정렬

- 정렬 이용 : 탐색, 리스트 확인

- 용어 : 객체의 집합에 대한 정보의 모임이 존재
  - 리스트(list): 주기억 용량에 모두 포함될 경우
  - 파일(file): 외부에 저장될 경우
  - 레코드: 객체 하나에 대한 정보
  - 필드: 레코드의 정보를 작은 단위로 나눈 것
  - 키: 레코드를 식별해주는 필드

  - 예)리스트:전화번호부, 이름, 주소, 전화번호 필드로 이루어진 레코드, 키는 전화번호 또는 이름

- 정렬되어 있지 않은 배열: 순차 탐색
- 정렬된 배열: 이진 탐색

```
int seqsearch (int list[], int searchnum, int n)
   {
   /* 크기가 n인 배열 list를 탐색, list[i] = searchnum인 경우 i
   를 반환, searchnum이 list에 없을 경우에는 –1을 반환 */
   int i ;
   list[n] = searchnum ;
   for (i=0; list[i] != searchnum; i++)
         ;
   return ((i<n)? i : -1) ;
}
```

- 성공시 평균 비교 횟수
  - (1 + 2 + … + n) / n ➔ O(n)

```
#define MAX_SIZE 1000 //리스트의 최대 크기 + 1
typedef struct {
  int key ;
  /* 그 밖의 필드들 */
  } element ;
element list[MAX_SIZE] ;
```

```
int binsearch (element list[], int serachnum, int n)
{
   /* list[0], …, list[n-1]을 탐색함 */
   int left = 0, right = n-1, middle;
   while (left <= right) {
        middle = (left + right) / 2 ;
        switch (COMPARE(list[middle].key, searchnum)) {
                case -1 : left = middle + 1 ;
                          break ;
                case 0 : return middle ;
                case 1 : right = middle -1 ;
        }
   }
   return -1 ;
 }
```

분석: j 번 탐색 후에는 n / 2^j 개의 탐색할 리스트➔ O(log n)

- 보간 탐색
  - 사람 전화번호부를 찾을 때 사용
  - l = ((k-f[l].key)/ (f[u].key – f[l].key) * n
  - l, u는 최소, 최대 키 원소

- 리스트 확인
  - 리스트들이 같은지 다른지 확인하기 위해 리스트를 비교
  - list1[I].key(0<=I <= n)과 list2[j].key(0<= j <= m) 비교
  - 세 종류의 오류 검출
    1. list1에 있고, list2에 없는 경우
    2. list1에 없고, list2에 있는 경우
    3. list1과 list2에 key는 같지만 값이 일치하지 않는 필드가 존재

- 리스트를 확인하는 함수
  - verify1 : O(m n)
  - verify2 : 정렬을 사용하는 경우 O(max[n log n, m log m])

```
void verify1(element list1[], element list2[], int n, int m)
 {/* 임의 순서로 배열된 두 리스트 list1, list2를 비교 */
  int i, j ;
  int marked[MAX_SIZE] ;

  for (i=0; i < m; i++)
       marked[i] = FALSE ;
  for (i=0; i<n; i++)
       if ((j=seqsearch(list2, m, list1[i].key)) < 0)
            printf("%d is not in list2 ₩n",list1[i].key);//오류 1
       else /* list1[i]와 list2[i]의 다른 필드들을 검사하여 일치
  하지 않는 내용을 출력 : 오류 3 */
                 marked[j] = TRUE ; //list2[j]는 list1에 존재
  for (i=0; i < m; i++)
       if (!marked[i])
         printf("%d is not in list1₩n", list2[i].key)  ; //오류 2
}
```

```
void verify2(element list1[], element list2[], int n, int m)
 {/* 임의 순서로 배열된 두 리스트 list1, list2를 비교 */
   int l, j ;
   sort (list1, n) ; sort (list2, m) ;
   l = j = 0 ;
   while (l<n && j<m)
         if (list1[l].key < list2[j].key) {
                  printf("%d is not in list2 ₩n", list1[l].key) ;
                  l++ ;
         }
         else if (list1[l].key == list2[j]key) {
         //list1[l]와 list2[l]의 다른 필드들을 검사하여 일치하지 않는 내용을 출력
                  l++; j++ ;
         }
         else {
                  printf("%d is not in list1 ₩n", list2[j].key) ;
                  j++ ;
         }
   for(; l<n; l++)
         printf("%d is not in list2 ₩n", list1[i].key) ;
   for); j<m; j++)
         printf("%d is not in list1 ₩n", list2[j].key) ;
}
```

