제7장 Sorting

경북대학교 임경식 교수

7.1 Motivation

We have seen two important uses of sorting: (1) as an aid in searching and (2) as a means for matching entries in lists. Sorting also finds application in the solution of many other more complex problems from areas such as optimization, graph theory and job scheduling. Consequently, the problem of sorting has great relevance in the study of computing. Unfortunately, no one sorting method is the best for all applications. We shall therefore study several methods, indicating when one is superior to the others.

We characterize sorting methods into two broad categories: **(1)** internal methods (i.e., methods to be used when the list to be sorted is small enough so that the entire sort can be carried out in main memory) and (2) external methods (i.e., methods to be used on larger lists). The following internal sorting methods will be developed: insertion sort, quick sort, merge sort, heap sort, and radix sort.

7.2 Insertion sort

In insertion sort, we begin with the ordered sequence *a* [1] and successively insert the records *a* [2], *a* [3], · · ·,*a [n].* Since each insertion leaves the resultant sequence ordered, the list with *n* records can be ordered making *n* - **1** insertions. The basic step in this method is to insert a new record into a sorted sequence of *i* records in such a way that the resulting sequence of size *i* + **1** is also ordered.

j	[1]	[2]	[3]	[4]	[5]
–	**5**	4	3	2	1
2	**4**	**5**	3	2	1
3	**3**	**4**	**5**	2	1
4	**2**	**3**	**4**	**5**	1
5	**1**	**2**	**3**	**4**	**5**

j	[1]	[2]	[3]	[4]	[5]
–	**2**	3	4	5	1
2	**2**	**3**	4	5	1
3	**2**	**3**	**4**	5	1
4	**2**	**3**	**4**	**5**	1
5	**1**	**2**	**3**	**4**	**5**

7.2 Insertion sort

```
void insertionSort(element a[], int n)
{/* sort a[1:n] into nondecreasing order */
    int j;
    for (j = 2; j <= n; j++) {
        element temp = a[j];
        insert(temp, a, j-1);
    }
}
void insert(element e, element a[], int i)
{/* insert e into the ordered list a[1:i] such that the
    resulting list a[1:i+1] is also ordered, the array a
    must have space allocated for at least i+2 elements */
    a[0] = e;
    while (e.key < a[i].key)
    {
        a[i+1] = a[i];
        i--;
    }
    a[i+1] = e;
}
```

j	[1]	[2]	[3]	[4]	[5]
–	**5**	4	3	2	1
2	**4**	**5**	3	2	1
3	**3**	**4**	**5**	2	1
4	**2**	**3**	**4**	**5**	1
5	**1**	**2**	**3**	**4**	**5**

7.2 Insertion sort

```
void insertionSort(element a[], int n)
{/* sort a[1:n] into nondecreasing order */
    int j;
    for (j = 2; j <= n; j++) {
        element temp = a[j];
        insert(temp, a, j-1);
    }
}
```

```
void insert(element e, element a[], int i)
{/* insert e into the ordered list a[1:i] such that the
    resulting list a[1:i+1] is also ordered, the array a
    must have space allocated for at least i+2 elements */
    a[0] = e;
    while (e.key < a[i].key)
    {
        a[i+1] = a[i];
        i--;
    }
    a[i+1] = e;
}
```

Analysis of *insertionSort:* In the worst case *insert* (e, a, i) makes $i + 1$ comparisons before making the insertion. Hence the complexity of *Insert* is $O(i)$. Function *insertionSort* invokes *insert* for $i = j - 1 = 1, 2, \cdots, n - 1$. So, the complexity of *insertionSort* is

$$O(\sum_{i=1}^{n-1} (i+1)) = O(n^2).$$

We can also obtain an estimate of the computing time of insertion sort based on the relative disorder in the input list. Record R_i is *left out of order* (LOO) iff $R_i < \max_{1 \le j < i}\{R_j\}$. The insertion step has to be carried out only for those records that are LOO. If k is the number of LOO records, the computing time is $O((k + 1)n) = O(kn)$. We can show that the average time for *insertionSort* is $O(n^2)$ as well. □

7.3 Quick sort

We now turn our attention to a sorting scheme with very good average behavior. The quick sort scheme developed by C. A. R. Hoare has the best average behavior among the sorting methods we shall be studying. In quick sort, we select a pivot record from among the records to be sorted. Next, the records to be sorted are reordered so that the keys of records to the left of the pivot are less than or equal to that of the pivot and those of the records to the right of the pivot are greater than or equal to that of the pivot. Finally, the records to the left of the pivot and those to its right are sorted independently (using the quick sort method recursively).

