Chap 7. Sorting (2)

Contents

7.4 How Fast Can We Sort?

- How quickly can we sort a list on *n* objects?
  - the best possible time: $O(n \cdot \log_2 n)$
- Decision tree describing the sorting process
  - *vertex : a key comparison*, branch : the result
  - Input sequence R_1, R_2, R_3 is labeled [1, 2, 3]

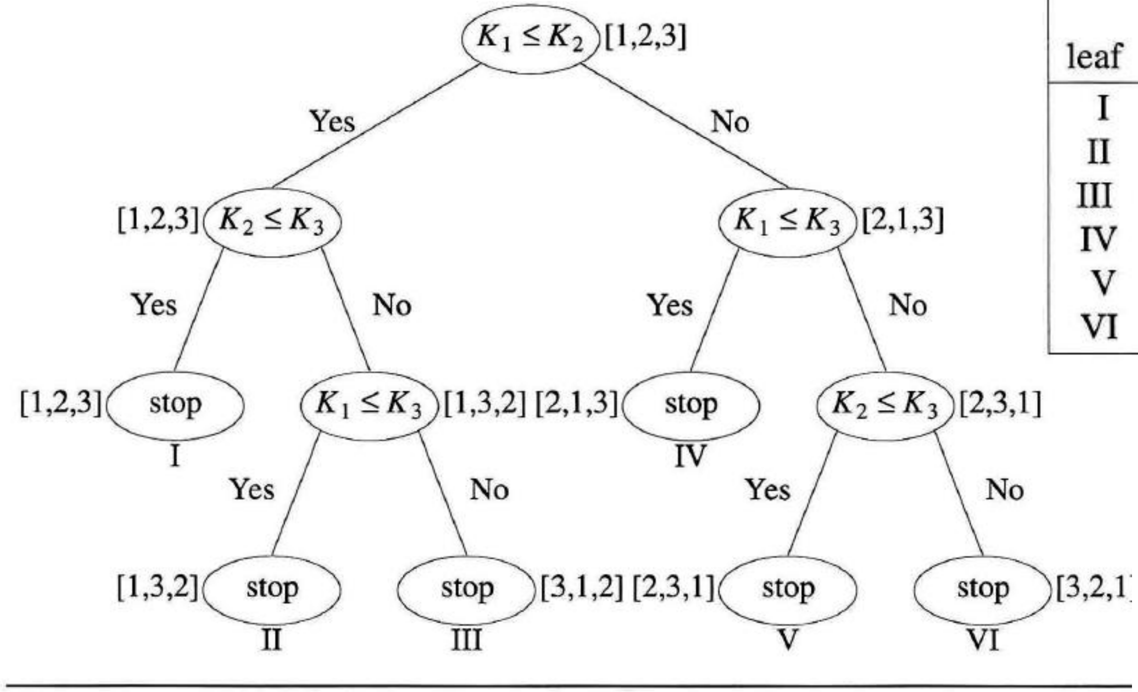


Figure 7.2: Decision tree for insertion sort

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]

Permutation 3! = 6
the maximum depth = 4

- **Theorem** : Any decision tree that sorts *n* distinct elements has a height of at least $\log_2(n!) + 1$
  - decision tree of *n* elements have *n*! leaves
  - number of leaves of a BT of height $k \leq 2^{k-1}$
  - $n! \leq 2^{k-1}$
  - $k \geq \log_2(n!) + 1$

- **Corollary** : Any algorithm that sorts by *comparisons* only must have a worst case computing time of $\Omega(n\log_2 n)$
  - By theorem, for every decision tree with n! leaves, there is a path of length $\log_2(n!)$
  - $n!=n(n-1)(n-2)\ldots(3)(2)(1) \geq (n/2)^{n/2}$
  - $\log_2(n!)\geq(n/2)\log_2(n/2) = \Omega(n\log_2 n)$

7.5 Merge Sort

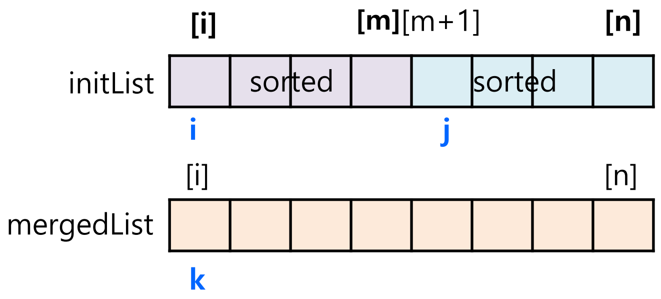
- Merge *two sorted lists* to *a single sorted list*.
  - initList[i:m] and initList[m+1:n]➔mergedList[i:n]

- Example

	A	B	C
1	2, 5, 6	1, 3, 8, 9, 10	
2	2, 5, 6	3, 8, 9, 10	1
3	5, 6	3, 8, 9, 10	1, 2
4	5, 6	8, 9, 10	1, 2, 3
5	6	8, 9, 10	1, 2, 3, 5
6		8, 9, 10	1, 2, 3, 5, 6
7			1, 2, 3, 5, 6, 8, 9, 10