- 정렬 문제의 정형화
  - 레코드의 리스트 (R_0, R_1, …, R_n-1)
  - R_I는 키 K_I를 가짐
  - 정렬 문제란 K_σ(I-1) <= K_σ(I)인 순열을 찾는 것
  - 원하는 순서는 (R_σ(0), R_σ(1), …, R_σ(n-1))

  - [정렬] K_σ(I-1) <= K_σ(I), 단 0< I < n-1
  - [안정성] 입력 리스트에서 I < j이고 K_I = K_j이면, 정렬된 리스트에서도 R_I는 R_j보다 앞에 위치

- 내부정렬: 리스트를 주기억 장치에서 모두 정렬
- 외부정렬: 주기억 장치에서 모두 정렬되지 않고, 파일의 일부가 주기억 장치에 옮겨져 처리되어 점차적으로 파일을 정렬

- 삽입정렬
  - 리스트의 각 레코드를 한 번에 하나씩 볼 수 있을 때 사용
  - R_i를 정렬된 R_0, R_1,..., R_i-1 사이에 정렬되도록 삽입

```
void insertion_sort (element list[], int n)
{
  int i, j ;
  element next ;
  for (i=1; i<n; i++) {
    next = list[i] ;
    for (j=i-1; j >= 0 && next.key < list[j].key; j--)
            list[j+1] = list[j] ;
    list[j+1] = next ;
  }
}
```

- insertion_sort 알고리즘의 분석
  - 최악의 경우, 안쪽 루프는 I번 수행
  - 바깥쪽은 I=1, 2, …, n-1이므로
  - 총 1 + 2 + … + n –1 ➔ O(n^2), 평균 시간도 O(n^2)

- 예제 7.1 : 최악의 경우는 역순으로 입력될 때
- LOO(left out of order)
  - R_I < max_{R_j}, 0<= j < i
- 예제 7.2 : R_4 만 LOO

- 삽입 정렬은 O((k+1)n) (k는 LOO인 레코드 수), 소수의 레코드 (k << n)인 경우에 사용하기 좋은 알고리즘

- 이진 삽입 정렬: 순차 탐색 기법 대신 이진 탐색을 사용하면 비교 회수를 감소, 레코드 이동 회수는 동일
- 리스트 삽입 정렬: 동적 연결 리스트를 사용할 경우 레코드 이동은 없지만, 순차 탐색을 사용하여야 함

- 퀵 정렬
  - 가장 좋은 평균 수행
  - Pivot key를 중심으로 두 개의 서브 파일로 나누어 정렬

- 예제 7.3

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	left	right
[26	5	37	1	61	11	59	15	48	19]	0	9
[11	5	19	1	15]	26	[59	61	48	37]	0	4
[1	5]	11	[19	15]	26	[59	61	48	37]	0	1
1	5	11	[19	15]	26	[59	61	48	37]	3	4
1	5	11	15	19	26	[59	61	48	37]	6	9
1	5	11	15	19	26	[48	37]	59	[61]	6	7
1	5	11	15	19	26	37	48	59	[61]	9	9
1	5	11	15	19	26	37	48	59	61		

```
void quicksort (element list[], int left, int right)
{//pivot key로 list[left].key로 선택
   int pivot, i, j ;          element temp ;
   if (left < right) {
         i = left; j = right + 1;
         pivot = list[left].key ;
         do {     /* 왼쪽 경계 < 오른쪽 경계인 경우
                  do
                           i++ ;
                  while (list[i].key < pivot) ;
                  do
                           j-- ;
                  while (list[j].key > pivot) ;
                  if (i < j )
                           SWAP(list[i], list[j], temp) ;
         } while (i < j) ;
         SWAP(list[left], list[j], temp) ; //list[j] < pivot 이다.
         quicksort (list, left, j-1) ;
         quicksort (list, j+1, right) ;
   }
}
```

- Quicksort 알고리즘의 분석
  - 최악의 경우 O(n^2): 연습문제 2
  - 평균적으로 한 레코드의 위치가 정확히 정해질 때마다 두 개의 n/2 인 서브 화일을 정렬하는 작업
  - O(n): 하나의 레코드를 위치시키는 데 필요한 소요시간
  - T(n): n 개의 레코드를 정렬하는데 소요되는 시간

  - T(n) <= cn + 2 T(n/2), 임의의 상수 c에 대해서

    <= cn + 2 (c n/2 + 2 T(n/4))

    <= 2cn + 4 T(n/4)

    ...

