Contents

- The sorting methods : insert, quick
  - Worst-case: **O(n^2)**
- Heapsort, Mergesort, and Quicksort all run in O($n$log$n$) best case running time
- Can we do any better? …

nlogn 보다 더 빠른 알고리즘은 없다

- If the only operations permitted on keys are comparisons and interchanges, then the best possible time: **O($n$log$n$)**
- **Decision tree** describes the sorting process
  - vertex : a key comparison
  - branch : the result

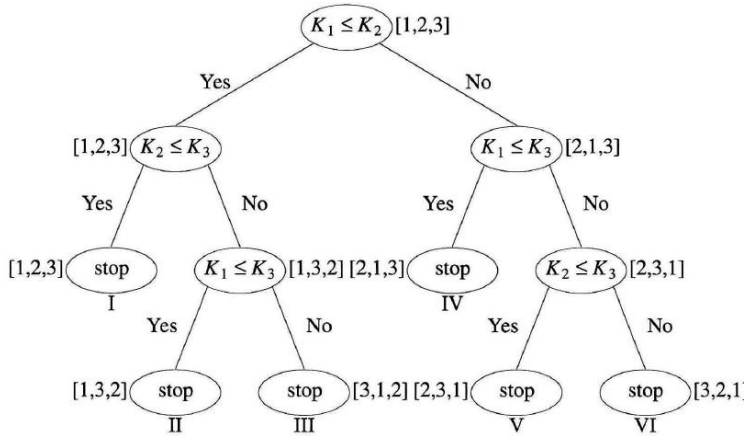


Figure 7.2: Decision tree for insertion sort

- Ex 7.4) Three records: [R_1, R_2, R_3]

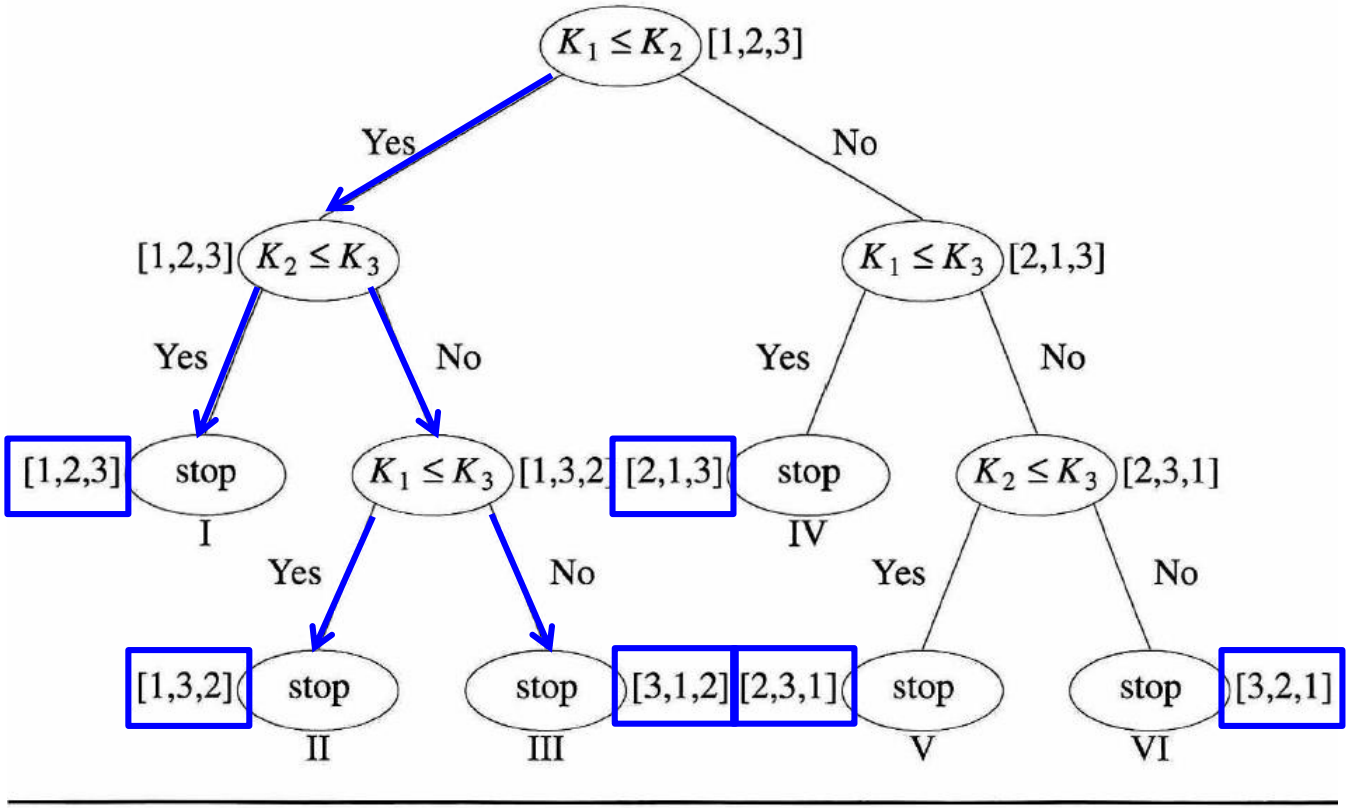


Figure 7.2: Decision tree for insertion sort

→ In decision tree, leaves represent …

- n records → $n!$ possible permutation
  - A path from the root to a leaf node
    - Represent one of $n!$ possibilities
  - The maximum depth of the tree: 3
    - Represent the number of comparisons

leaf	permutation	sample input key values that give the permutation
I	1 2 3	[7, 9, 10]
II	1 3 2	[7, 10, 9]
III	3 1 2	[9, 10, 7]
IV	2 1 3	[9, 7, 10]
V	2 3 1	[10, 7, 9]
VI	3 2 1	[10, 9, 7]

