제 7 장 정렬(계속)

7.5 합병 정렬

(1) **합병**

- **이미 정렬된 2개의 리스트를 1개의 정렬된 리스트로 만드는 것**
- **예**:

```
 initList:     3   4   8   10   2   6   8   9
               ↑           ↑   ↑           ↑
               l           m  m+1          n
              i1                 i2
mergedList: 2   3   4   6    8   8   9   10
```

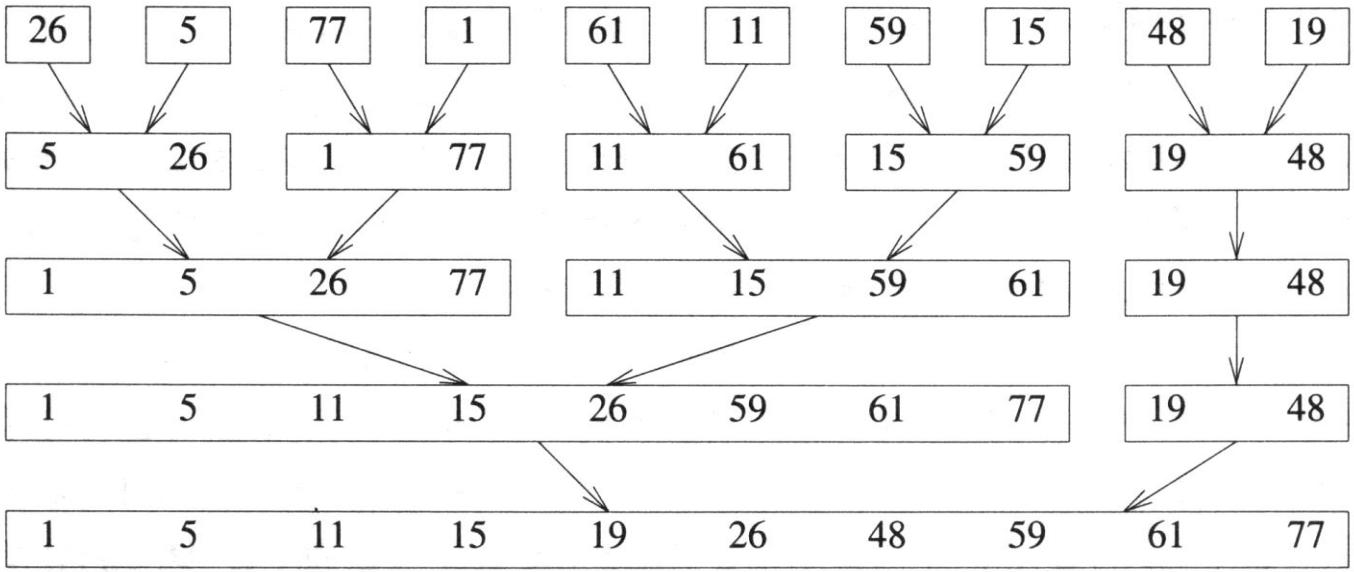
- 배열을 이용할 경우, 합병 결과를 또 다른 배열에 저장해야 한다.
- 프로그램 7.7: for 루프를 반복할 때마다 iResult가 1만큼 증가하므로 O(n-l+1)이다.

합병 함수(프로그램 7.7)

```
Template <class T>
void Merge(T *initList, T *mergedList, const int l, const  int m, const int n)
{
  for(i1=l, iResult=l, i2=m+1; i1<=m && i2<=n; iResult++)
     if(initList[i1]<= initList[i2])
     {              //앞쪽 리스트의 원소가 더 작은 경우
       mergedList[iResult] = initList[i1];
       i1++;     }
     else
     {              //뒤쪽 리스트의 원소가 더 작은 경우
       mergedList[iResult] = initList[i2];
       i2++;     }
  if(i1 > m)     // 뒤쪽 리스트의 원소가 남았음
        for (int t=i2; t<=n; t++)
          mergedList[iResult+t-i2] = initList[t];
    else          // 앞쪽 리스트의 원소가 남았음
        for (int t=i1; t<=m; t++)
          mergedList[iResult+t-i1] = initList[t];
}
```

(2) 반복 합병 정렬

- 입력 열의 길이가 1인 n개의 정렬된 리스트를 합병하여 길이가 2인 리스트들을 만든다.
- 길이가 2인 n/2 개의 리스트를 합병하여 길이가 4인 리스트를 만든다.
- 길이가 n인 리스트로 최종 합병될 때까지 계속한다.



