m-원 합병 알고리즘

```
mWayMerge()    // m-원 합병 알고리즘
// m           : 입력 화일의 수
// FILE[]       : 화일 변수들의 배열
// outfilenum    : 출력 화일을 포함하는 화일 배열의 인덱스
// r            : 현 단계에서 합병되지 않고 입력 화일에 남아 있는 런수
// runcount     : 현 단계에서 생성된 런의 수

// 합병 단계
do {
  openFile(FILE[1], ..., FILE[m]);           // for reading
  openFile(FILE[m+1]);                       // for writing
  runcount ← 0;
  do {
    mergeRuns(FILE[1], ..., FILE[m] ⇒ FILE[m+1]);
    runcount ← 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 ← r + 1;
    else outfilenum ← i;
  }
  runcount ← runcount + r;
```

```

// 화일을 배열에 재할당
FILE[1], ..., FILE[m+1] ← FILE[1], ..., FILE[outfilenum-1],
    FILE[outfilenum+1], ..., FILE[m+1], ..., FILE[outfilenum];
closeFile(FILE[1], ..., FILE[m]);

// 런들의 분산 단계
openFile(FILE[m]); // for reading
openFile(FILE[1], ..., FILE[m-1]); // for writing
// 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 ← 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 ← 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 목표 화일은 FILE[m]
end mWayMerge()

```

## ▶ 선택 트리(selection tree)

- ◆ **m개의 런을 하나의 큰 런으로 정렬**
  - m개의 런 중 가장 작은 키 값의 레코드를 계속 선택, 출력
- ◆ **가장 작은 키 값을 선택하는 방법**
  - 직관적 방법
    - ◆ m-1번 비교
  - 선택 트리 : 비교 횟수를 줄일 수 있음
    - ◆  $\log_2 m$  번 비교
- ◆ **선택 트리의 종류**
  - 승자 트리(winner tree)
  - 패자 트리(loser tree)