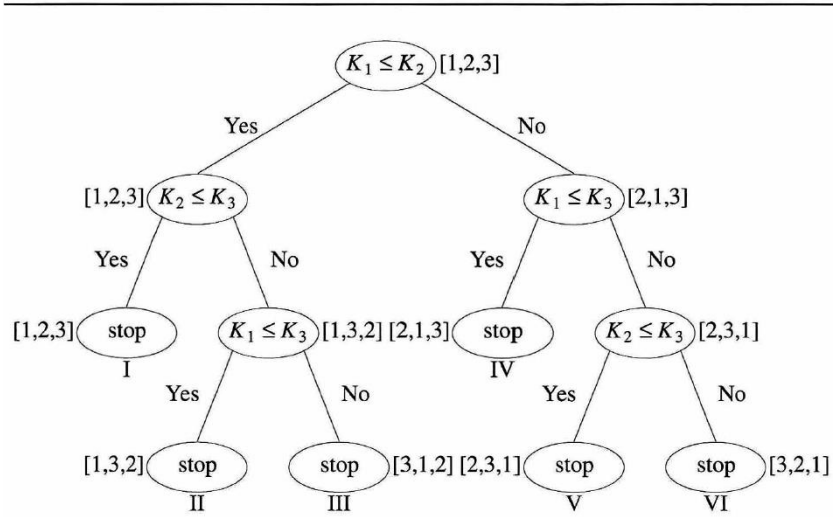


Figure 7.2: Decision tree for insertion sort

- Theorem 7.1: Any decision tree that sorts n distinct elements has a **height of at least $\log_2(n!) + 1$**
  - A decision tree is a BT; if its height is k → at most 2^{k-1} leaves
  - Every decision tree for sorting must have at least $n!$ leaves:

    $n! = 2^{k-1}$

    → $\log_2(n!) = k-1$

    → $k = \log_2(n!) + 1$

- Corollary : Any algorithm that sorts by *comparisons* only must have a **worst case computing time** of **$\Omega(n\log_2 n)$**
  - $n! = n(n-1)(n-2) \cdots (3)(2)(1) >= (n/2)^{n/2}$
  - $\log_2(n!) \geq (n/2)\log_2(n/2) = \Omega(n\log_2 n)$

7.5 MERGE SORT

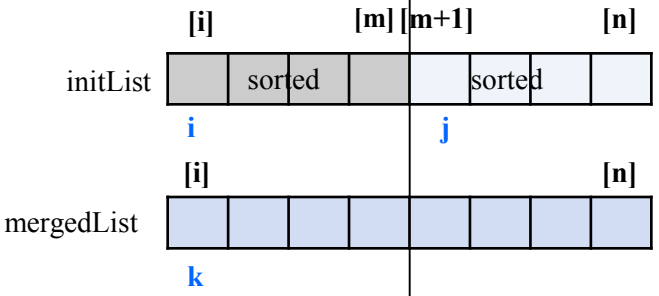
7.5.1 Merging

- Merge two sorted lists to get a single sorted list
  - initList[i:m] and initList[m+1:n] into mergedList[i:n]
  - Ex)

initLst: 4, 5, 7, 11, 2, 3, 6, 8, 14

mergedLst:

```
void merge(element initList[], element mergedList[], int i, int m, int n)
{ /* the sorted lists initList[i:m] and initList[m+l:n] are merged to obtain
the sorted list mergedList[i:n] */
    int j,k,t;
    j = m+1;            /* index for the second sublist */
    k = i;              /* index for the merged list */

    while( i <= m && j <= n) {
        if (initList[i].key <= initList[j].key)
            mergeList[k++] = initList[i++];
        else
            mergeList[k++] = initList[j++];
    }
    if (i > m)          /* mergedList[k:n] = initList[j:n]*/
        for(t = j; t <= n; t++)
            mergeList[t] = initList[t];
    else                /* mergedList[k:n] = initList[i:m] */
        for(t = i; t <= m; t++)
            mergeList[k+t-i] = initList[t];
}
```



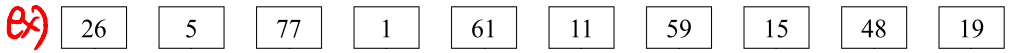
Stable

Time complexity:
O(...)

