

---

# CSE 206 - 파일 처리론 (File Processing)

## 5 장. File Sort and Merge

# Contents

---

- **5.1. An overview of File sort and Merge**
- **5.2. Sort & Merge**
  - Sort
  - Merge

---

## 5.1. An overview of File sort and Merge

# Sorting (정렬)

## • Sort란?

- 데이터를 Sort Key(기준 값)의 순서에 따라 줄세우는 것
- 앞의 Key가 뒤의 Key보다 값이 작으냐 크냐에 따라 오름차순 정렬 (Ascending Sort)과 내림차순 정렬(Descending Sort)로 분류가능.

학번	이름	나이	본적	성
1243	홍길동	10	서울	남
1257	김철수	20	경기	남
1332	박영희	19	충청	여
1334	이기수	21	전라	남
1367	정미영	20	경상	여
1440	최미숙	21	강원	여

<정렬키 학번으로 오름차순 정렬된 데이터>

## Sorting (정렬)

- 모든 일은 Sorting에서 시작해서 Sorting으로 끝난다
  - 사전의 단어 순서
  - 게임에서 랭킹
  - 성적
  - 인터넷 쇼핑
    - 가격으로 정렬해서 살펴본다. 출시일자로 정렬해서 살펴본다.
  - 분류기 (Classifier)
    - 여러 개의 클래스에 대해 계산한 확률 중 확률이 높은 순서로 k개 클래스 분류



Dog : 0.6  
Human : 0.3  
Cat : 0.1

## Sorting Algorithm : Quick Sort

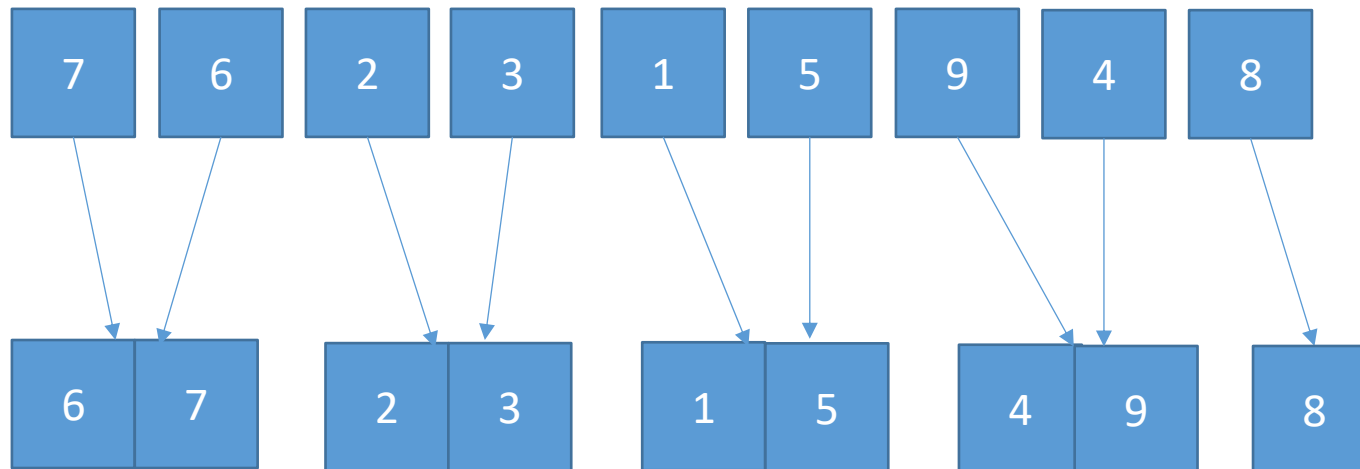
---

- 데이터 구조 혹은 알고리즘 과목에서 Sorting Algorithm은 빠지지 않음.
- **Quick Sort**
  - 평균적으로 가장 정렬 속도가 빠른 알고리즘.
    - 속도 뿐만 아니라 sort에 요구되는 memory space도 작다.
    - Cache locality를 활용할 수 있다.
  - 이미 구현되어 제공되는 Sorting Algorithm의 대부분 Quick sort로 구현되어 있다.
    - 자세히는 시중에서 구할 수 있는 Data structure 혹은 알고리즘 책들 참조.
    - 본 수업에서는 메모리에서 빠른 Sorting을 구현하기 위해 Quick sort를 사용할 수 있다 정도를 알아 두자.

# Sorting Algorithm : Merge sort

## • Merge sort: Sorting Algorithm 중 하나

- 입력 데이터 리스트(List)가 있을 때, 리스트의 인접 데이터 2개씩 비교해 정렬된 더 큰 리스트 (Merged list)를 여럿 만들고, 다시 만들어진 리스트들을 인접 리스트 2개씩 비교하여 정렬된 더 큰 리스트를 여러 개 만드는 작업을 반복한다.
- 리스트 크기가 커질수록 리스트의 숫자가 줄어드는데, 이를 남아 있는 리스트 수가 1개가 될 때까지 반복한다.



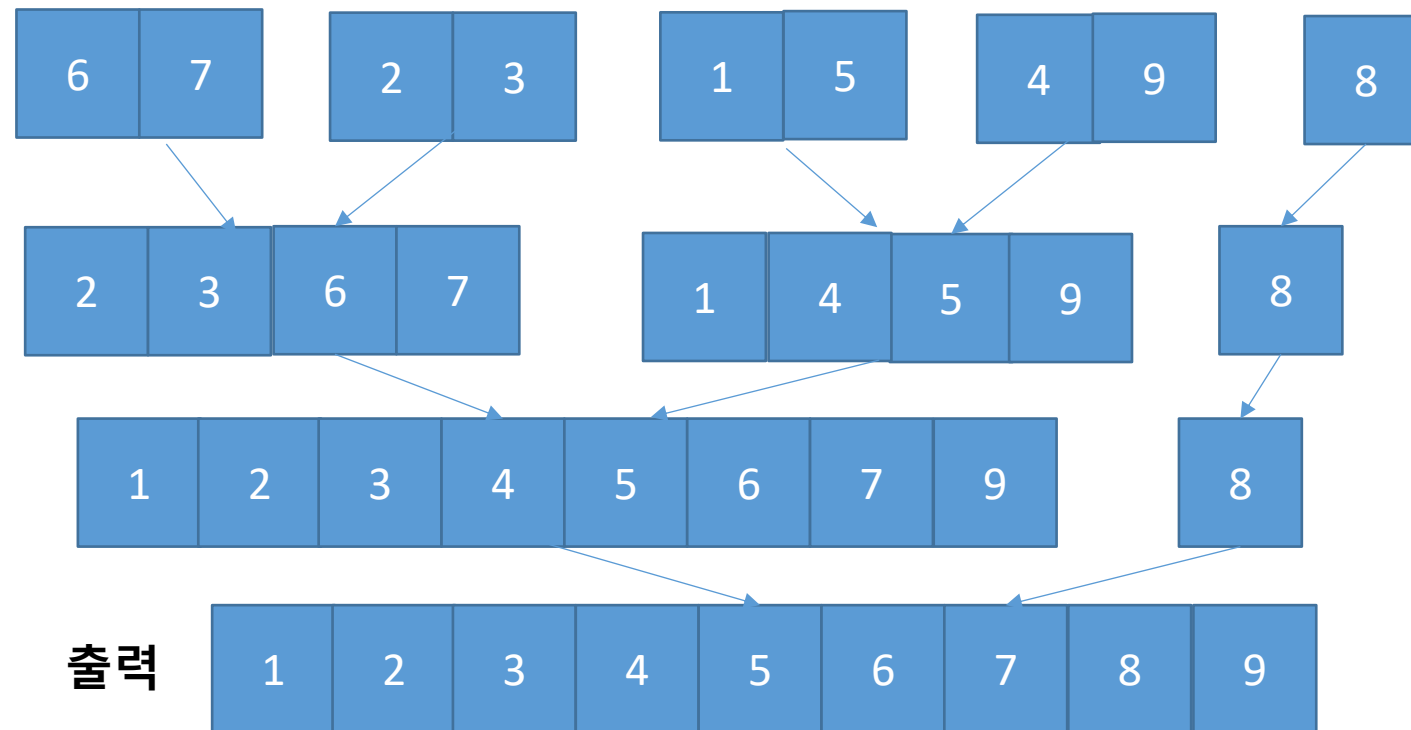
**Step 1:** 인접 데이터 두 개씩 비교 하여 정렬된 더 큰 리스트 생성 (데이터 1개는 Sorting된 길이 1인 리스트로도 볼 수 있다.)

## Sorting Algorithm : Merge sort (Cont'd)

**Step 2:** 인접 리스크 두 개씩 비교 하여 정렬된 더 큰 리스트 생성.

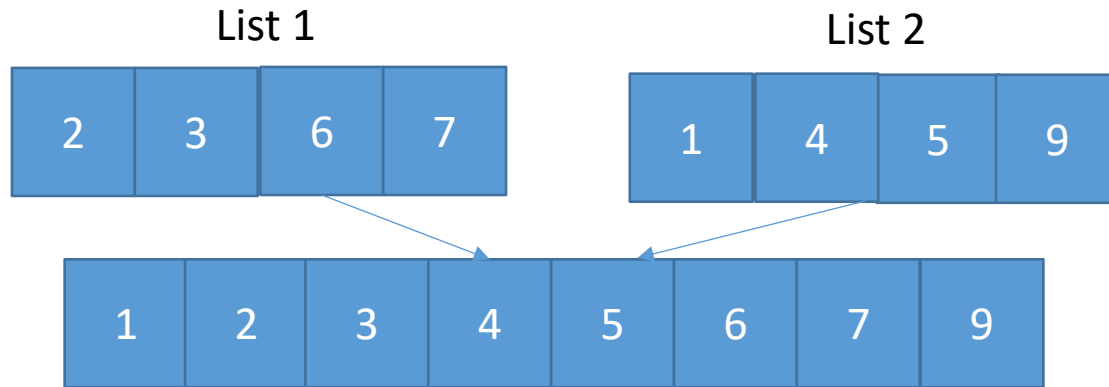
**Step 3:** 인접 리스크 두 개씩 비교 하여 정렬된 더 큰 리스트 생성.

**Step 4:** 인접 리스크 두 개씩 비교 하여 정렬된 더 큰 리스트 생성.





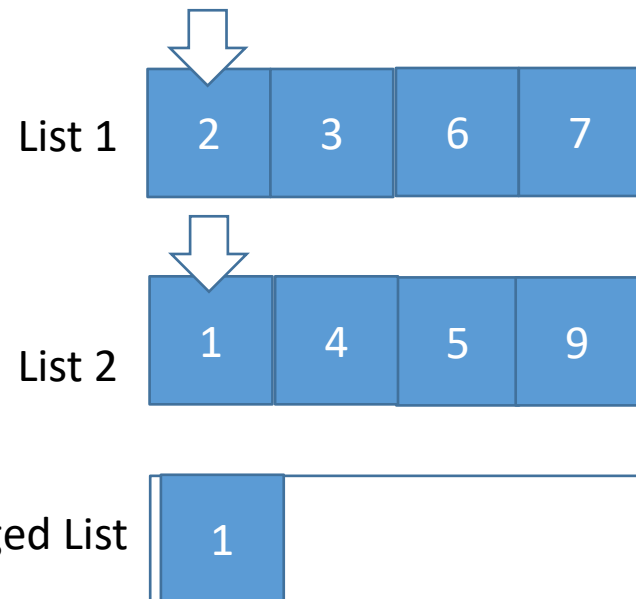
## Sorting Algorithm : Merge sort (Cont'd)



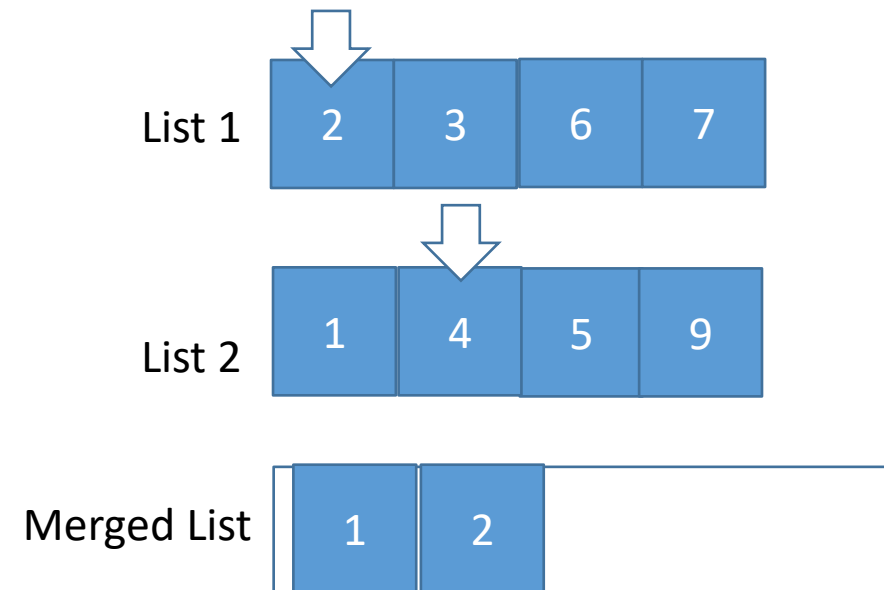
**Merge 알고리즘** : List 1과 List2의 제일 앞을 비교하여 작은 쪽 (내림 차순 정렬의 경우 큰 쪽)을 원래 리스트에서 빼서 Merged List에 넣는 작업을 List1, List2 가 모두 빌 때 까지 계속 함.  
List 1, 2 중 한쪽이 먼저 빌 경우 다른 쪽을 그대로 Merged List뒤에 붙여 넣는다.