  – Compare the smallest elements of A and B and merge the smaller into C.
  – When one of A and B becomes empty, append the other list to C.

```
void merge(element initList[], element mergedList[],
           int i, int m, int n)
{/*  the sorted lists initList[i:m] and initList[m+1: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)
         mergedList[k++] = initList[i++];
      else
         mergedList[k++] = initList[j++];
   }
   if (i > m)
   /* mergedList[k:n] = initList[j:n] */
      for (t = j; t <= n; t++)
         mergedList[t] = initList[t];
   else
   /* mergedList[k:n] = initList[i:m] */
      for (t = i; t <= m; t++)
         mergedList[k+t-i] = initList[t];
}
```



Program 7.7: Merging two sorted lists

- **Analysis of *merge*:**
  - Total increment in *k* is *n*-*i*+1.
  - O(*n*-*i*+1) ➔ O(*n*)
  - Stable sorting

7.5.2 Iterative Merge Sort

- Start with sorted lists of size 1 and do pairwise merging of these sorted lists.

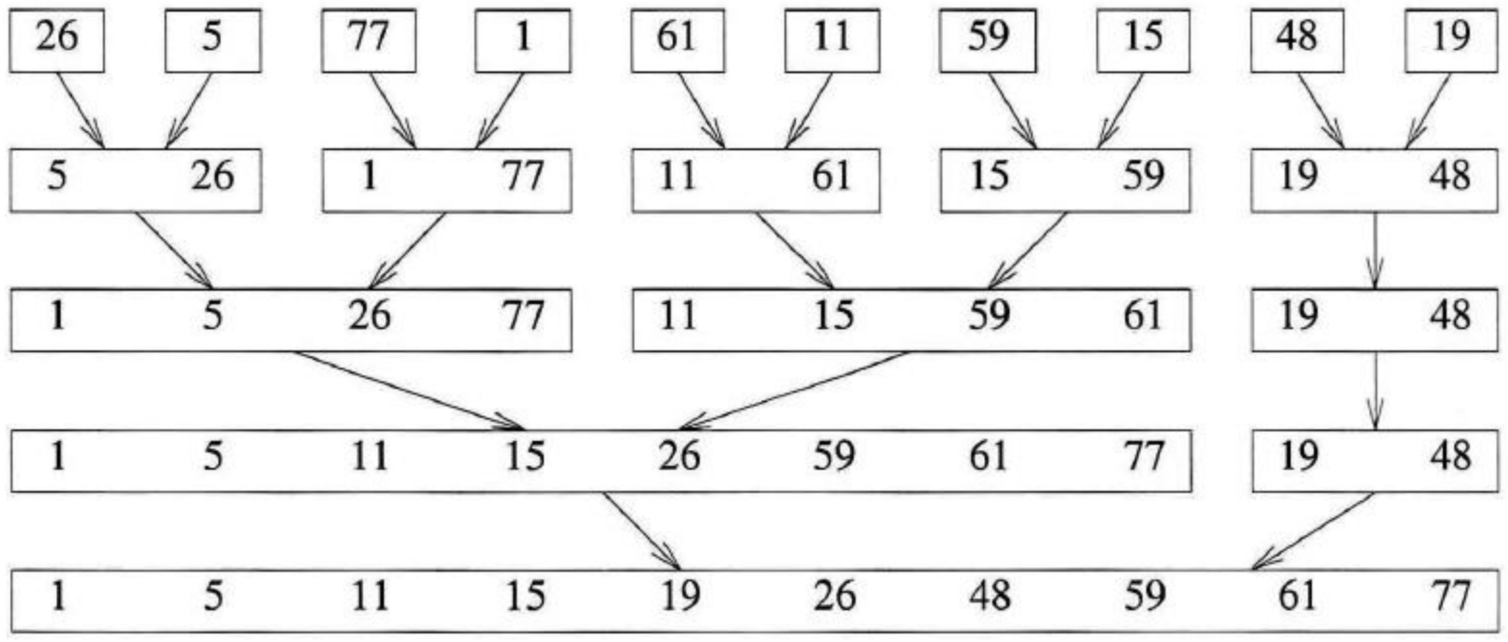
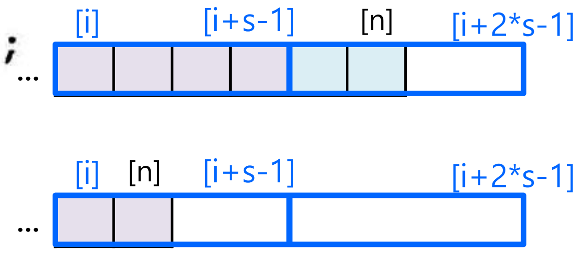
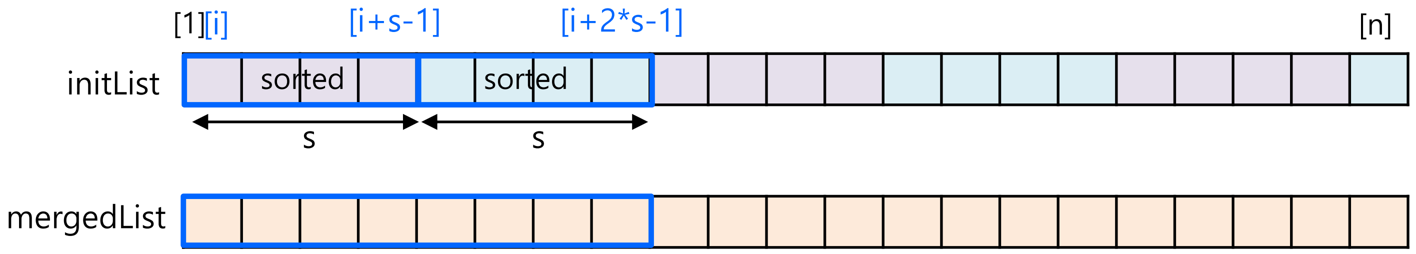