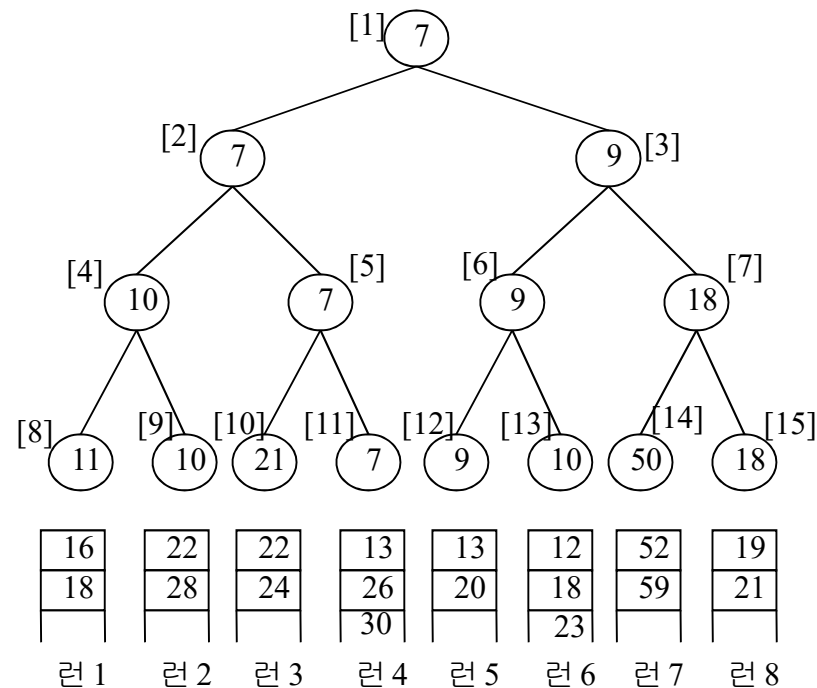
# 선택 트리 – 승자 트리

## ◆ 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( $m=8$ )인 경우 승자 트리 예



## – 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 값을 가진 원소

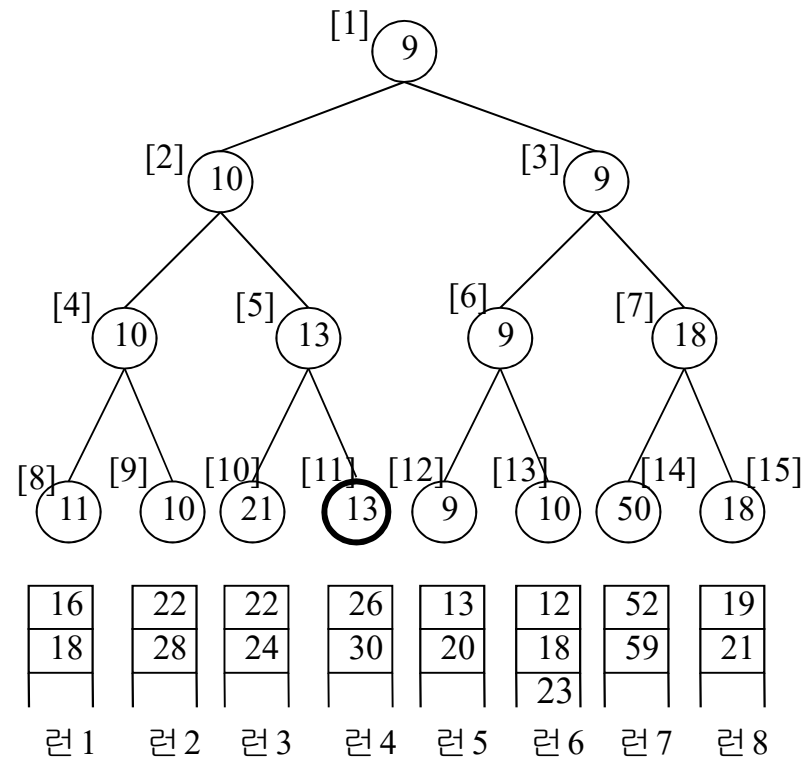
## – 승자 트리의 표현

- ◆ 승자트리는 완전 이원트리이기 때문에 순차 표현이 유리
- ◆ 인덱스 값이  $i$ 인 노드의 두 자식 인덱스는  $2i$ 와  $2i+1$

## – 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

- 다시 만들어진 승자트리의 예



- ◆ 이런 방식으로 순서 순차구축을 계속함

# 선택 트리 – 패자 트리

## ◆ 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)

## – 패자 트리 구축 과정

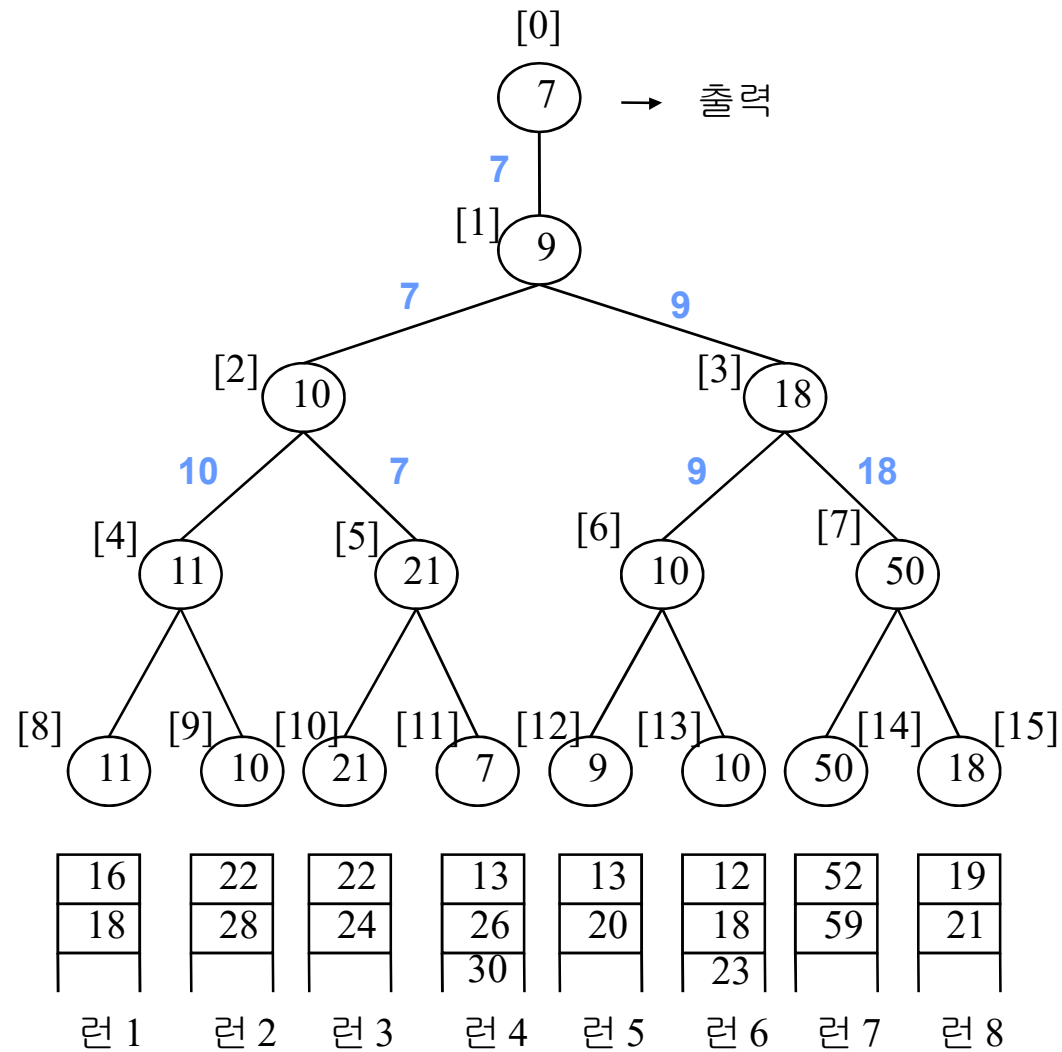
- ◆ 단말 노드: 각 런의 최소 키값 원소
- ◆ 내부 노드
  - 두 자식 노드들이 부모노드에서 토너먼트 경기를 수행
  - 패자는 부모 노드에 남음
  - 승자는 그 위 부모 노드로 올라가서 다시 토너먼트 경기를 계속
- ◆ 1번 루트 노드
  - 마찬가지로 패자는 1번 루트 노드에 남음
  - 승자는 전체 토너먼트의 승자로서 0번 노드로 올라가 순서순차에 출력됨