    <= c n log_2 n + nT(1) = O(n log_2 n)

- 보조 정리 7.1 : T_avg (n)을 함수 quicksort가 n개의 레코드를 가지는 파일을 정렬하는 데 필요한 예상 시간이라고 할 때 n이 2보다 크거나 같은 경우 T_avg (n) <= k n log_e n 을 만족하는 상수 k가 존재

- 퀵 정렬의 순환구현에서는 스택공간이 필요하다. 최악의 경우 크기가 n-1과 0인 서브화일로 나눠질 경우 순환 깊이는 n이 되어 O(n)의 스택공간 필요

- 변형
  - Pivot key로 median {K_left, K_(left+right)/2, K_right} 사용 가능
  - median {10, 5, 7} = 7

합병 정렬

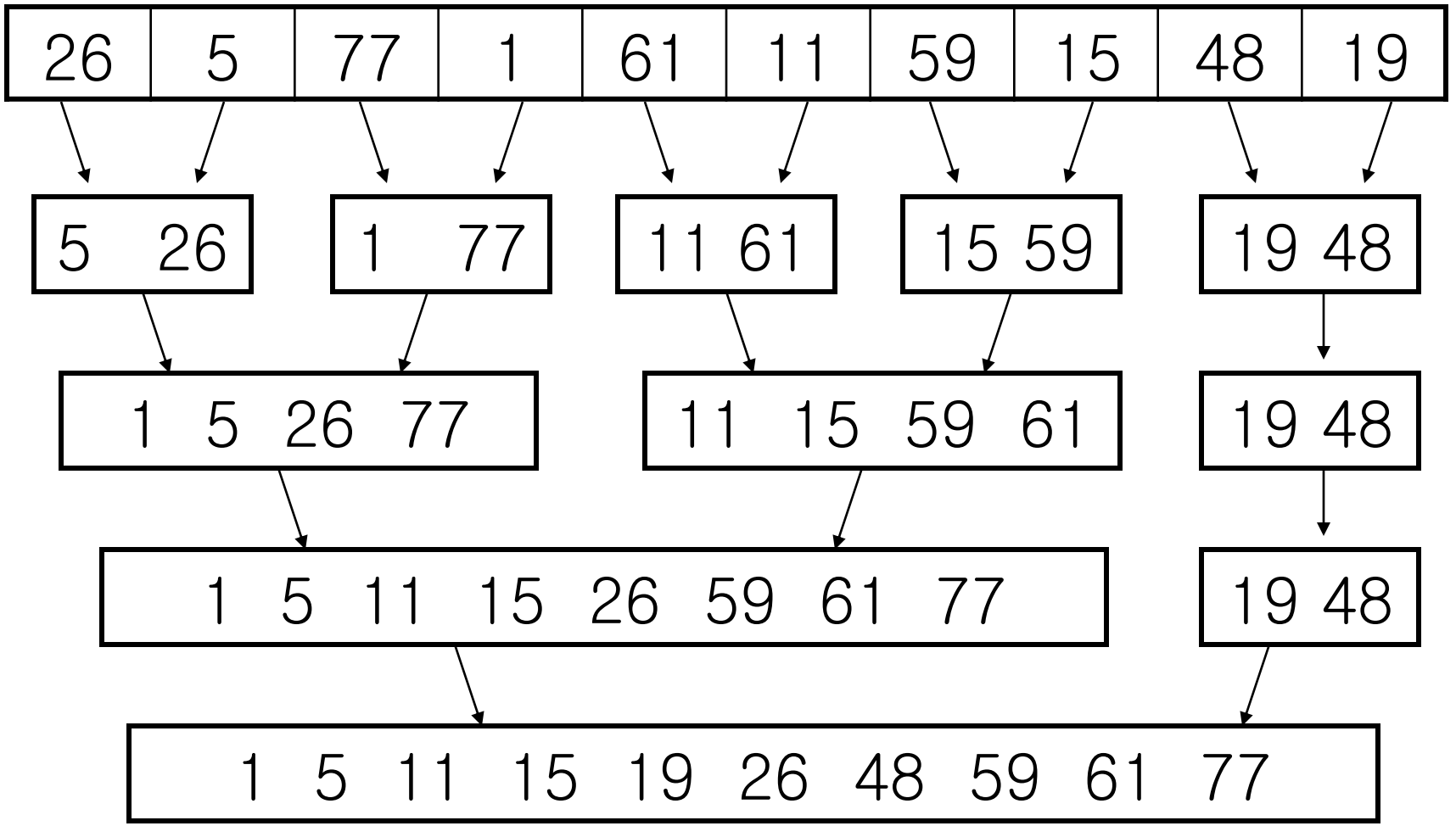
- 두 리스트를 합병하는 함수

```
void merge (element list[], element sorted[], int l, int m, int n)
/* 두 개의 정렬된 리스트 (list[l], ..., list[m])과 (list[m+1], ..., list[n])을 합
   병하여 하나의 정렬된 리스트 (sorted[l], ...,sorted[n])을 생성한다. */
{
   int j, k, t;
   j = m + 1 ;   /* 두번째 리스트에 대한 인덱스 */
   k = l;        /* 정렬된 리스트에 대한 인덱스 */
   while (l <= m && j <= n) {
         if (list[l].key <= list[j].key)
                  sorted[k++] = list[l++] ;
         else
                  sorted[k++] = list[j++] ;
   }
```

```
    if (l > m) /* sorted[k], ..., sorted[n] = list[j], ..., list[n] */
        for (t=j; t<=n; t++)
            sorted[k+t-j] = list[t] ;
    else  /* sorted[k], ..., sorted[n] = list[i], ..., list[m] */
        for (t=i; t<=m; t++)
            sorted[k+t-i] = list[t] ;
}
```