Figure 7.4: Merge tree

```
void mergePass(element initList[], element mergedList[],
                    int n, int s)
{/* perform one pass of the merge sort, merge adjacent
    pairs of sorted segments from initList[] into mergedList[],
    n is the number of elements in the list, s is
    the size of each sorted segment */
   int i,j;        i+2*s-1 <= n
(1) for (i = 1; i <= n - 2 * s + 1; i += 2 * s)
     merge(initList,mergedList,i,i + s - 1,i + 2 * s - 1);
(2) if (i + s - 1 < n)
     merge(initList,mergedList,i,i + s - 1,n);
(3) else
     for (j = i; j <= n; j++)
        mergedList[j] = initList[j];
}
```



Program 7.8: A merge pass



```
void mergeSort(element a[], int n)
{/* sort a[1:n] using the merge sort method */
   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

- **Analysis of *mergeSort*:**
  - ith pass merges segments of size 2^{i-1}
  - The number of passes is $\lceil \log_2 n \rceil$.
  - Each merge pass takes $O(n)$ time.
  - Total time is $O(n \log n)$.
  - Need $O(n)$ additional space for the merge.
  - Stable sorting

7.5.3 Recursive Merge Sort

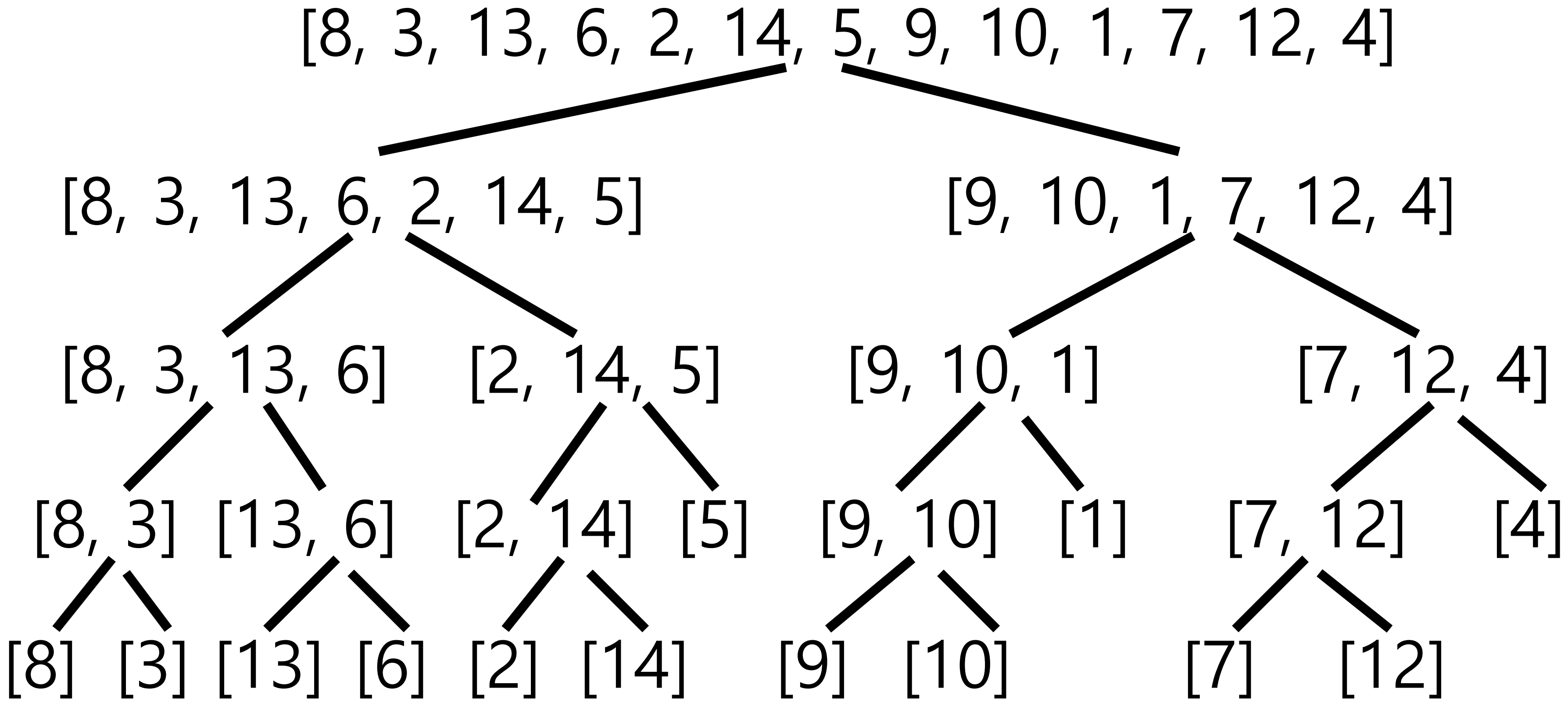
- *Divide* the list to be sorted into two roughly equal parts.
  - Left sublist : $\lceil n/2 \rceil$ elements
  - Right sublist : $\lfloor n/2 \rfloor$ elements

- These sublists are *sorted recurively*.

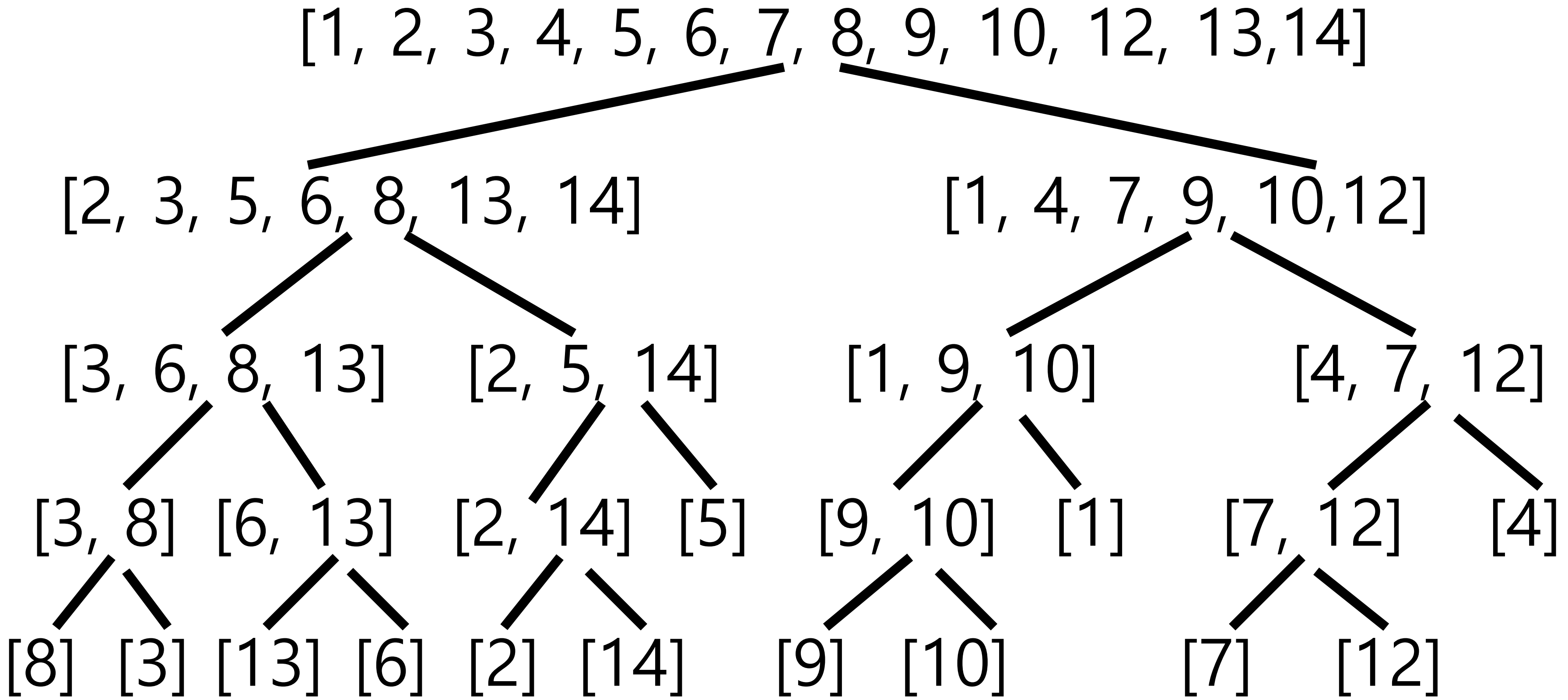
- The sorted sublists are *merged*.

- *Downward pass* over the recursion tree.
  - Divide large list into small lists.
- *Upward pass* over the recursion tree.
  - Merge pairs of sorted lists.
- Number of leaf nodes is n.
- Number of nonleaf nodes is $n-1$.

Example : downward pass



Example : upward pass



