CHAPTER 7

SORTING

All the programs in this file are selected from

Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
"Fundamentals of Data Structures in C",

Sequential Search

- Example

  44, 55, 12, 42, 94, 18, 06, 67
- Unsuccessful search
  - n+1
- Successful search

$$\sum_{i=0}^{n-1}(i+1)/n = \frac{n+1}{2}$$

```
# define MAX-SIZE 1000 /* maximum size of  list plus one */
typedef struct {
          int key;
          /* other fields */
          } element;
element list[MAX_SIZE];
```

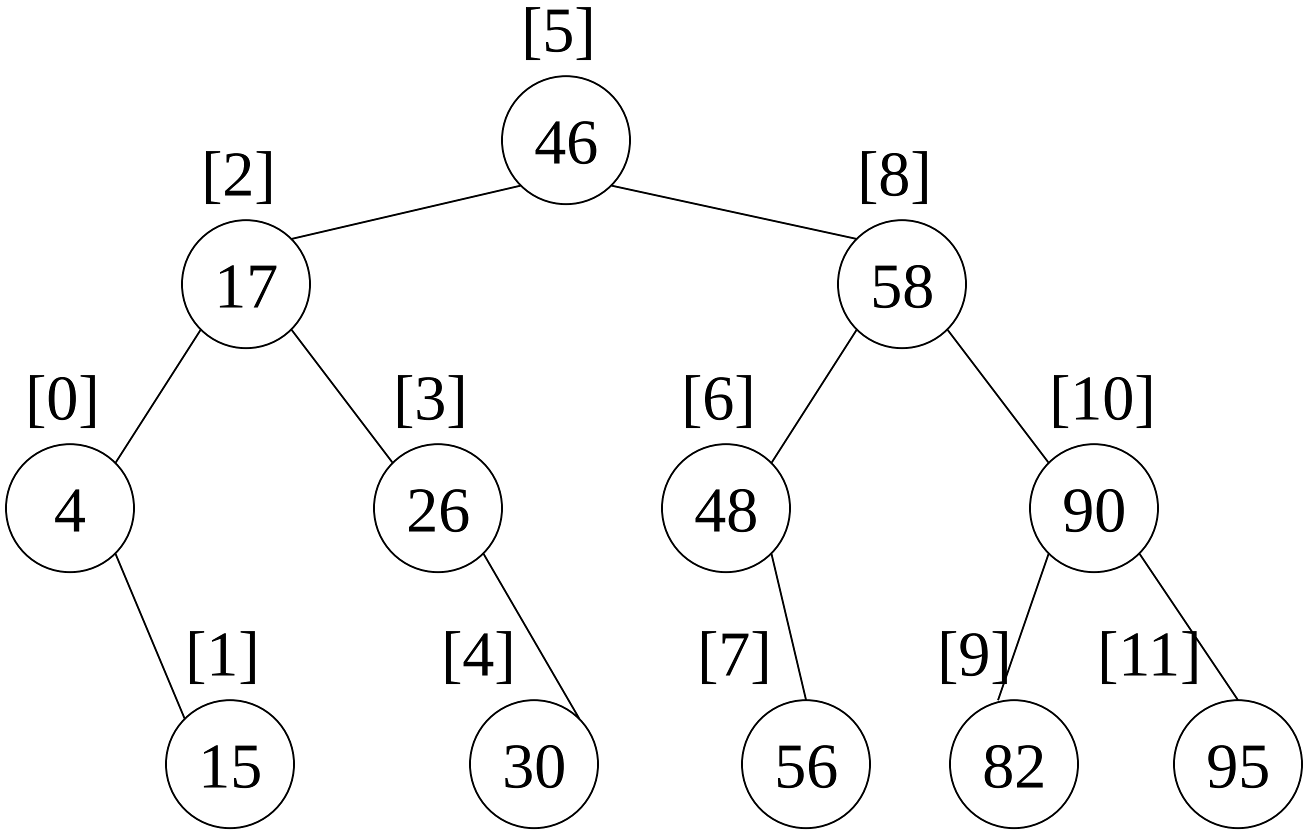
***Program 7.1:**Sequential search

```
int seqSearch(element a[], int k, int n)
{
  /* 在a[1:n]中尋找最小的i值，使得a[i].key = k
     如果k在串列中找不到，則回傳0。*/
   int i;
   for(i=1; i<=n && a[i].key !=k; i++)
        ;
   if (i>n) return 0;
   return i;
}
```

*Program : Binary search

```
int binsearch(element list[ ], int searchnum, int n)
{
/* search list [0], ..., list[n-1]*/
    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;  //right
                    break;
        case  0: return middle;
        case  1:right = middle - 1;  //left
                   break;
        }
      }
      return -1;
}
```

$O(\log_2 n)$



4, 15, 17, 26, 30, 46, 48, 56, 58, 82, 90, 95

***Figure 7.1:**Decision tree for binary search

List Verification

- Compare lists to verify that they are identical or identify the discrepancies.
- Example
  - International revenue service (e.g., employee vs. employer)
  - List1 is the employer list; List2 is the employee list.
  - Various employers stating how much they paid their employees;
  - Individual employees stating how much they received.
- Complexities
  - random order: $O(mn)$
  - ordered list: $O(tsort(n)+tsort(m)+m+n)$

***Program 7.2:** verifying using a sequential search

```
void verify1(element list1[], element list2[ ], int n, int m)
/* compare two unordered lists list1 and list2 */
{
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 list 2\n ", list1[i].key);
 else
 /* check each of the other fields from list1[i] and list2[j], and
print out any discrepancies */
```

(a) all records found in list1 but not in list2
(b) all records found in list2 but not in list1
(c) all records that are in list1 and list2 with
    the same key but have different values
    for different fields.

```
    marked[j] = TRUE;
for ( i=0; i<m; i++)
    if (!marked[i])
        printf(“%d is not in list1\n”, list2[i].key);
}
```

*Program 7.3:Fast verification of two lists (p.325)

```
void verify2(element list1[ ], element list2 [ ], int n, int m)
/* Same task as verify1, but list1 and list2 are sorted */
{
  int i, j;
  sort(list1, n);
  sort(list2, m);
  i = j = 0;
  while (i < n && j < m)
      if  (list1[i].key < list2[j].key) {
          printf (“%d is not in list 2 \n”, list1[i].key);
          i++;  //fixed j, control i
      }
      else if (list1[i].key == list2[j].key) {
      /* compare list1[i] and list2[j] on each of the other field
        and report any discrepancies */
      i++; j++;
      }
```

```
 else {
     printf(“%d is not in list 1\n”, list2[j].key);
     j++;
 }
for(; i < n; i++)
     printf (“%d is not in list 2\n”, list1[i].key);
for(; j < m; j++)
     printf(“%d is not in list 1\n”, list2[j].key);
}
```

Sorting Problem

R: record   K: key value

- Definition
  - given $(R_0, R_1, \ldots, R_{n-1})$, where R_i = key + data
    find a permutation σ, such that $K_{\sigma(i-1)} \leq K_{\sigma(i)}$, $0<i<n-1$
- Sorted
  - $K_{\sigma(i-1)} \leq K_{\sigma(i)}$, $0<i<n-1$
- Stable
  - if $i < j$ and $K_i = K_j$ then R_i precedes R_j in the sorted list
- internal sort vs. external sort
- Criteria
  - # of key comparisons
  - # of data movements

Insertion Sort

Find an element smaller than K.

	●								
26	5	77	1	61	11	59	15	48	19
		◆							
5	26	77	1	61	11	59	15	48	19
			◆						
5	26	77	1	61	11	59	15	48	19
				◆					
1	5	26	77	61	11	59	15	48	19
					◆				
1	5	26	61	77	11	59	15	48	19
						◆			
1	5	11	26	61	77	59	15	48	19
							◆		
1	5	11	26	59	61	77	15	48	19
								◆	
1	5	11	15	26	59	61	77	48	19
									◆
1	5	11	15	26	48	59	61	77	19

Insertion Sort

```
void insert(element e, element a[], int i)
{ /* 將e插入到一個已排序過串列 a[1:i] 中，並使得插入過後的串列
a[1:i+1]仍然是依序排好。此陣列a至少必須分配到大小為i+2個
element */
  a[0] = e;
  while (e.key < a[i].key)
   {
        a[i+1] = a[i];
        i--;
   }
 a[i+1] = e;
}
```

```
void insertionSort( element a[], int n)
{ /* 將a[1:n] 排序成依序遞增 */
     int j;
     for ( j =2; j<=n; j++) {
          element temp = a[j];
          insert(temp, a, j-1);
     }
}
```

worse case

i	0	1	2	3	4
-	5	4	3	2	1
1	4	5	3	2	1
2	3	4	5	2	1
3	2	3	4	5	1
4	1	2	3	4	5

$$\boxed{O(\sum_{j=0}^{n-2} i) = O(n^2)}$$

k: # of records L

n: # of records

best case

Computing time: O(

i	0	1	2	3	4
-	2	3	4	5	1
1	2	3	4	5	1
2	2	3	4	5	1
3	2	3	4	5	1
4	1	2	3	4	5

$$\boxed{O(n)}$$

left out of order (LOO)

Variation

- Binary insertion sort
  - sequential search --> binary search
  - reduce # of comparisons,
    # of moves unchanged
- List insertion sort
  - array --> linked list
  - sequential search, move --> 0

Quick Sort (C.A.R. Hoare)

- Given $(R_0, R_1, \ldots, R_{n-1})$
- K_i: key
  if K_i is placed in $S(i)$,
  then $K_j \le K_{s(i)}$ for $j < S(i)$,
  $K_j \ge K_{s(i)}$ for $j > S(i)$.
- $\underline{R_0, \ldots, R_{S(i)-1},}\ R_{S(i)},\ \underline{R_{S(i)+1}, \ldots, R_{S(n-1)}}$

two partitions

Example for Quick Sort

pivot

R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	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}	pivot 11	{19	15}	26	{ 59	61	48	37}	0	1
1	5	11	15	19	26	{ 59	61	48	37}	3	4
1	5	11	15	19	26	{ 48	37}	59	{ 61}	6	9
1	5	11	15	19	26	37	48	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		

Quick Sort

```
void quicksort(element list[], int left,int right)
{
  int pivot, i, j;
  element temp;
  if (left < right) {
    i = left;    j = right+1;
    pivot = list[left].key;
    do {
      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);
 //exchange the number between groups using pivot
    SWAP(list[left], list[j], temp);
    quicksort(list, left, j-1);
    quicksort(list, j+1, right);
  }
}
```

Analysis

- Assume that each time a record is positioned, the list is divided into the rough same size of two parts.
- Position a list with n element needs O(n)
- $T(n)$ is the time taken to sort n elements

  $T(n)<=cn+2T(n/2)$ for some c

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

  ...