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	*left*	*right*
[26	5	37	1	61	11	59	15	48	19]	1	10
[11	5	19	1	15]	26	[59	61	48	37]	1	5
[1	5]	11	[19	15]	26	[59	61	48	37	1	2
1	5	11	[19	15]	26	[59	61	48	37]	4	5
1	5	11	15	19	26	[59	61	48	37]	7	10
1	5	11	15	19	26	[48	37]	59	[61]	7	8
1	5	11	15	19	26	37	48	59	[61]	10	10
1	5	11	15	19	26	37	48	59	61		

7.3 Quick sort

```
void quickSort(element a[], int left, int right)
{/* sort a[left:right] into nondecreasing order
    on the key field; a[left].key is arbitrarily
    chosen as the pivot key;  it is assumed that
    a[left].key <= a[right+1].key */
  int pivot,i,j;
  element temp;
  if (left < right) {
     i = left; j = right + 1;
     pivot = a[left].key;
     do {/* search for keys from the left and right
            sublists, swapping out-of-order elements until
            the left and right boundaries cross or meet */
        do i++; while (a[i].key < pivot);
        do j--; while (a[j].key > pivot);
        if (i < j) SWAP(a[i],a[j],temp);
     } while (i < j);
     SWAP(a[left],a[j],temp);
     quickSort(a,left,j-1);
     quickSort(a,j+1,right);
  }
}
```

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	*left*	*right*
[26	5	37	1	61	11	59	15	48	19]	1	10
[11	5	19	1	15]	26	[59	61	48	37]	1	5
[1	5]	11	[19	15]	26	[59	61	48	37	1	2
1	5	11	[19	15]	26	[59	61	48	37]	4	5
1	5	11	15	19	26	[59	61	48	37]	7	10
1	5	11	15	19	26	[48	37]	59	[61]	7	8
1	5	11	15	19	26	37	48	59	[61]	10	10
1	5	11	15	19	26	37	48	59	61		

7.3 Quick sort

Analysis of *quickSort:* The worst-case behavior of *quickSort* is examined in Exercise 2 and shown to be $O(n^2)$. However, if we are lucky, then each time a record is correctly positioned, the sublist to its left will be of the same size as that to its right. This would leave us with the sorting of two sublists, each of size roughly $n/2$. The time required to position a record in a list of size n is $O(n)$. If $T(n)$ is the time taken to sort a list of n records, then when the list splits roughly into two equal parts each time a record is positioned correctly, we have

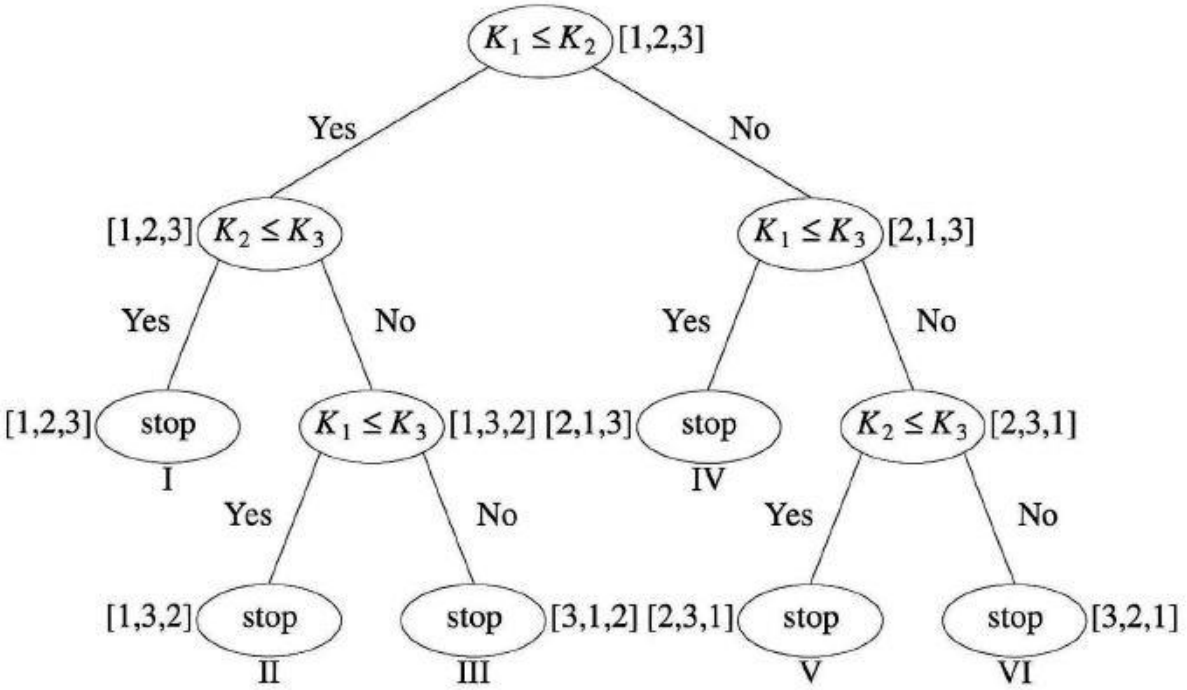
$$
\begin{aligned}
T(n) &\le cn + 2T(n/2), \text{ for some constant } c \\
&\le cn + 2(cn/2 + 2T(n/4)) \\
&\le 2cn + 4T(n/4) \\
&\;\;\vdots \\
&\le cn \log_2 n + nT(1) = O(n \log n)
\end{aligned}
$$