- merge 알고리즘의 분석
  - 정렬 리스트에 추가 되는 레코드 수는 n – i + 1
  - O(n – i +1)

- 반복 합병 정렬
  - 입력열을 길이가 1인 n개의 정렬 리스트로 간주
  - 크기가 2인 n/2개의 리스트를 얻기 위하여 쌍쌍이 합병
  - 이들을 같은 방식으로 합병하여 최종 정렬 리스트 얻음



```
void merge_pass (element list[], element sorted[], int n, int length)
{
/* 합병 정렬의 한 회전만을 수행한다. list로부터 인접한 서브화일 한 쌍을 합
   병하여 그 결과를 sorted에 넣는다. n은 list에 있는 원소수이고 length는
   서브화일의 길이이다. */

   int i, j ;
   for (i=0; i <= n - 2*length; i += 2 * length)
         merge(list, sorted, i, i+length-1, i+2*length-1) ;
   if (i+length < n)
   /*길이가 length인 서브화일과 length보다 작은 길이의 서브화일 */
         merge(list, sorted, i, i+length-1, n-1) ;
   else
         for (j=i; j<n; j++)
               sorted[j] = list[j] ;
}
```

```
void merge_sort (element list[], int n)
{
  int length = 1 ; /*합병될 서브화일 길이 */
  element extra[MAX_SIZE] ;
  while (length < n) {
      merge_pass (list, extra, n, length) ;
      length *= 2 ;
      merge_pass (extra, list, n, length) ;
      length *= 2 ;
  }
}
```

- Merge_sort 알고리즘의 분석
  - 1번째 회전에서는 크기가 1인 리스트를 합병하고 2번째 회전에서는 크기가 2인 리스트를 합병하고, I번째 회전에서는 크기가 2^(I-1) 인 리스트를 합병한다. 총 처리 과정 횟수는 O(log_2 n),
  - 두개의 정렬된 리스트에 대한 합병은 O(n)이므로, 알고리즘의 총 연산 시간은 O(n log n)

- 순환 합병 정렬
  - Recursive merge sort를 위한 레코드 구조

```
typedef struct {
  int key ;
  /* 다른 필드들 */
  int link ; /* 초기값은 –1, 합병이 되면서 다음 레코드의 첨
  자를 갖는다 */
} element ;
```

  - rmerge는 정렬리스트의 시작점을 가리키는 정수를 반환
  - listmerge가 두개의 정렬 리스트 fist와 second를 입력으로 받아 이 리스트들을 포함하는 새로운 정렬 리스틀 반환
  - start = rmerge(list, 0, n-1)로 호출

```
int rmerge(element list[], int lower, int upper)
/* 리스트 (list[lower],...,list[upper])를 정렬한다. 각 레코드내
   에 있는 링크 필드는 초기에 –1로 지정되어 있다. */
{
   int middle ;
   if (lower >= upper)
         return lower ;
   else {
         middle = (lower + upper) / 2 ;
         return listmerge(list, rmerge(list, lower, middle),
                                  rmerge(list, middle+1, upper) ;
   }
}
```

```
int listmerge(element list[], int first, int second)
{ /* first와 second가 가리키는 리스트들을 합병한다 */
   int start = n ;
   while (first != -1 && second != -1)
         if (list[first].key <= list[second].key) {
         /* first 리스트내에 있는 키값이 작다면 이 원소를 start에 연결시키고
         start는 first를 가리키도록 한다. */
                  list[start].link = first ;
                  start = first ;
                  first = list[first].link ;
         }
         else {
         /* second 리스트 키값이 작다면 이 원소를 일부 정렬된 리스트에
         연결시킨다. */
                  list[start].link = second ;
                  start = second ;
                  second = list[second].link ;
         }
   /* while의 끝 */ start는 while을 마치기전 마지막 원소
```