  $<=cn\log n+nT(1)=O(n\log n)$

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

Recurrence Solving: Review

- $T(n) = 2T(n/2) + cn$, **with** $T(1) = 1$.
- **By term expansion.**

$$\begin{aligned} T(n) &= 2T(n/2) \; + \; cn \\ &= 2\left(2T(n/2^2) + cn/2\right) + cn \; = \; 2^2T(n/2^2) \; + \; 2cn \\ &= 2^2\left(2T(n/2^3) + cn/2^2\right) + 2cn \; = \; 2^3T(n/2^3) \; + \; 3cn \\ &\vdots \\ &= 2^iT(n/2^i) \; + \; icn \end{aligned}$$

- **Set** $i = \log_2 n$. **Use** $T(1) = 1$.
- **We get** $T(n) \; = \; n \; + \; cn(\log n) \; = \; O(n \log n)$.

Time and Space for Quick Sort

- Space complexity:
  - – Average case and best case: O(log n)
  - – Worst case: O(n)
- Time complexity:
  - – Average case and best case: O(n log n)
  - – Worst case: O(n^2)

Merge Sort

- Given two sorted lists

  (initList[i], …, initList[m])

  (initList[m+1], …, initList[n])

- O(n) space vs. O(1) space

  Generate a single merge list

  (mergeList[i], …, mergeLis[n])

Merge Sort (O(n) space)

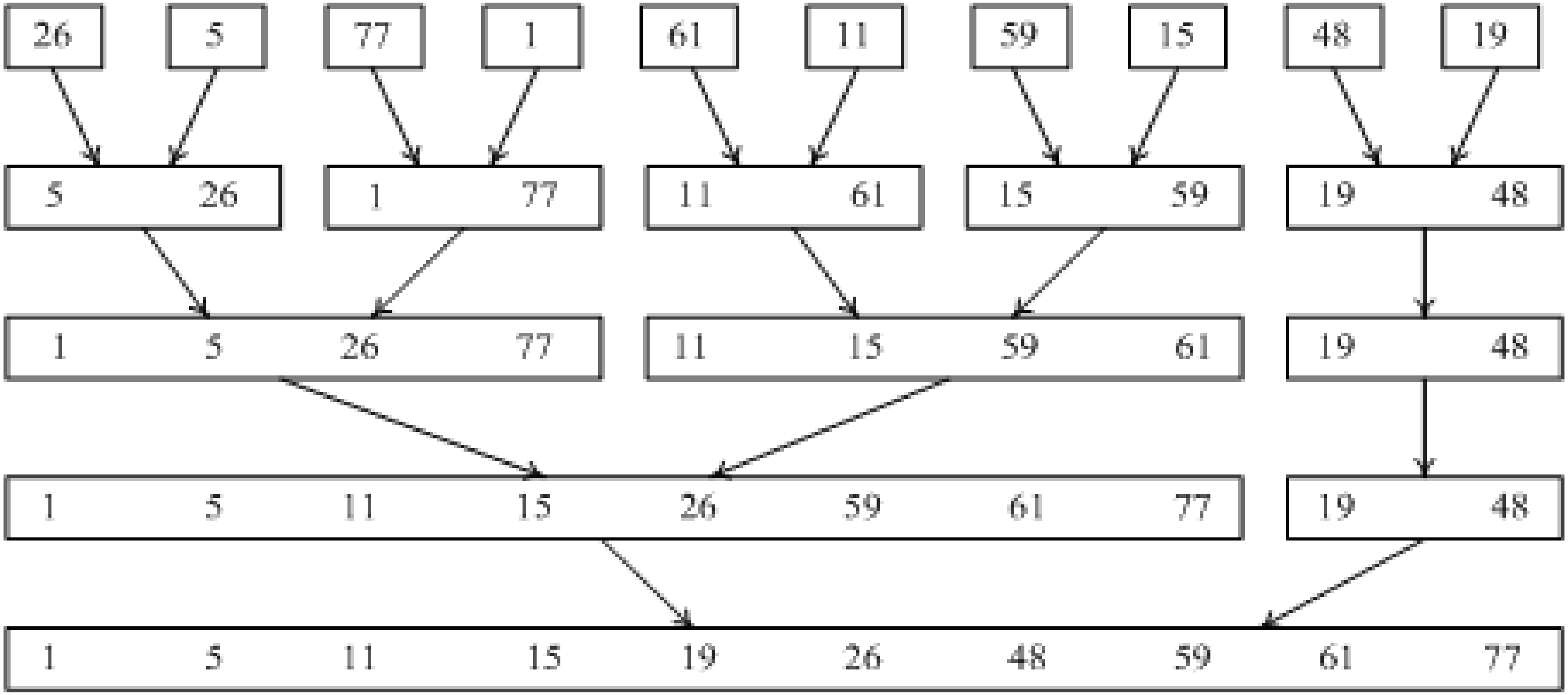
```
void merge(element initList[], element mergeList[]
               int i, int m, int n)
{
  int j, k, t;
  j = m+1; /*第二個子串列的索引值*/
  k = i; /*合併串列的起始索引值*/
  while (i<=m && j<=n) {
    if (initList[i].key<= initList[j].key)
        mergeList[k++]= initList[i++];
    else mergeList[k++]= initList[j++];
  }
  if (i>m)  /* mergedList[k:n] = initList[j:n] */
    for (t=j; t<=n; t++)
     mergeList [t]= initList[t];
  else for (t=i; t<=m; t++)/* mergedList[k:n] = initList[i:m]
     mergeList[k+t-i] = initList[t];
}
```

addition space: n-i+1

# of data movements: M(n-

Analysis

- array vs. linked list representation
  - array: $O(s(n-i+1))$ where s: the size of a record
    for copy
  - linked list representation: $O(n-i+1)$
    $(n-i+1)$ linked fields



26 5 77 1 61 11 59 15 48 19

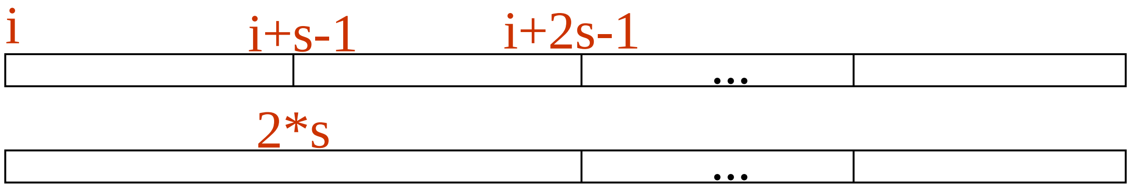
5 26 | 1 77 | 11 61 | 15 59 | 19 48

1 5 26 77 | 11 15 59 61 | 19 48

1 5 11 15 26 59 61 77 | 19 48

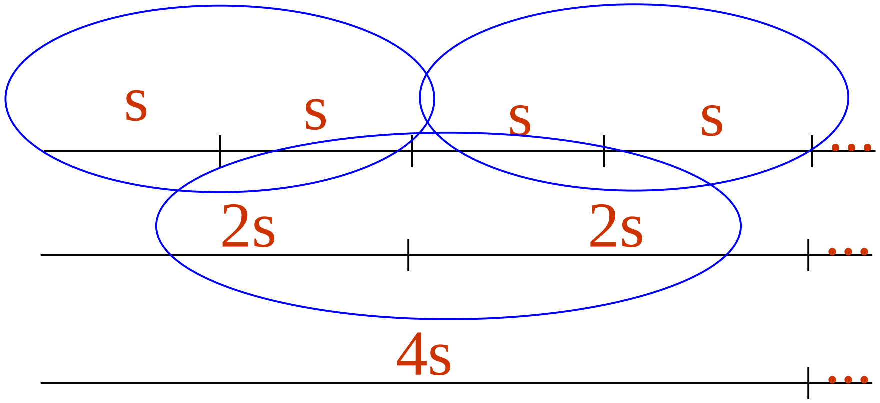
1 5 11 15 19 26 48 59 61 77

```
void mergePass(element initList[], element
mergedList[], int n, int s)
{/* 執行一回合的合併排序，將initList[]中兩兩相鄰的
排序過的區段合併到mergedList[]。n 為串列中元素個數
，s代表每一個區段大小。 */
  int i, j;
  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)  One complement segment and one partial segment
    merge(initList, mergedList, i, i+s-1, n);
  else
    for( j=i; j<=n; j++)  Only one segment
          mergedList[j] = initList[j];
}
```