## – 합병의 진행

- ◆ 출력된 원소가 속한 런 4의 다음 원소, 즉 키값이 13인 원소를 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



- ◆ 런이 8개(m=8)인 패자 트리의 예



## ❖ 균형 합병 (balanced merge)

### ◆ 목적

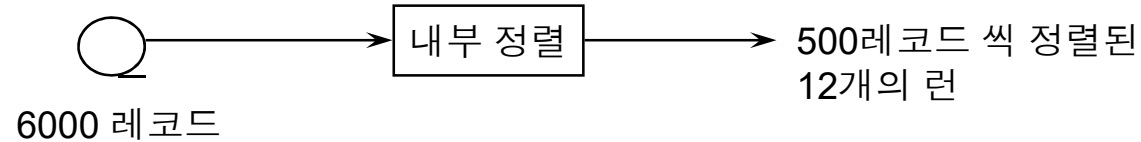
- 자연합병시 화일 재분배에 의해 소요되는 많은 I/O 회수를 줄이기 위한 방안

### ◆ 균형합병

- 출력을 미리 다음 단계의 입력화일로 재분배
  - ◆ m-원 자연합병 :  $m + 1$  개의 화일
  - ◆ m-원 균형합병 :  $2m$  개의 화일 (m 입력화일, m 출력화일)
  - ◆ “균형”은 입출력화일의 수가 동일하다는 의미
- 각 합병 단계 후
  - ◆ 런의 총수는 합병 차수(m)로 나눈 만큼 감소
  - ◆ 런의 길이는 합병 차수(m)의 두배씩 증가
- 소요 합병단계 =  $O(\log_m N)$ 
  - ◆ N: 초기 런의 수