```
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 */
}
```

Program 7.10: Recursive merge sort

※ We use chains to eliminate record copying.
(**n chains** each of which has one node)

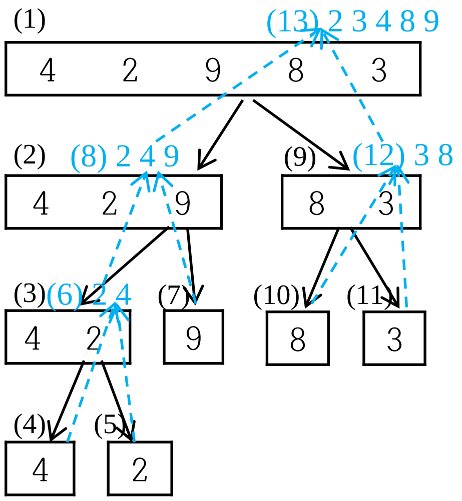
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[n]
link	0	0	0	0	0	0	0	0	0	0	0
a	-	26	5	77	1	61	11	59	15	48	19
		left		node		mid					right

First call :
rmergeSort(a, link, 1, n);

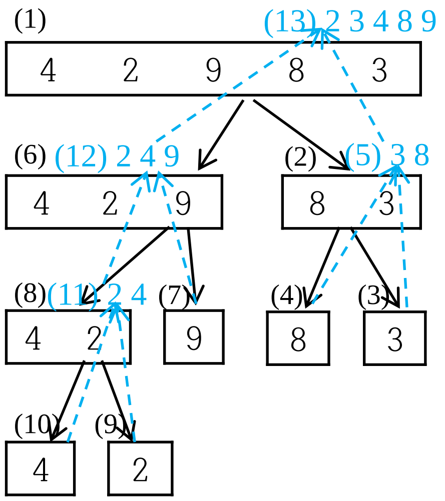
```
int listMerge(element a[], int link[], int start1, int start2)
{/* sorted chains beginning at start1 and start2,
    respectively, are merged; link[0] is used as a
    temporary header; returns start of merged chain */
   int last1, last2, lastResult = 0;
   for (last1 = start1, last2 = start2; last1 && 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

- **Recursive Merge Sort : rmergeSort + listMerge**



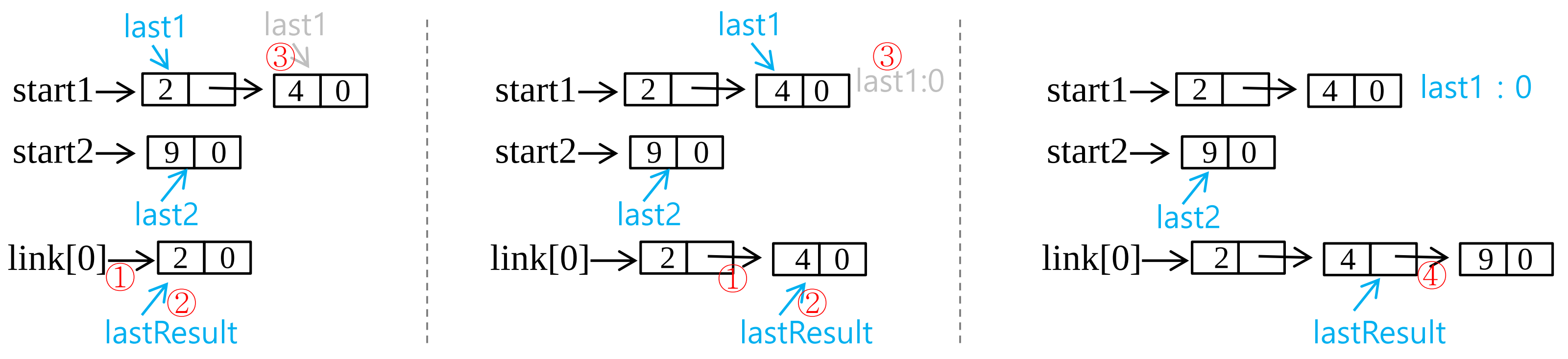
(a) Modified version of LRV



(b) Modified version of RLV

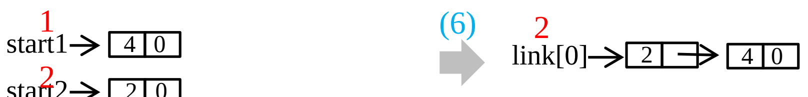
※(1)~(13) : the order of function calls

- **listMerge(a, link, 2, 3)**



```
int listMerge(element a[], int link[], int start1, int start2)
{
    int last1, last2, lastResult = 0;
    for (last1 = start1, last2 = start2; last1 && last2;)
        if (a[last1] <= a[last2]) {
          ① link[lastResult] = last1;
          ② lastResult = last1;③last1 = link[last1];
        }
        else {
            link[lastResult] = last2;
            lastResult = last2; last2 = link[last2];
        }

    /* attach remaining records to result chain */
    if (last1 == 0)④link[lastResult] = last2;
    else link[lastResult] = last1;
    return link[0];
}
```

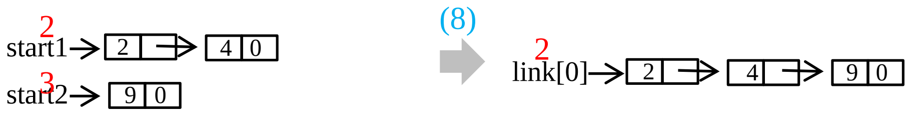
- **all calls to listMerge**



listMerge(a, link, 1, 2);