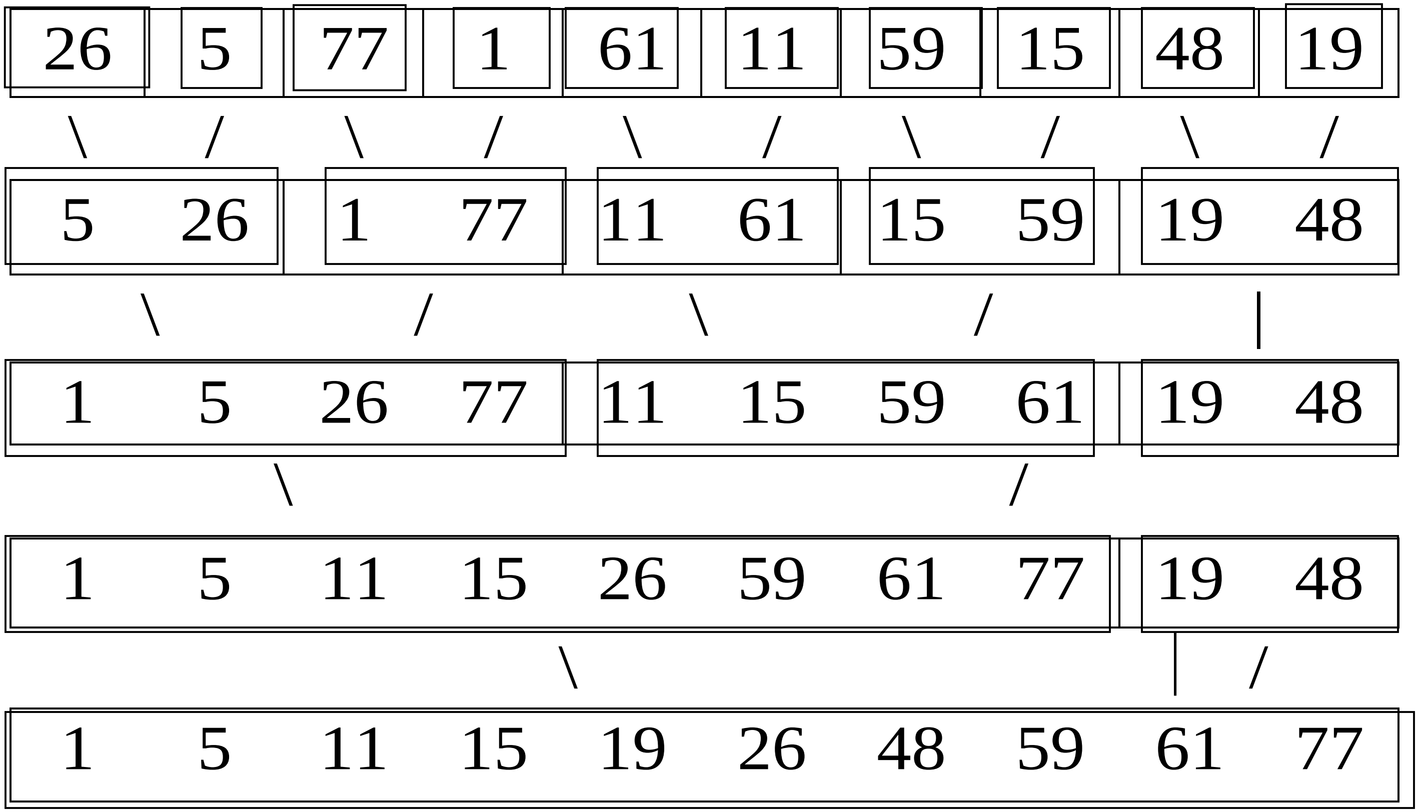
```
void mergeSort(element a[], int n)
{ /* 使用合併排序法將a[1:n]排序 */
  int s =1;     /*現在區段大小*/
  element extra[MAX_SIZE];

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



Interactive Merge Sort

Sort 26, 5, 77, 1, 61, 11, 59, 15, 48, 19



26 5 77 1 61 11 59 15 48 19

5 26 | 1 77 | 11 61 | 15 59 | 19 48

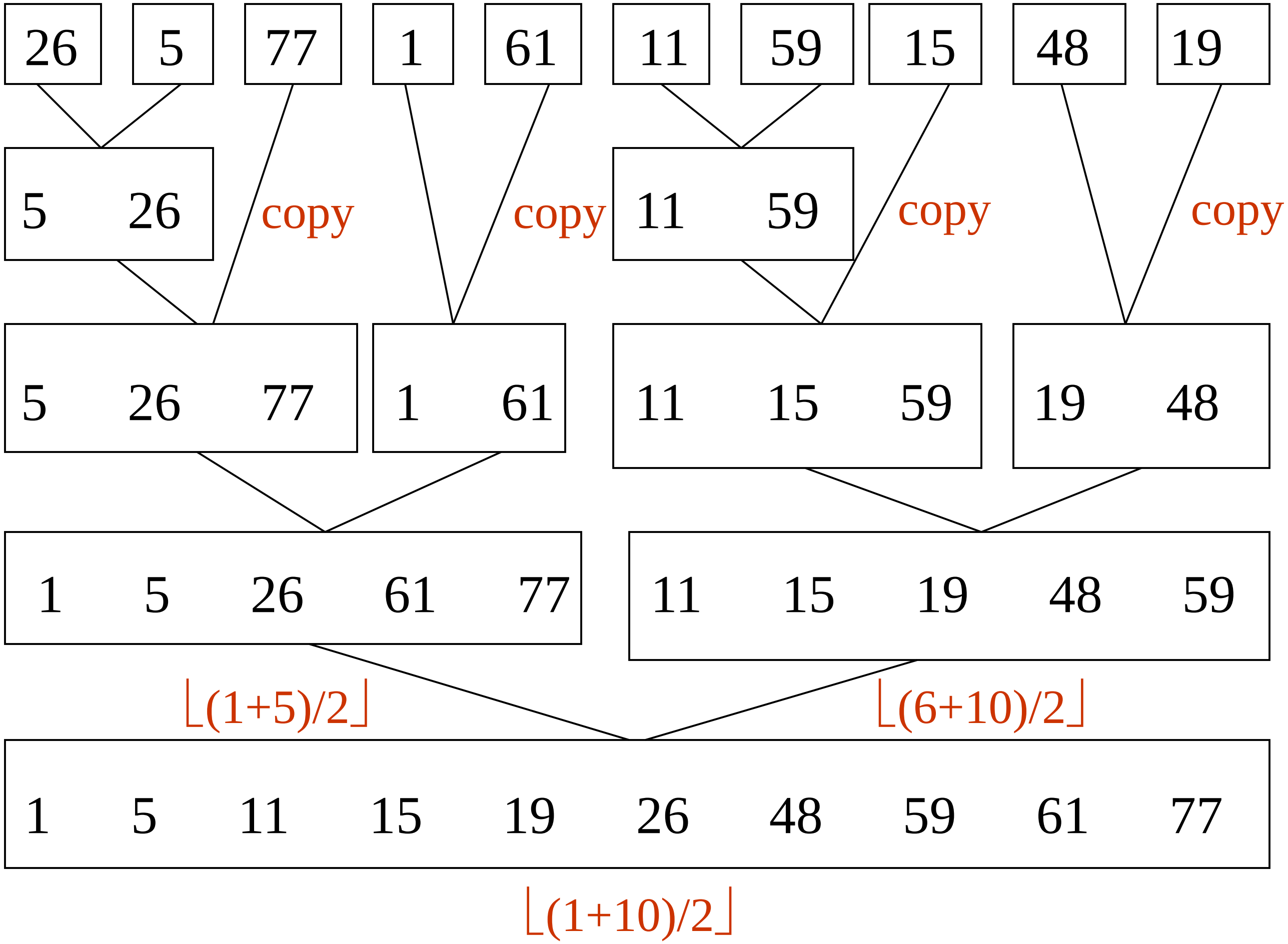
1 5 26 77 | 11 15 59 61 | 19 48

1 5 11 15 26 59 61 77 | 19 48

1 5 11 15 19 26 48 59 61 77

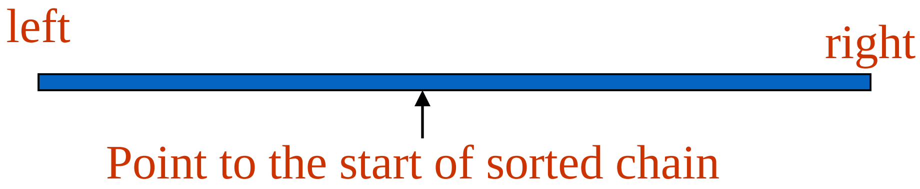
O(nlog$_2$n): $\lceil \log_2 n \rceil$ passes, O(n) for each pass

Recursive Formulation of Merge Sort

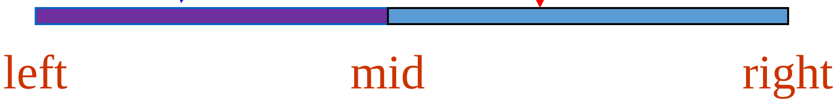


Data Structure: array (copy subfiles) vs. linked list (no copy)

Recursive Merge Sort



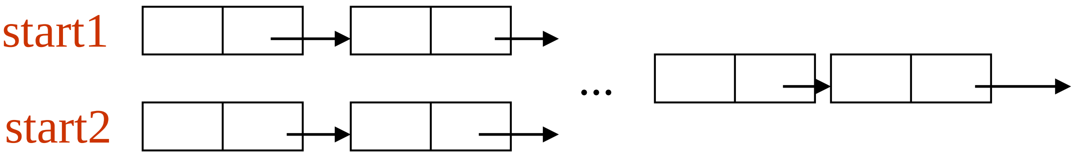
```
int rmergeSort(element a[], int link[],int
  left,int right)
{
  int mid;
  if (left >= right) return left;
  else {
    mid = (left+right)/2;
    return listMerge(a, link,
              rmergeSort(a,link,left,mid),
              rmergeSort(a,link,mid+1,right));
  }
}
```



ListMerge

```
int  listMerge(element  a[],  int  link[],  int  start1,  int
start2)
{
  /*兩個排序好的chains分別從start1及start2開始，將它們合併
將link[0]當作一個暫時的標頭。回傳合併好的chains的開頭。*/
 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];
            }
```

key in first lis
is lower, link th
element to start
and change start
point to first



```
/* 將其餘的記錄附接至最後的鏈結*/
if(last1 == 0) link[lastResult] = last2;
else link[lastResult] = last1;