\* 아래 예제에서 화살표는 리스트 제일 앞을 표시

**S1**



**S2**

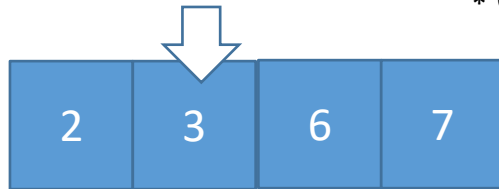


## Sorting Algorithm : Merge sort (Cont'd)

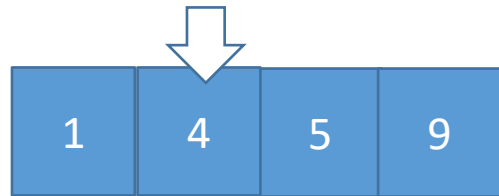
\* 아래 예제에서 화살표는 리스트 제일 앞을 표시

**S3**

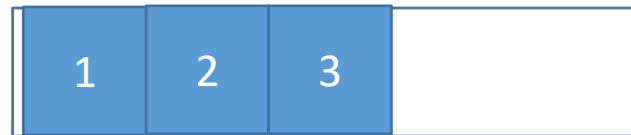
List 1



List 2

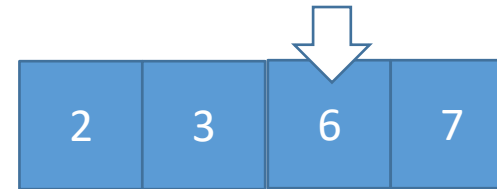


Merged List

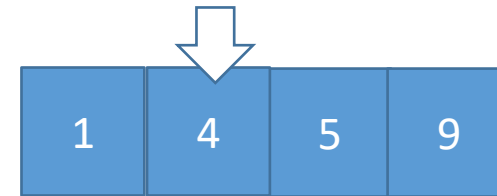


**S4**

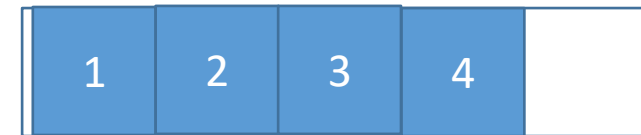
List 1



List 2



Merged List



...

**Merge 알고리즘** : List 1과 List2의 제일 앞을 비교하여 작은 쪽 (내림 차순 정렬의 경우 큰 쪽)을 원래 리스트에서 빼서 Merged List에 넣는 작업을 List1, List2 가 모두 빌 때 까지 계속 함.  
List 1, 2 중 한쪽이 먼저 빌 경우 다른 쪽을 그대로 Merged List뒤에 붙여 넣는다.

## 파일 구조에서의 Sorting

- 모든 일은 Sorting 에서 시작해서 Sorting 으로 끝난다
- 파일 구조에서도 Sorting은 중요하다.
  - Key Sequenced File을 만들려면 Key를 Sort Key(정렬키)로 하여 File에 저장될 Record를 정렬하여야 한다.

학번	이름	나이	본적	성
1243	홍길동	10	서울	남
1257	김철수	20	경기	남
1332	박영희	19	충청	여
1334	이기수	21	전라	남
1367	정미영	20	경상	여
1440	최미숙	21	강원	여

예) 왼쪽 데이터가 파일에 들어 있는데  
나이를 Key로 Key Sequenced File 을 만들 필요가 있을 경우.

<정렬키 학번으로 오름차순 정렬된 데이터>

## File Record Sorting: 메모리 부족 문제

---

Idea 1. 파일 내의 Record를 Quick sort를 사용하여 정렬하자.  
=> 정렬할 Record를 모두 메모리 위에 올려야 한다.

**문제 : 파일 내의 Record를 모두 메모리 위에 올리기는 메모리가 너무 작다.**

- 파일은 Hard Disk에 안 들어가면 안되지만 메모리에는 못 들어가도 되지만
    - 일반적으로 파일 전체 Record 중 필요한 Record만 메모리에 Load되어 있으면 연산 수행 가능.
  - Key Sequenced File과 같은 파일은 일괄 처리에 적합한 데이터를 저장하므로 일반적으로 크기가 크다. (많은 수의 Record를 포함함)
  - 정렬은 파일에 저장된 “모든 Record를 키를 근거로 정렬”해야 한다.
  - 메모리로 디스크와 같은 용량을 구현하기에는 비용이 너무 많이 듭
- 메모리 : 8, 16, 32 GB (16 GB memory : 8 만원)
- Hard Disk : 1 ~ 2 TB : (2 TB HDD: 4 – 5 만원)

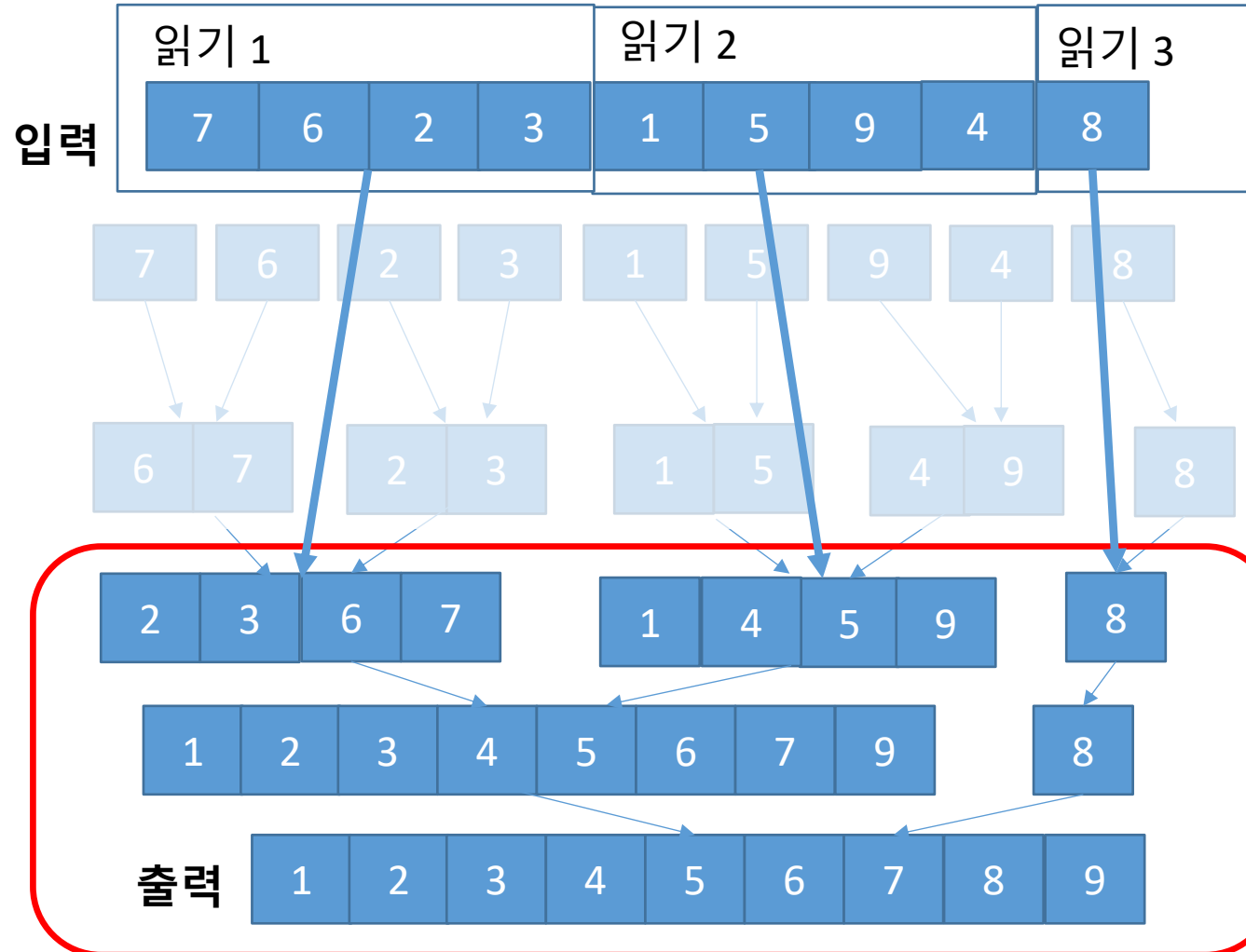
## File Record Sorting: 그래도 Sorting작업은 메모리에 Load된 데이터로 해야 한다.

---

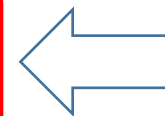
- 컴퓨터의 연산은, CPU가 메모리의 데이터를 사용하여 수행하고 그 결과도 1차적으로는 메모리로 저장된다.
- 따라서 Disk에 저장된 데이터도 연산하려면 메모리에 전송해 와야 한다.
- 문제: 그렇지만 Disk에 저장되어 있는 파일의 모든 Record를 메모리에 들고 올 수는 없다.

# File Record Sorting: (Divide & Conquer) : Sort & Merge

## • Sort & Merge (정렬/합병)



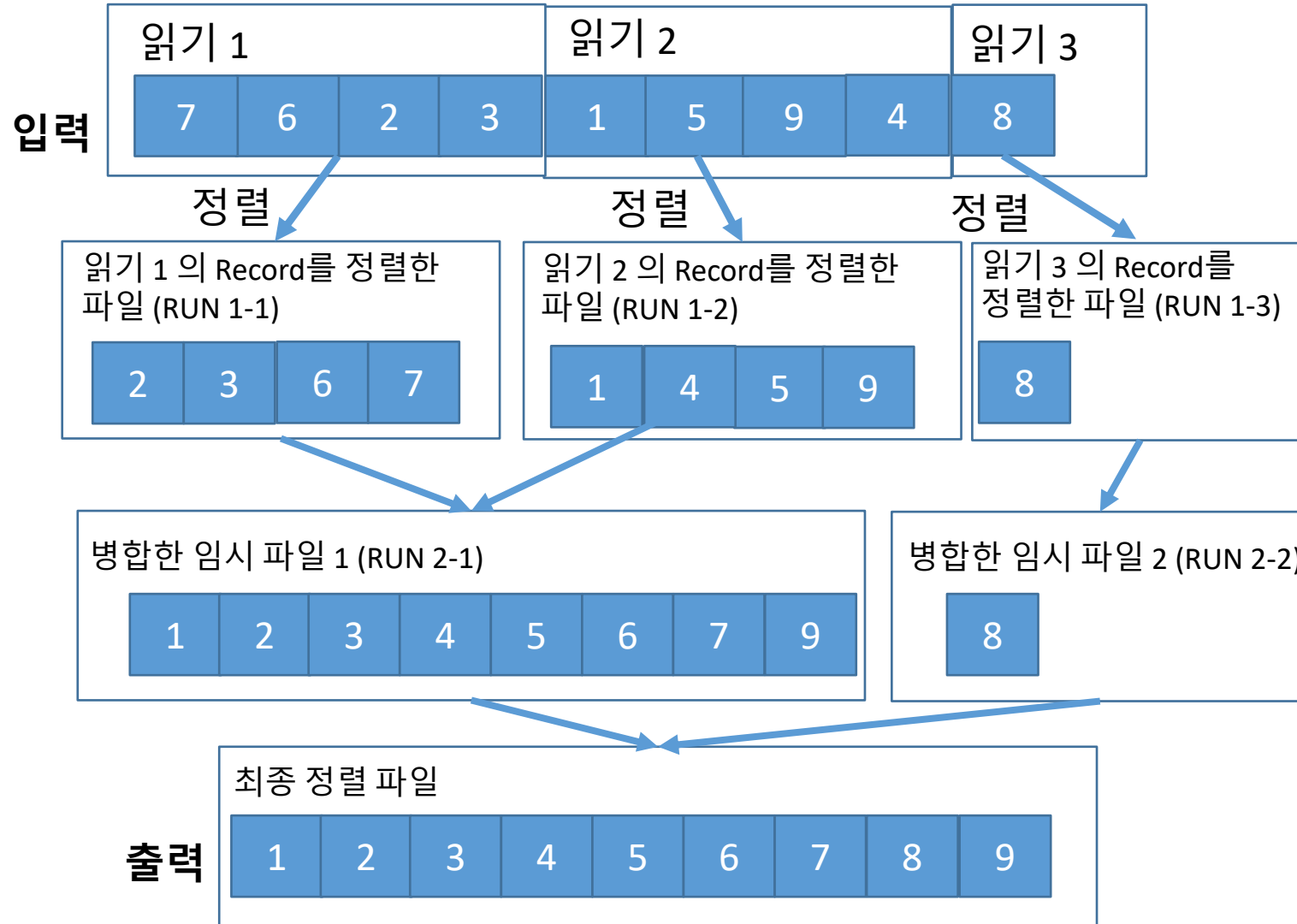
**Idea:** 메모리 크기가 한정되어 있다면 원래 파일에서 Record를 처음부터 메모리 크기만큼 읽어 들여 정렬한 다음에 파일에 출력하고, Merge sort 처럼 작은 파일들을 병합하는 것을 반복하면서 더 큰 파일을 만들어 가면 어떨까?