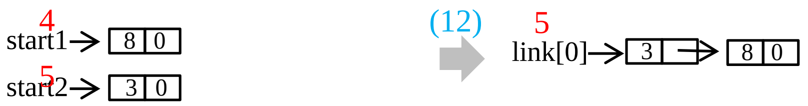
	[0]	[1]	[2]	[3]	[4]	[5]
link	0	0	0	0	0	0
a	-	4	2	9	8	3

	[0]	[1]	[2]	[3]	[4]	[5]
link	2	0	1	0	0	0
a	-	4	2	9	8	3



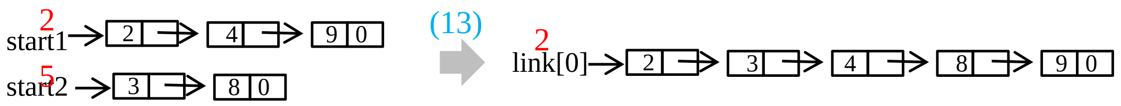
	[0]	[1]	[2]	[3]	[4]	[5]
link	2	0	1	0	0	0
a	-	4	2	9	8	3

	[0]	[1]	[2]	[3]	[4]	[5]
link	2	3	1	0	0	0
a	-	4	2	9	8	3



	[0]	[1]	[2]	[3]	[4]	[5]
link	2	3	1	0	0	0
a	-	4	2	9	8	3

	[0]	[1]	[2]	[3]	[4]	[5]
link	5	3	1	0	0	4
a	-	4	2	9	8	3



	[0]	[1]	[2]	[3]	[4]	[5]
link	5	3	1	0	0	4
a	-	4	2	9	8	3

	[0]	[1]	[2]	[3]	[4]	[5]
link	2	4	5	0	3	1
a	-	4	2	9	8	3

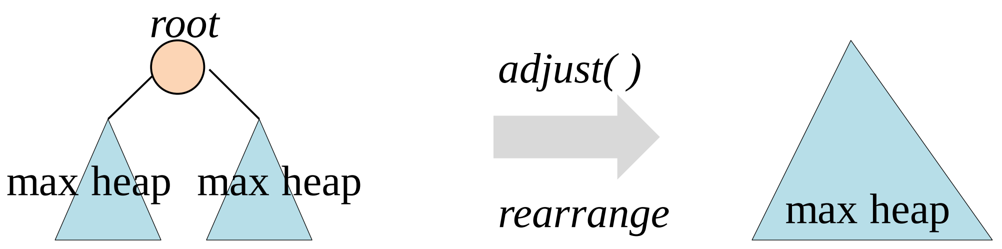
- The addition of the array of links
  - Record copying is replaced by *link changes*
  - The runtime becomes independent of the size *s* of a record.
  - Additional space required is O(*n*).

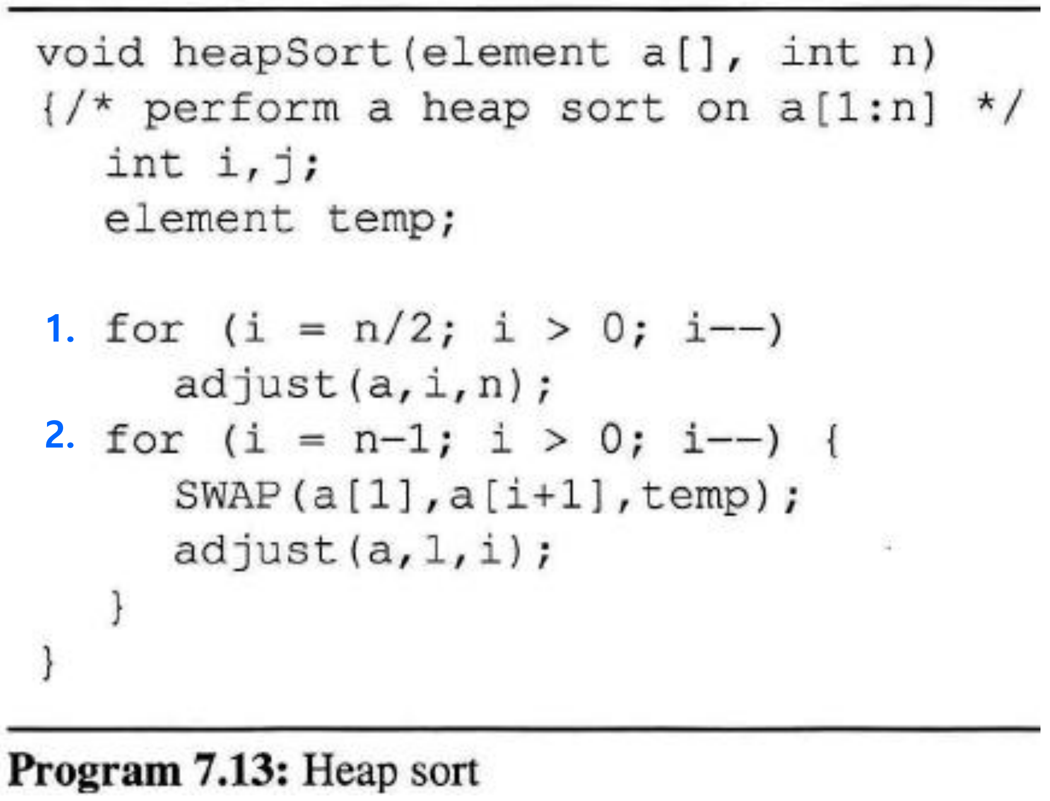
- **Analysis of *rmergeSort*:**
  - Stable sorting
  - Downward pass
    - O(1) time at each node.
    - O(*n*) total time at all nodes.
  - Upward pass
    - O(*n*) time merging at each level that has a nonleaf node.
    - Number of levels is O(log *n*).
    - Total time is O(*n* log *n*).