return link[0];
}
```

first is exhausted.

second is exhausted.

$O(n\log_2 n)$

Heap Sort

***Figure 7.7:** Array interpreted as a binary tree

Index:	1	2	3	4	5	6	7	8	9	10
Number:	26	5	77	1	61	11	59	15	48	19

input file

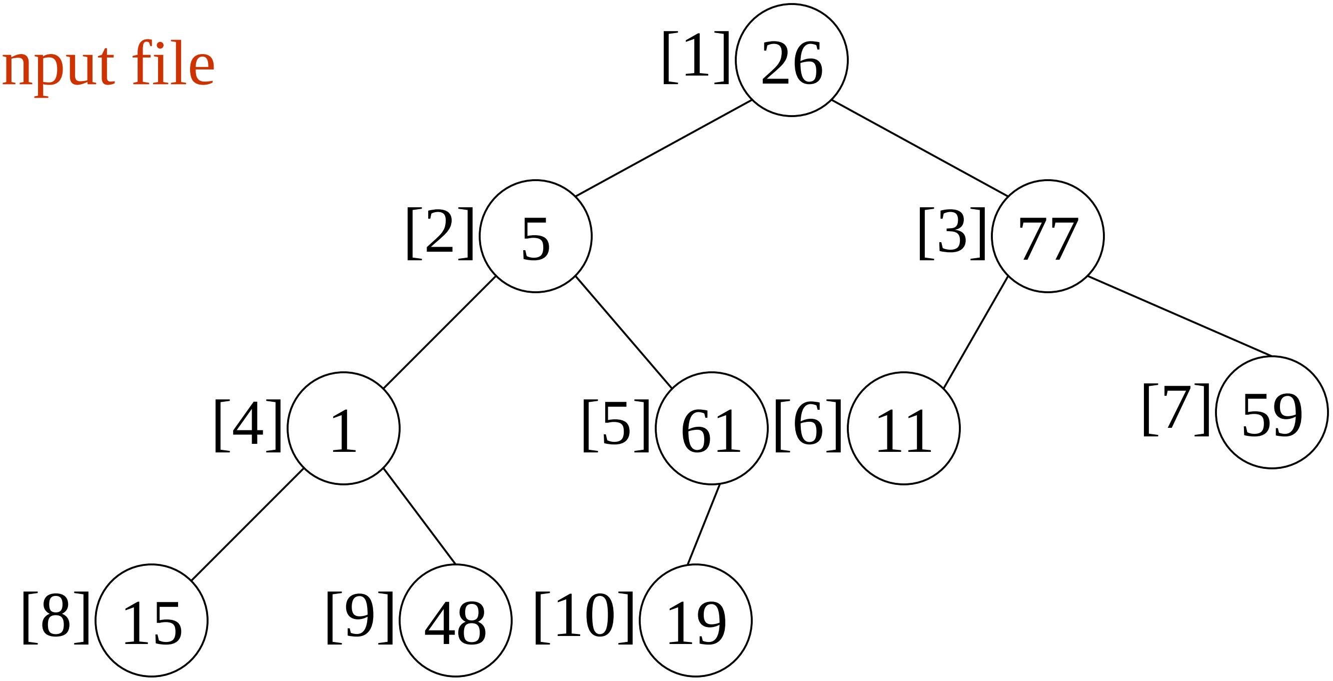


*Figure 7.7: Max heap following first **for** loop of *heapsort*

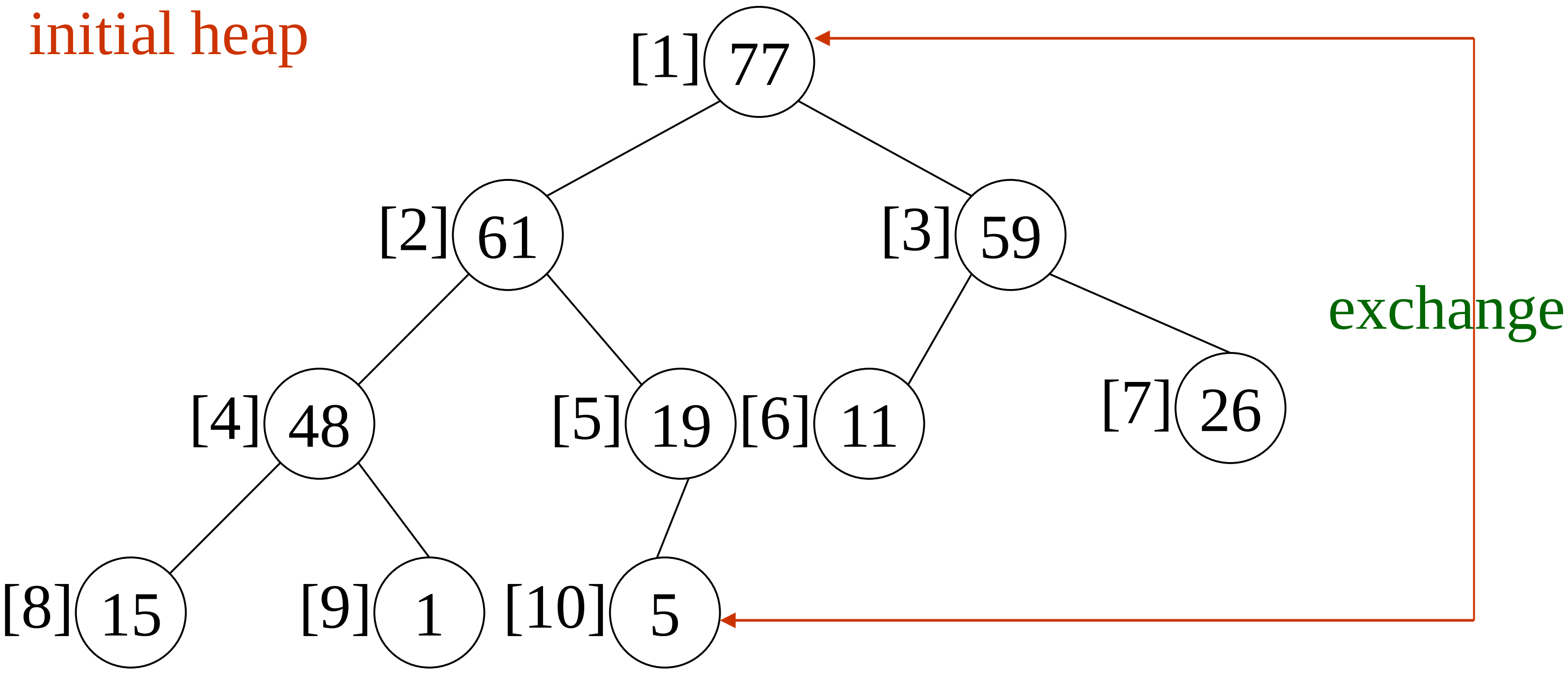


Figure 7.8: Heap sort example

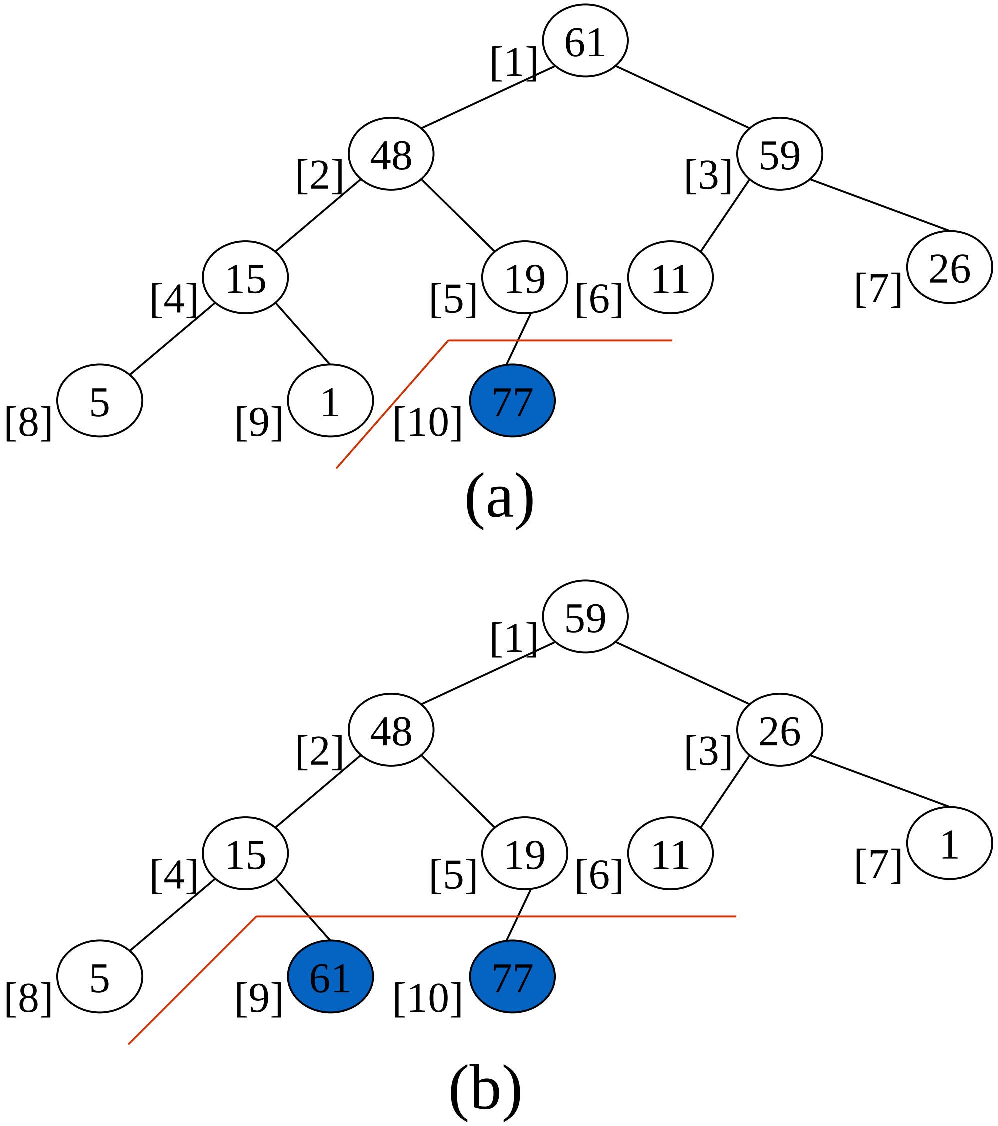
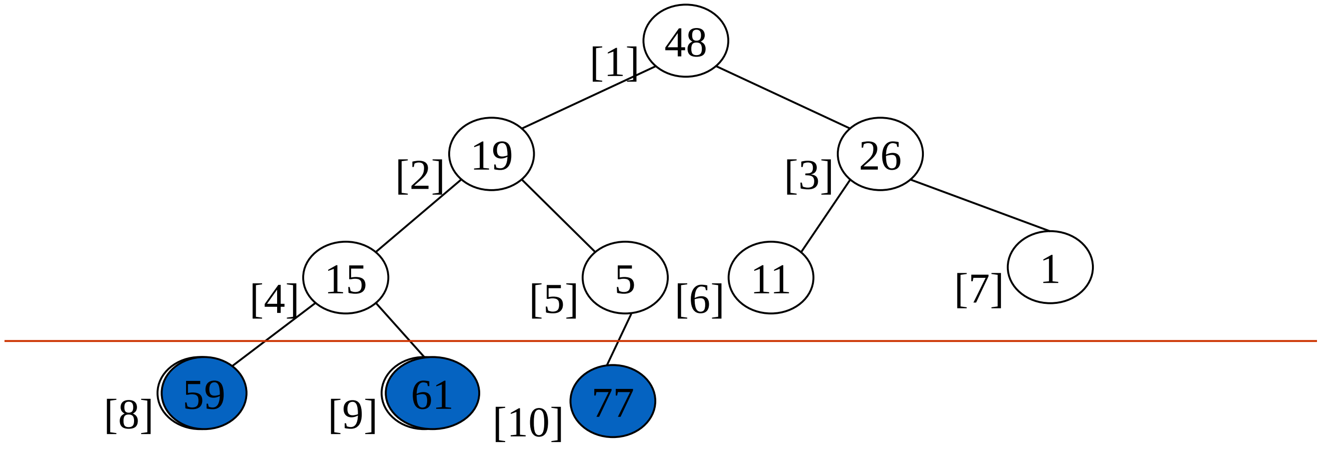
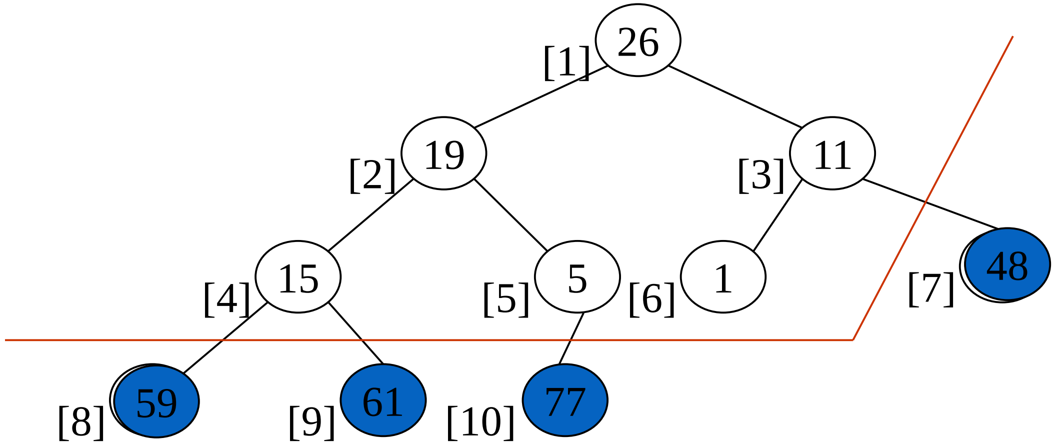


Figure 7.8(continued): Heap sort example



(c)

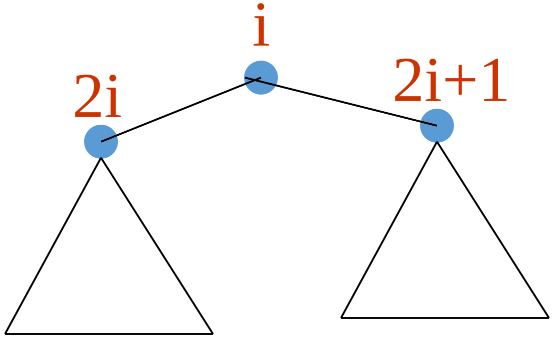


(d)

Heap Sort