Lemma 7.1 shows that the average computing time for function *quickSort* is $O(n \log n)$. Moreover, experimental results show that as far as average computing time is concerned, Quick sort is the best of the internal sorting methods we shall be studying.

7.4 How fast can we sort?

Both of the sorting methods we have seen so far have a worst-case behavior of $O(n^2)$. It is natural at this point to ask the question, What is the best computing time for sorting that we can hope for? The theorem we shall prove shows that if we restrict our question to sorting algorithms in which the only operations permitted on keys are comparisons and interchanges, then $O(n \log n)$ is the best possible time.

The method we use is to consider a tree that describes the sorting process. Each vertex of the tree represents a key comparison, and the branches indicate the result. Such a tree is called a *decision tree*. A path through a decision tree represents a sequence of computations that an algorithm could produce.



7.4 How fast can we sort?

Theorem 7.1: Any decision tree that sorts n distinct elements has a height of at least $\log_2(n!) + 1$.

Proof: When sorting n elements, there are $n!$ different possible results. Thus, every decision tree for sorting must have at least $n!$ leaves. But a decision tree is also a binary tree, which can have at most 2^{k-1} leaves if its height is k. Therefore, the height must be at least $\log_2 n! + 1$. □

Corollary: Any algorithm that sorts only by comparisons must have a worst-case computing time of $\Omega(n \log n)$.

Proof: We must show that for every decision tree with $n!$ leaves, there is a path of length $cn\log_2 n$, where c is a constant. By the theorem, there is a path of length $\log_2 n!$. Now

$$n! = n(n-1)(n-2) \cdots (3)(2)(1) \geq (n/2)^{n/2}$$

So, $\log_2 n! \geq (n/2)\log_2(n/2) = \Omega(n \log n)$. □

7.5 Merge sort

7.5.2 Iterative Merge Sort

This version of merge sort begins by interpreting the input list as comprised of n sorted sublists, each of size 1. In the first merge pass, these sublists are merged by pairs to obtain $n/2$ sublists, each of size 2 (if n is odd, then one sublist is of size 1). In the second merge pass, these $n/2$ sublists are then merged by pairs to obtain $n/4$ sublists. Each merge pass reduces the number of sublists by half. Merge passes are continued until we are left with only one sublist. The example below illustrates the process.