개념적으로는 도중에서 시작하는 Merge sort와 같다.

# File Record Sorting: (Divide & Conquer) : Sort & Merge

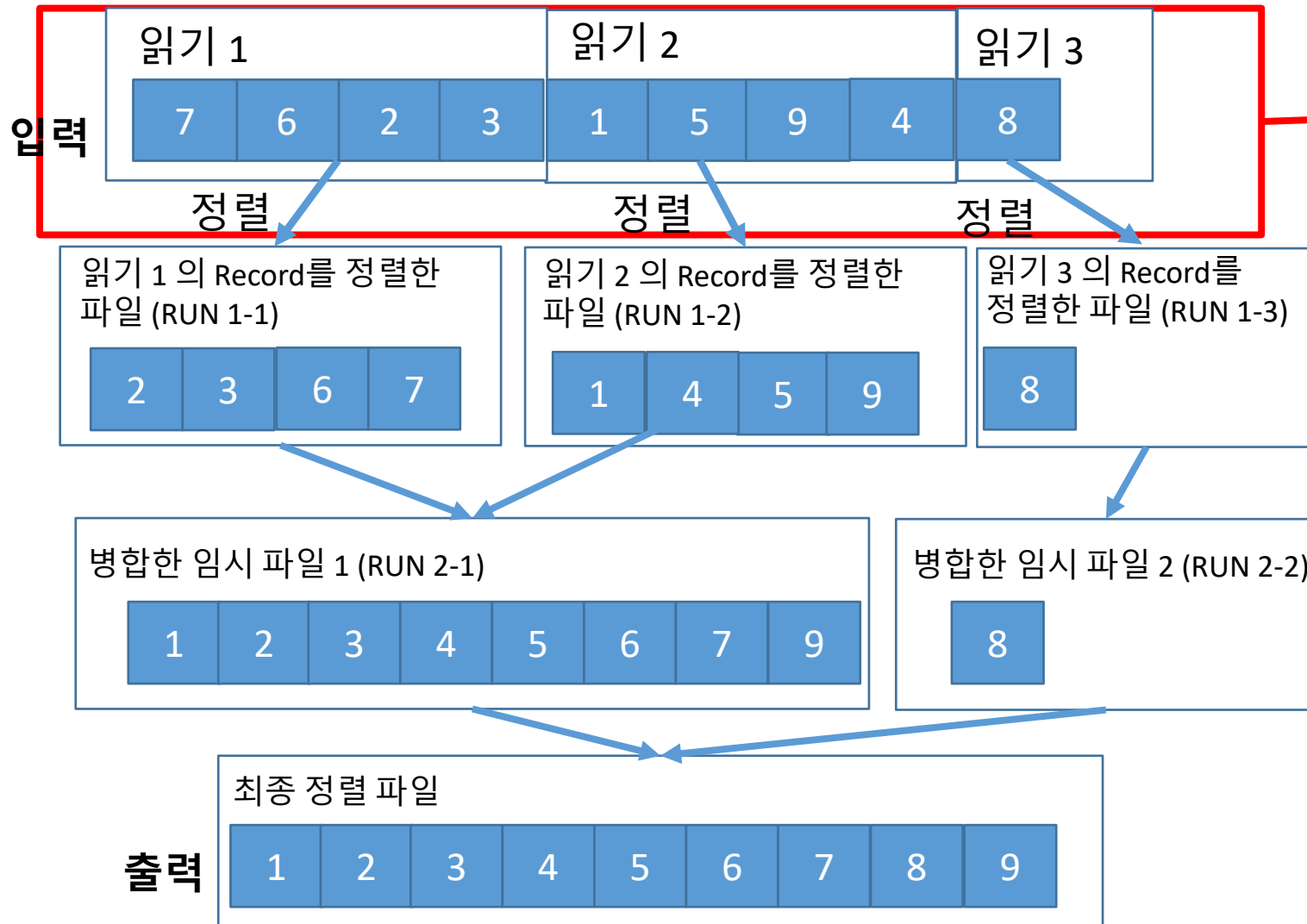
## • Sort & Merge (정렬/합병)



**Sort & Merge (정렬/합병):** 원래 파일에서 Record를 처음부터 메모리 크기만큼 읽어 들여 정렬한 다음에 파일에 출력하고, Merge sort 처럼 파일들의 병합을 반복하면서 최종 정렬 파일 생성

# Sort & Merge – Internal Sort vs External Sort

## Sort & Merge (정렬/합병)



**Internal Sort :** 메모리 내부에서 모든 필요한 정렬이 가능한 정렬  
 예) 각각의 읽기에서 읽은 Record에 대해서 정렬하기만 하면 충분할 경우

**External Sort :** Record가 많아서 메인 메모리의 용량을 초과하여 보조 저장장치에 저장된 Record를 정렬

**런(run) :** 하나의 파일을 여러 개로 분할하여 내부 정렬 기법으로 정렬시킨 서브파일(subfile)



# Internal Sort vs External Sort – 속도 계산

---

- **Internal Sort 알고리즘 사이의 속도 비교**

- 어떤 정렬 알고리즘이 더 빠르냐는 정렬해야 할 데이터 수  $n$ 에 대해 실행되는 비교 연산의 수로 나타낸다.
  - 왜?
  - 일반적으로 Big-O notation으로 나타낸다.
  - **Merge-sort** : Average  $O(n \log(n))$
  - **Quick-sort** : Average  $O(n \log n)$

- **External Sort 알고리즘 사이의 속도 비교**

- Disk I/O를 더 줄인 정렬 알고리즘이 더 빠르다.
  - Memory 연산에 비해 10만 배 이상 느린 Disk I/O 가 존재한다.
  - 메모리에서의 정렬 연산이 10 배정도 느려도 티가 안 난다.

# Internal Sort vs External Sort – 정리

---

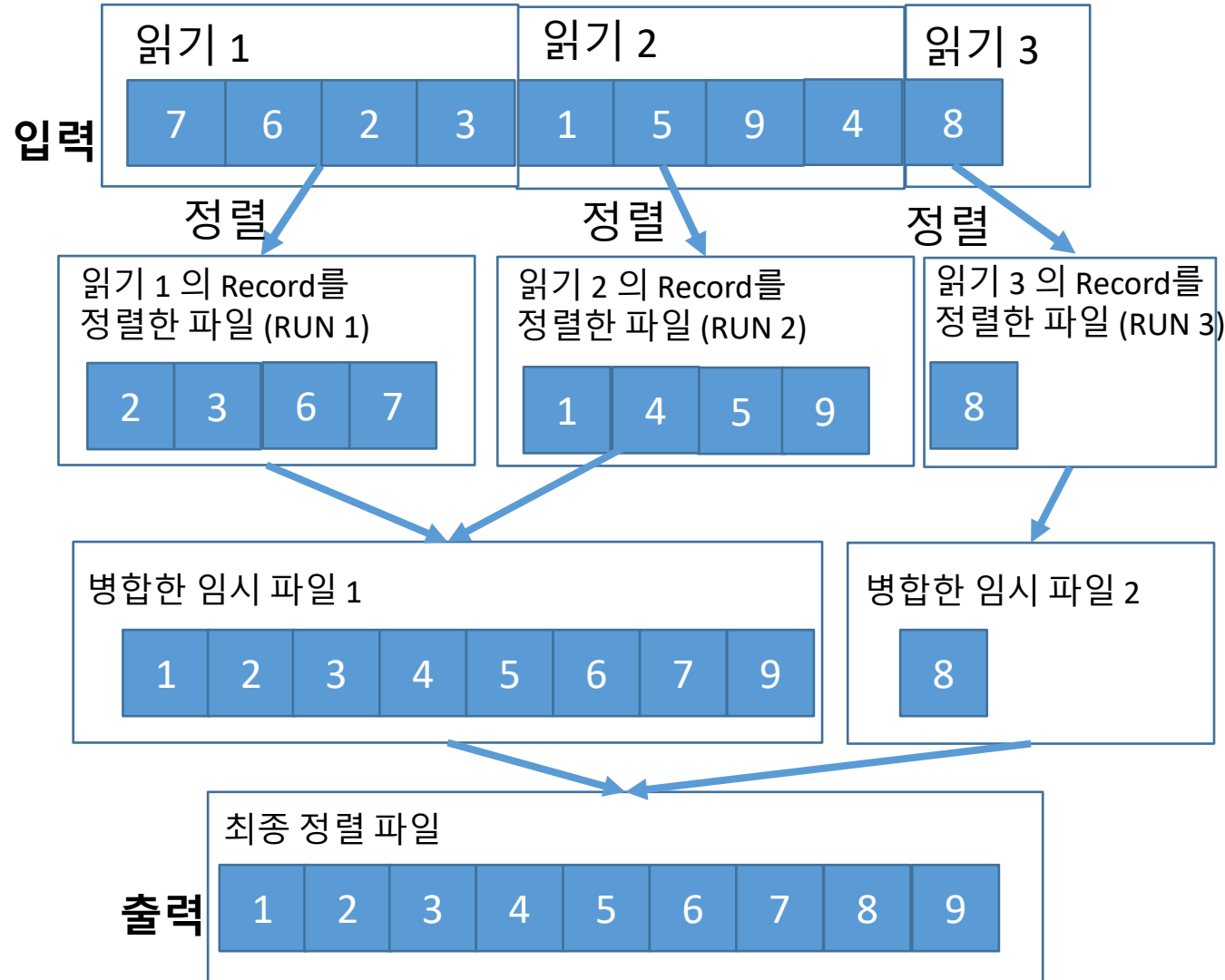
- **Internal sorting (내부 정렬)**

- 데이터가 적어서 메인 메모리 내에 모두 저장시켜 정렬 가능할 때
- Record Read, Write에 걸리는 시간이 문제되지 않는다.

- **External sorting (외부 정렬)**

- 데이터가 많아서 메인 메모리의 용량을 초과하여 보조 저장장치에 저장된 파일을 정렬할 때
- Record Read, Write에 걸리는 시간이 중요
- 정렬/합병(sort/merge)
  - Run : 하나의 파일을 여러 개로 분할하여 내부 정렬 기법으로 정렬시킨 서브파일(subfile)
  - Run들을 합병(merge)하여 원하는 하나의 정렬된 파일로 만든다.