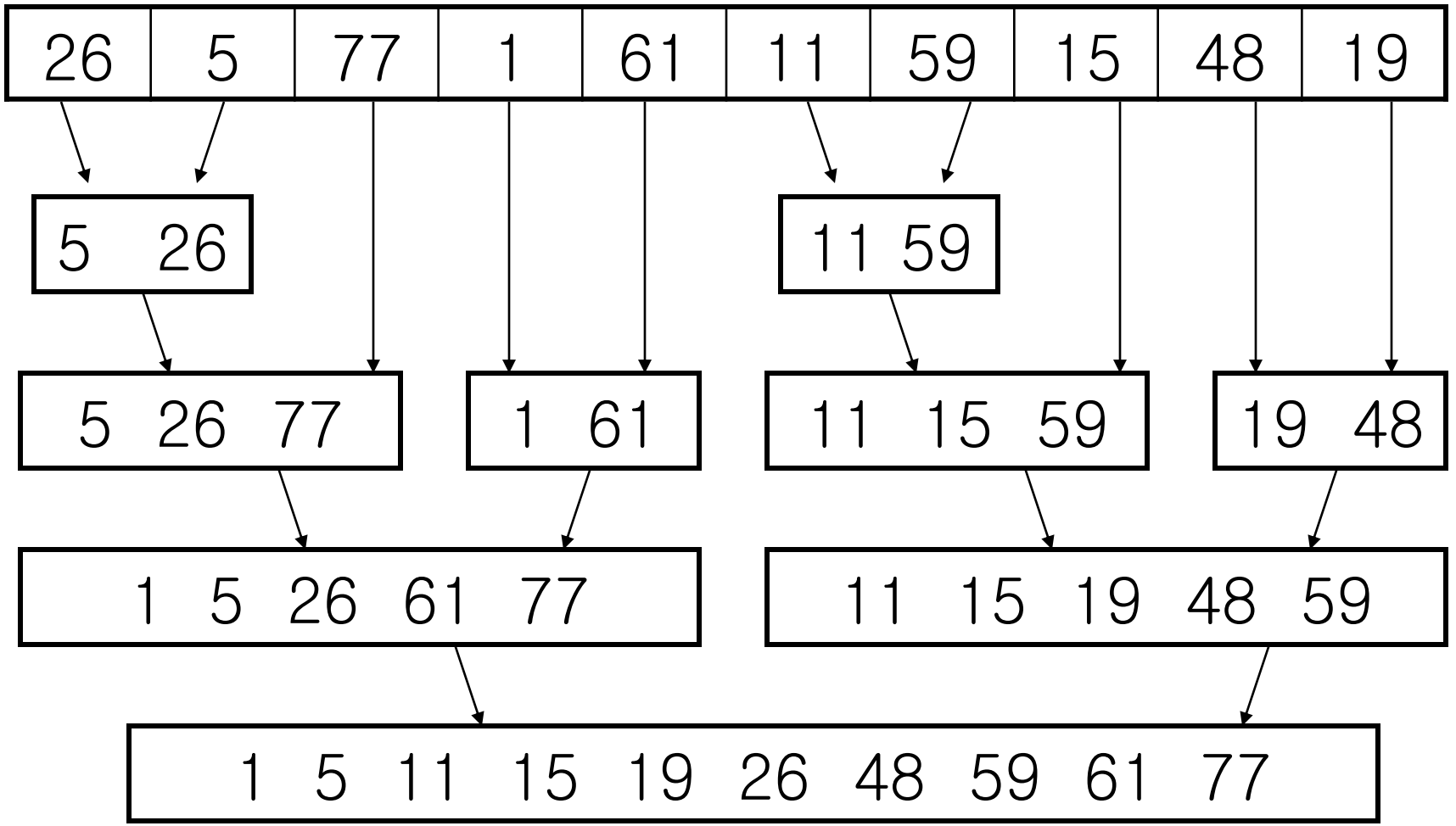
```
    /* 나머지를 이동 시킨다. */
    if ( first == -1) //first뒤에는 연결된 원소 없다.
            list[start].link = second ;
    else
            list[start].link = first ;

    return list[n].link ; /* 새로운 리스트의 start 점 */
}
```

rmerge 알고리즘의 분석

O(n log n)