Example 7.5: The input list is (26, 5, 77, 1, 61, 11, 59, 15, 48, 19). The tree of Figure 7.4 illustrates the sublists being merged at each pass. □

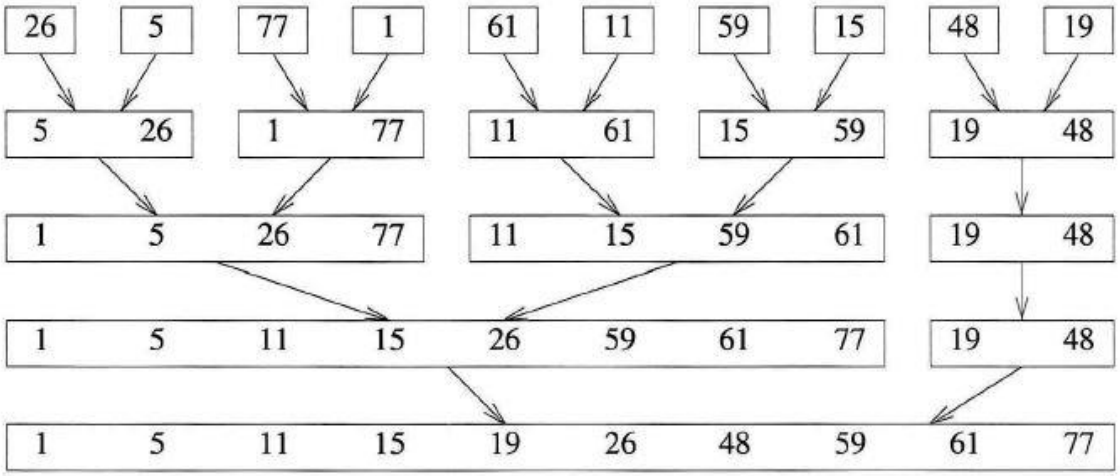


Figure 7.4: Merge tree

7.5 Merge sort

7.5.3 Recursive Merge Sort

In the recursive formulation we divide the list to be sorted into two roughly equal parts called the left and the right sublists. These sublists are sorted recursively, and the sorted sublists are merged.

Example 7.6: The input list (26, 5, 77, 1, 61, 11, 59, 15, 49, 19) is to be sorted using the recursive formulation of merge sort. If the sublist from *left* to *right* is currently to be sorted, then its two sublists are indexed from *left* to $\lfloor (left + right)/2 \rfloor$ and from $\lfloor (left + right)/2 \rfloor + 1$ to *right*. The sublist partitioning that takes place is described by the binary tree of Figure 7.5. Note that the sublists being merged are different from those being merged in *mergeSort*. □

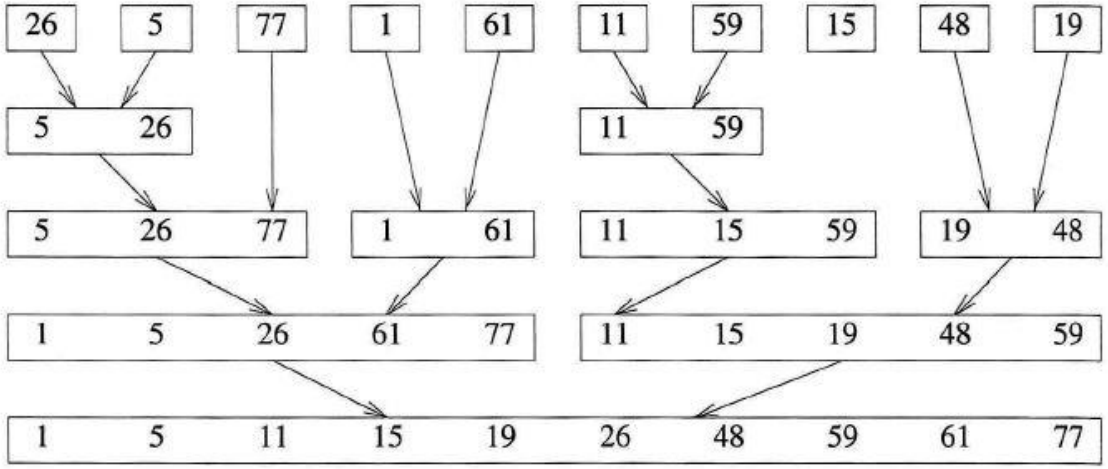
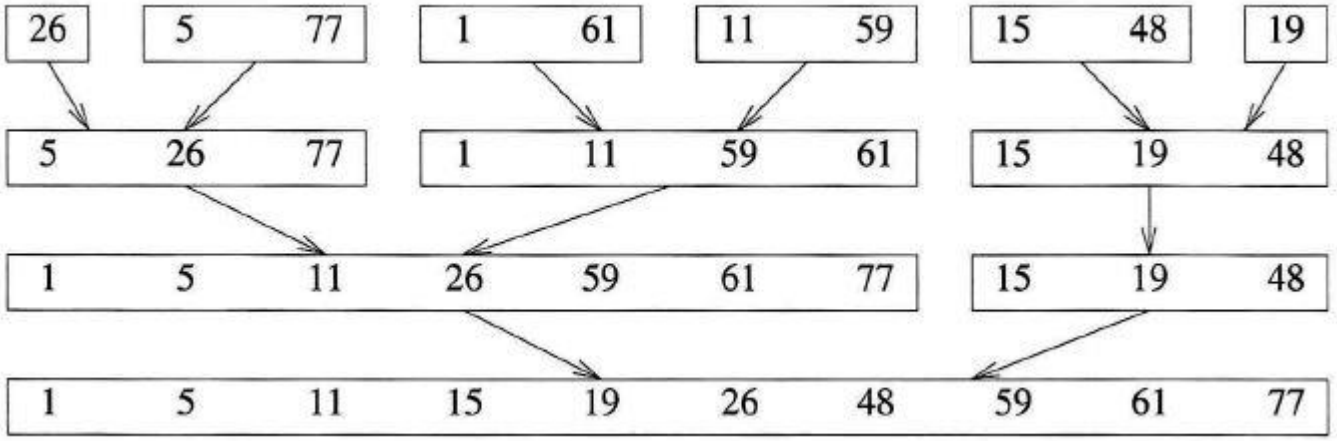