# File Sort/Merge Phase



## 1. Sort phase (정렬 단계)

- 정렬할 파일의 Record들을 지정된 길이의 서브파일(메모리 크기보다 작다)로 분할해서 정렬하여 Run을 만들어 임시 파일 (Merge Phase의 입력)로 저장하는 단계

## 2. Merge phase (합병 단계)

- 정렬된 Run들을 합병해서 보다 큰 Run으로 만들고, 이것들을 다시 입력 파일로 재분배하여 합병하는 방식으로 모든 Record들이 하나의 Run에 포함되도록 만드는 단계

---

## 5.2. Sort & Merge - Sort

## Sort Phase

---

- 첫 단계 Run 생성 방법
  - 내부 정렬 (internal sort)
  - 대체 선택 (replacement selection)
  - 자연 선택 (natural selection)
- 입력 파일의 예
  - 설명의 간소화를 위해 Record 키 값만 표기

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				

## 용어 정리

---

- **Buffer** : Sorting을 위해 확보해 둔 메모리 공간.
  - 이전 장에서 Buffer는 디스크에 저장할 Record 혹은 디스크에서 읽어온 Record를 저장해 두는 메모리의 공간을 뜻하였다.
  - 이번 장의 정렬 단계의 설명에 한해서 특별한 언급이 없는 한, “Buffer”는 “정렬 작업을 위해 메모리에 확보해 둔 공간”을 뜻한다.

## Internal sort (내부 정렬)

---

- 제일 간단하고 단순한 방법
  - 요약: Buffer에 한번에 읽을 수 있는 만큼 읽고 정렬해서 파일로 쓴다.
  - Run 생성 방법
    - Buffer에 한번에 읽을 수 있는 크기가 m Record라 하면
      1. 파일을 m Record 씩 여러 개의 서브 파일로 분할
      2. 분할된 Record들을 각각 정렬
- Run 생성 결과
  - 마지막 Run을 제외하고 모두 길이가 동일

# Internal sort (내부 정렬)

m 이 5일 경우의 입출력 예

## 입력 파일

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
81	59	10	38	76	16	24	85							

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			

**장점:** 간단하고 구현이 쉽다.

**단점?** 파일의 모든 Record 수를  $f$  이라 하면 항상  $\left\lceil \frac{f}{m} \right\rceil$  개의 초기 Run이 생성된다. => 파일 수가 많다.

Ex) Record  $f = 52$ 개, 버퍼에 한번에  $m = 5$ 개 들어간다면  $\left\lceil \frac{52}{5} \right\rceil = 11$  개 Run이 생성



## 대체 선택 (replacement selection)

- **목적 : Run의 길이를 길게 할 수 있을까?**
  - Run 하나당 길이가  $m$ 보다 큰  $m'$  ( $m < m'$ )로 만들 수 있다면  $\left\lceil \frac{f}{m} \right\rceil > \left\lceil \frac{f}{m'} \right\rceil$  이므로 초기 런의 개수가 줄어든다. => Merge할 파일 **개수**가 준다.
- **풀어야 할 문제 :** Memory의 Buffer에다 한번에 읽어 sorting 알고리즘을 적용할 수 있는 Record 수가  $m$  인데 어떻게  $m$  보다 긴 초기 Run을 만들 수 있는가?
- **아이디어 :** 파일에 있는 Record의 전체 배열이 아닌 “부분” 배열을 보았을 때 부분 배열이 모두 다 정렬과 역순으로 저장되어 있는 것은 아니다.
- 즉, 원래 정렬 순서대로 저장되어 있는 부분이 있다.

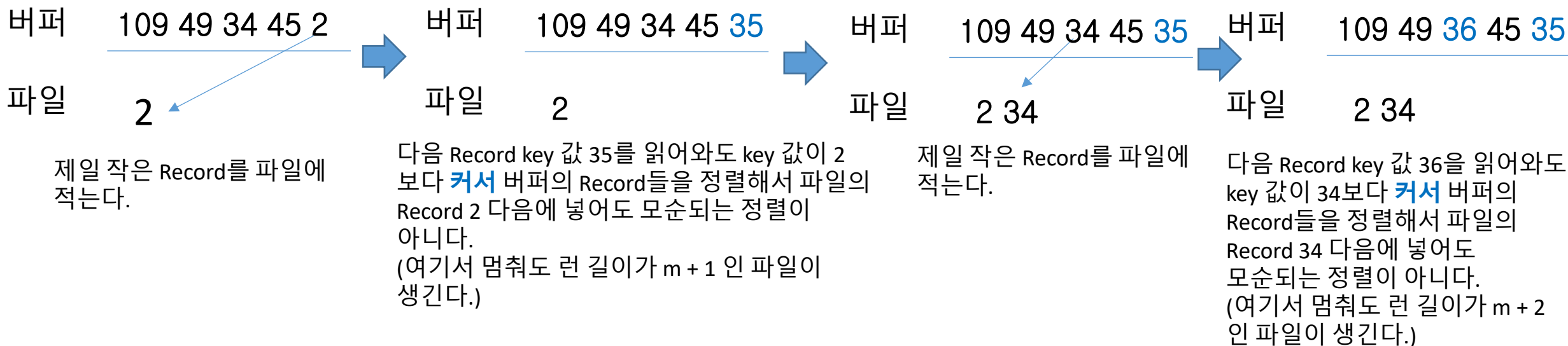
## 대체 선택 (replacement selection)

- 아이디어** : 현재 버퍼에서 key가 제일 작은 Record를 파일에 적은 다음 남은 공간에 다음 Record를 읽어 넣어도 버퍼 내에 존재하는 Record의 키들의 크기가 이미 파일에 적어 넣은 Key와 정렬하는데 모순되지 않는 경우가 있다. 이를 활용하면 버퍼 크기  $m$  이상의 Run이 생성 가능.

유리한 예 : 버퍼에 읽어 올 수 있는 Record 개수가 최대  $m = 5$  이라 하자.

입력 파일 Record

109 49 34 45 2 35 36 37 38 43 47 ...



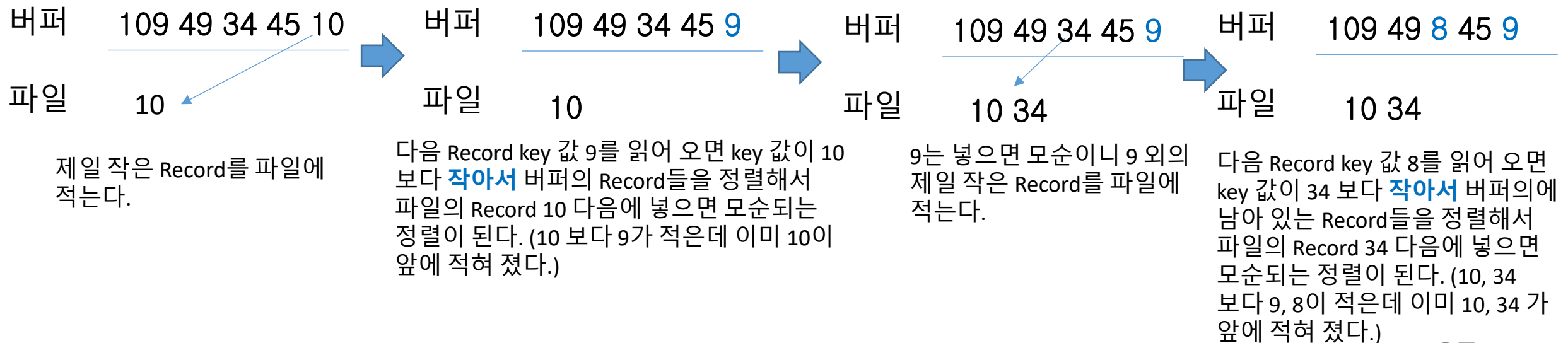
## 대체 선택 (replacement selection)

- **아이디어** : 현재 버퍼에서 key가 제일 작은 Record를 파일에 적은 다음 남은 공간에 다음 Record를 읽어 넣어도 버퍼 내에 존재하는 Record의 키들의 크기가 이미 파일에 적어 넣은 Key와 정렬하는데 모순되지 않는 경우가 있다. 이를 활용하면 버퍼 크기  $m$  이상의 Run이 생성 가능.

**불리한 예 : 버퍼에 읽어 올 수 있는 Record 개수가 최대  $m = 5$  이라 하자.**

입력 파일 Record

109 49 34 45 10 9 8 7 6 5 ...



# 대체 선택 (replacement selection)

## Run 생성 방법

Step 1. 입력 파일에서 Buffer (정렬에 사용하는 Buffer)에 m개의 Record를 읽어 들임

- 여기서 읽은 직후의 Record들은 모두 동결(Freeze)되지 않음.

Step 2. Run을 저장할 파일을 새로 하나 만들

Step 3. 버퍼로부터 Key 값이 가장 작은 “동결되지 않은” Record를 선정해서 파일에 적음

Step 4. Record 출력으로 버퍼에 자리가 하나 났으므로 입력 파일에서 다음 Record를 하나 읽어 버퍼에 넣음.

Step 5 : if (방금 파일에 적은 Record의 Key 값 > 방금 새로 읽은 Record의 Key 값) {

    새로읽은 Record에 “동결(frozen)” 표시 하고.

    버퍼안의 모든 Record가 동결 되었다면 Step 6로 가고 아니면 Step 3로 가서 계속함.

    } else {

        Step 3로 가서 계속함.

    }

Step 6. Step 2에서 만든 파일을 저장하고, 동결된 Record들을 모두 해제하고 단계 2로 돌아감.

- 대체 선택 **Run**의 평균 예상 길이는 대략 **2 m**이 된다고 함. {경험적}

# 대체 선택 동작 예 (m = 5를 가정)

## 입력 파일

```
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
```

Step	설명	Memory (m = 5)	File content	File name
1	m 개 읽음.	109 49 34 68 45		
2	File Run 저장용 파일 새로 하나 만듦	109 49 34 68 45		run1
3	동결되지 않은 Record 중 제일 작은 Record 출력	45 49 [34] 68 109	34	run1
4	다음 Record 읽음	45 49 [2] 68 109	34	run1
5	방금 파일에 쓴 34 보다 읽은 2 가 작으므로 동결	45 49 [2] 68 109	34	run1
6	동결되지 않은 Record 중 제일 작은 Record 출력	[45] 49 68 2 109	34 45	run1
7	다음 Record 60 읽음	[60] 49 68 2 109	34 45	run1
8	방금 파일에 쓴 45 보다 읽은 60 이 크므로 아무일 안함	[60] 49 68 2 109	34 45	run1
9	동결되지 않은 Record 중 제일 작은 Record 출력	60 [49] 68 2 109	34 45 49	run1
10	다음 Record 읽음	60 [38] 68 2 109	34 45 49	run1
11	방금 파일에 쓴 49 보다 읽은 38 이 작으므로 동결	60 [38] 68 2 109	34 45 49	run1
12	동결되지 않은 Record 중 제일 작은 Record 출력	[60] 38 68 2 109	34 45 49 60	run1
13	다음 Record 읽음	[28] 38 68 2 109	34 45 49 60	run1
14	방금 파일에 쓴 60 보다 읽은 28 이 작으므로 동결	[28] 38 68 2 109	34 45 49 60	run1

# 대체 선택 동작 예 (m = 5를 가정) – Cont'd

## 입력 파일

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

Step	설명	Memory (m = 5)	File content	File name
	이전 페이지 마지막 상태	<b>28 38 68 2 109</b>	<b>34 45 49 60</b>	<b>run1</b>
15	동결되지 않은 Record 중 제일 작은 Record 출력	<b>28 38 [68] 2 109</b>	<b>34 45 49 60 68</b>	<b>run1</b>
16	다음 Record 읽음	<b>28 38 [47] 2 109</b>	<b>34 45 49 60 68</b>	<b>run1</b>
17	방금 파일에 쓴 68 보다 읽은 47 이 작으므로 동결	<b>28 38 [47] 2 109</b>	<b>34 45 49 60 68</b>	<b>run1</b>
18	동결되지 않은 Record 중 제일 작은 Record 출력	<b>28 38 47 2 [109]</b>	<b>34 45 49 60 68 109</b>	<b>run1</b>
19	다음 Record 읽음	<b>28 38 47 2 [16]</b>	<b>34 45 49 60 68 109</b>	<b>run1</b>
20	방금 파일에 쓴 109 보다 읽은 16 이 작으므로 동결	<b>28 38 47 2 [16]</b>	<b>34 45 49 60 68 109</b>	<b>run1</b>
21	파일 저장하고 동결 해제. 그리고 step 2로	<b>2 38 28 47 16</b>	<b>34 45 49 60 68 109</b>	<b>run1</b>
22	File Run 저장용 파일 새로 하나 만듦	<b>2 38 28 47 16</b>		<b>run2</b>
23	동결되지 않은 Record 중 제일 작은 Record 출력	<b>38 28 47 [2] 16</b>	<b>2</b>	<b>run2</b>
24	다음 Record 읽음	<b>38 28 47 [19] 16</b>	<b>2</b>	<b>run2</b>
25	방금 파일에 쓴 2 보다 읽은 19 가 크므로 아무일 안함	<b>38 28 47 [19] 16</b>	<b>2</b>	<b>run2</b>
26	동결되지 않은 Record 중 제일 작은 Record 출력	<b>38 28 47 19 [16]</b>	<b>2 16</b>	<b>run2</b>
:	이어짐	:	:	:

## 대체 선택 동작 예 ( $m = 5$ 를 가정) – Cont'd

### Run 생성 결과

```
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 - 계산 비용 분석

## 대체 선택 min 알고리즘의 계산 비용 :

- Step 3의 Record하나를 파일에 쓰기 위한 min 연산은  $m - 1$  개의 Record key 비교가 필요.
- 입력 파일에  $f$ 개의 Record가 있다면  $f$  번 파일에 Record를 쓰기 위해  $f$ 번 min 연산이 필요.

⇒ 대체 선택 전체에서 min 알고리즘은  $O(mf)$  의 계산 비용을 가짐  
 $O((m-1)*f) = O(mf)$

## 대체 선택에서 Run 파일 쓰기 비용 :

- 입력 파일에  $f$ 개의 record가 있다면, 각 record를 한번씩 읽어서 새 파일에 써야 하므로  $f$  번 파일 읽기 연산과,  $f$ 번 파일 쓰기 연산, 합해서  $2f$ 번의 File I/O가 필요

- $b$ 개 Record를 한 블록으로 모아서 디스크에서 읽고 쓴다고 해도  $2 \left\lceil \frac{f}{b} \right\rceil$  읽기/쓰기 연산이 필요.

⇒ 보통 “ $2 \left\lceil \frac{f}{b} \right\rceil$  번 디스크 읽기/쓰기 시간” >> “ $m*f$  번 비교연산에 걸리는 시간” 이므로 Sort&Merge 성능향상을 위해서는 min연산의 효율화는 특별히 하지 않는다.  
(가능하다면 Disk I/O 횟수 줄이기에 집중한다.)



# 자연 선택 (Natural Selection) : 대체 선택과 자연 선택의 차이점

## • 해결하려는 문제점

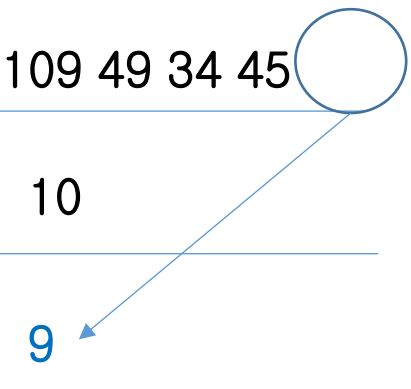
- 현재 Run(파일)의 뒤에 붙일 수 있는 Record가 더 있어도 길이 m의 버퍼가 동결된 Record로 가득차면 현재 Run을 저장하고 새로운 Run을 만들어야 한다.

대체 선택 : Freeze된 Record를  
버퍼에 그냥 놓아 둔다.  
=> 얼면 탐색 중지.

버퍼	109 49 34 45 <b>9</b> 11 12 ...
파일	10

자연 선택 : 저장소에 Freeze된 Record를  
저장할 수 있는 저장소가 따로 있다.  
=> 얼어도 얼음 저장소에 옮겨 놓고 조금 더  
탐색할 수 있다.

버퍼	109 49 34 45 <span style="border: 1px solid black; border-radius: 50%; padding: 2px;">  </span> 11 12 ...
파일	10
얼음 저장소	<b>9</b>



## 자연 선택 (natural selection)

---

- 해결하려는 문제점

- 현재 Run(파일)의 뒤에 붙일 수 있는 Record가 더 있어도 길이  $m$ 의 버퍼가 동결된 Record로 가득차면 현재 Run을 저장하고 새로운 Run을 만들어야 한다.

- 자연 선택

- 특징
  - 동결된 Record들을 버퍼에 방치하지 않고 보조 저장장치의 저장소(reservoir)에 별도 저장
  - 저장소가 다 차면 저장소의 얼음들 (freeze 된 Record) 중  $m$ 개를 버퍼로 옮겨와서 녹여서 다음 Run 생성 시작
  - 하나의 Run 생성 작업은 저장소가 꽉 차거나 입력 파일의 끝에 도달할 때까지 계속
  - 대체 선택보다 Run을 길게 만들어 Run 수를 줄임으로써 합병 비용을 줄임

## 자연 선택 동작 예 ( $m = 5$ 를 가정)

### 입력 파일

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		

# Run 생성 방법의 비교 정리

---

- 내부 정렬 (internal sort)
  - 마지막 Run을 제외하고 모든 Run들의 길이가 같음
  - Run의 길이를 예측할 수 있으므로 합병 알고리즘이 간단
- 대체 선택 (replacement selection)
  - 내부 정렬보다 평균적으로 훨씬 긴 Run을 생성
    - Run의 길이가 길수록 합병 비용이 적음 (왜?)
    - Run의 길이가 일정치 않아 정렬/합병 알고리즘이 복잡