7.6 Heap Sort

- Using the *max heap* introduced in Chapter 05
  - n records are inserted into an empty max heap.
  - The records are extracted from the max heap one at a time.

- Using the *max heap* by function *adjust*
  - Faster than by inserting introduced in Chapter 05





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

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

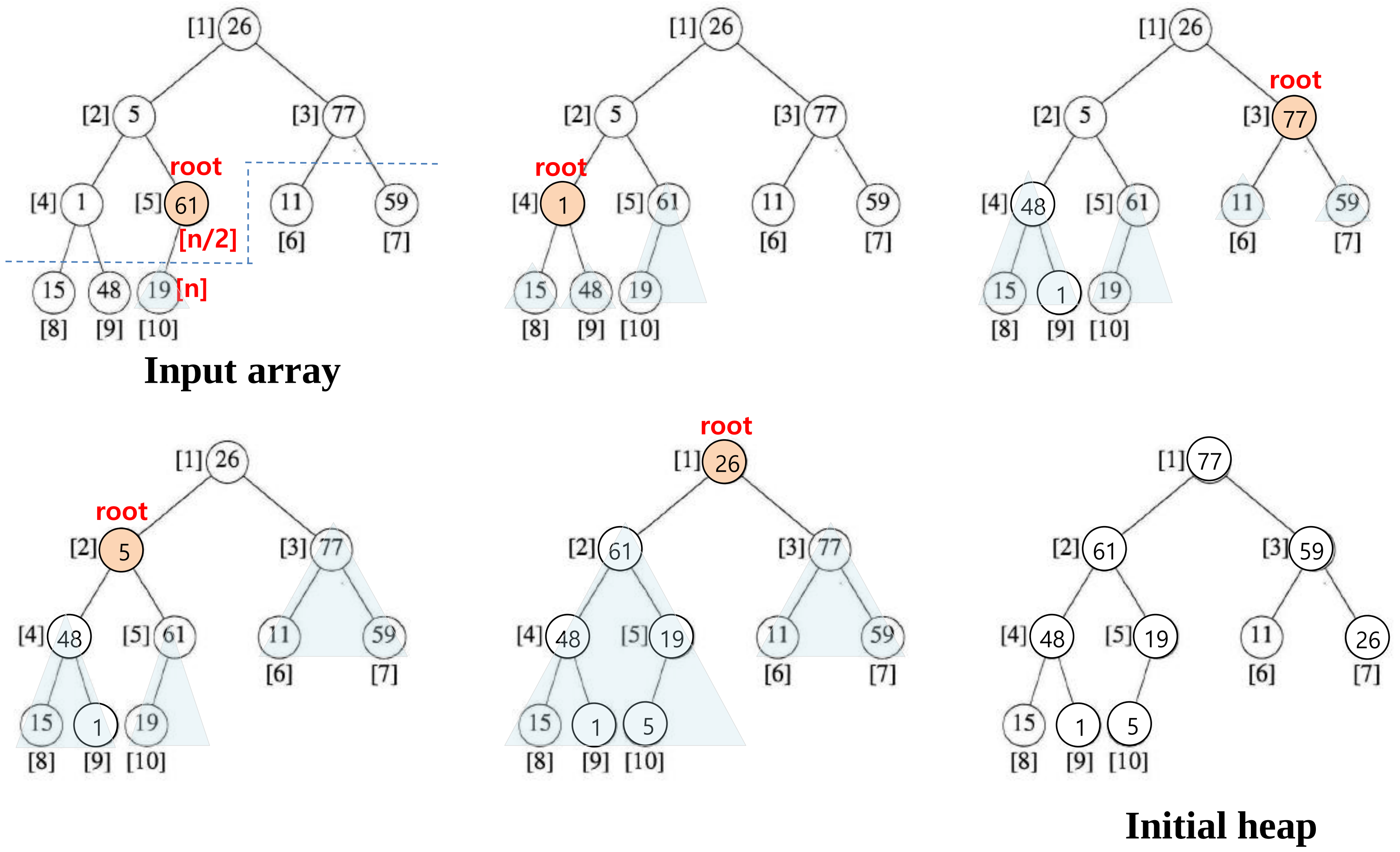
Program 7.13: Heap sort

1. Create an initial *max heap* by using *adjust* repeatedly.
2. Repeat the following pass *n-1* times to *sort* an array a[1:*n*].
   ① Swap the *first* and *last* records in the heap
   ② Decrease the heap size and re*adjust* the heap