합병 트리

합병 정렬을 위한 패스 함수

- 길이 s 의 인접한 서브리스트 한 쌍씩을 합병한다.
  → one pass 라 하며, O(n) 시간 걸린다.

```
void MergePass(T *initList, T *resultList, const int n, const int s)
// 합병 정렬의 한 패스만을 수행한다. 길이 s의 인접한 서브리스트 한 쌍을
// initList로부터 합병하여 그 결과를 resultList에 넣는다. n은 initList에 있는
// 원소 수이다.
{
   for (int i=1;   // i는 합병한 두 서브리스트 중에서 첫 번째 리스트의 첫 번째의 위치
    i <= n-2*s+1; // 길이 s의 두 서브리스트가 되기에 충분한 원소가 남아 있는가?
    i += 2*s)
        Merge(initList, resultList, i, i+s-1, i+2*s-1):

   // 길이가 2*s보다 작게 남아 있는 리스트를 합병한다.
   if((i < n-s+1) Merge(initList, resultList, i, i+s-1, n);
   else for (int t=i; t<=n; i++) resultList[t] = initList[t];
}
```

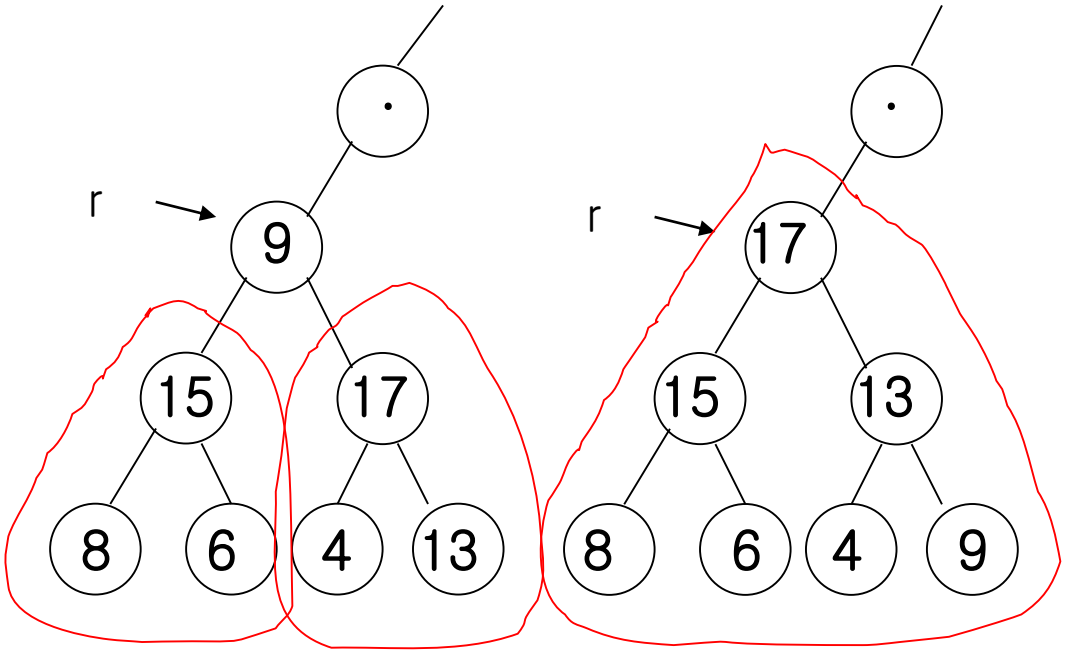
합병 정렬 함수

- 길이 l = 1, 2, 4, 8, ... (l < n) 에 대해 MergePass를 적용한다.
- 첫번째 패스에서는 길이가 1, 두번째 패스에서는 길이가 2, i번째 패스에서는 길이가 2^{i-1}인 리스트가 합병된다. 따라서 총 패스수는 log n 이며, 전체 시간은 O(n log n)이다.

```
void MergeSort(T *a, const int n)
// a[1:n]을 비감소순으로 정렬한다.
{
   T *tempList = new T[n+1];
    // l 은 현재 합병되고 있는 서브 리스트의 길이이다.
   for(int  l=1;  l<n;  l *= 2)
   {
     MergePass(a, tempList, n, l);
     l *= 2;
     MergePass(tempList, a, n, l); // a와 tempList의 역할을 교환
   }
   delete [] tempList;
}
```

7.6 히프 정렬

- 5장에서 소개된 최대 히프 구조를 이용한다.
- 조정(adjustment): r 이 가리키는 노드를 루트로 하는 부분 트리를 최대 히프로 만든다. 이때 r 이 가리키는 노드의 왼쪽, 오른쪽 서브트리는 이미 최대 히프이다.