- 자연 선택 (natural selection)
  - 대체 선택보다 더 긴 Run을 생성 가능
    - Run의 길이가 길수록 합병 비용이 적음
    - Run의 길이가 일정치 않아 정렬/합병 알고리즘이 복잡
  - 저장소로의 입출력이 문제가 됨
  - 긴 Run 생성에 따른 효율화가 저장소 전송 비용보다 이익이 될 수도 있음

---

## 5.3. Sort & Merge - Merge

M-way Merge

# Merge 수행 방법

---

- Merge 수행 방법
  - **m-원 합병(m-way merge)**
  - 균형 합병(balanced merge)
  - 다단계 합병(polyphase merge)
  - 계단식 합병(cascade merge)

## 2 원 합병 (2-way merge)

---

- 특징

- $m$  이 2인  $m$  원 합병
- 한번에 2개의 입력 파일을 Merge(합병)하여 하나의 출력 파일을 생성
- 합병 방법은 앞에서 설명한 Merge sort의 합병 방법과 같음

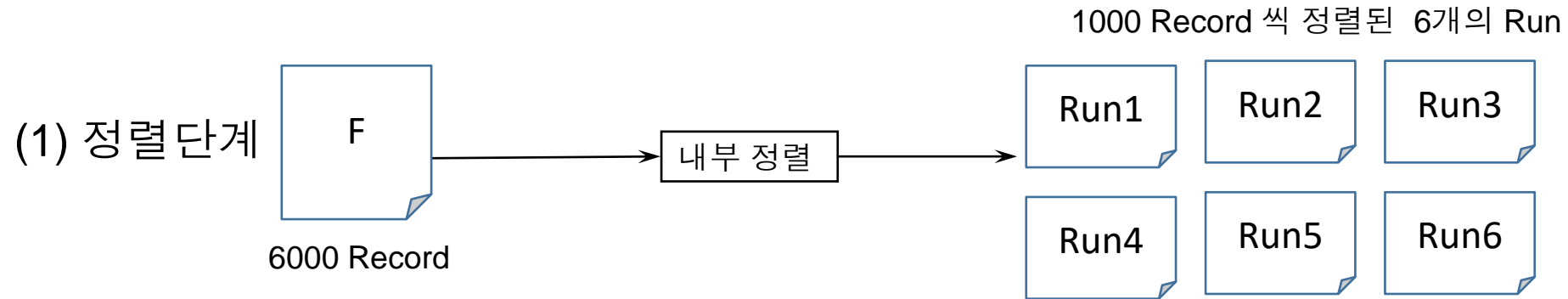
## 2 원 합병 (2-way merge) (Cont'd)

---

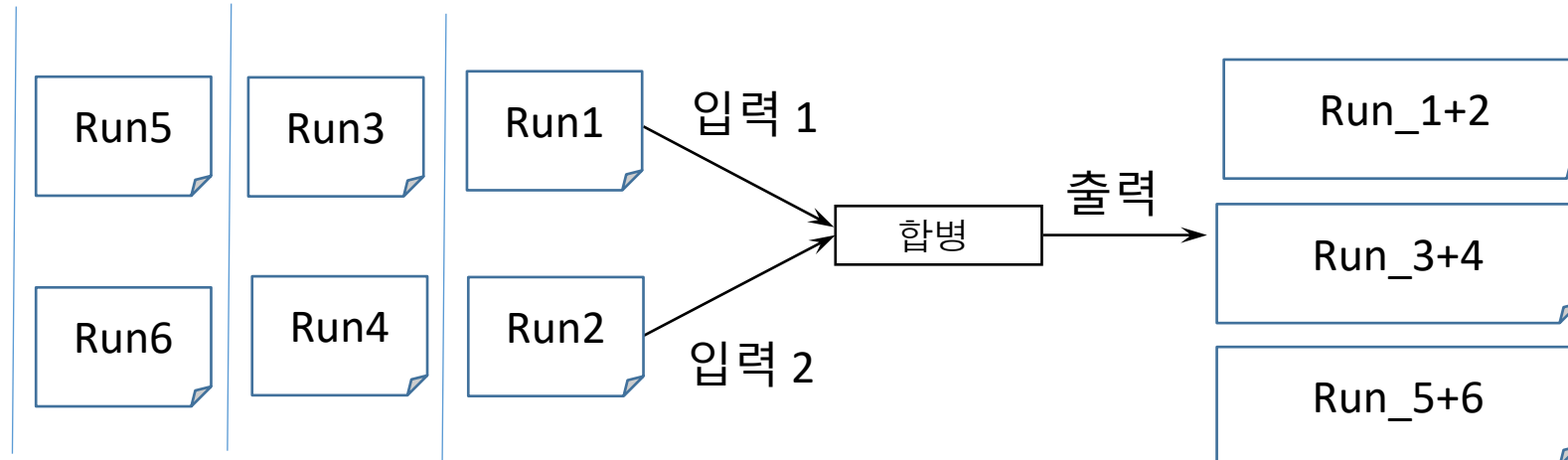
- 한번의 pass를 수행 후 : 합병된 Run의 크기는 2배, Run의 수는 반
  - Pass : 입력 Run 구성의 변경 없이 한번에 계속할 수 있는 연속된 Merge과정
  - 2원 합병에서의 1 패스(pass): 주어진 Merge 단계에서, Merge하기 위한 모든 Run을 1 쌍씩 입력으로 사용하여 모든 입력에 대해 Merge된 출력 Run을 출력 하는 연산 과정을 1 패스라 한다.
  - 예) “Run 6개를 2원 합병으로 Merge => 3 개의 Merge된 Run이 생성”



## 2 원 합병 예제

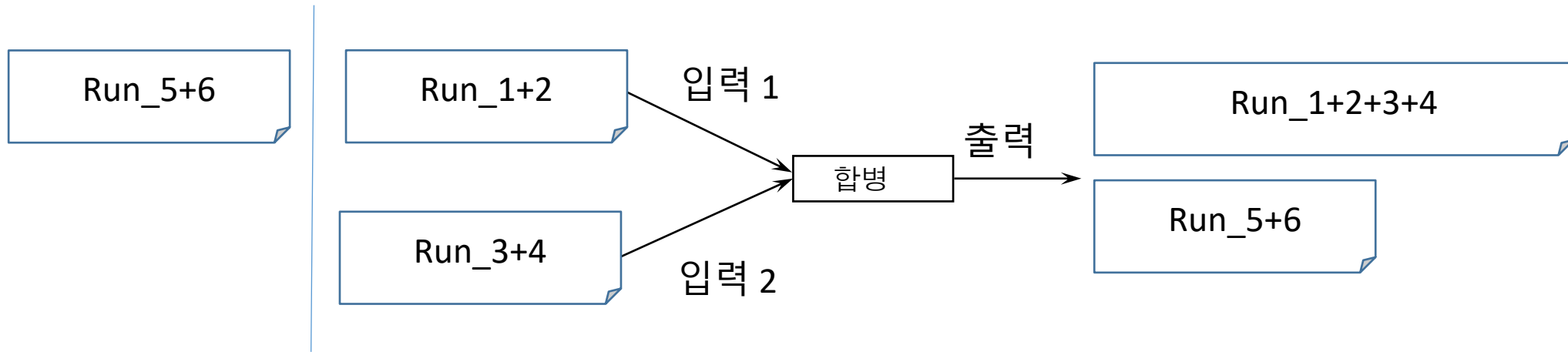


### (2) 1차 Merge (1<sup>st</sup> pass)

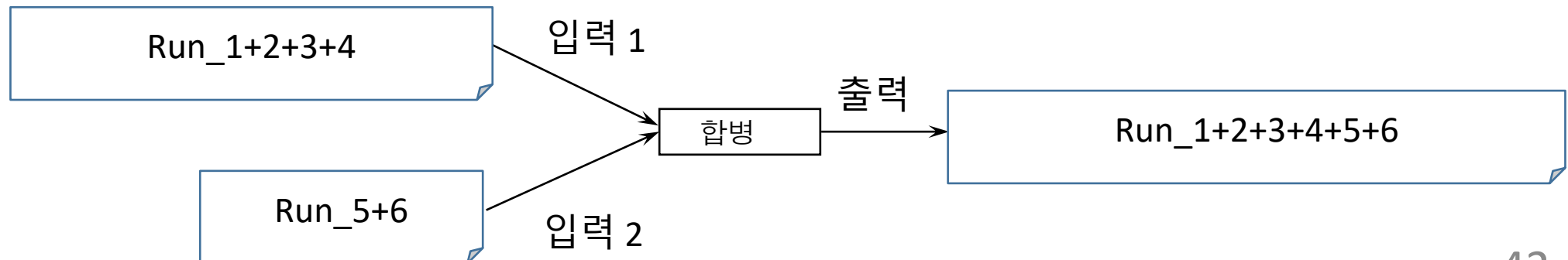


## 2 원 합병 예제 (Cont'd)

(2) 2차 Merge (2<sup>nd</sup> pass)



(3) 3차 Merge (3<sup>rd</sup> pass)



## 2 원 합병 (2-way merge) (Cont'd)

---

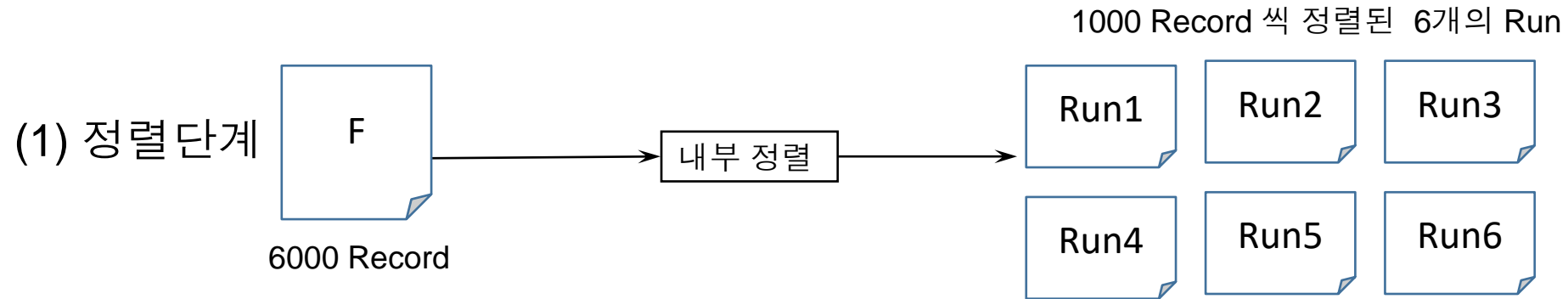
- N개의 Run으로 분할된 파일을 정렬하기 위한 Pass 수 :
  - N 개의 Run을 2원 합병으로 최종적으로 1개의 Merged파일로 만들기 위한 Pass 수는  $\lceil \log_2 N \rceil$  이다.

## m-원 합병(m-way merge)

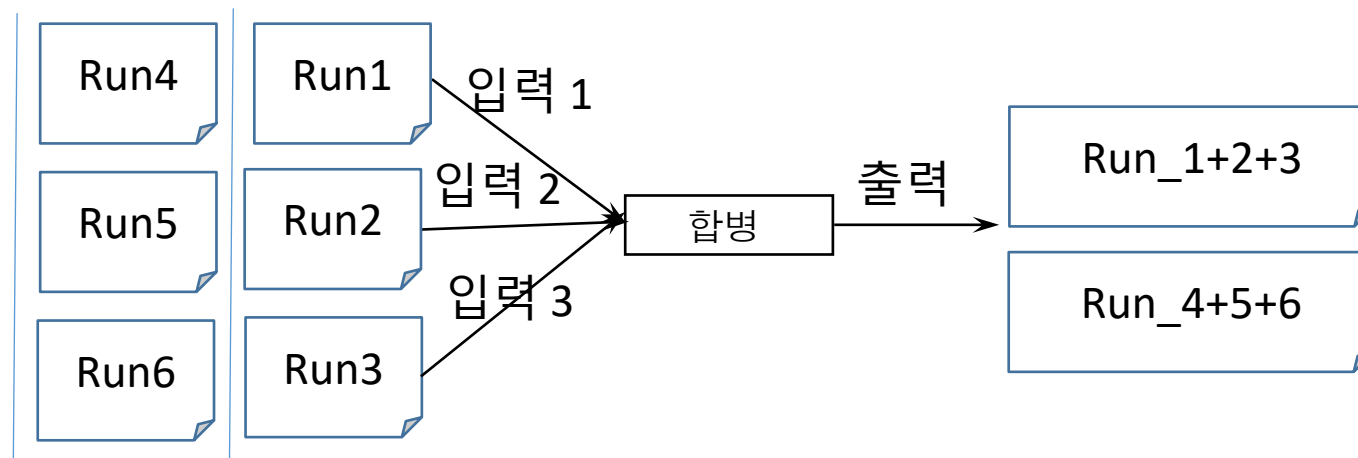
---

- 특징
  - m개의 입력 파일을 동시에 처리하는 합병
  - m개의 입력 파일을 합병하여 하나의 출력 파일을 생성
    - 한번의 합병에 총 m+1 (입력 m 개, 출력 1 개) 의 파일을 사용
- **많은 입출력 : Pass**가 여러 번이면 Pass 한번마다 입력 파일에 포함되어 있는 전체의 Record 수 만큼의 파일 읽기와 쓰기가 발생함.
  - N 개의 초기 Run을 m 원 합병으로 합병하기 위해 필요한 Run 수  $\lceil \log_m N \rceil$
- **이상적 Sort & Merge** : m개의 Run에 대해 한번의 m-원 합병으로 완료

## 3-원 합병 예제

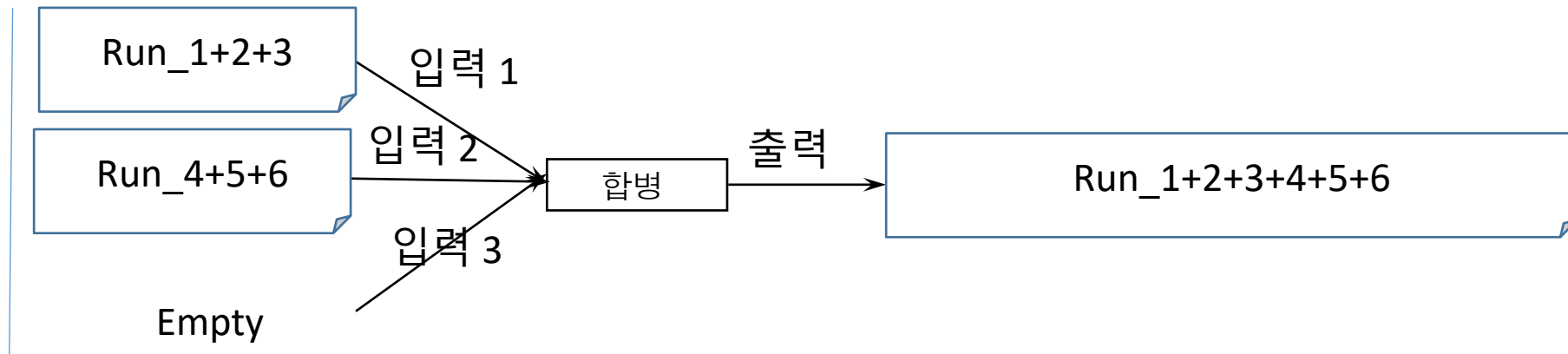


### (2) 1차 합병 (1<sup>st</sup> pass)



## 3-원 합병 예제 (Cont'd)

(2) 2차 합병 (2<sup>nd</sup> pass)



## Selection tree (선택 트리)

---

- **Selection Tree**

- m-way merge에서 m개의 Run의 앞의 값 중에서 제일 작은 값을 찾아내는데 필요한 비교 횟수를 m-1 회보다 줄이기 위한 자료 구조.
- m-way merge 알고리즘 (Remind)