Program 7.7: Merging two sorted lists

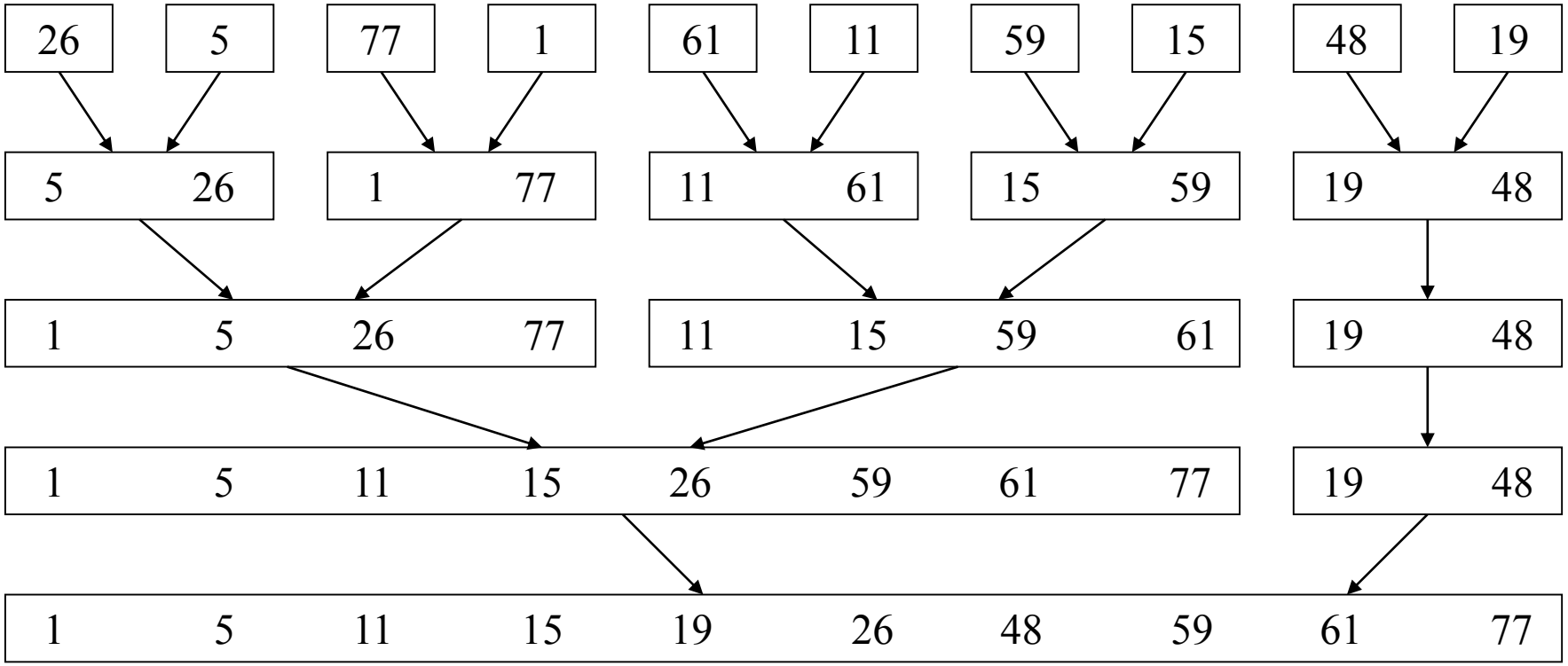
7.5.2 Iterative Merge Sort

- Assume:
  Input sequence has ***n* sorted lists**, each of length 1



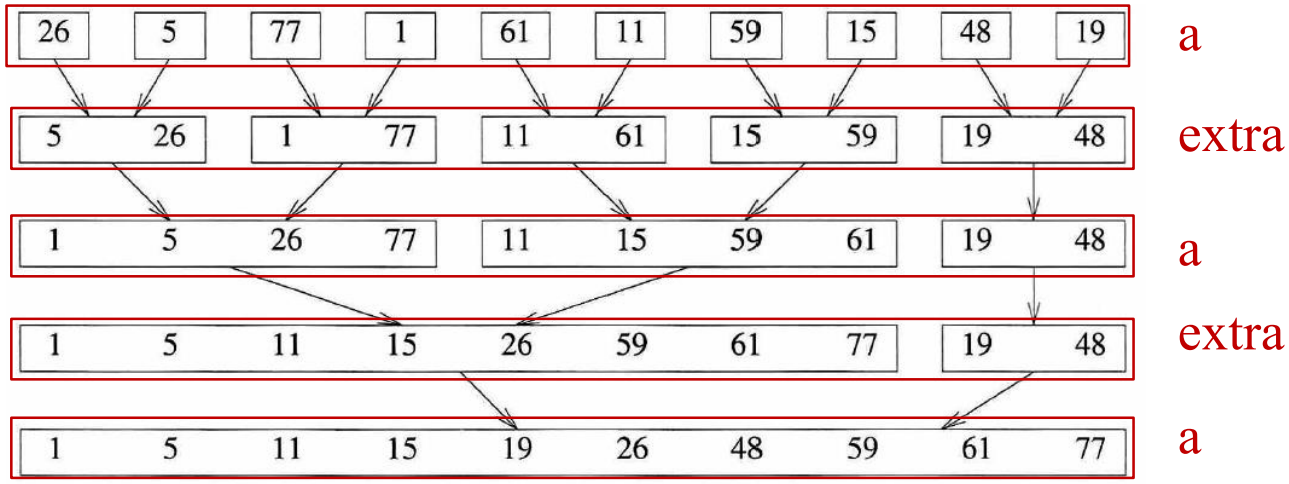
- Merge Sort
  1) Merge these lists pairwise to obtain $n/2$ lists, each of size 2
  2) Merge the $n/2$ lists pairwise to obtain $n/4$ sublists
  3) Continue until only one sublist is left

- Ex 7.5)
  input list: (26, 5, 77, 1, 61, 11, 59, 15, 48, 19)



```
void mergeSort(element a[], int n) {
     int s = 1;        /* current segment size */
     element extra[MAX_SIZE];

     while (s<n)       {
          mergePass(a, extra, n, s);
          s *= 2;
          mergePass(extra , a, n , s);
          s *= 2;
     }
}
```

Program 7.9: Merge sort



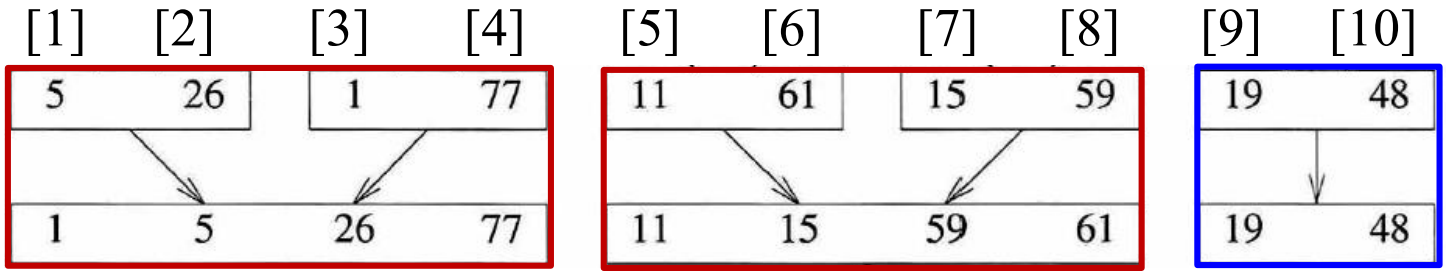
mergePass(a, extra, 10, 2);

```
void mergePass(element initList[], element mergedList[], int n, int s)
{/* n: the number of elements in the list,
     s: the size of each sorted segment */

      for(i = 1;  i <= n-2*s+1; i+= 2*s)
            merge(initList, mergedList, i, i+s-1, i+2*s-1);

      if((i+s-1)<n) merge(initList, mergedList, i, i+s-1, n);
      else
            for(j=i; j <= n; j++)
                  mergedList[j] = initList[j];
}
```