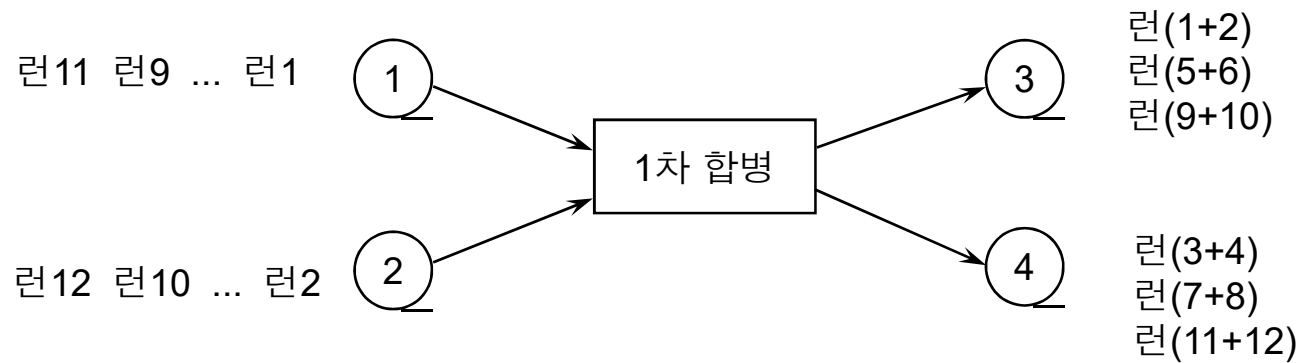
# ▶ 12개의 런에 대한 2-원 균형 합병

(1) 정렬단계

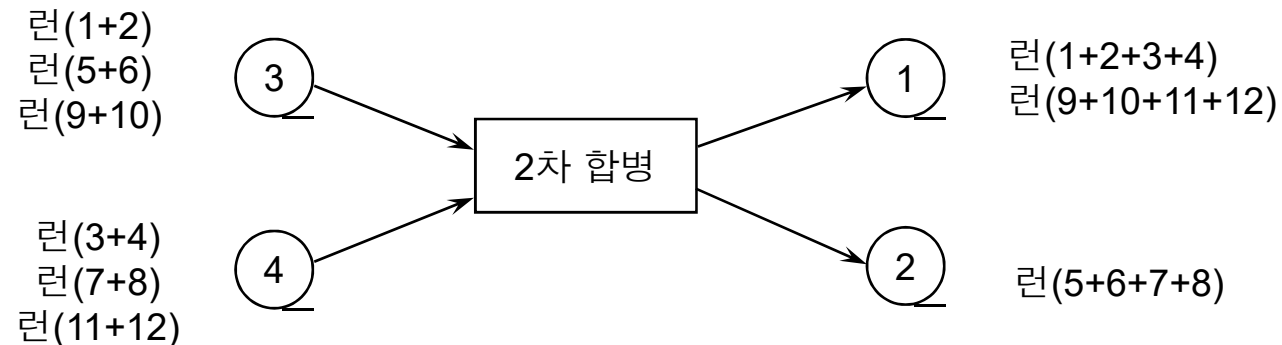


생성된 12개의 런을 2개의 화일에 분배한다.