- 예제 7.6
  - 각 순환 호출시 두 개의 서브리스트 [left, (left+right)/2]와 [(left+right)/2 + 1, right]로 나뉘어 각각 순환적으로 정렬
  - 그림 7.7의 반복 합병정렬과는 다름



- 그림 7.9: start = rmerge(list, 0, n-1)로 호출 후

start = 3

I	R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
Key	26	5	77	1	61	11	59	15	48	19
link	8	5	-1	1	2	7	4	9	6	0

히프 정렬

- 5장에서 최대 히프 구조를 이용
  - adjust() 함수를 사용하여 root 이하의 트리에서 n까지의 노드들을 최대 히프로 구성
  - 루트를 제외하고 왼쪽 및 오른쪽 서브트리 모두가 히프의 성질을 만족하는 이진 트리를 사용하여 전체 트리가 히프의 조건을 만족하도록 함

```
void adjust (element list[], int root, int n)
{ /* 히프를 구성하기 위하여 이진 트리를 조정하는 알고리즘 */
   int child, rootkey ;
   element temp ;
   temp = list[root] ;
   rootkey = list[root].key ;
   child = 2 * root ; /* 왼쪽 자식 */
   while (child <= n) { 많아야 트리의 깊이 만큼 수행
         if ( (child < n) && (list[child].key < list[child+1].key)) 값이 큰자식
                  child++ ;
         if (rootkey > list[child].key) /* 부모와 최대값의 자식과 비교 */
                  break ;
         else {
                  list[child/2] = list[child] ; /*부모로 이동 */
                  child *= 2 ;
         }
   }
   list[child/2] = temp ;
}
```

```
void heapsort (element list[], int n)
{ /* 배열에 대한 히프 정렬을 수행하는 알고리즘 */
   int k, j ;
   element temp ;

   for(k=n/2; k > 0; k--)  /* 자식이 있는 노드들에 대해서만 수행 */
         adjust(list, k, n) ;

   for(k=n - 1; k > 0; k--) {
         SWAP (list[1], list[k+1], temp) ; //가장 큰 값을 뒤쪽으로 옮김
         adjust(list, 1, k) ;
   }
}
```

- heapsort 알고리즘의 분석
  - 2^k-1 <= n < 2^k 라면 트리는 k개의 레벨
  - l 레벨의 노드수는 2^l-1
  - 첫번째 for 루프는 O(n) = sum {노드수 x 깊이}

  - 두번째 for 루프: 최대 깊이 ceil(log_2 (n+1)) 로 adjust를 n-1번 호출하므로 O(n log n)