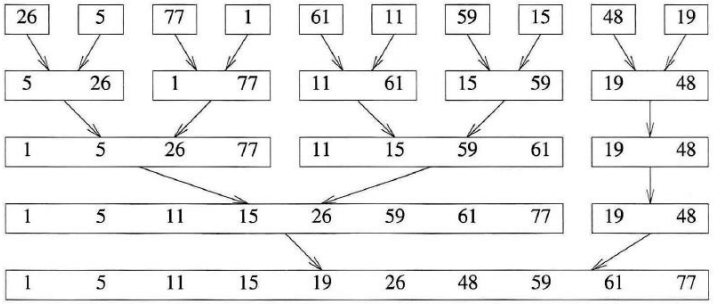
```
i=1
merge(initList, mergedList, 1, 2, 4);
i=5
merge(initList, mergedList, 5, 6, 8);
i=9
```

Program 7.8: A merge pass

< case: n=10, s=2 >



- Analysis of *mergeSort*
  - Sorted segment size: 1, 2, 4, 8, …
  - # of merge passes: $\lceil \log_2 n \rceil$
  - Each merge pass: O(n)
  - Total computing time: O(n log n)
  - Need O(n) additional space for the merge
  - Stable sorting



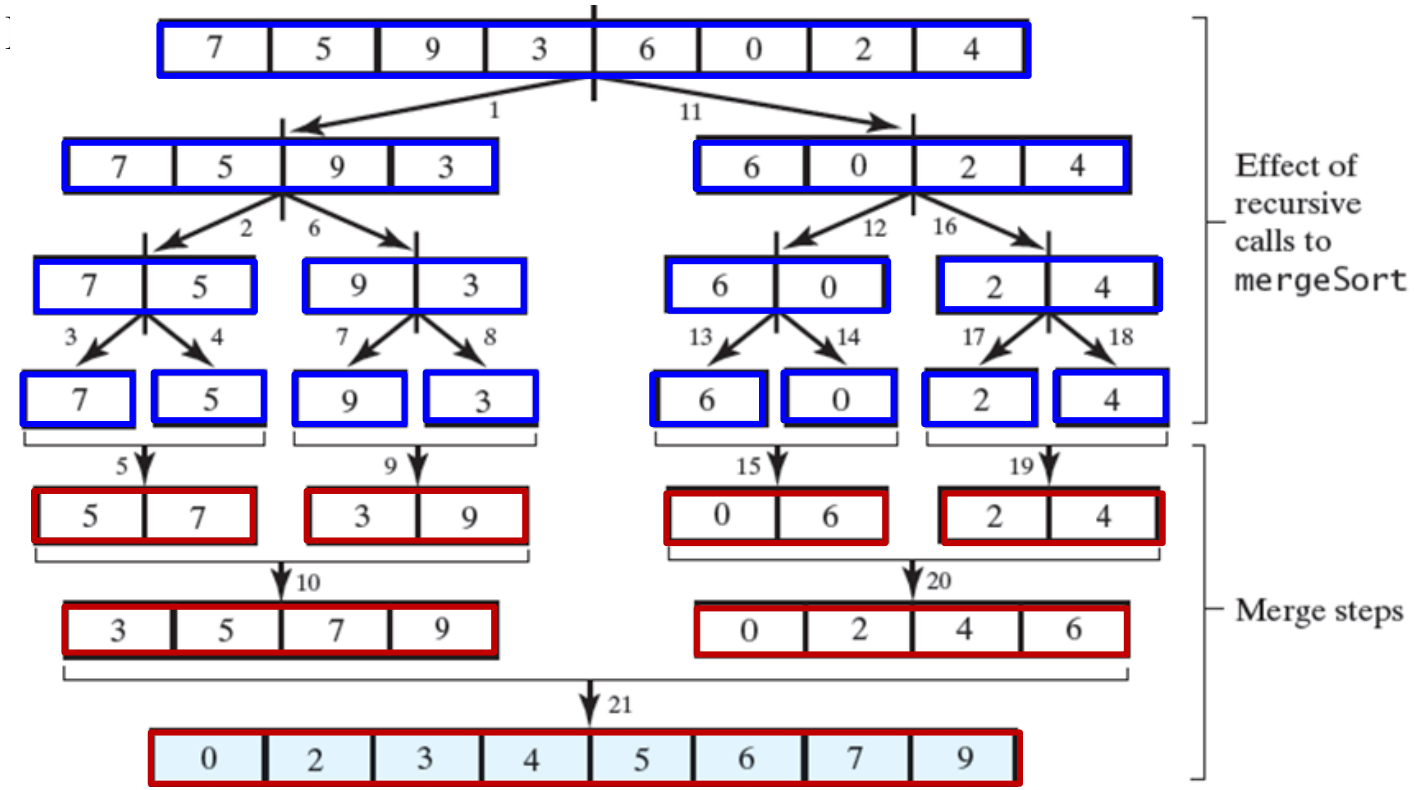
7.5.3 Recursive Merge Sort

1) Recursively divide the input list into smaller sublists
   - Until the sublists are trivially sorted
2) Merge the sublists
   - while returning up the call chain

```
int rmergeSort(element a[], int link[], int left, int right)
{/ * a[left:right] is to be sorted, link[i] is initially 0 for all i, returns the index of the first
element in the sorted chain */
    if (left >= right) return left;
    int mid = (left + right)/2;
    return listMerge(a, link, rmergeSort(a, link, left, mid),
                              rmergeSort(a, link, mid+1, right));
}
```