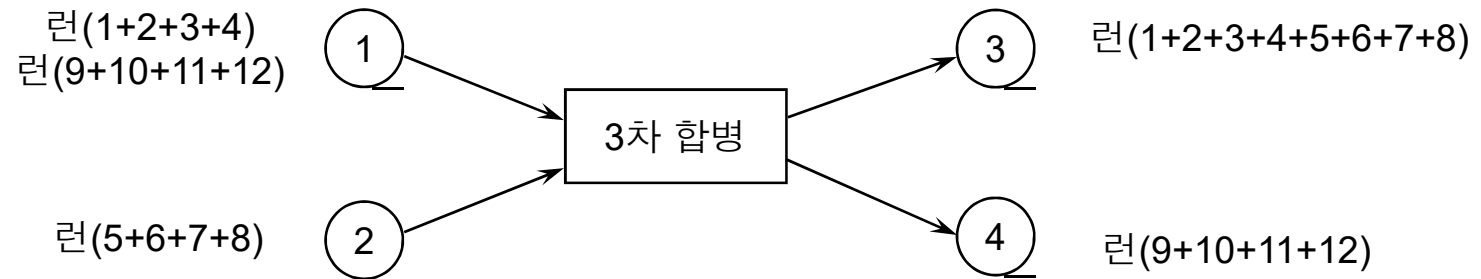
(2) 1차 합병



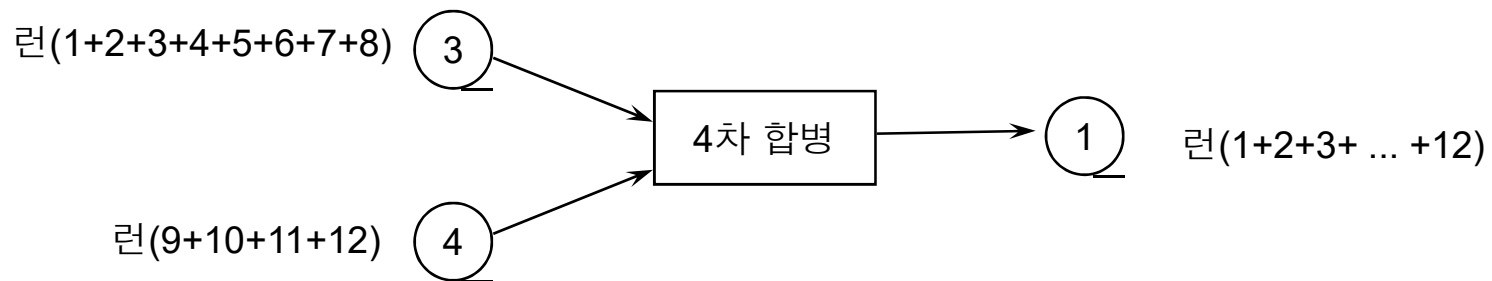
(3) 2차 합병



(4) 3차 합병



(5) 4차 합병



**주의** 런(1)은 4번 판독/기록되었음.

## ▶ m-원 균형합병 알고리즘

```
balancedMerge()  // m-원 균형합병 알고리즘
// m              : 입력 파일 수(모두  $2m$ 개의 파일 사용)
// FILE[]         : 파일 변수들의 배열
// input-set-first : 현 단계에서 입력과 출력 세트를
                  : 구분하기 위한 플래그
// base          : 출력 세트의 파일 중 첫 번째 파일의 번호
// outfilenum     : 현 단계의 출력 파일 번호
// runcount       : 현 단계에서 생성된 런의 수
```

```
input-set-first ← false;
do {
    if (input-set-first) {
        input-set-first ← false;
        openFile(FILE[1], ..., FILE[m]);    // for reading;
        openFile(FILE[m+1], ..., FILE[2m]); // for writing;
        base ← m + 1;
    } else {
        input-set-first ← true;
        openFile(FILE[1], ..., FILE[m]);    // for writing;
        openFile(FILE[m+1], ..., FILE[2m]); // for reading;
        base ← 1;
    }
}
```

```
// 합병 단계의 수행
outfilenum ← 0;
runcount ← 0;

do {
    mergeRun(input files ⇒ FILE[base+outfilenum]);
    runcount ← 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
end balancedMerge()
```

## ◆ 균형 합병의 성능

- $m$ 개 출력파일중  $m-1$ 개 파일은 항상 휴무 상태
  - ◆ 어느 단계에서도 임의의 순간을 보면,  $m$ 개의 파일로부터 런을 읽어 한 개의 파일로 출력
  - ◆  $m-1$  개 파일은 항상 유휴 상태
- 개선책
  - ◆ 불균형 합병: 다단계 합병, 계단식 합병