조정 함수

```
void Adjust(T *a, const  int  r, const  int  n)
// r 을 루트 노드로 하는 이진 트리를 히프 성질을 만족하도록 조정한다
// r 의 왼쪽과 오른쪽 서브트리는 히프 성질을 이미 만족한다
// 어떤 노드도  n보다  큰  인덱스를 갖지 않는다
{
   T e = a[r] ;
    // e에 대한 적절한 위치를 탐색
   for(int j = 2 * r; j <= n; j *= 2)
   { // 먼저 왼쪽과 오른쪽 자식의 최대를 찾는다
      if(j < n &&  a[j] < a[j+1]) j++ ;
    // 최대 자식을 e 와 비교한다.  e가 최대이면 완료
      if(e >= a[j]) break;
      a[j/2] = a[j]; // j번째 레코드를 트리 위로 이동시킨다
   }
   a[j/2] = e;
}
```

- 시간은 트리의 높이에 비례하므로, 완전이진트리의 높이가 O(log n)이므로 걸리는 시간은 최대 O(log n)이 된다.

히프 정렬을 위한 2단계

(i) a를 초기 히프로 변환한다.
(ii) 정렬작업을 수행한다.

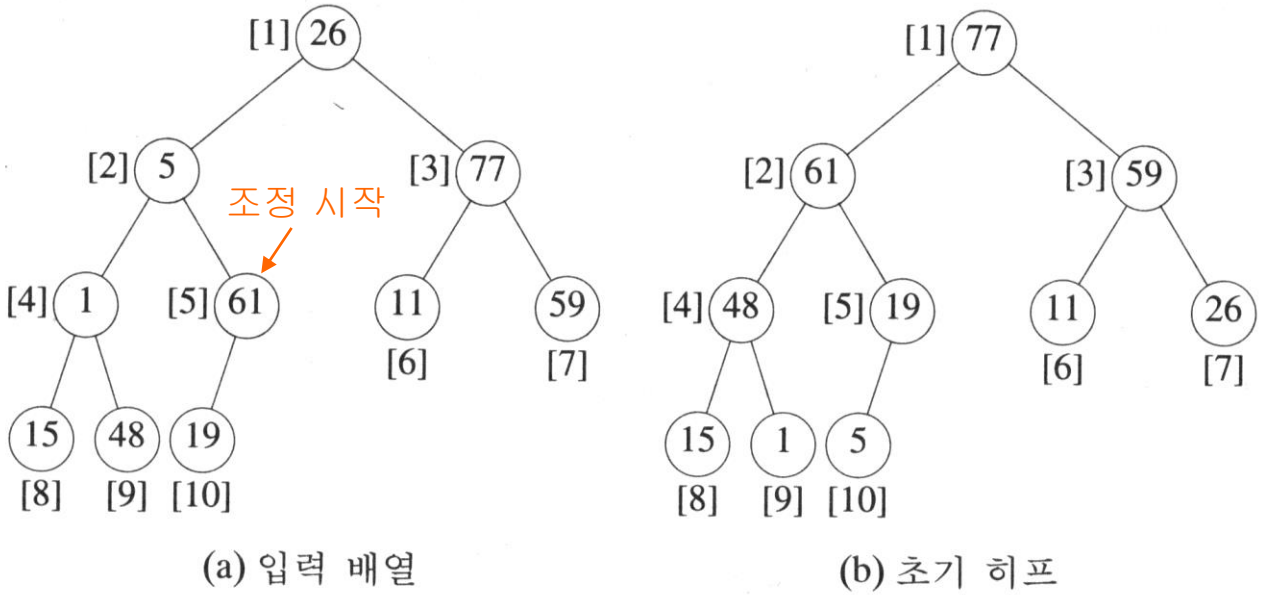


그림 7.9 : 이진 트리로 해석되는 배열

히프 정렬 과정

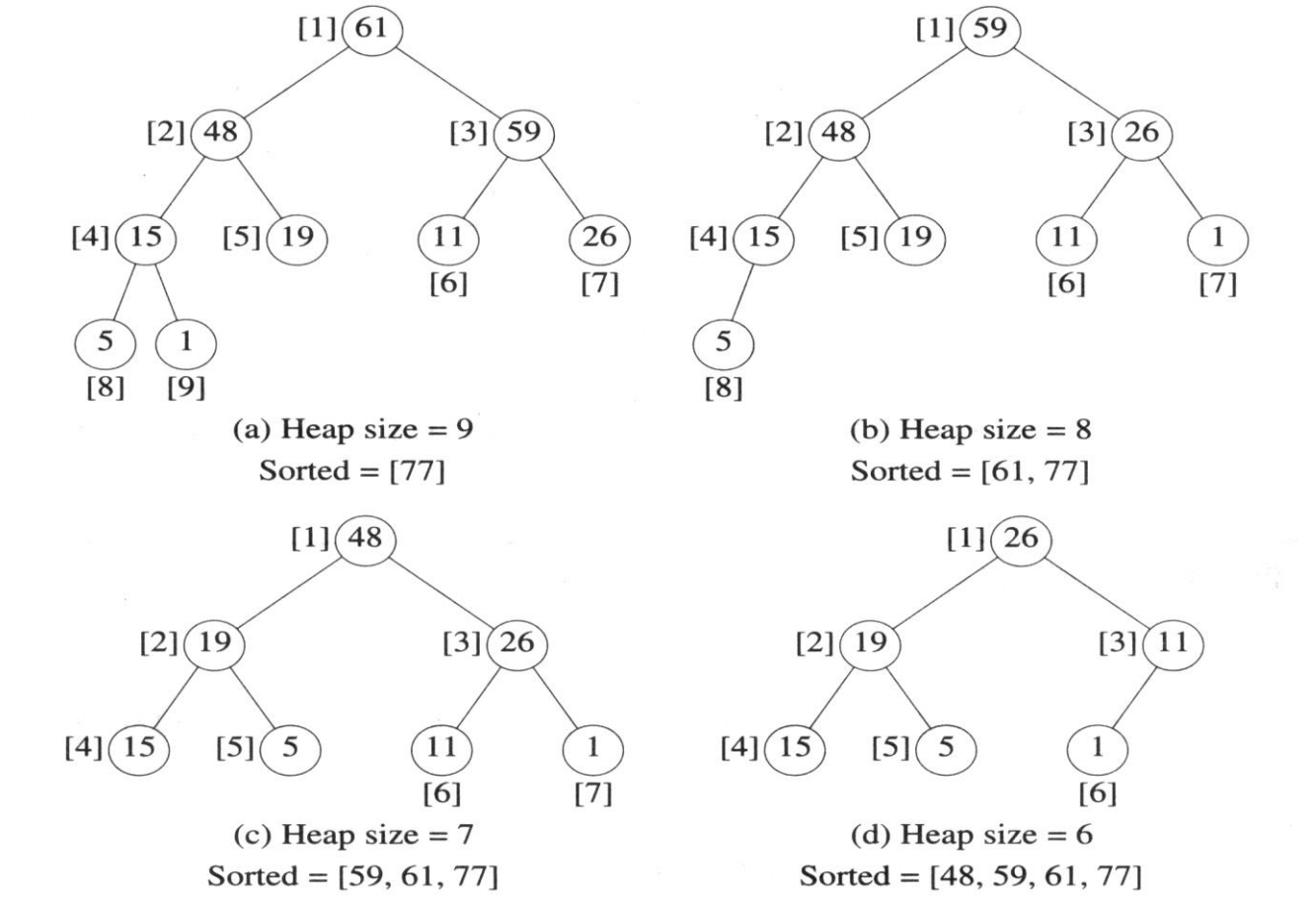
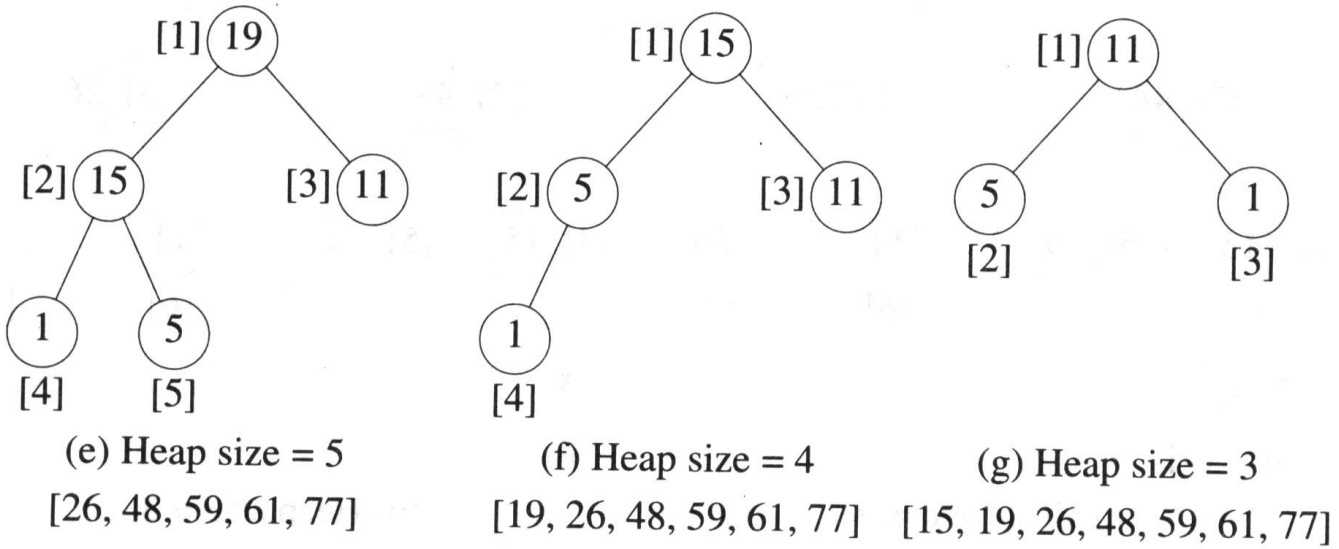