Figure 7.5: Sublist partitioning for recursive merge sort

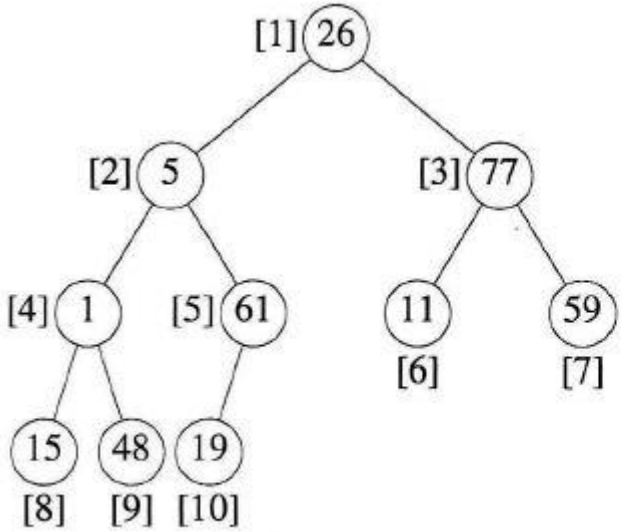
7.5 Merge sort

Variation—Natural Merge Sort: We may modify *mergeSort* to take into account the prevailing order within the input list. In this implementation we make an initial pass over the data to determine the sublists of records that are in order. Merge sort then uses these initially ordered sublists for the remainder of the passes. Figure 7.6 shows natural merge sort using the input sequence of Example 7.6.

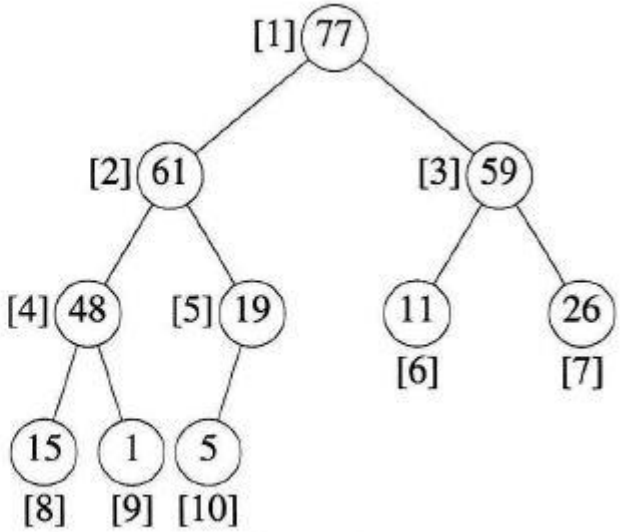


7.6 Heap sort

Although the merge sort scheme discussed in the previous section has a computing time of $O(n \log n)$, both in the worst case and as average behavior, it requires additional storage proportional to the number of records to be sorted. The sorting method we are about to study, heap sort, requires only a fixed amount of additional storage and at the same time has as its worst-case and average computing time $O(n \log n)$. However, heap sort is slightly slower than merge sort.