## ❖ 다단계 합병 (Polyphase Merge)

### ◆ 목적

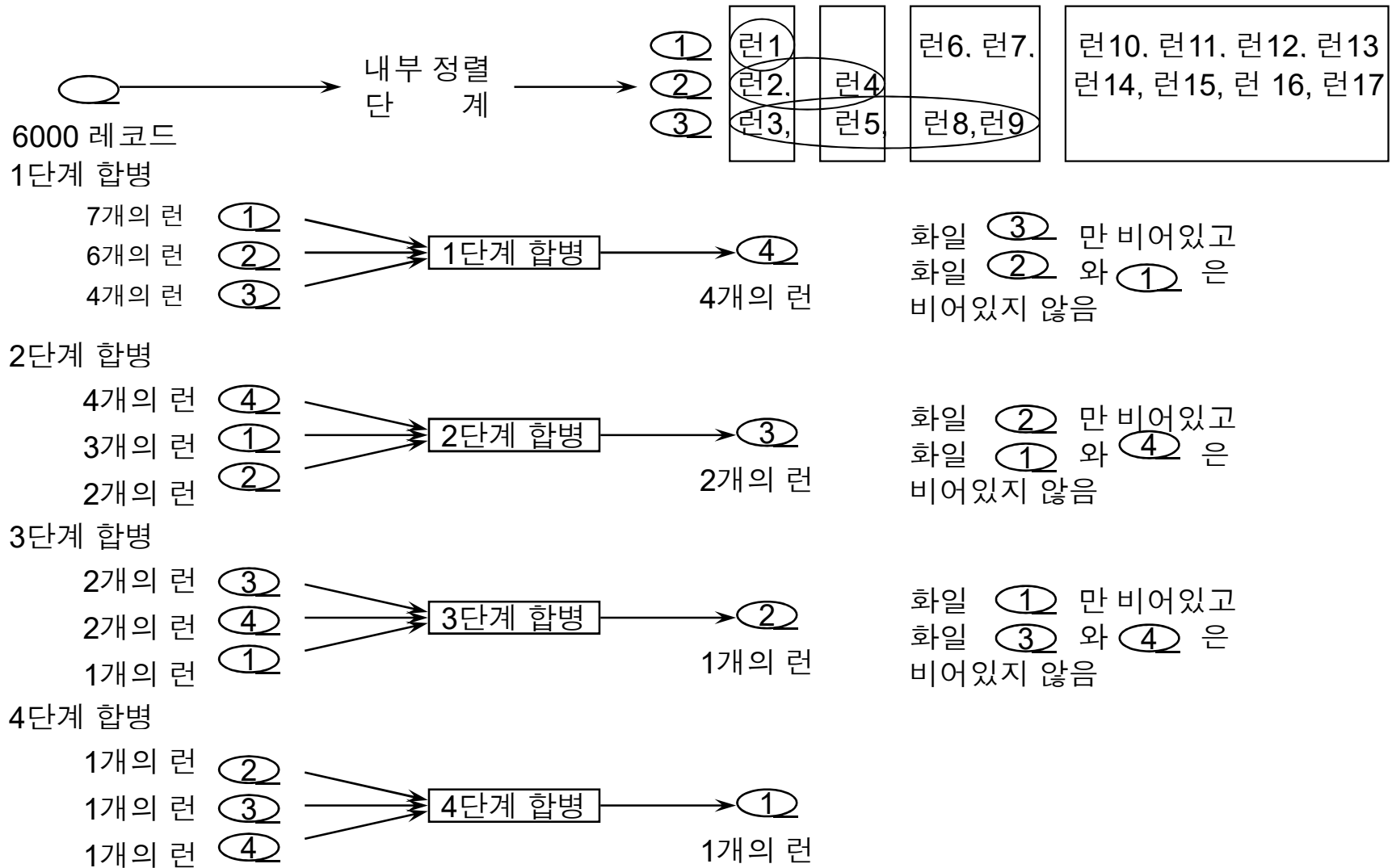
- 화일의 활용도 증가

### ◆ m-원 다단계 합병

- "불균형" 합병의 한 형태
  - ◆ m 개의 입력 화일, 1 개의 출력 화일
- 1. 초기 입력 화일에 대한 런의 분배 단계
  - ◆ 피보나치 수열 이용
- 2. 합병 단계
  - ◆ 입력 화일의 어느 하나가 공백이 될 때까지 런들을 합병
  - ◆ 공백이 된 입력 화일이 다음 합병 단계의 출력 화일이 됨
  - ◆ 다음 합병 단계는 그 전단계 입력 파일에 남아있는 런들에 대해 작업하므로, 레코드 복사량이 감소



## ▶ 3-원 다단계 합병



## ▶ 초기 런 분배 방법

- ◆ 런수의 변화 ( $m = 3$ )

1, 1, 1, 3, 5, 9, 17, 31, ...

- ◆ 피보나치(Fibonacci) 수열

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

$T_i = 1, \quad i \leq 3$  이 된다.

- ◆ 일반형

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

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

# ★ 초기 런 분배 방법 ( $m = 3$ , 화일수 = 17)

	제 1 차	제 2 차	제 3 차	제 4 차
HDD 1	1	1	3	7
2	1	2	2	6
3	1	2	4	4
합 계	3	5	9	17

1차분배: 모든 파일에 1개씩 분배

2차분배: 파일1에 있는 런의 개수만큼 파일2,3에 분배

3차분배: 파일2에 있는 런의 개수만큼 파일3,1에 분배

4차분배: 파일3에 있는 런의 개수만큼 파일1,2에 분배

## ★ 각 합병 단계에서 화일당 런 수의 변화

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

## ▶ 3-원 다단계 합병의 예

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

HDD1 : 50 95 110

HDD2 : 15 30 100 60 70 130

HDD3 : 40 120 150 20 80 140

HDD4 : 공백

(a) 정렬 단계 결과

HDD1 : 공백

HDD2 : 60 70 130

HDD3 : 20 80 140

HDD4 : 15 30 40 50 95 100 110 120 150

(b) 1차 합병 결과

HDD1 : 15 20 30 40 50 60 70 80 95 100 110 120 130 140 150

HDD2 : 공백

HDD3 : 공백

HDD4 : 공백

(c) 2차 합병 결과

## ▶ 2-원 다단계 합병의 예

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

HDD1 : 50 95 110 60 70 130 20 80 140

HDD2 : 15 30 100 40 120 150

HDD3 : 공백

(a) 정렬 단계 결과

HDD1 : 20 80 140

HDD2 : 공백

HDD3 : 15 30 50 95 100 110 40 60 70 120 130 150

(b) 1차 합병 결과

HDD1 : 공백

HDD2 : 15 20 30 50 80 95 100 110 140

HDD3 : 40 60 70 120 130 150

(c) 2차 합병 결과

HDD1 : 15 20 30 40 50 60 70 80 95 100 110 120 130 140 150

HDD2 : 공백

HDD3 : 공백

(d) 3차 합병 결과

## ▶ m-원 다단계 합병 알고리즘

```
polyphaseMerge()    // m-원 다단계 합병 알고리즘
// m    : 입력 파일의 수
// FILE[] : 파일 변수들의 배열
// (초기 입력 파일에는 피보나치 수열에 따라 런들이 분배되어 있다고 가정)

// 파일의 준비
for (i ← 1; i ≤ m; i++)
    openFile(FILE[i]); // for reading;
openFile(FILE[m+1]);  // for writing;
// 합병 단계
do {
    do {
        mergeRun(FILE[1], ..., FILE[m] ⇒ FILE[m+1]);
    } while (!end-of-file(FILE[m]));
    rewind(FILE[m] and FILE[m+1]);
    closeFile(FILE[m] and FILE[m+1]);
    openFile(FILE[m+1]);      // for reading
    openFile(FILE[m]);        // for writing
    // 파일을 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 목표 파일은 FILE[1]
end polyphaseMerge()
```