그림 7.10 : 히프 정렬 예제(계속)

(계속)

히프 정렬 과정(계속)



(e) Heap size = 5
[26, 48, 59, 61, 77]

(f) Heap size = 4
[19, 26, 48, 59, 61, 77]

(g) Heap size = 3
[15, 19, 26, 48, 59, 61, 77]

그림 7.10 : 히프 정렬 예제

히프 정렬 함수와 분석

```
void HeapSort(T *a, const int n)
// a[1:n]을  오름차순으로 정렬한다.
{
   for(int i = n/2;  i >= 1;  i--) // 히프로 변환
      Adjust(a, i, n);
   for(i=n-1; i >= 1; i--) // a를 정렬
   {
     T  t = a[i+1] ; // list1 과 listi+1을 교환
     a[i+1] = a[1];
     a[1] = t;
     Adjust(a, 1, i); // 트리의 루트로부터 히프 조정
   }
}
```

- 분석: (i) 초기 히프의 변환: O(n)
  (ii) for 문이 n-1 번 반복하고, 각 반복에서 조정작업이 한번씩 일어나므로 O(n log n) 시간 걸린다.
  따라서 전체 시간은 O(n log n)이 된다.

7.7 여러 키에 의한 정렬

- r 개의 키 K^1, K^2, ... K^r (K^1 는 최대 유효 키, K^r 은 최소 유효키)를 갖는 레코드의 정렬 ==> 사전 정렬(lexical sort),
  즉 $(K_i^1, K_i^2, \ldots, K_i^r) \leq (K_j^1, K_j^2, \ldots, K_j^r)$ 의 순서로 정렬
- 예: 카드 정렬: 2개의 키
  K^1 (무늬): ♣ < ◆ < ♥ < ♠
  K^2 (숫자): 2 < 3 < ... < 10 < J < Q < K < A
- MSD 정렬방법: 같은 무늬를 무늬 크기 순으로 놓은 다음,
  각 무늬에서 숫자 크기로 배열한다.
  먼저 4 묶음이 된 후, 모두 정렬된다.
- LSD 정렬방법: 숫자를 크기 순으로 놓은 다음, 각 숫자에서
  무늬를 크기 순으로 배열한다.
  먼저 13 묶음이 된 후, 모두 정렬된다.