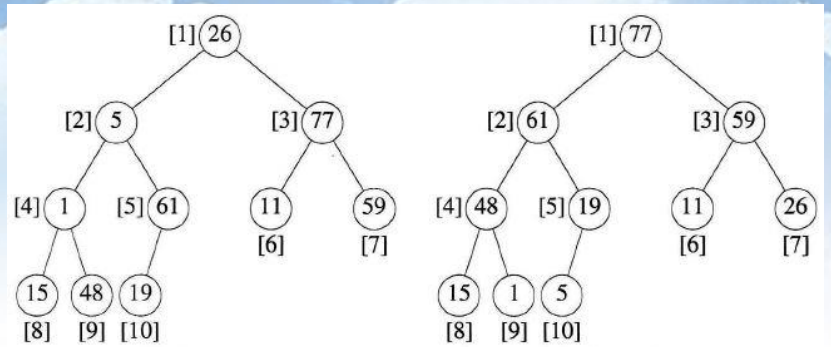


(a) Input array

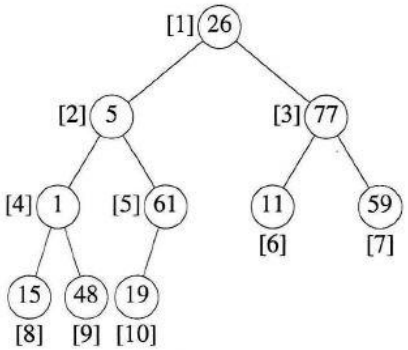


(b) Initial heap

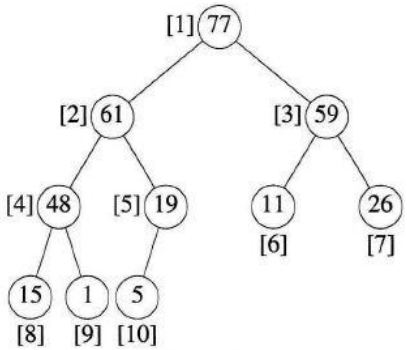
7.6 Heap sort



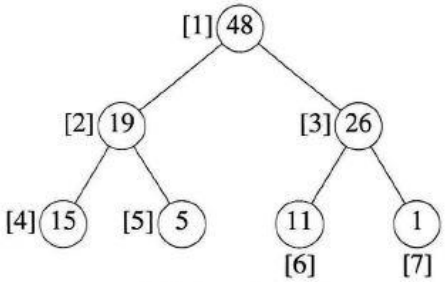
(a) Input array (b) Initial heap



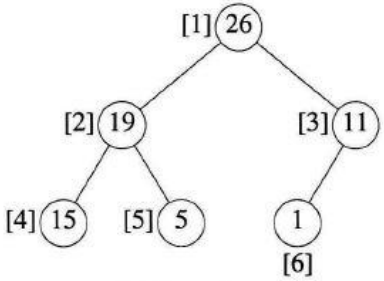
(a) Input array



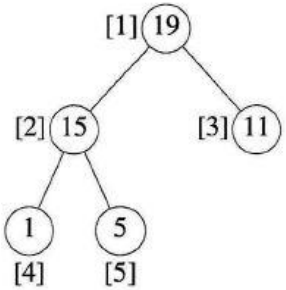
(b) Initial heap



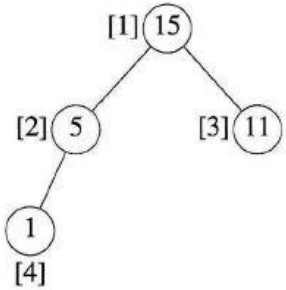
(c) Heap size = 7
Sorted = [59, 61, 77]



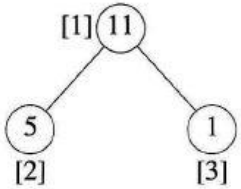
(d) Heap size = 6
Sorted = [48, 59, 61, 77]



(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]

7.6 Heap sort

```
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);
   }
}
```

The total computing time is $O(n \log n)$.

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

Thank You !

노력 없이 이룰 수 있는 것 아무것도 없다.