  - m개의 Run(Record) 중 가장 작은 키 값의 Record를 계속 선택, 출력
  - 제일 단순한 비교 방법 : 각 Run에서 출력하지 않은 Record 중 제일 처음 레코드의 키 값을 비교하여 키 값이 제일 작은 Record 출력
    - m-1 번의 비교가 필요.

- **Selection Tree의 종류**

- 승자 트리(winner tree)
- 패자 트리(loser tree)

# 승자 트리

- 승자 트리(winner tree)

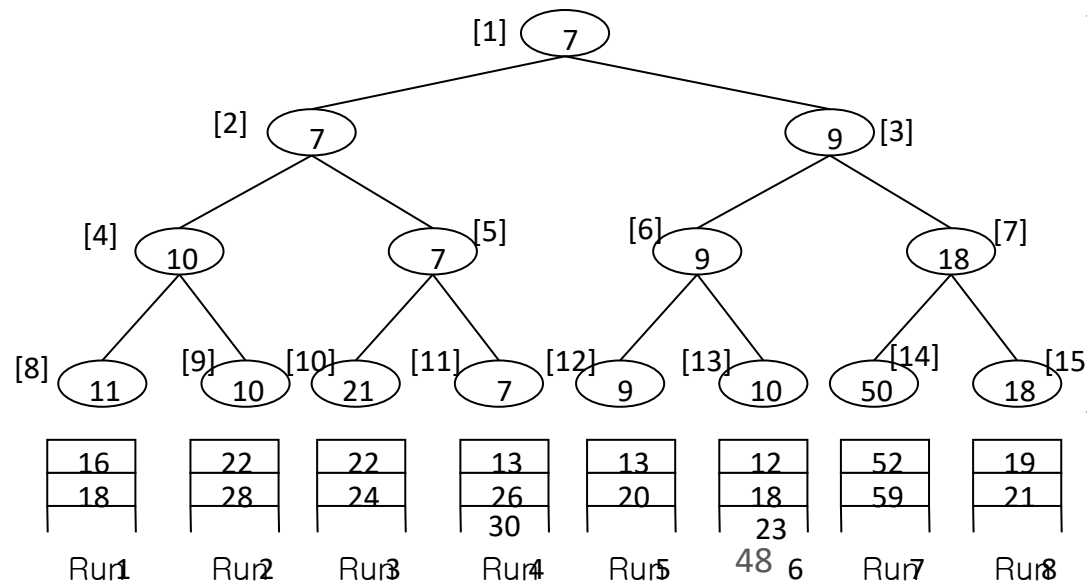
- 특징

- 완전 이진 트리

- 각 단말 노드는 각 Run의 최소 Key 값 Record를 나타냄

- 내부 노드는 그의 **두 자식 중에서 가장 작은 Key 값을 가진 Record (이진 Key)**를 나타냄

- Run이 8개(k=8)인 경우 승자 트리 예



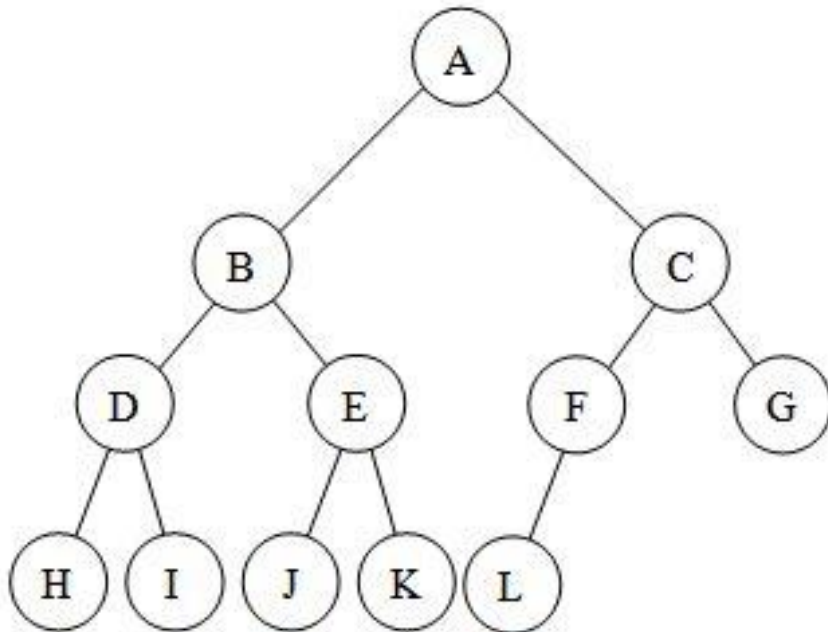
각 입력 Run의 제일 앞 Record (아직 출력 Run에 출력하지 않은 Record 중에 key 값이 제일 작은 Record) 에 대해 트리를 구축함.

각 입력 Run의 제일 앞 Record (아직 출력 Run에 출력하지 않은 Record 중에 key 값이 제일 작은 Record)



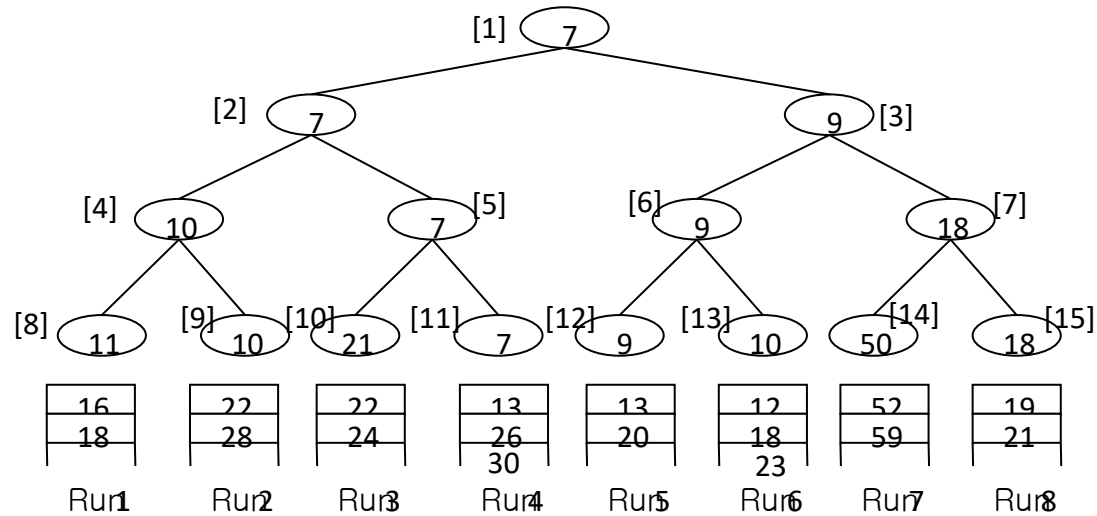
## 완전 이진 트리

- 다음 성질을 모두 만족하면 완전 이진 트리 (Complete Binary Tree)라 한다.
  - 1. Root에서 “트리 높이 - 1”번 만에 갈 수 있는 노드와 leaf 노드 (= Leaf 노드와 그 부모 노드)를 제외하고는 모든 노드는 자식이 둘이다.
  - 2. Leaf 노드는 tree의 왼쪽부터 채워진다.

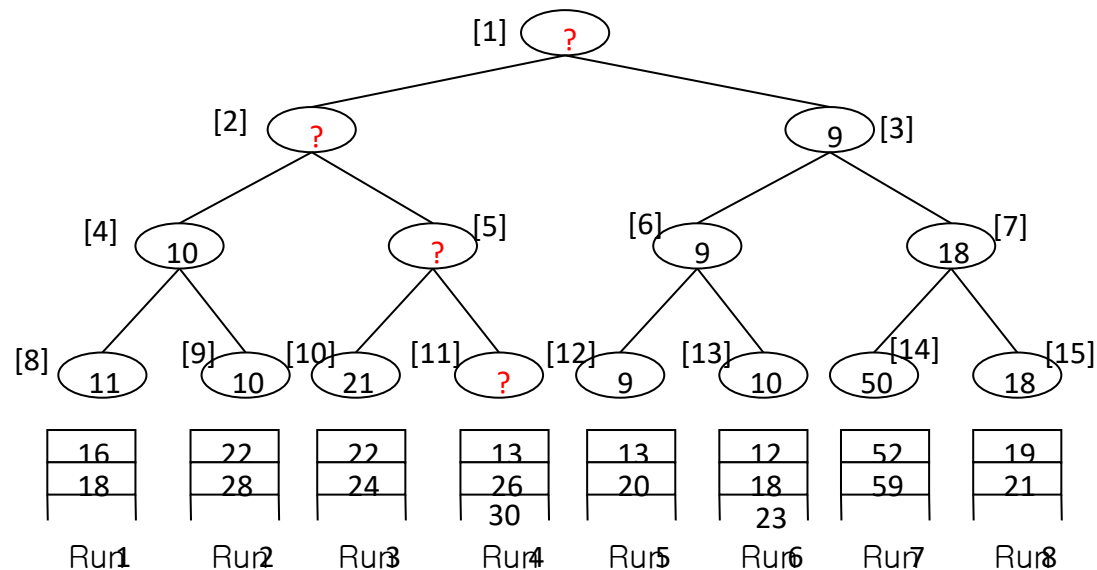


그림의 트리의 노드 L이 노드 F의 오른쪽 자식이면  
그림의 트리는 완전 이진 트리가 아니다.

## 승자 트리 갱신 예

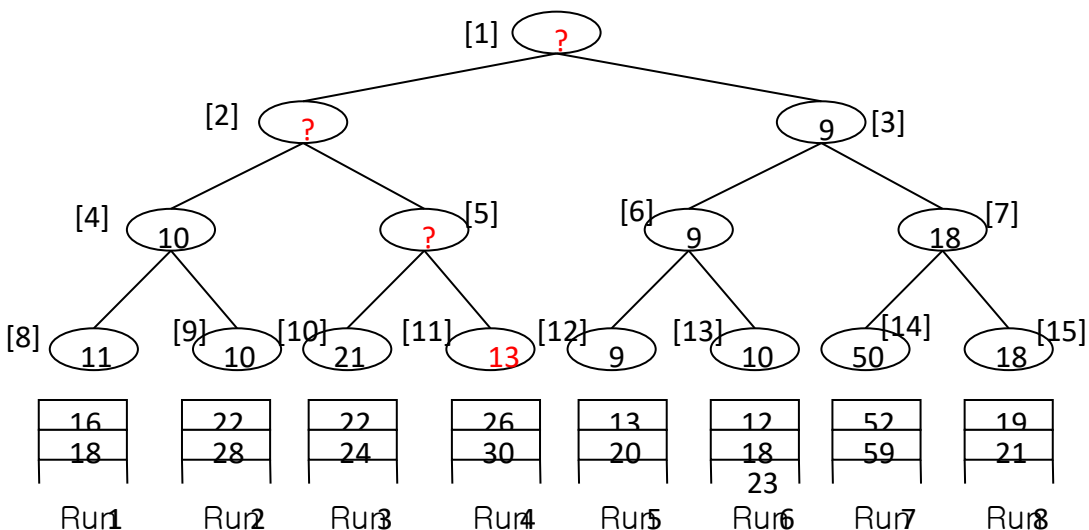


1. 현재 제일 key 값이 적은 Record는 Run 4의 Record 7 이므로, 해당 Record가 출력 Run (합병 결과 파일) 에 출력됨.

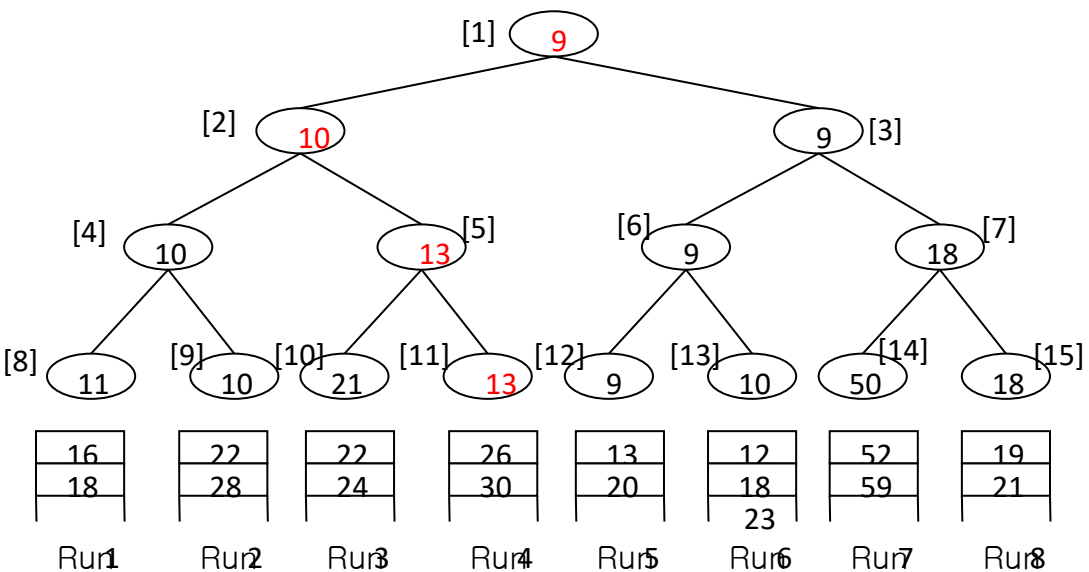


7이 출력되었으므로 왼쪽 승자 Tree에서 7이 의미가 없어짐

## 승자 트리 갱신 예 (Cont'd)



2. Run 4의 Record 130이 Run 4에서 가장 작은 key 값을 가진 Record가 됨.



3. 승자 트리의 node [5], [2], [1] 순으로 자식 두 개를 비교하여 승자를 계산함.

최대 7 회 비교해야 할 min Record 찾기가 3회 만에 끝남.

비교  $\lceil \log_2 N \rceil$  회. N : 입력 Run 수

## 승자 트리 (Cont'd)

- 승자 트리 구축 과정

- 가장 작은 키 값을 가진 Record가 승자로 올라가는 토너먼트 경기로 표현
- 트리의 각 내부 노드: 두 자식 노드 Record의 토너먼트 승자
  - 매 경기의 승자를 적어 놓은 토너먼트 표라 이해하면 됨
- 루트 노드: 전체 토너먼트 승자, 즉 트리에서 가장 작은 Key값 가진 Record

- 합병의 진행

- 1.루트가 결정되는 대로 출력 Run (병합 파일)에 루트에 들어 있는 Record 출력
- 2.Record가 출력된 Run 에서 다음 Record가 승자 트리로 들어감
- 3.승자 트리를 다시 재구성
  - Record가 새로 들어간 노드에서부터 루트까지의 경로를 따라가면서 형제 노드간 토너먼트 진행
- 앞 페이지의 예제 참조

# 패자 트리

- 패자 트리(loser tree)

- 루트 위에 0번 노드가 추가된 완전 이원 트리

- 성질

- (1) 단말 노드 (Leaf node) : 각 Run 의 최소 키 값을 가진 Record

- (2) 내부 노드 : 토너먼트의 **승자 대신 패자 Record가 기록됨 (=해당 토너먼트 패자의 비석이라 볼 수 있음)**

- 승부를 가려서 패자가 내부 노드에 기록되지만,