기수 정렬(radix sort)

- 하나의 키를 가진 리스트에 MSD 또는 LSD를 적용하는 정렬 방법이다.
- 기수 r을 이용하여 정렬 키를 몇 개의 숫자로 분해하여 MSD 또는 LSD 정렬 방법을 사용한다.
- 예: r=10, 3개의 키로 분해

  342 → 3개의 키: 3, 4, 2

  58 → 3개의 키: 0, 5, 8

  745 → 3개의 키: 7, 4, 5

  7 → 3개의 키: 0, 0, 7
- 리스트로 저장되며, 각각의 빈(bin)을 구현하기 위해 큐를 사용한다.

기수 정렬의 예

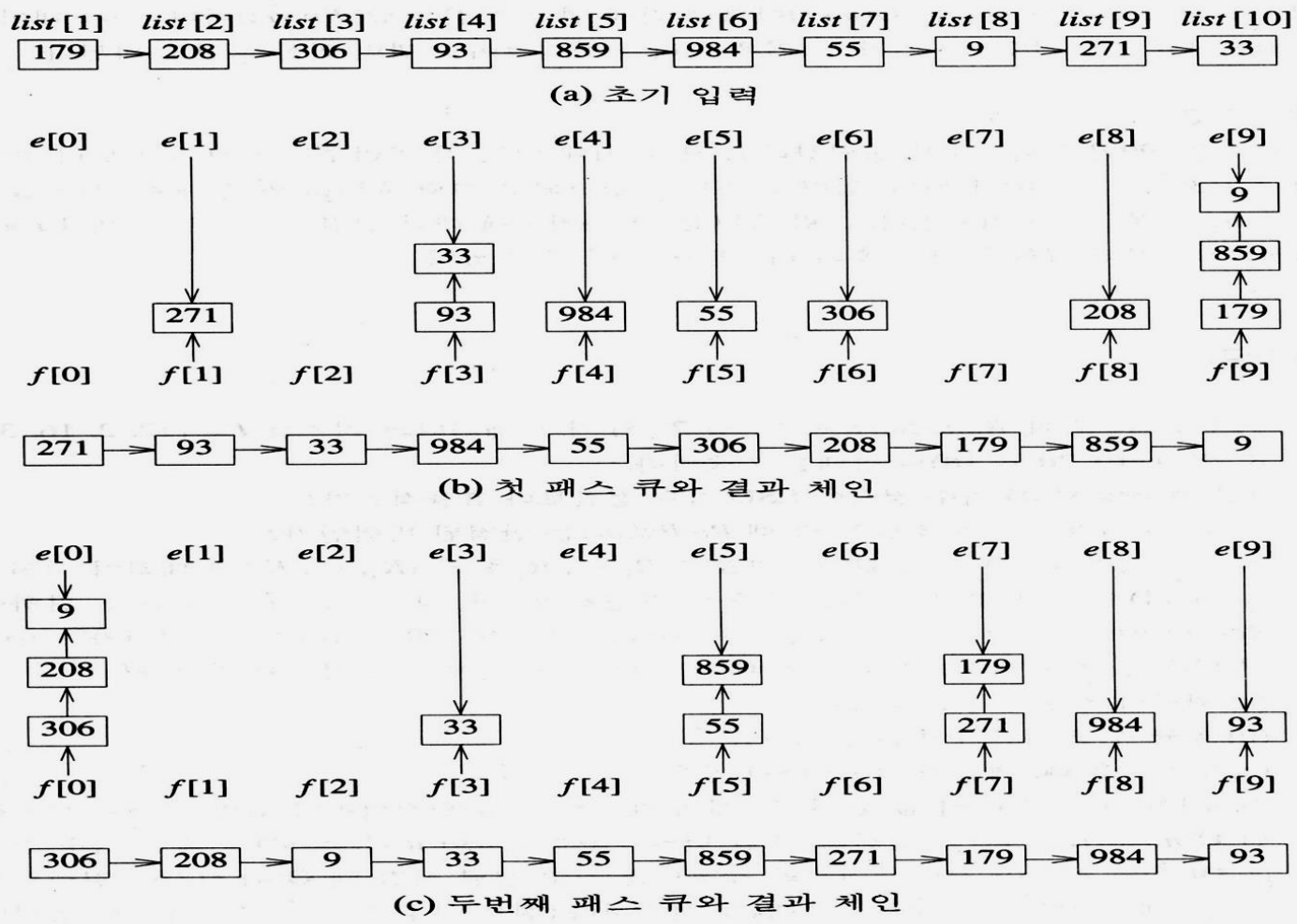
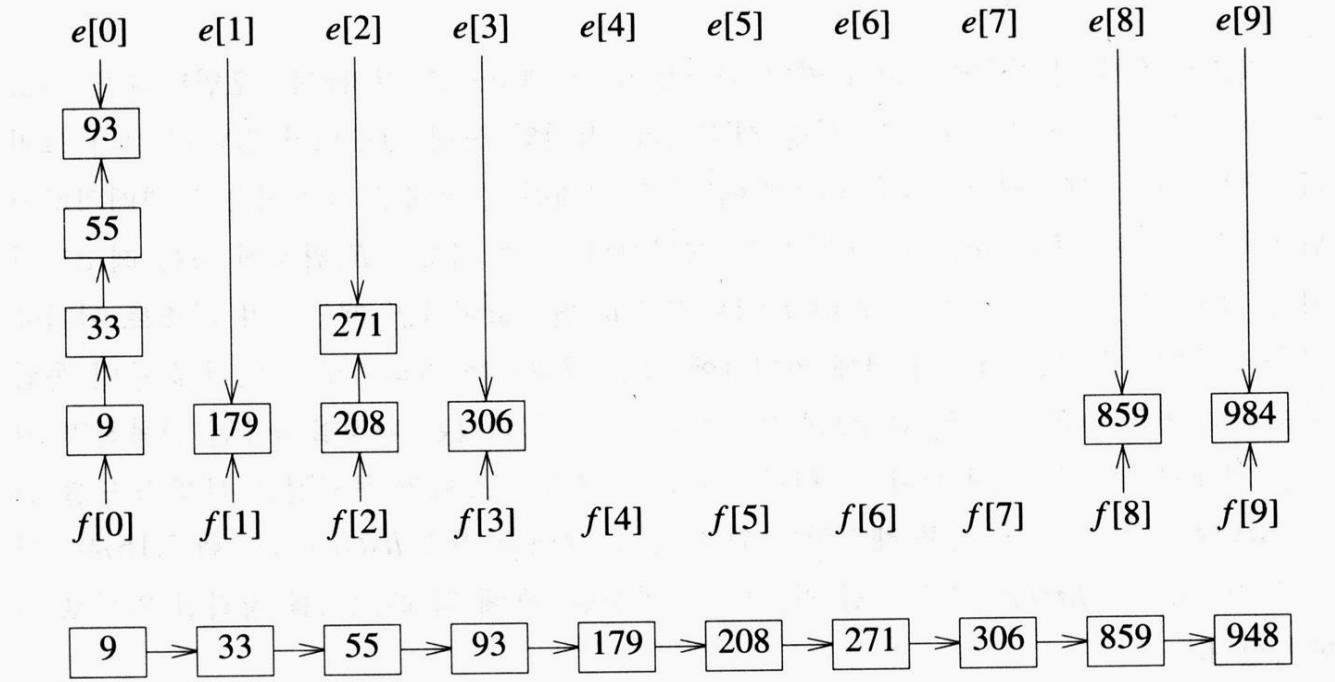