```
void adjust(element a[], int root, int n)
{
  int child, rootkey;   element temp;
  temp=a[root];     rootkey=a[root].key;
  child=2*root; //左子樹
  while (child <= n) {
    if ((child < n) &&
        (a[child].key < a[child+1].key))
           child++;

    if (rootkey > a[child].key)
    /*比較樹根和最大子樹*/
      break;
    else {//move to parent
      a[child/2] = a[child];
      child *= 2;
    }
  }
  a[child/2] = temp;
}
```

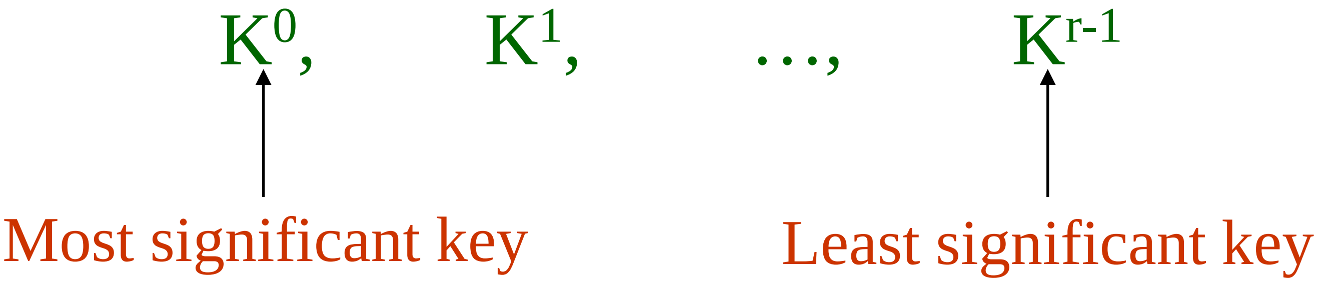


Heap Sort