Program 7.10: Recursive merge sort

```
int rmergeSort(element a[], int link[], int left, int right)
{/ * a[left:right] is to be sorted, link[i] is initially 0 for all i, returns the index of the first
element in the sorted chain */
    if (left >= right) return left;
    int mid = (left + right)/2;
    return listMerge(a, link, rmergeSort(a, link, left, mid),
                                                /* sort left half */
                              rmergeSort(a, link, mid+1, right));
                                                / * sort right half */
}
```



```
int listMerge(element a[], int link[], int start1, int start2)
{/ *sorted chains beginning at startl and start2, respectively, are merged;
    link[0] is used as a temporary header; returns start of merged chain */
   int last1, last2, lastResult=0;
   for(last1 =start1, last2 = start2; last1 && last2; )
       if( a[last1] <= a[last2] ) {
            link[lastResult] = last1;
            lastResult = last1; last1 = link[last1];
       }
       else {
            link[lastResult] = last2;
            lastResult = last2; last2 = link[last2];
       }
    /* attach remaining records to result chain */
    if(last1 == 0)      link[lastResult] = last2;
    else                link[lastResult] = last1;
    return link[0];
}
```

Program 7.11: Merging sorted chains

```
int listMerge(element a[], int link[], int start1, int start2)
{/ *sorted chains beginning at startl and start2, respectively, are merged;
    link[0] is used as a temporary header; returns start of merged chain */
   int last1, last2, lastResult=0;
   for(last1 =start1, last2 = start2; last1 && last2; )
       if( a[last1] <= a[last2] ) {
            link[lastResult] = last1;
            lastResult = last1; last1 = link[last1];
       }
       else {
            link[lastResult] = last2;
            lastResult = last2; last2 = link[last2];
       }
    /* attach remaining records to result chain */
    if(last1 == 0)        link[lastResult] = last2;
    else                  link[lastResult] = last1;
    return link[0];
}
```

※ chains : to eliminate record copying

	[0]	[1]								[n]
link	0	0	0	0	0	0	0	0	0	0
a	-	26	5	77	1	61	11	59	15	48	19

link[0] = 2

lastresult = 2, last2 = link[2] = 0

Program 7.11: Merging sorted chains

```
int listMerge(element a[], int link[], int start1, int start2)
{/ *sorted chains beginning at startl and start2, respectively, are merged;
    link[0] is used as a temporary header; returns start of merged chain */
   int last1, last2, lastResult=0;
   for(last1 =start1, last2 = start2; last1 && last2; )
      if( a[last1] <= a[last2] ) {
         link[lastResult] = last1;
         lastResult = last1; last1 = link[last1];
      }
      else {
         link[lastResult] = last2;
         lastResult = last2; last2 = link[last2];
      }
   /* attach remaining records to result chain */
   if(last1 == 0)        link[lastResult] = last2;
   else                  link[lastResult] = last1;
   return link[0];
}
```

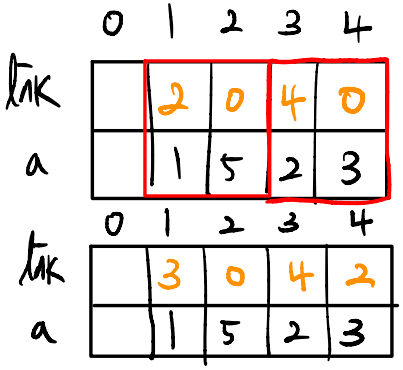
※ chains : to eliminate record copying

	[0]	[1]									[n]
link	0	0	0	0	0	0	0	0	0	0	0
a	-	26	5	77	1	61	11	59	15	48	19

	[0]	[1]	[2]
link	2	0	1
a	-	26	5

link[2] = 1

Program 7.11: Merging sorted chains



1은 3번째 인덱스 2를 가리킴
2는 4번째 인덱스 3을 가리킴

listMerge(a, link, 1, 2);

start1	start2	lastResult	last1	last2
1	2	2	1	0

- Natural Merge Sort
  - Take into account the prevailing order within the input list
    - Need an *initial pass* over the data to determine the sublists of records that are in order 각각의 sublist는 정렬되어 있다고 가정
  - Ex) Input list: (26, 5, 77, 1, 61, 11, 59, 15, 48, 19)

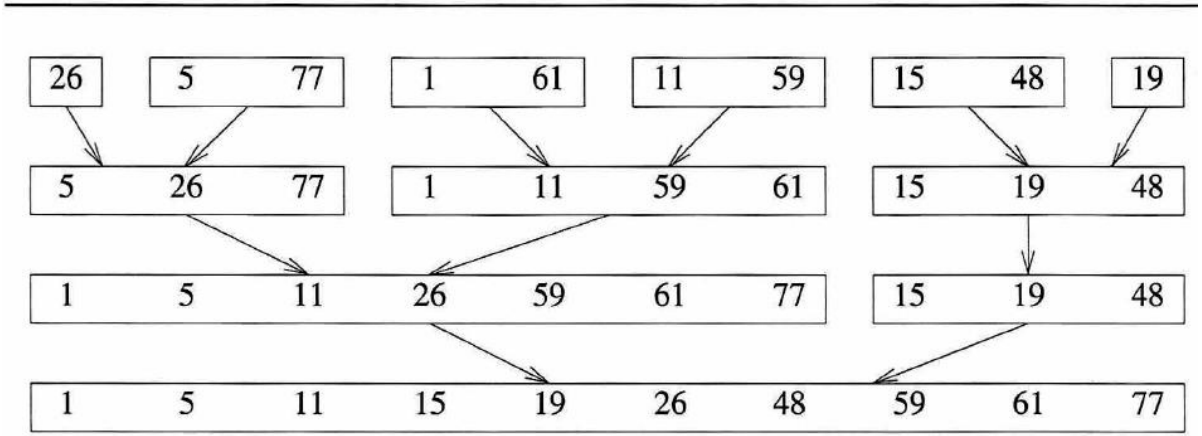


Figure 7.6: Natural merge sort

7.6 HEAP SORT

- Merge sort
  - Require additional storage proportional to the number of records to be sorted