## ❖ 계단식 합병 (Cascade Merge)

### ◆ 목적

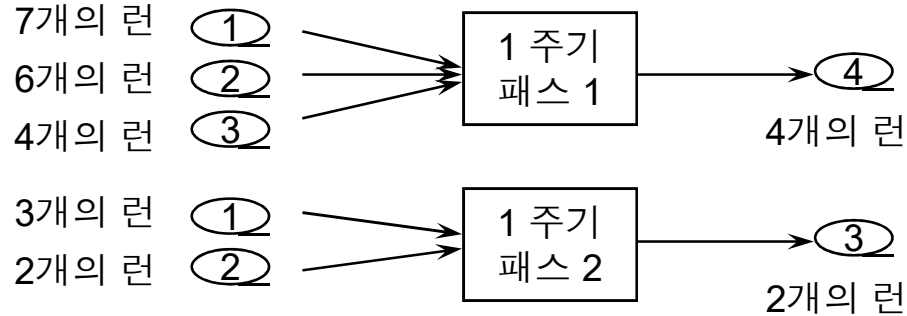
- 정렬/합병 과정에서 런의 복사작업을 줄이려는 불균형 합병의 또 다른 형태

### ◆ m-원 계단식 합병(m-way cascade merge)

- "불균형" 합병의 한 형태
  - ◆ 주기 :  $m, m-1, m-2, \dots$ , 그리고 마지막에 2개의 입력 화일을 사용
- 1. 초기 입력 화일에 대한 런의 분배 단계
  - ◆ 피보나치 수열 이용
- 2. 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

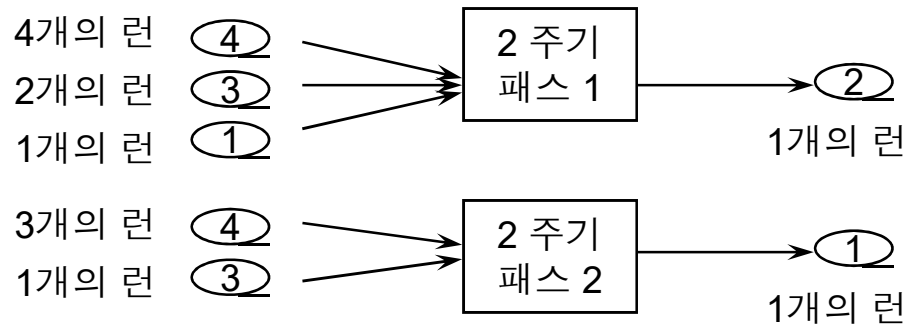


### 합병 1주기



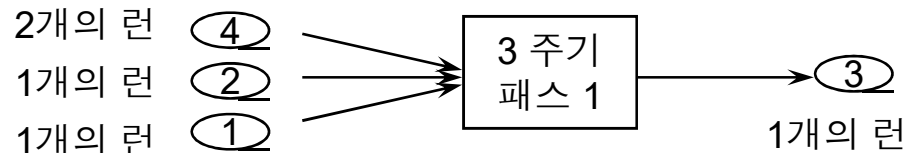
화일 ③ 은 공백  
 화일 ④ 는 대기  
 화일 ② 는 공백  
 화일 ③ 은 대기

### 합병 2주기



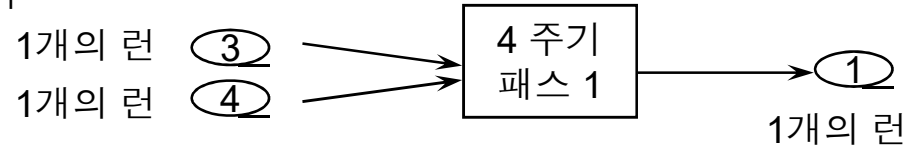
화일 ① 은 공백  
 화일 ② 는 대기  
 화일 ③ 은 공백  
 화일 ① 은 대기