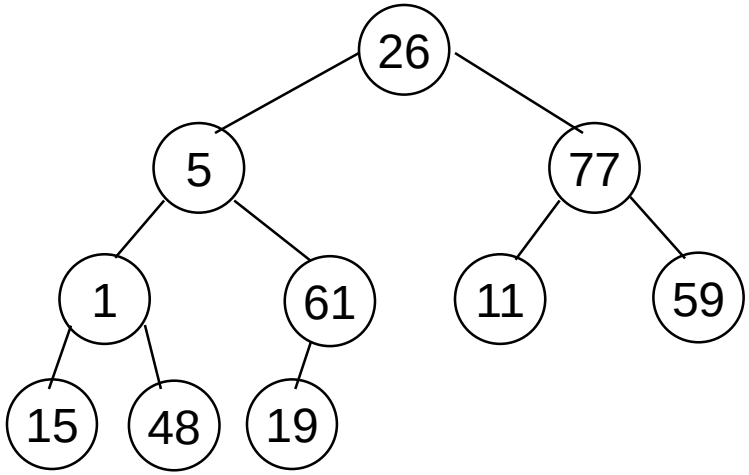


그림 7.11 이진트리로 표현된 배열

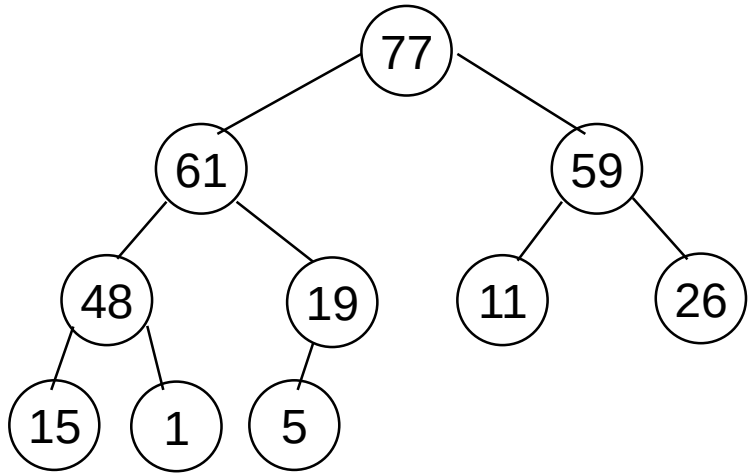


그림 7.12 heapsort의 첫번째 for 루프를 수행한 이후의 최대히프

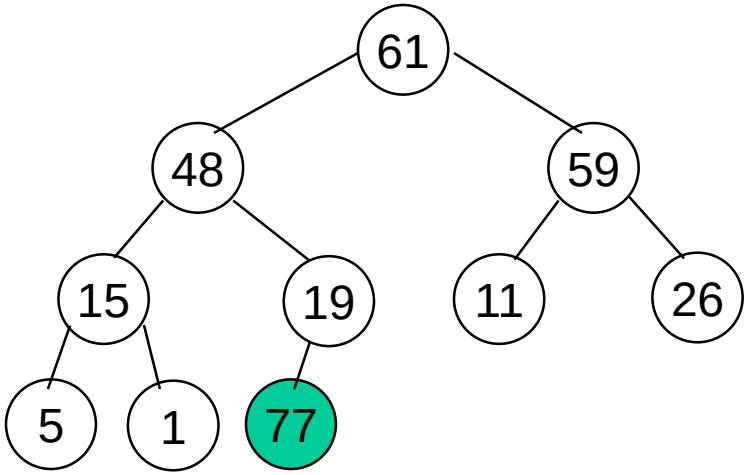


그림 7.13 heap 정렬의 예 (a)

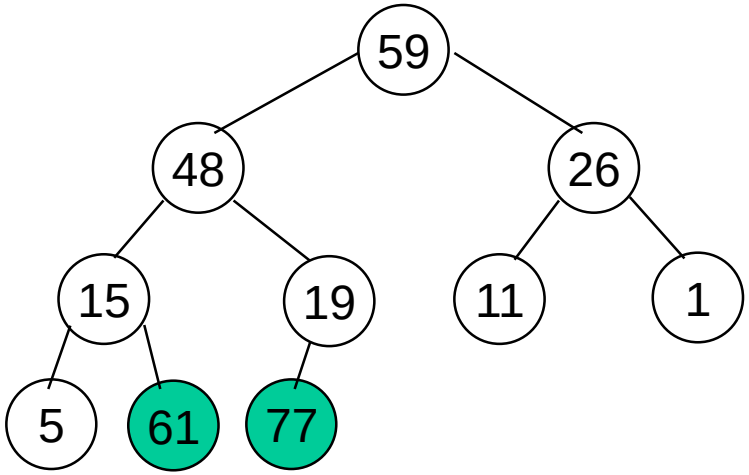


그림 7.13 heap 정렬의 예 (b)

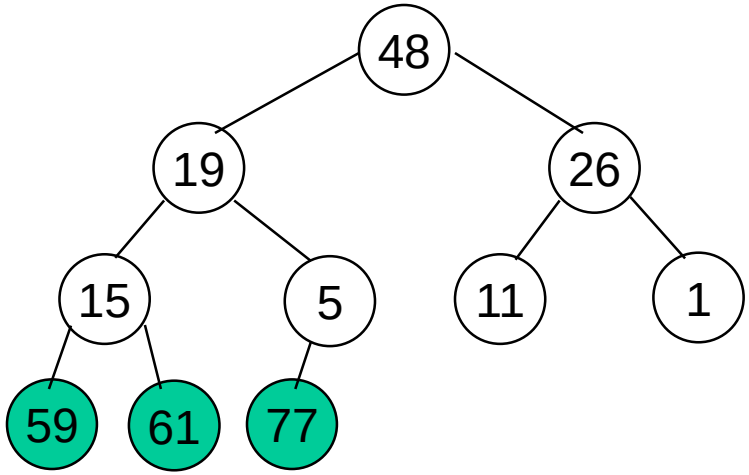


그림 7.13 heap 정렬의 예 (c)