- 다음 토너먼트에 진출 할 수 있는 자는 패자가 아닌 승자.

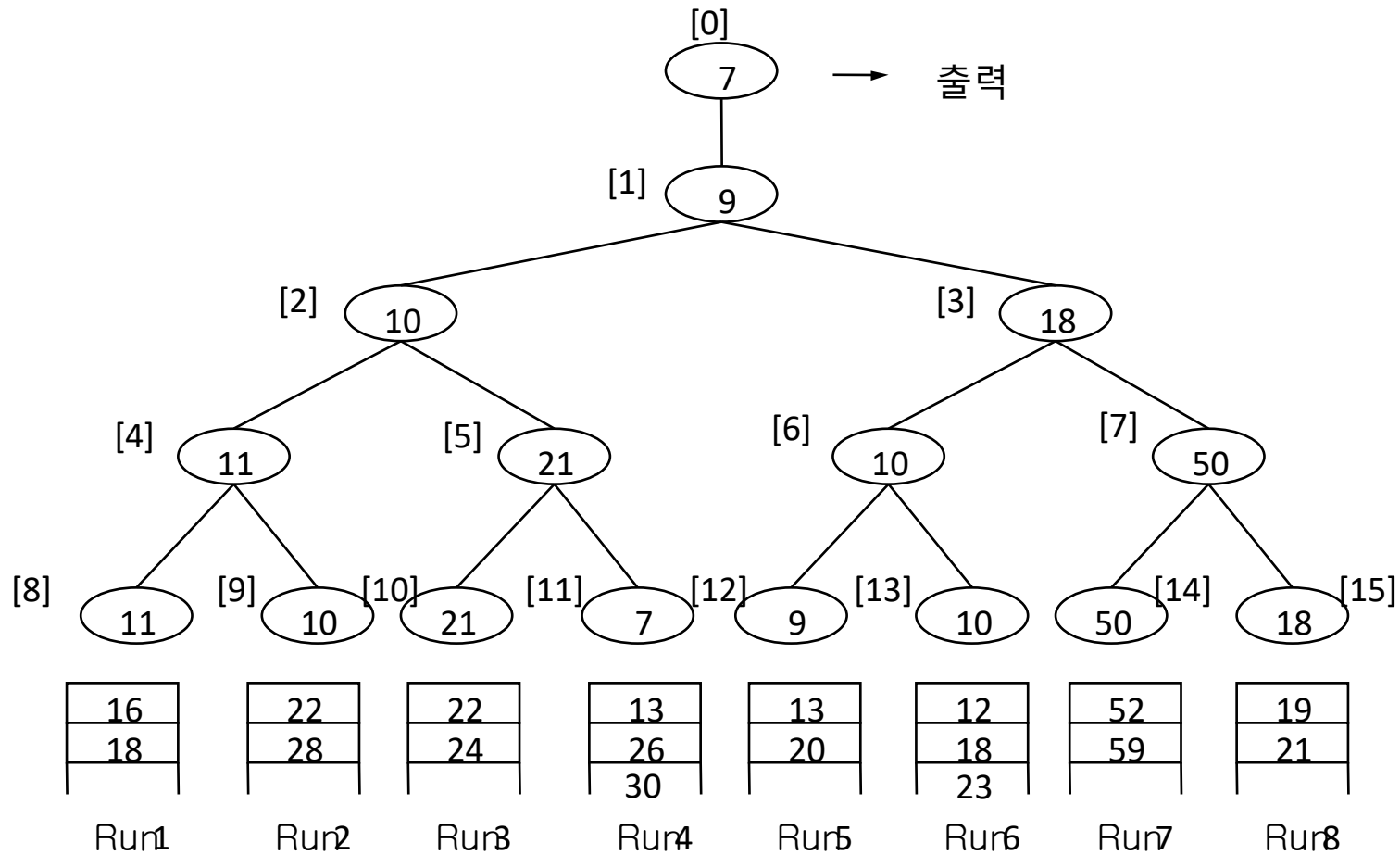
- 각 팀이 올라간 최종 경기가 기록되어 있는 토너먼트 표를 생각하면 됨.

- (3) 루트(1번 노드) : **준우승자**: 결승 토너먼트의 패자

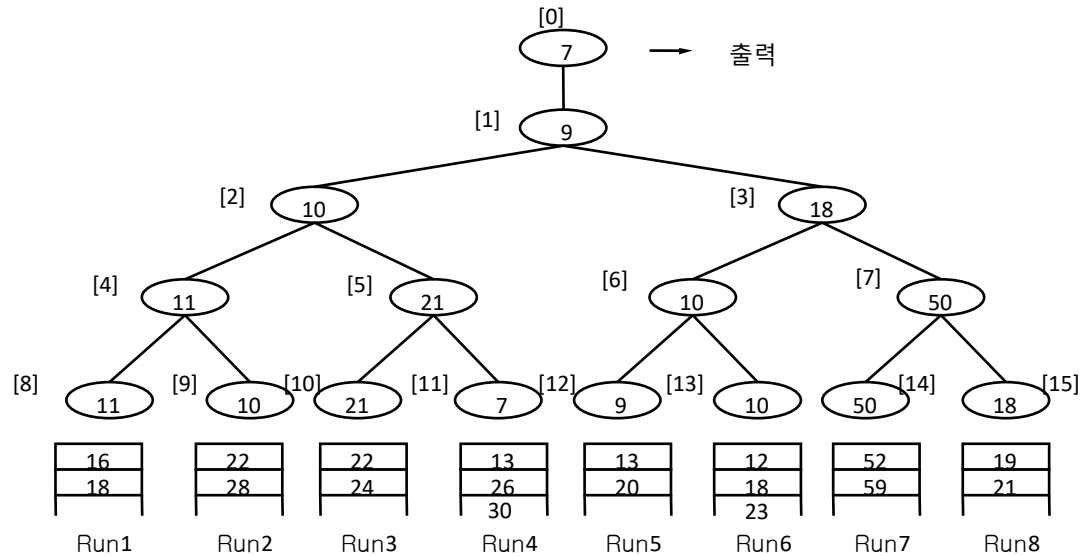
- (4) **0번 노드 : 우승자**: 전체 승자(루트 위에 별도로 위치)

## 패자 트리 (Cont'd)

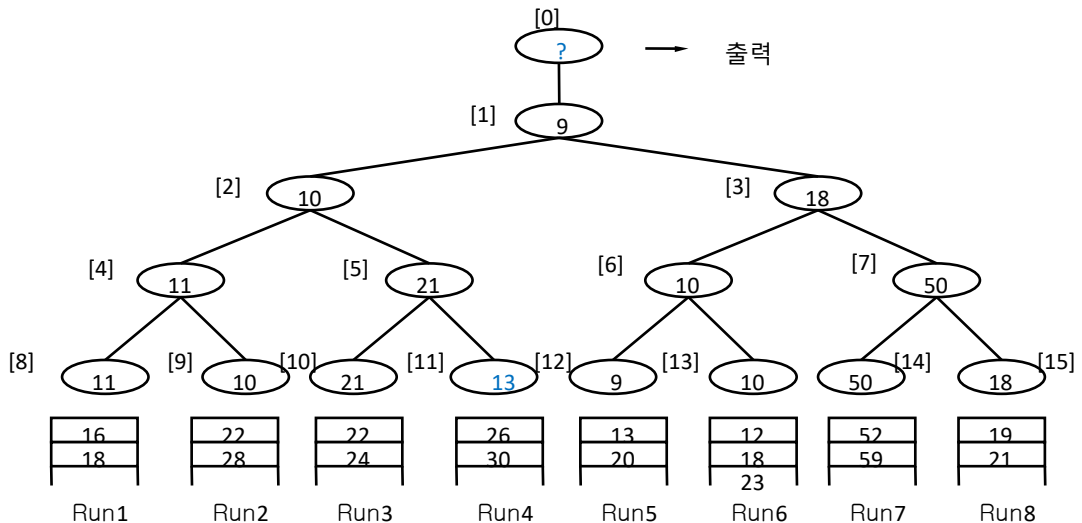
- Run이 8개(k=8)인 패자 트리의 예



# 패자 트리 갱신 예

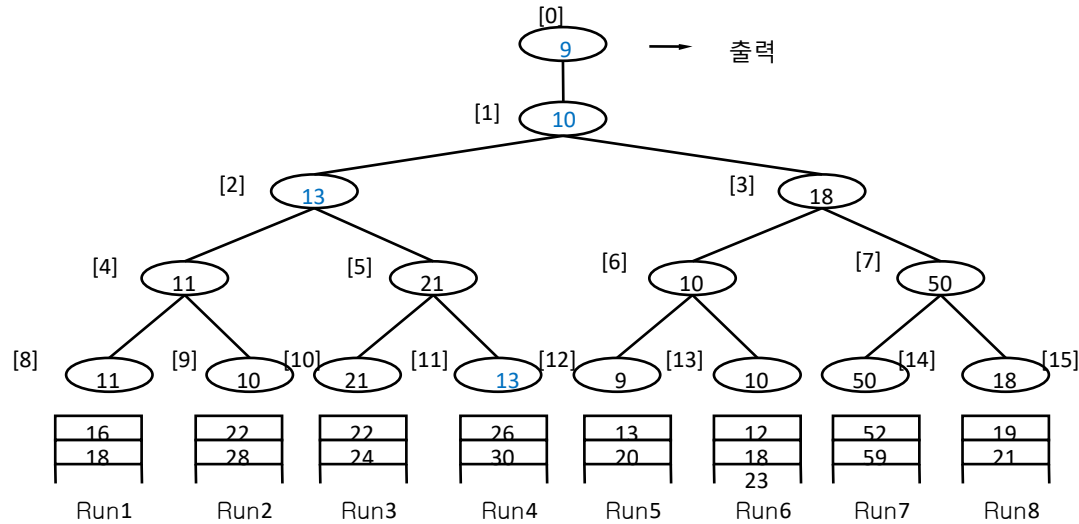


1. 현재 제일 key 값이 적은 Record (루트 위 0번 노드의 Record)는 Run 4의 Record 7 이므로, 해당 Record가 출력 Run 파일 (합병 결과 파일) 에 출력됨.



2. Run 4의 Record 13이 Run 4에서 가장 작은 key 값을 가진 Record가 됨.

# 패자 트리 갱신 예



3. 승자 트리의 node [5], [2], [1] 순으로 13과 해당 노드에 기록되어 있는 노드와 토너먼트를 치른다.  
Node [2]에서 13이 10에 졌으므로 13은 Node[2]에 기록되고 10 이 node [1]로 올라간다.

10이 node [1]에서 이전 준우승자인 9에 져서 10은 node[1]에 기록되고 9가 최종 승자가 된다.



## 패자 트리 (Cont'd)

- 패자 트리 구축 과정

- 단말 노드: 각 Run 의 최소 키 값 Record
- 내부 노드
  - 두 자식 노드들이 부모 노드에서 토너먼트 경기를 수행
  - 패자는 부모 노드에 남음 (기록됨)
  - 승자는 그 위 부모 노드로 올라가서 다시 토너먼트 경기를 계속
  - **각 팀이 올라간 최종 경기가 기록되어 있는 토너먼트 표를 생각하면 됨. (혹은 패자의 비석)**
- 1번 루트 노드
  - 마찬가지로 패자는 1번 루트 노드에 남음
  - 승자는 전체 토너먼트의 승자로서 0번 노드로 올라가 출력 Run (병합 파일)에 출력됨

- 합병의 진행 (앞페이지 예제 참고)

- 출력된 Record가 속한 Run 4의 다음 Record, 즉 키 값이 13인 Record를 패자 트리 노드 11에 삽입
- 패자 트리를 다시 재구성
  - 토너먼트는 노드 11에서부터 루트 노드 1까지의 경로를 따라 경기를 진행
    - 진 Record는 해당 노드에 기록되고 이긴 Record가 다음 노드로 올라간다.
  - 다만 경기는 형제 노드 대신 형식상 부모 노드와 경기를 함

# Sort & Merge - 여러 기법들 사이의 차이

---

- **Sort & Merge 기법의 차별화 요소**

- 기법들의 목적은 모두 다 Sort & Merge 를 목적으로 하므로 같다. 아래 요소들로 차별화 된다.
  - 적용하는 내부 정렬 방식
  - 내부 정렬을 위해 할당된 메인 버퍼의 크기
  - 정렬된 Run들의 보조 저장장치에서의 저장 분포
    - Tape 저장 장치일 때 매우 중요했음. HDD의 경우 특히 중요하지 않음
  - **Sort & Merge** 단계에서 동시에 처리할 수 있는 Run의 수

- **위 차별화 요소에 따라 초기 Run의 수와 합병의 pass 수가 결정**

- 패스(pass): 입력 Run 구성의 변경 없이 한번에 계속할 수 있는 연속된 병합 과정을 1 패스라 한다.
- 예) “Run 6개를 2원 합병으로 합병 => 3 개의 합병된 Run이 생성” => 1 pass 종료.

## Sort & Merge - 여러 기법들 사이의 차이 (Cont'd)

---

- **Sort & Merge 기법의 성능 비교**
  - 정렬 단계에서 만들어지는 Run의 수와 합병의 Pass 수 비교
    - Run의 수가 적을 수록, Pass 수가 적을수록 빠르다.
  - Record들의 Read/Write 횟수
    - 가능한 Run의 길이를 길게 만들면 합병해야 될 Run의 수는 최소화
    - 동시에 합병할 수 있는 Run의 수를 늘리면 합병의 Pass 수는 감소