### 합병 3주기



화일 ① 과 ② 는 공백  
 화일 ③ 은 대기

### 합병 4주기



화일 ② ③ ④ 는 공백

## ▶ 3-원 계단식 합병

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

HDD1 : 50 110 95

HDD2 : 15 30 100 60 70 130

HDD3 : 40 120 150 20 80 140

HDD4 : 공백

(a) 정렬 단계 결과

HDD1 : 공백

HDD2 : 60 70 130

HDD3 : 20 80 140

HDD4 : 15 30 40 50 95 100 110 120 150

(b) 합병 1주기 1차 합병 결과

HDD1 : 20 60 70 80 130 140

HDD2 : 공백

HDD3 : 공백

HDD4 : 15 30 40 50 95 100 110 120 150

(c) 합병 1주기 2차 합병 결과

HDD1 : 공백

HDD2 : 15 20 30 40 50 60 70 80 95 100 110 120 130 140 150

HDD3 : 공백

HDD4 : 공백

(d) 합병 2주기 1차 합병 결과

## ▶ m-원 계단식 합병 알고리즘

```
cascadeMerge()           // m-원 계단식 합병 알고리즘
// m           : 입력 파일의 수
// FILE[]      : 파일 변수들의 배열
// (초기 입력 파일에는 피보나치 수열에 따라 런들이 분산되어 있다고 가정)

// 파일의 준비
for (i ← 1; i ≤ m; i++)
    openFile(FILE[i]); // for reading;
    openFile(FILE[m+1]); // for writing;
// 합병 단계
do {
    i ← 0;
    do {
        i ← i + 1;
        do {
            mergeRuns(FILE[1], ..., FILE[m-i+1] ⇒ FILE[m-i+2]);
        } while (!end-of-file(FILE[m-i+1]));
        rewind(FILE[m-i+1] and FILE[m-i+2]);
        closeFile(FILE[m-i+1] and FILE[m-i+2]);
        // 하나 이상의 파일이 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 ← i + 1;
        k ← k + 1;
        rewind(FILE[m-i+1-k]);
        closeFile(FILE[m-i+1-k]);
    } else empty-file ← false; }
    // 다음 단계의 출력 화일 준비
    openFile(FILE[m-i+1]);      // for writing
} while (i ≠ m-1);
// 화일을 런 수의 내림차순으로 정렬한 후 배열에 재할당
reallocateFiles(FILE[1], ..., FILE[m+1],
    run count of FILE[1] ≥ run count of FILE[2] ≥
    ... ≥ run count of FILE[m+1]);
// 남은 화일 수의 재조정
no-more-empty ← false;
do {
    if (end-of-file(FILE[k])) m ← m - 1;
    else no-more-empty ← true;
} while (m ≠ 1 && !no-more-empty);
} while (m ≠ 1);
// 최종 정렬된 목표 화일은 FILE[1]
end cascadeMerge()

```

## ❖ 정렬/합병 유틸리티

### ◆ 정렬 합병 유틸리티 (utility)

- 범용의 화일 정렬/합병 작업을 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

### ◆ 정렬/합병 패키지 사용시 명세 내용

- (1) 정렬/합병할 화일의 이름
- (2) 정렬/합병의 키 필드의 데이터 타입, 길이, 위치
- (3) 키 필드들의 순서(주에서 보조순으로)
- (4) 각 키 필드에 적용할 배열 순서 (ASC, 또는 DES)
- (5) 각 키 필드에 적용될 순서 기준
- (6) 정렬/합병 결과를 수록할 출력 화일의 이름

## ▶ 사용자 정의 사항

- (1) 사용자가 정의한 정렬 순서 및 기준
- (2) 내부 정렬 단계에서 사용할 알고리즘  
(예 : quick, heap sort)
- (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) 회복을 위한 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)

★ job control card에 의한 sort

## ❖ 저장 장치와 정렬/합병

- ◆ 순차 접근만 허용되는 자기 테이프의 경우
  - 각 화일이 서로 다른 릴에 저장
  - 테이프 감는 시간 : 시작점으로 되감기가 되어 있어야 함
- ◆ 임의 접근이 가능한 자기 디스크의 경우
  - 정렬/합병될 화일들을 하나의 디스크에 저장
  - 디스크 입/출력 연산(탐구시간+회전지연시간)
    - ◆ 테이프(시작 시간+정지시간)보다 더 많은 오버헤드 수반
    - ◆ 자기테이프보다 데이터 전송률이 훨씬 빠름
  - 탐구와 회전 지연에 따른 접근 오버헤드
    - ◆ 화일들을 2개 이상의 디스크로 분산, 중첩시켜 감소
    - ◆ 화일의 병렬적 수행(디스크마다 별도의 디스크 제어기)