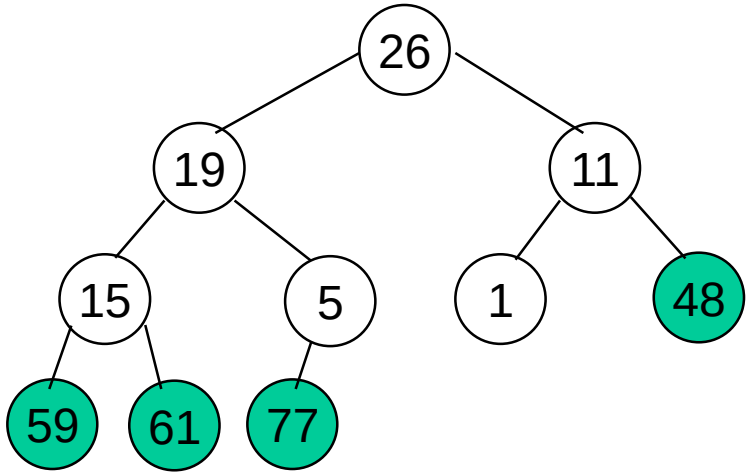


그림 7.13 heap 정렬의 예 (d)

기수 정렬

- 몇 개의 키 $K^0, K^1, \cdots K^{r-1}$ 를 갖는 레코드들을 정렬하는 문제
  - 최대 유효키 : K^0 , 최소 유효키 : K^{r-1}
- 최대 유효 숫자 (most significant digit ; MSD) 정렬
  - 최대 유효키에 따라 정렬한 후, 각 pile에 대해 그 다음 유효키로 정렬
- 최소 유효 숫자 (least significant digit ; LSD) 정렬
  - 최소 유효키에 따라 정렬한 후, 각 pile에 대해 그 다음 유효키로 정렬
- 예제 : p360
- LSD 방법은 subpile을 각각 정렬할 필요가 없으므로 오버헤드가 적다.

- 기수 r을 이용하여 정렬 키를 몇 개의 숫자로 분해
- 레코드가 d 튜플의 키 (x_1, x_2, ..., x_d) 를 갖는다.
  - 예) r=10을 사용할 경우 0 <= k <= 999는 d=3인 세 개의 키를 갖음
  - 예제 7.8 : 그림 7.16

front[i] : i번째 빈의 첫번째 레코드를 가리키는 포인터, 0<= i <= r
rear[i] : i번째 빈의 마지막 레코드를 가리키는 포인터, 0<= i <= r

```
#define MAX_DIGIT 3 /* 0에서 999까지의 수 */
#define RADIX_SIZE 10
typedef struct list_node *list_pointer ;
typedef struct list_node {
    int key[MAX_DIGIT] ;
    list_pointer link ;
} ;
```

```
list_pointer radix_sort (list_pointer ptr) /*연결 리스트를 이용한 기수 정렬*/
{    list_pointer front[RADIX_SIZE], rear[RADIX_SIZE] ;
     int i, j, digit ;
     for (i=MAX_DIGIT-1 ; i >= 0; i--) {
          for (j=0; j < RADIX_SIZE; j++)
               front[j] = rear[j] = NULL ;
          while (ptr) {
               digit = ptr→key[i] ;
               if (!front[digit]) //큐에 연결할 때 큐가 비어 있는 경우
                    front[digit] = ptr ;
               else //정상적인 큐 연결
                    rear[digit]→link = ptr ;
               rear[digit] = ptr ;
               ptr = ptr→link ;
          }
          /* 다음 수행을 위하여 연결 리스트를 재설정한다. */
          ptr = NULL ;
          for (j= RADIX_SIZE -1 ; j >= 0 ; j--)
               if (front[j]) {
                    rear[j]→link = ptr; ptr = front[j] ;
               }
     }
     return ptr ;
}
```

정렬 알고리즘 비교

알고리즘	최선	평균	최악
삽입	$O(n)$	$O(n^2)$	$O(n^2)$
선택, 버블	$O(n^2)$	$O(n^2)$	$O(n^2)$
셸	$O(n)$	$O(n^{1.5})$	$O(n^{1.5})$
퀵	$O(nlog_2^n)$	$O(nlog_2^n)$	$O(n^2)$
합병	$O(nlog_2^n)$	$O(nlog_2^n)$	$O(nlog_2^n)$
히프	$O(nlog_2^n)$	$O(nlog_2^n)$	$O(nlog_2^n)$
기수	$O(dn)$	$O(dn)$	$O(dn)$

성능 실험 결과	퀵 > 합병 > 히프 > 셸 >>> 삽입 > 선택 > 버블