```
void heapsort(element a[], int n)
{ ascending order (max heap)
    int i, j;
    element temp;              bottom-up
    for (i=n/2; i>0; i--)
       adjust(a, i, n);
    for (i=n-1; i>0; i--) { top-down  n-1 cylces
        SWAP(a[1], a[i+1], temp);
        adjust(a, 1, i);
    }
}
```

Radix Sort

Sort by keys



K^0, K^1, …, K^{r-1}

Most significant key

Least significant key

$R_0, R_1, \ldots, R_{n-1}$ are said to be sorted w.r.t. $K_0, K_1, \ldots, K_{r-1}$ iff

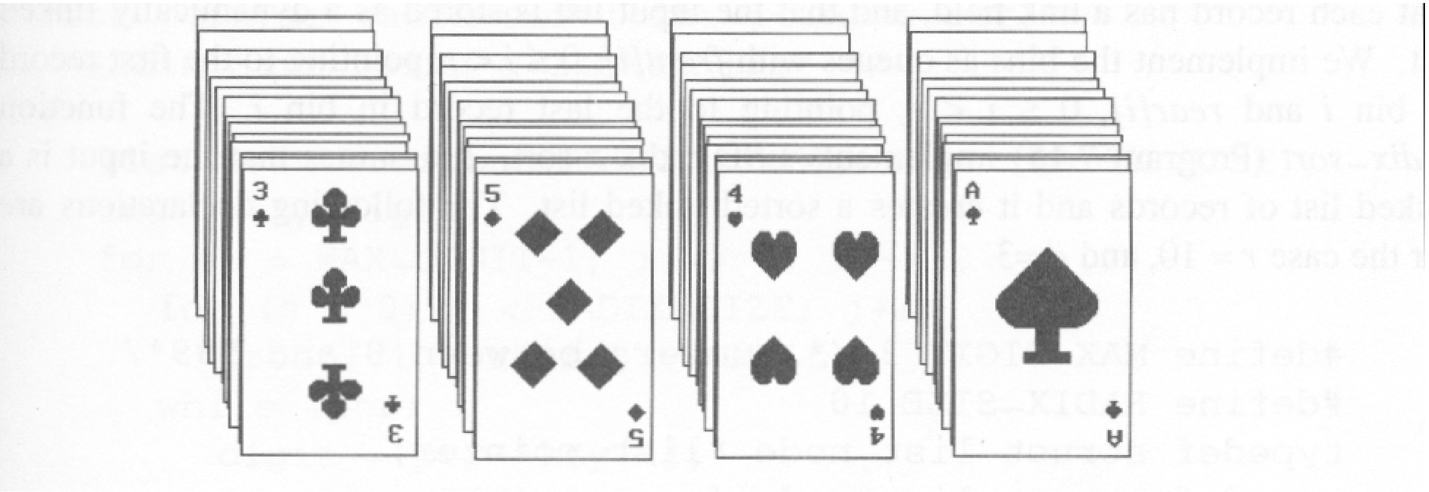
$$(k_i^0, k_i^1, \ldots, k_i^{r-1}) <= (k_{i+1}^0, k_{i+1}^1, \ldots, k_{i+1}^{r-1}) \quad 0 \le i < n-1$$

with respect to

Most significant digit first: sort on K^0, then K^1, ...

Least significant digit first: sort on K^{r-1}, then K^{r-2}, ...

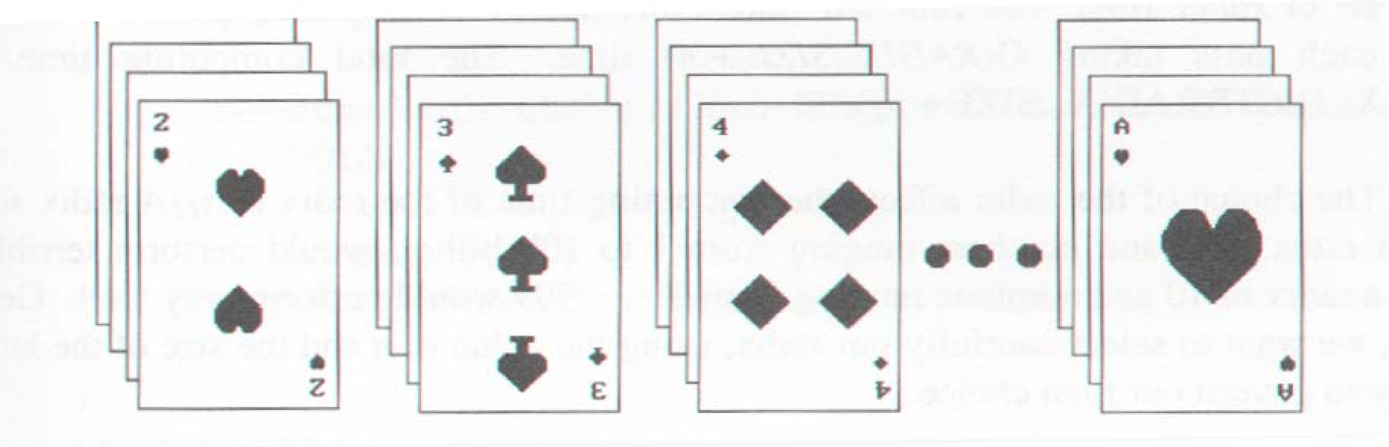
Figure 7.14: Arrangement of cards after first pass of an MSD sort



Suits: ♣ < ♦ < ♥ < ♠

Face values: 2 < 3 < 4 < … < J < Q < K < A

Figure 7.15: Arrangement of cards after first pass of LSD sort



(1) MSD sort first, e.g., bin sort, four bins ♣ ♦ ♥ ♠
LSD sort second, e.g., insertion sort

(2) LSD sort first, e.g., bin sort, 13 bins
2, 3, 4, …, 10, J, Q, K, A
MSD sort, e.g., bin sort four bins ♣ ♦ ♥ ♠

Radix Sort

$0 \leq K \leq 999$

$(K^0, \quad K^1, \quad K^2)$

MSD LSD

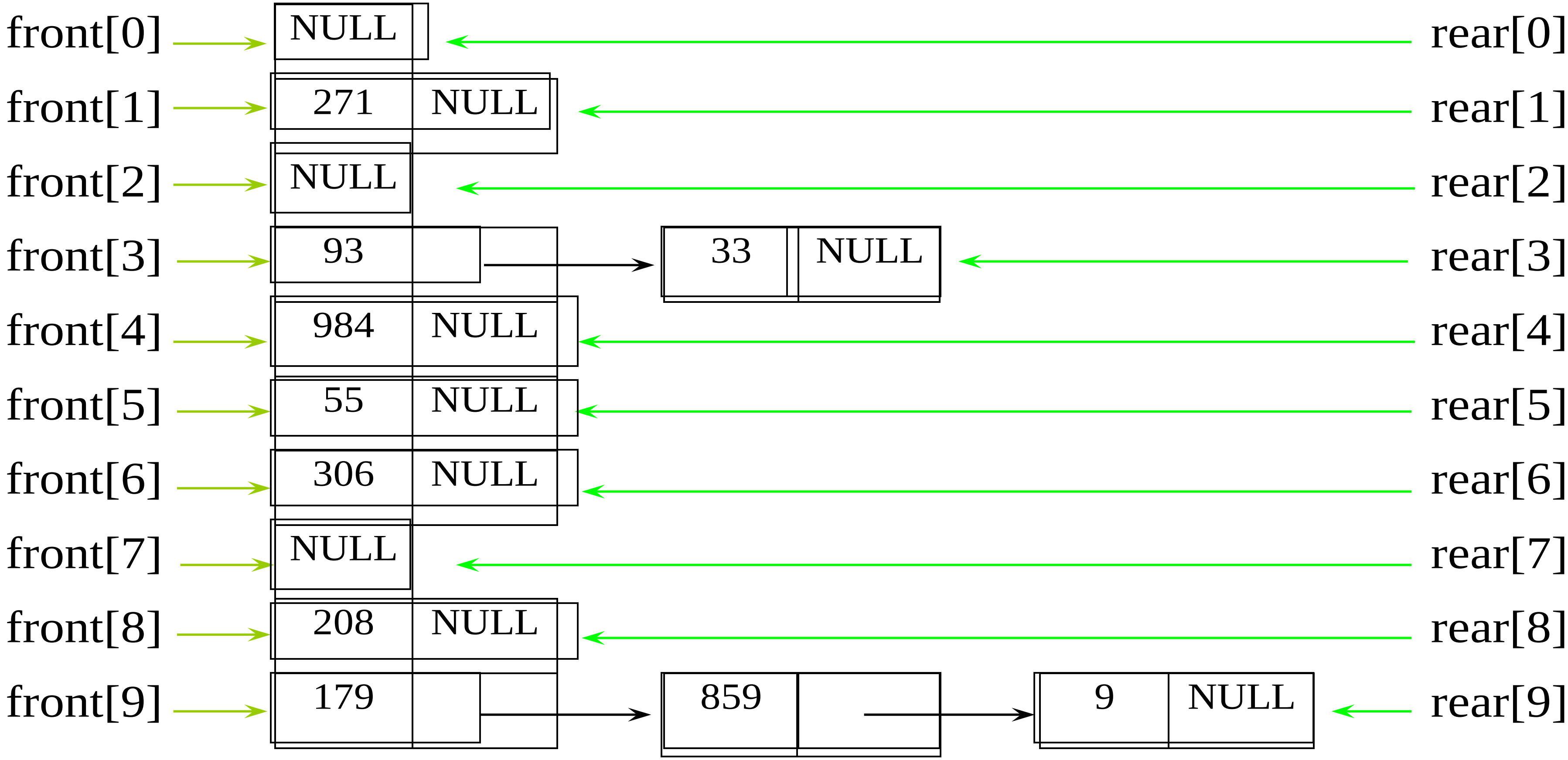
0-9 0-9 0-9

Example for LSD Radix Sort

d (digit) = 3, r (radix) = 10 ascending order

179, 208, 306, 93, 859, 984, 55, 9, 271, 33

Sort by digit



front[0] NULL rear[0]

front[1] 271 NULL rear[1]

front[2] NULL rear[2]

front[3] 93 33 NULL rear[3]

front[4] 984 NULL rear[4]

front[5] 55 NULL rear[5]

front[6] 306 NULL rear[6]

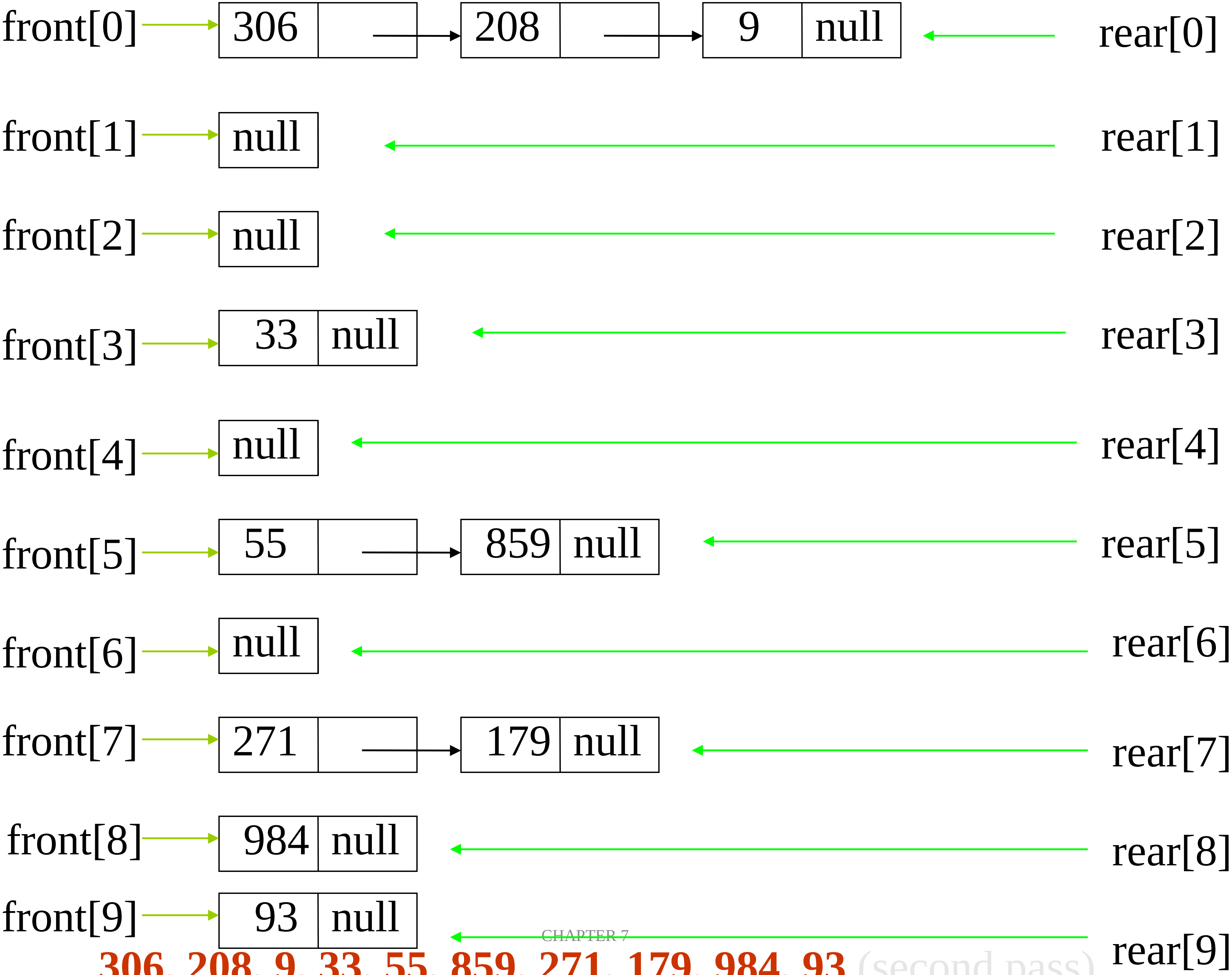
front[7] NULL rear[7]

front[8] 208 NULL rear[8]

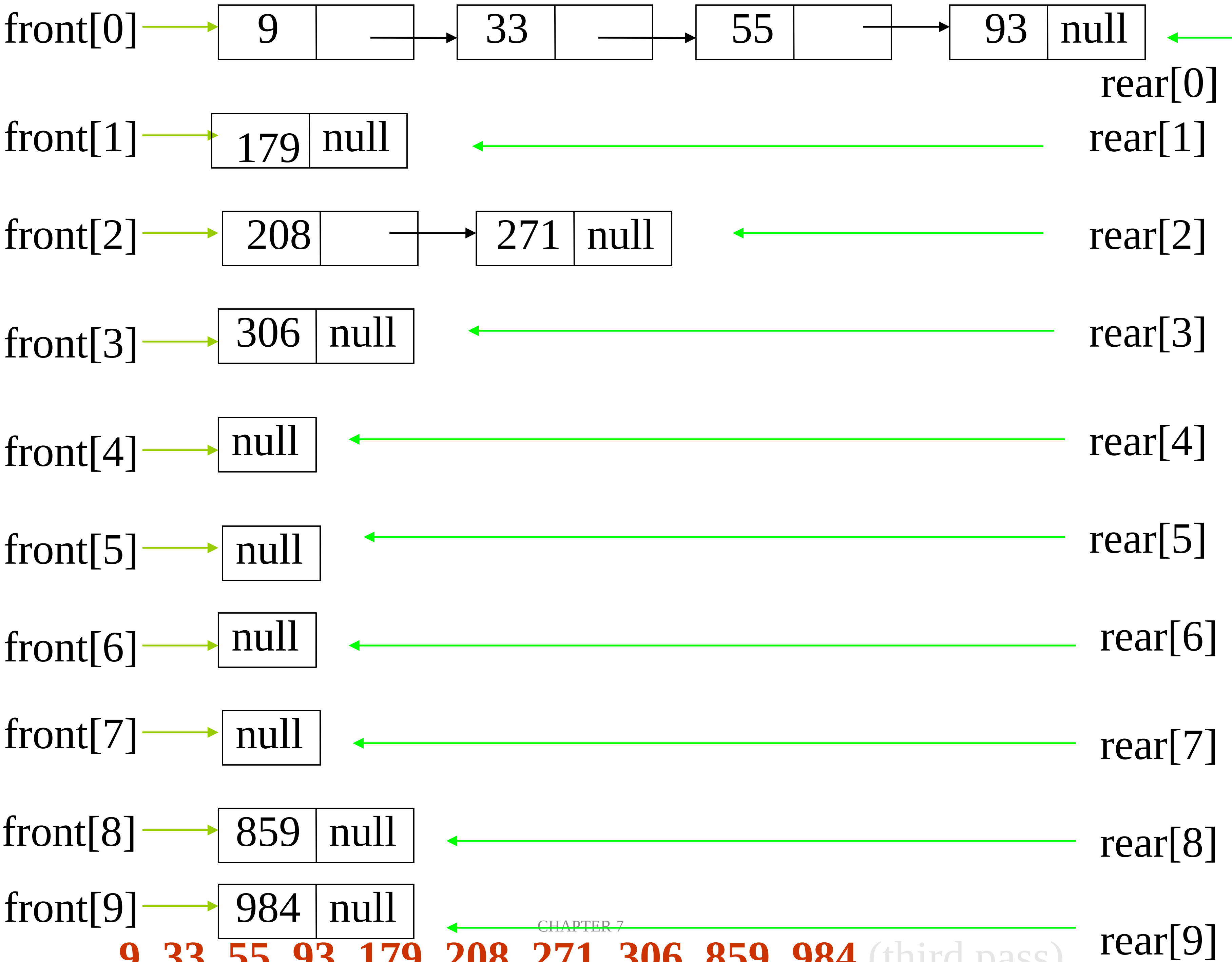
front[9] 179 859 9 NULL rear[9]

271, 93, 33, 984, 55, 306, 208, 179, 859, 9 After the first pass

concatenate



306, 208, 9, 33, 55, 859, 271, 179, 984, 93 (second pass)



9, 33, 55, 93, 179, 208, 271, 306, 859, 984 (third pass)

Data Structures for LSD Radix Sort

- An LSD radix *r* sort,
- R_0, R_1, ..., R_{n-1} have the keys that are *d*-tuples (x_0, x_1, ..., x_{d-1})