- Heap sort
  - Require only *a fixed* amount of additional storage → merge는 데이터 개수에 비례 공간이 필요하지, heap 정렬은 그렇지 않다
  - Worst/average time complexity: $O(n\log n)$
    - Slightly slower than merge sort
  - Utilize the **max-heap** structure

- Ex 7.7)
  Input list: (26, 5, 77, 1, 61, 11, 59, 15, 48, 19)

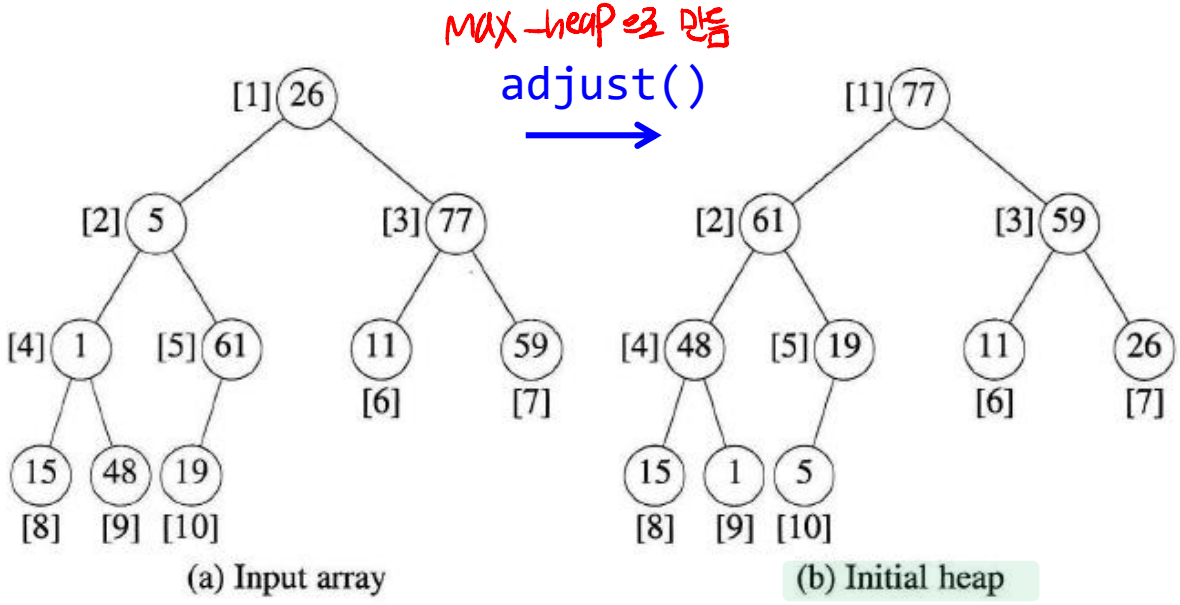


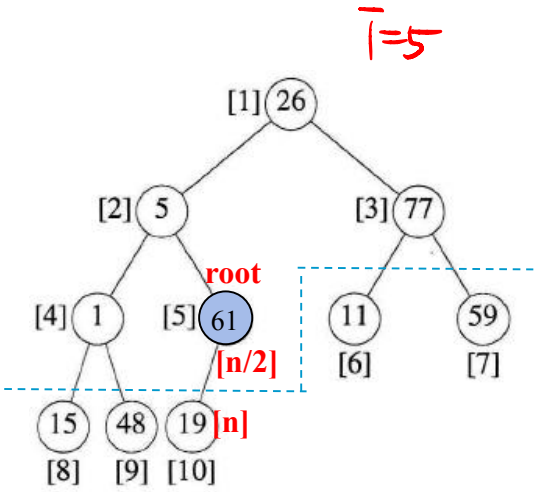
Figure 7.7: Array interpreted as a binary tree

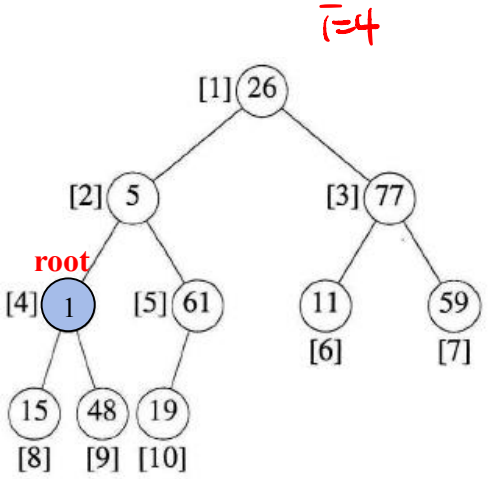
```
void heapSort(element a[], int n)
{/* perform a heap sort on a[1:n] */
   int i,j;
   element temp;

   for (i = n/2; i > 0; i--)
      adjust(a,i,n);
   for (i = n-1; i > 0; i--) {
      SWAP(a[1],a[i+1],temp);
      adjust(a,1,i);
   }
}
```

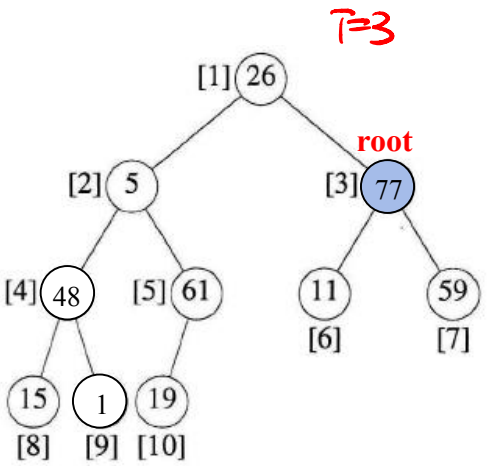
Program 7.13: Heap sort

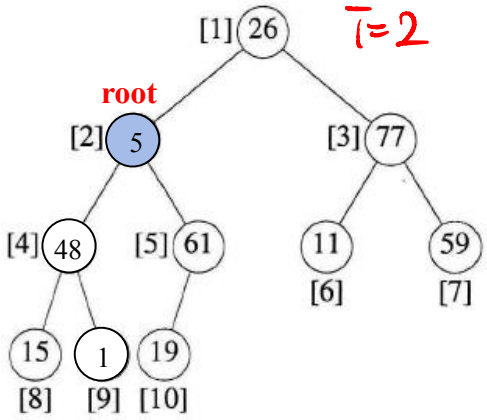
```
for (i = n/2; i > 0; i--)
   adjust(a,i,n);
```