그림 7.11 : 기수 정렬 예제(계속)

기수 정렬의 예(계속)



(d) 세번째 패스 큐와 결과 체인

그림 7.11 : 기수 정렬 예제

기수 정렬의 분석

- 분석:

  d = 키를 r로 분해했을 때 숫자 키의 수,

  n = 레코드의 수,

  이때, d 번의 패스가 필요하며, 각각의 패스에서 r개의 큐가 사용된다.

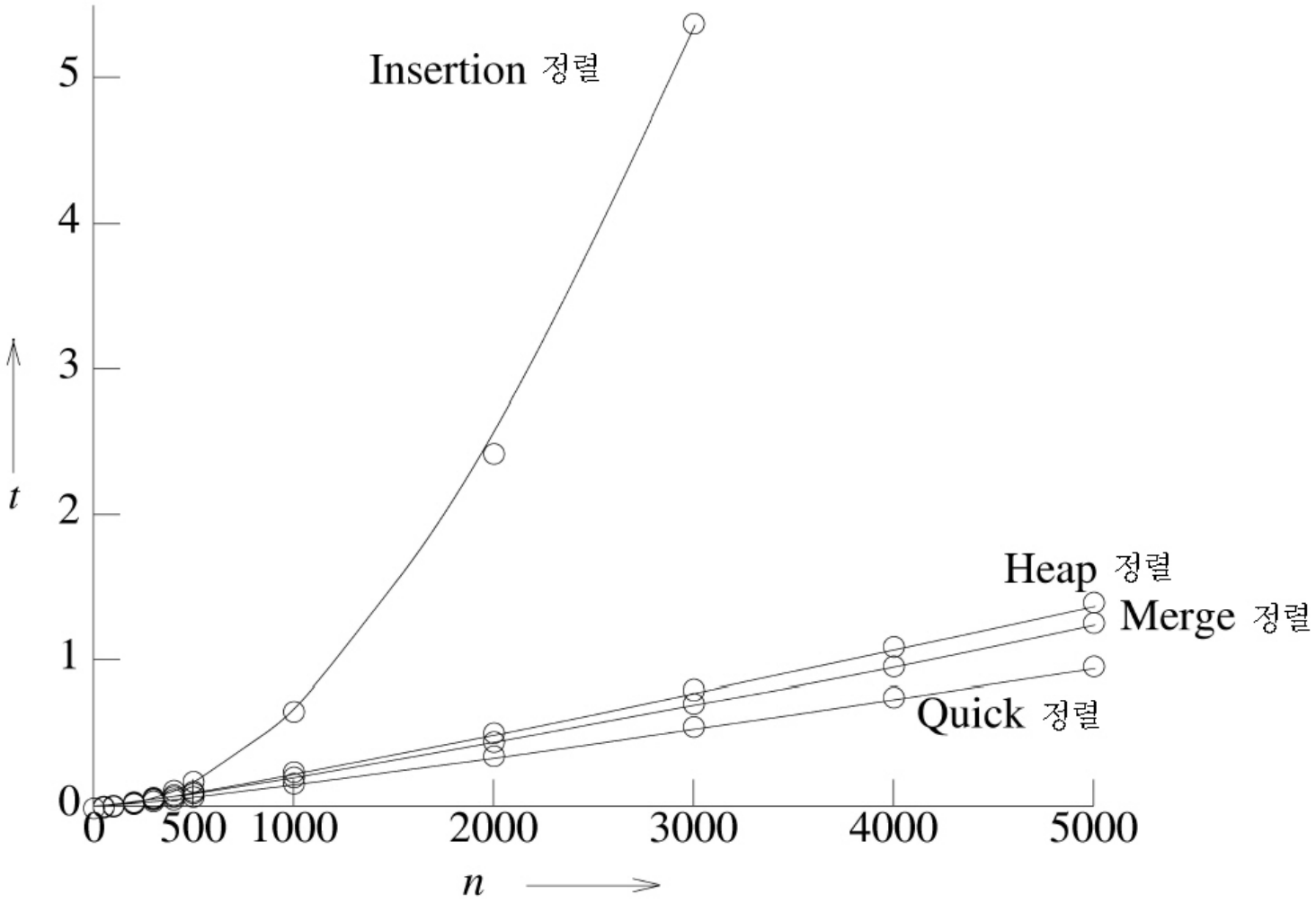
- 시간 복잡도: 각 패스의 연산은 O(n+r)이므로 전체 연산 시간은 O(d(n+r))이 된다.

7.9 내부 정렬의 요약

- 512MB RAM을 가진 1.7GHz Intel Pentium 4 PC와 Microsoft Visual Studio .NET 2003으로 얻은 결과 (시간은 msec)

n	삽입	히프	합병	퀵
0	0.000	0.000	0.000	0.000
50	0.004	0.009	0.008	0.006
100	0.011	0.019	0.017	0.013
200	0.033	0.042	0.037	0.029
300	0.067	0.066	0.059	0.045
400	0.117	0.090	0.079	0.061
500	0.179	0.116	0.100	0.079
1000	0.662	0.245	0.213	0.169
2000	2.439	0.519	0.459	0.358
3000	5.390	0.809	0.721	0.560
4000	9.530	1.105	0.972	0.761
5000	15.935	1.410	1.271	0.970

평균 시간 그래프


Insertion 정렬
Heap 정렬
Merge 정렬
Quick 정렬
t
n
0
1
2
3
4
5
0
500
1000
2000
3000
4000
5000