```
#define MAX_DIGIT 3
#define RADIX_SIZE 10
typedef struct list_node *list_pointer;
typedef struct list_node {
    int key[MAX_DIGIT];
    list_pointer link;
}
```

```
  int radixSort(element a[], int link[], int d,
int r, int n)
  {/* 利用一個d位數、基數r的基數排序法來排序a[1:n]
      digit(a[i], j, r) 回傳a[i]以r為基數的鍵值在第j
個位數(從左邊)。每一個位數的範圍是[0, r)，同一個位數內的
排序是使用容器排序法 */

  int front[r], rear[r]; /*佇列的開頭和結尾指標*/
  int i, bin, current, first, last;
  /*建立一個從first開始的記錄起始鏈*/
  first = 1;
  for( i = 1; i < n; i++) link[i] = i+1;
  link[n] = 0;
  for( i = d-1; i>=0; i--)
  {   /*根據第i位數來排序*/
       /*將容器初始化成空的佇列*/
      for(bin = 0; bin<r; bin++) front[bin] = 0;
```

```
    for(current = first; current; current = link[current])
     {      /*把記錄放到佇列/容器中*/
            bin = digit(a[current], i, r);
            if( front[bin] == 0) front[bin] = current;
            else link[rear[bin]] = current;
            rear[bin] = current;
     }
            /*找出第一個非空的佇列/容器*/
     for(bin = 0; !front[bin]; bin++);
     first = front[bin]; last = rear[bin];

            /*連接其餘的佇列*/
     for(bin++; bin < r; bin++)
            if(front[bin])
            {
              link[last] =front[bin]; last = rear[bin];}
              link[last] = 0;
            }
  return first;
 }
```

Practical Considerations for Internal Sorting

- Data movement
  - slow down sorting process
  - insertion sort and merge sort → linked file
- Perform a linked list sort + rearrange records

List and Table Sorts

- Many sorting algorithms require excessive data movement since we must physically move records following some comparisons
  - If the records are large, this slows down the sorting process
- We can reduce data movement by using a linked list representation

List and Table Sorts

- However, in some applications, we must physically rearrange the records so that they are in the required order

- We can achieve considerable savings by first performing a linked list sort and then physically rearranging the records according to the order specified in the list.

Rearranging Sorted Linked List (1)

Sorted linked list, first=4

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	26	5	77	1	61	11	59	15	48	19
linka	9	6	0	2	3	8	5	10	7	1

Add backward links to become a doubly linked list, first=4

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	26	5	77	1	61	11	59	15	48	19
linka	9	6	0	2	3	8	5	10	7	1
linkb	10	4	5	0	7	2	9	6	1	8

Rearranging Sorted Linked List (2)

R_1 is in place. first=2

i	**R_1**	R_2	R_3	**R_4**	R_5	R_6	R_7	R_8	R_9	R_{10}
key	**1**	5	77	**26**	61	11	59	15	48	19
linka	**2**	6	0	**9**	3	8	5	10	7	**4**
linkb	**0**	4	5	**10**	7	2	9	6	**4**	8

R_1, R_2 are in place. first=6

i	R_1	**R_2**	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	1	**5**	77	26	61	11	59	15	48	19
linka	2	**6**	0	9	3	8	5	10	7	4
linkb	0	**4**	5	10	7	2	9	6	4	8

Rearranging Sorted Linked List (3)

R_1, R_2, R_3 are in place. first=8

i	R_1	R_2	**R_3**	R_4	R_5	**R_6**	R_7	R_8	R_9	R_{10}
key	1	5	**11**	26	61	**77**	59	15	48	19
linka	2	6	**8**	9	**6**	**0**	5	10	7	4
linkb	0	4	**2**	10	7	**5**	9	**3**	4	8

R_1, R_2, R_3, R_4 are in place. first=10

i	R_1	R_2	R_3	**R_4**	R_5	R_6	R_7	**R_8**	R_9	R_{10}
key	1	5	11	**15**	61	77	59	**26**	48	19
linkb	2	6	8	**10**	6	0	5	**9**	7	**8**
linkb	0	4	2	**3**	7	5	9	**10**	**8**	8

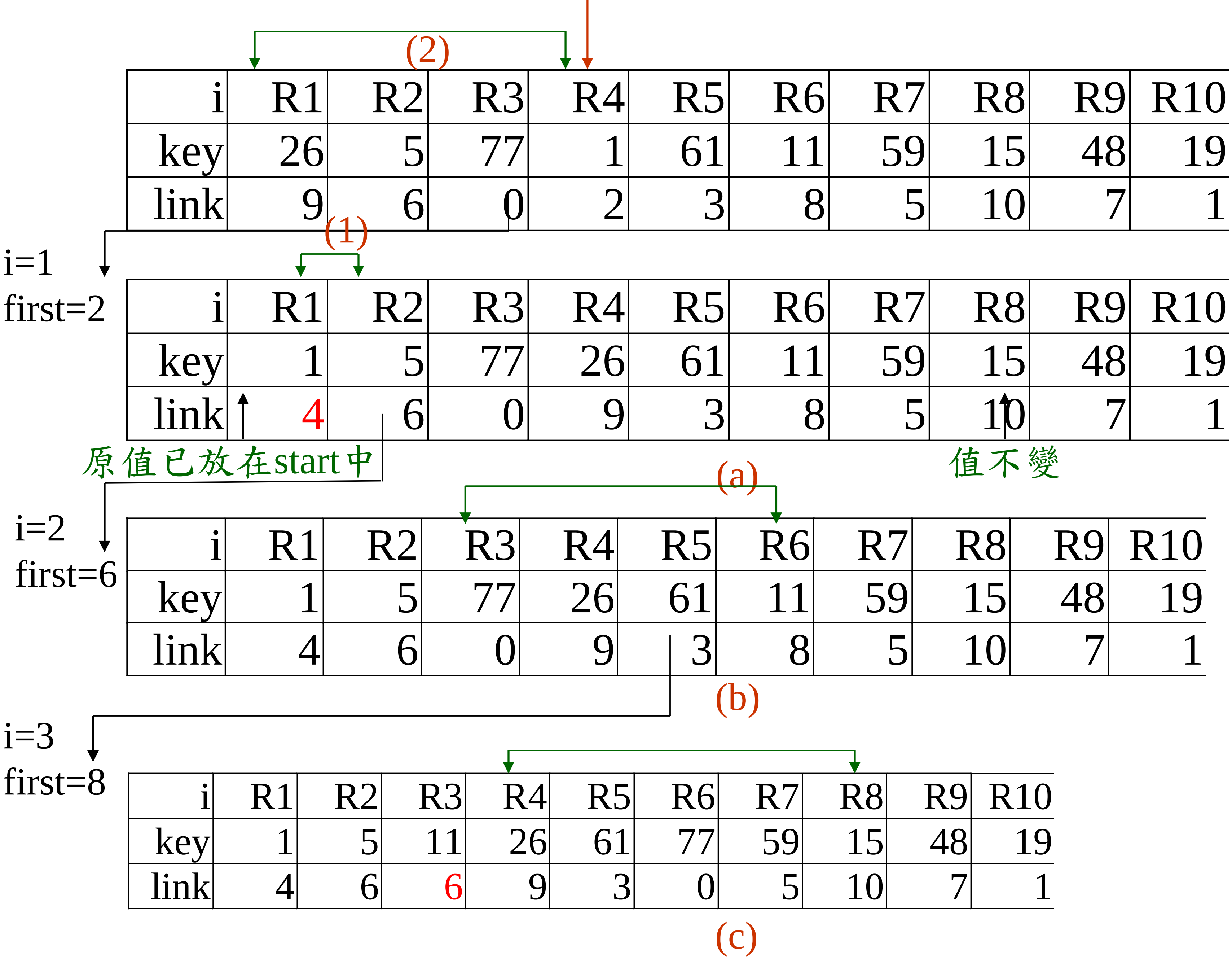
Algorithm for List Sort

```
void listSort1(element a[], int linka[], int n, int first)
{
/* 重新排列從first開始已排序的鍊，使得記錄a[1:n]為已排序的順序 */
   int linkb[MAX_SIZE]; /* 反向鏈結陣列 */
   int i, current, prev = 0;
   element temp;
   for (current = first; current; current = linka[current])
      {/* 轉換鍊為雙向鏈結串列 */
         linkb[current] = prev;
         prev = current;
      }

   for (i = 1; i < n ; i++)
   /* 當維護串列時移動a[first]到位置i */
      {
         if (first != i) {
            if (linka[i])
            linkb[linka[i]] = first;
            linka[linkb[i]] = first;
            SWAP(a[first], a[i], temp);
            SWAP(linka[first], linka[i], temp);
            SWAP(linkb[first], linkb[i], temp);
         }
         first = linka[i];
      }
}
```

Convert start into a doubly lined list

O(mn)



(2)

i	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
key	26	5	77	1	61	11	59	15	48	19
link	9	6	0	2	3	8	5	10	7	1

(1)

i=1
first=2

i	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
key	1	5	77	26	61	11	59	15	48	19
link	4	6	0	9	3	8	5	10	7	1

原值已放在start中

值不變

(a)

i=2
first=6

i	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
key	1	5	77	26	61	11	59	15	48	19
link	4	6	0	9	3	8	5	10	7	1

(b)

i=3
first=8

i	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
key	1	5	11	26	61	77	59	15	48	19
link	4	6	6	9	3	0	5	10	7	1

(c)

i=4
first=10

i	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
key	1	5	11	15	61	77	59	26	48	19
link	4	6	6	8	3	0	5	9	7	1

值不變

(d)

i=5
first=1

i	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
key	1	5	11	15	19	77	59	26	48	61
link	4	6	6	8	10	0	5	9	7	3

(e)