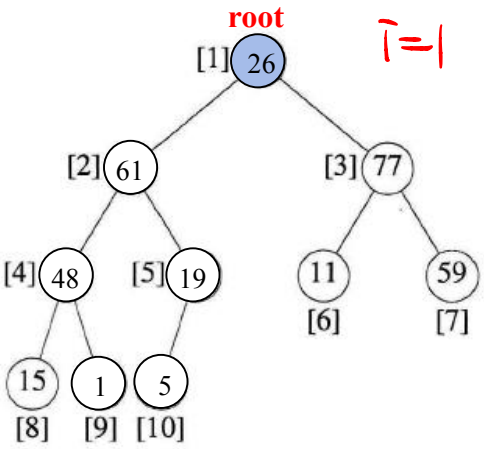
bottom-up

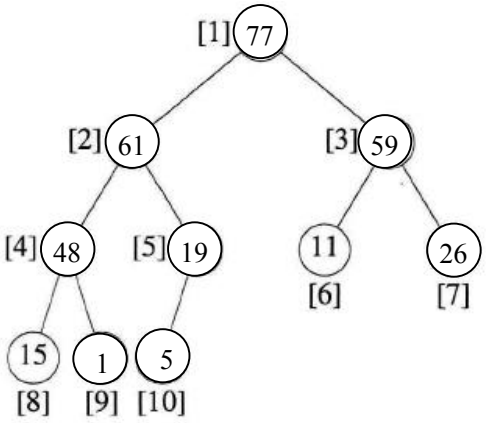




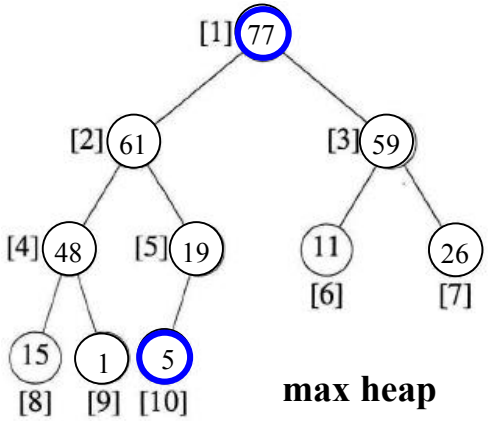








max heap



max heap

Sorting:

1) Swap the first and last records
2) Decrement the heap size and re-adjust the heap

```
void heapSort(element a[], int n)
{/* perform a heap sort on a[1:n] */
   int i,j;
   element temp;

   for (i = n/2; i > 0; i--)
      adjust(a,i,n);
   for (i = n-1; i > 0; i--) {
      SWAP(a[1],a[i+1],temp);
      adjust(a,1,i);
   }
}
```

top-down

Program 7.13: Heap sort

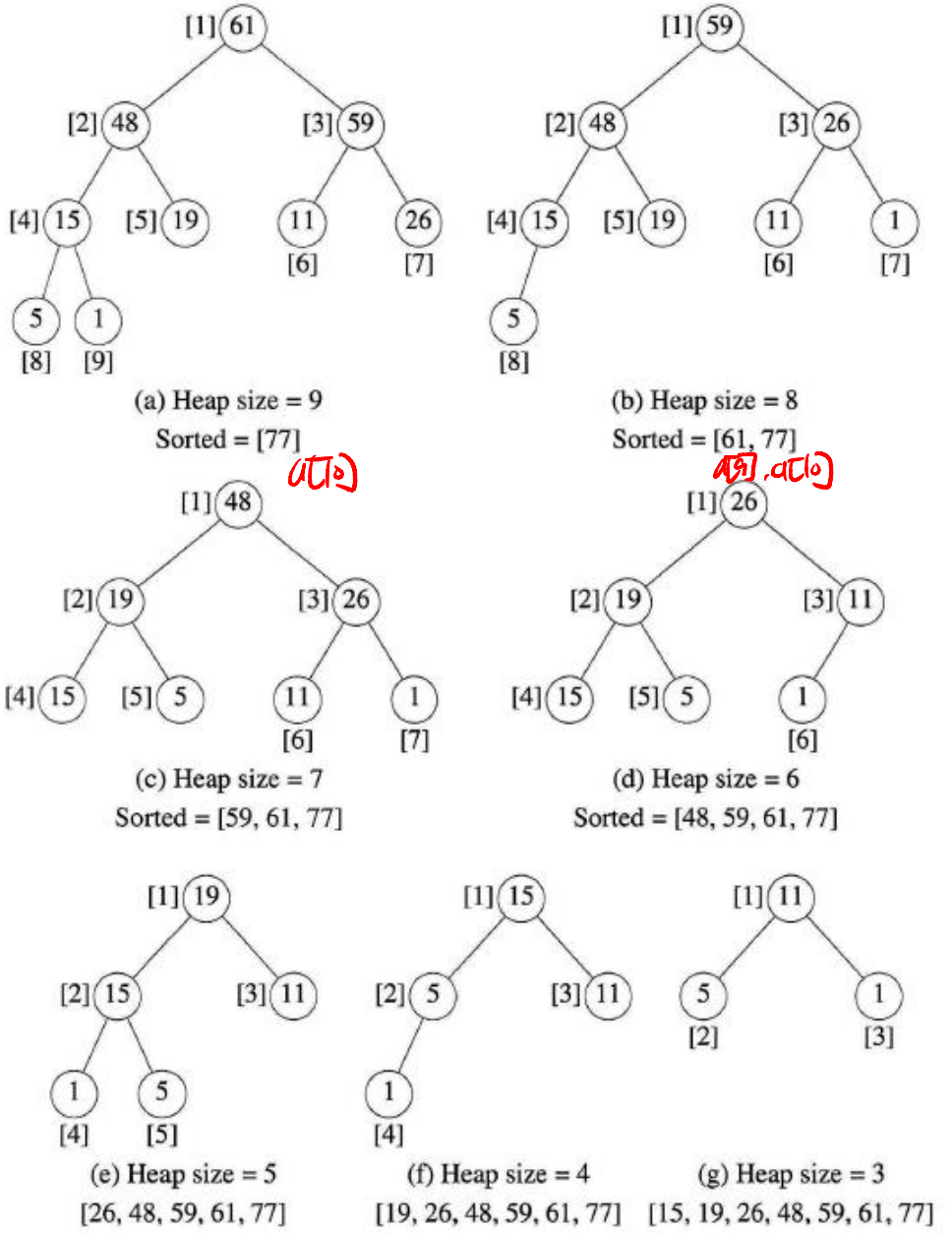
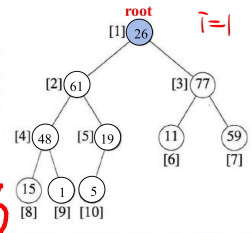