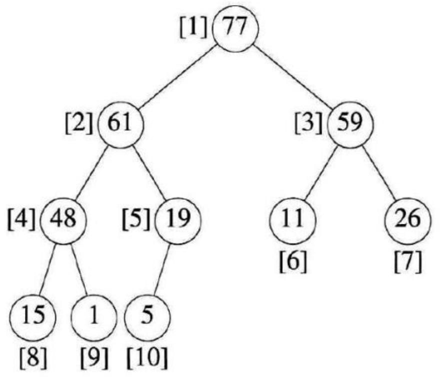
```
void adjust(element a[], int root, int n)
{/* adjust the binary tree to establish the heap */
  int child,rootkey;
  element temp;
  temp = a[root];
  rootkey = a[root].key;
  child = 2 * root;                        /* left child */
  while (child <= n) {
    if ((child < n) &&
    (a[child].key < 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;
}
```

Program 7.12: Adjusting a max heap

1. Creating an initial *max heap*



2. Sorting the array a[1:n]


Initial heap

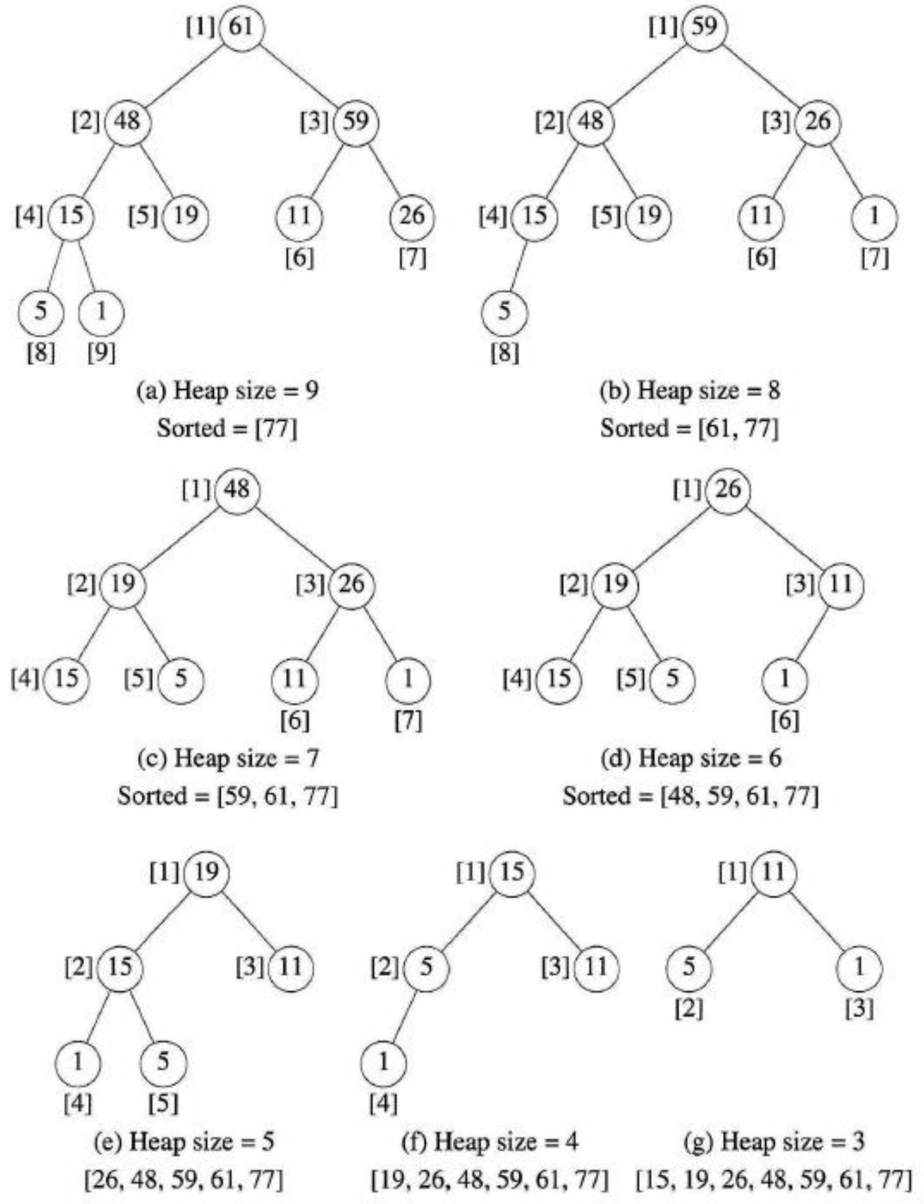


Figure 7.8: Heap sort example

- **Analysis of *heapSort*:**
  - average case : $O(n \cdot \log_2 n)$
  - function *adjust* : $O(d)$, where d: depth of tree
  - worst case :

    $\lfloor \log_2 n \rfloor + \lfloor \log_2 (n-1) \rfloor + \ldots + \lfloor \log_2 2 \rfloor = O(n \cdot \log_2 n)$