Figure 7.8: Heap sort example

```
void adjust(element list[], int root, int n)
{
   int child,rootkey;
   element temp;
   temp = list[root];
   rootkey = list[root].key;
   child = 2 * root;        /* left child */
   while (child <= n) {
      if ((child < n) &&
      (list[child].key < list[child+1].key))
         child++;
      if (rootkey > list[child].key)
      /* compare root and max. child */
         break;
      else {                  /* move to parent */
         list[child / 2] = list[child];
         child *= 2;
      }
   }
   list[child/2] = temp;
}
```

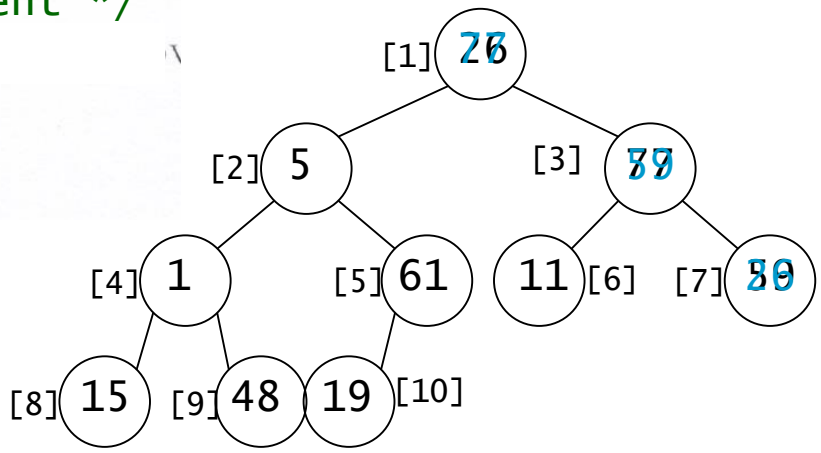


root = 1

n = 10

rootkey = 26

child = 4



```
void heapsort(element list[], int n)
/* perform a heapsort on the array */
{
   int i,j;
   element temp;

   for (i = n/2; i > 0; i--)
      adjust(list,i,n);
   for (i = n-1; i > 0; i--) {
      SWAP(list[1],list[i+1],temp);
      adjust(list,1,i);
   }
}
```

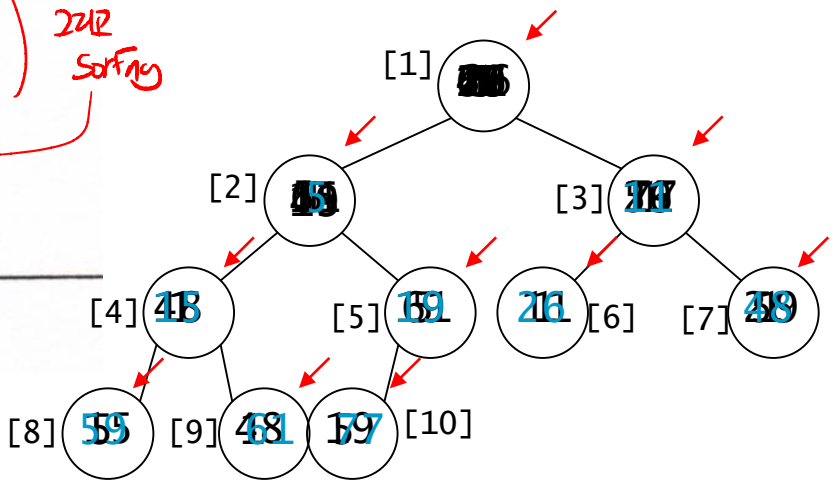
bottom-up

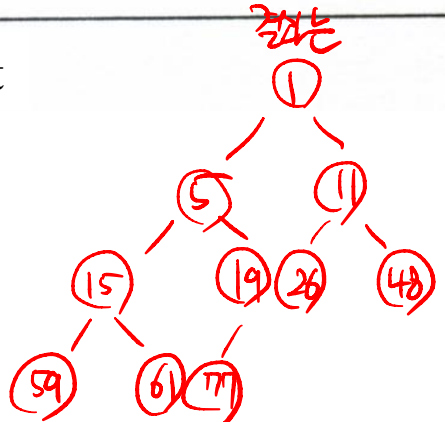
top-down

n = 10

i =

max heap





Program 7.13: Heap sort

Contents