```
void listSort2(element a[], int linka[], int n, int first)
{
 /* 與list1相同的函式，除了不需要第二個鏈結陣列linkb以外 */
   int i;
   element temp;
   for (i = 1; i < n ; i++)
     {
      /*尋找在第i個位置上正確的記錄，它的索引值 ≥i因為在位置
      1, 2, …, i-1位置上的記錄已經放在正確的位置上 */
        while (first < i) first = link[first];
         int q = link[first];
       /* a[q]是下一筆將被排到正確位置的記錄 */
         if (first != i)
            {
  /*a[first]的鍵值是第i小的，並將與a[i]互換以將鏈結值更新*/
            SWAP(a[i], a[first], temp);
            link[first] = link[i];
            link[i] = first;
           }
           first = q;
     }
}
```

Table Sort

- The list-sort technique is not well suited for quick sort and heap sort.
- One can maintain an auxiliary table, *t*, with one entry per record, an indirect reference to the record.
- Initially, *t*[*i*] = *i*. When a swap are required, only the table entries are exchanged.
- After sorting, the list a[t[1]], a[t[2]], a[t[3]]…are sorted.
- Table sort is suitable for all sorting methods.

Permutation Cycle

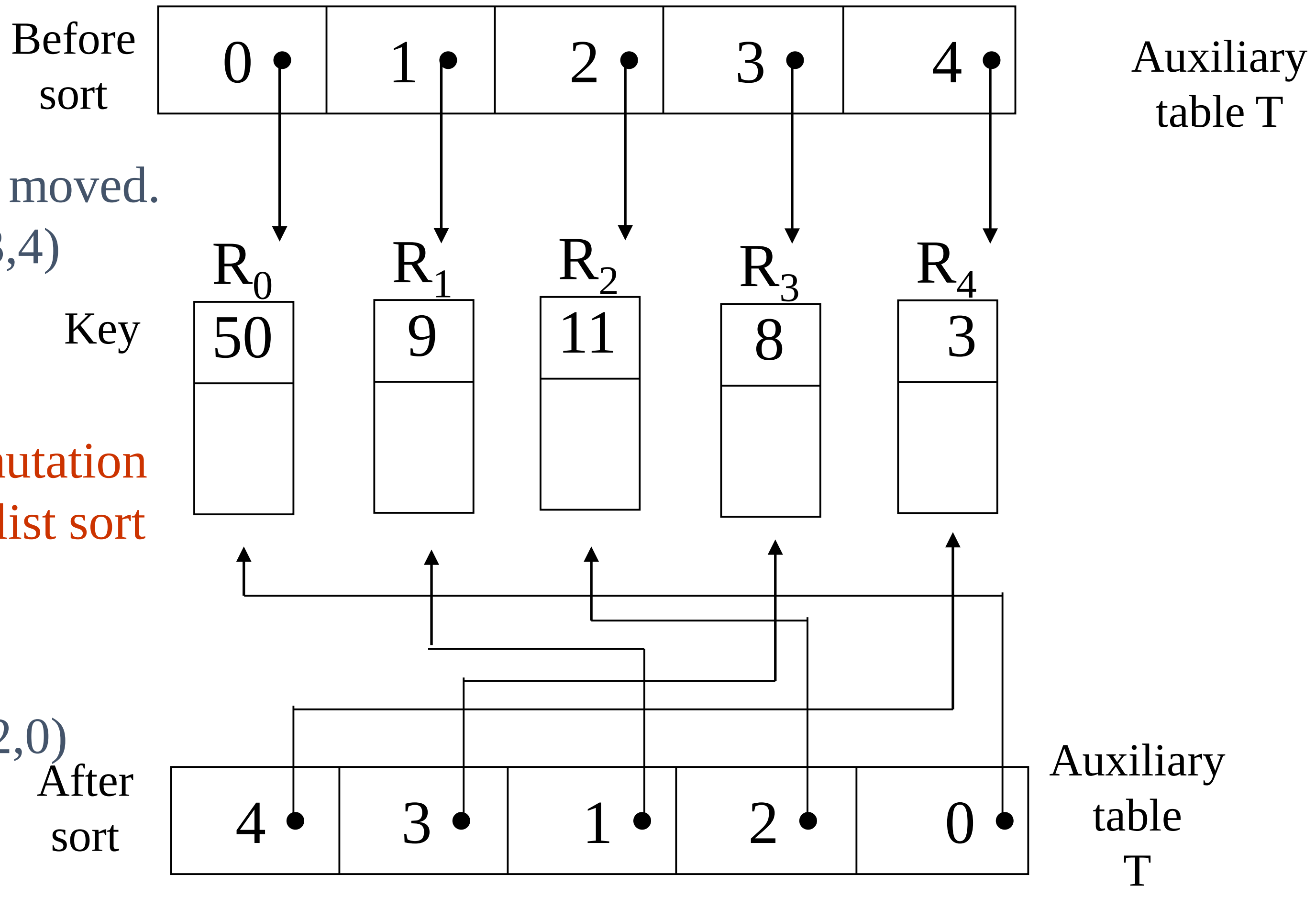
- After sorting (nondecreasing):

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
key	35	14	12	42	26	50	31	18
t	3	2	8	5	7	1	4	6

- Permutation [3 2 8 5 7 1 4 6]
- Every permutation is made up of disjoint permutation cycles:
  - (1, 3, 8, 6) nontrivial cycle
    - R1 now is in position 3, R3 in position 8, R8 in position 6, R6 in position 1.
  - (4, 5, 7) nontrivial cycle
  - (2) trivial cycle

Table Sort

Data are not moved.
(0,1,2,3,4)



Before sort

0	1	2	3	4

Auxiliary table T

	R_0	R_1	R_2	R_3	R_4
Key	50	9	11	8	3

A permutation
linked list sort

(4,3,1,2,0)

After sort

4	3	1	2	0

Auxiliary
table
T

第0名在第4個位置，第1名在第3個位置，…，第4名在第0個位置

(0,1,2,3,4)

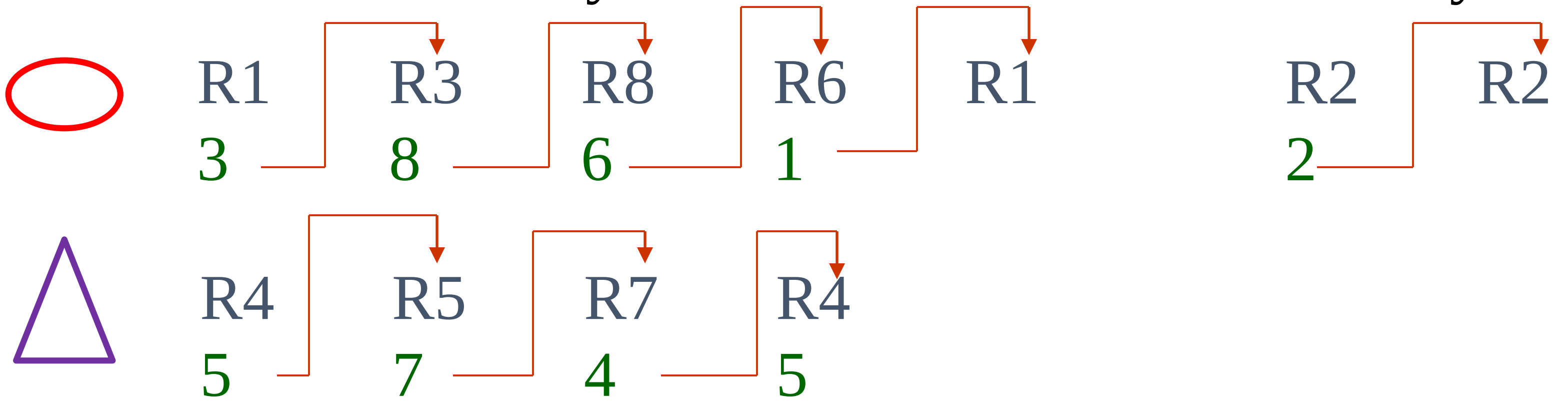
Every permutation is made of disjoint cycles.

Example

	R1	R2	R3	R4	R5	R6	R7	R8
key	35	14	12	42	26	50	31	18
table	3	2	8	5	7	1	4	6

two nontrivial cycles

trivial cycle

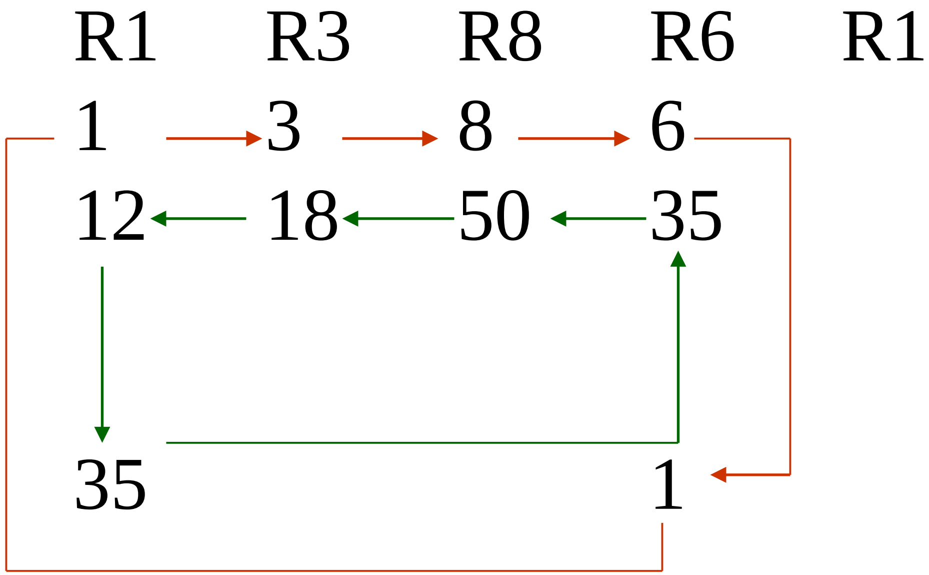


R1 3 → R3 8 → R8 6 → R6 1 → R1

R4 5 → R5 7 → R7 4 → R4 5

R2 2 → R2

(1)



R1 R3 R8 R6 R1

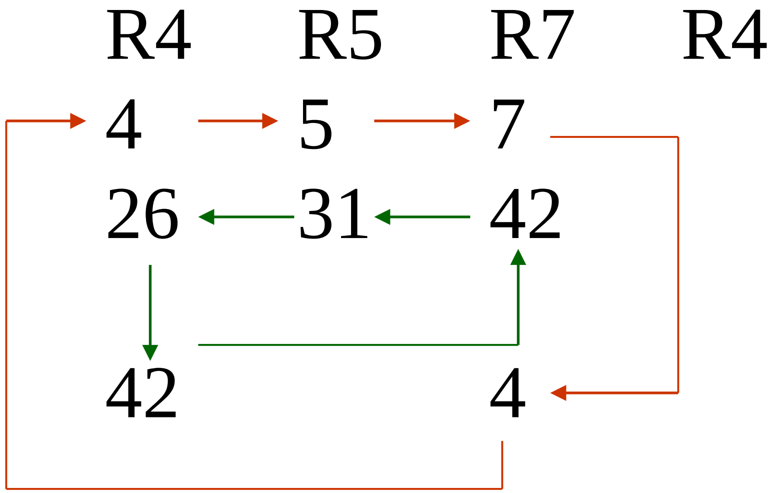
1 3 8 6

12 18 50 35

35 1

(2) i=1,2 t[i]=i

(3)

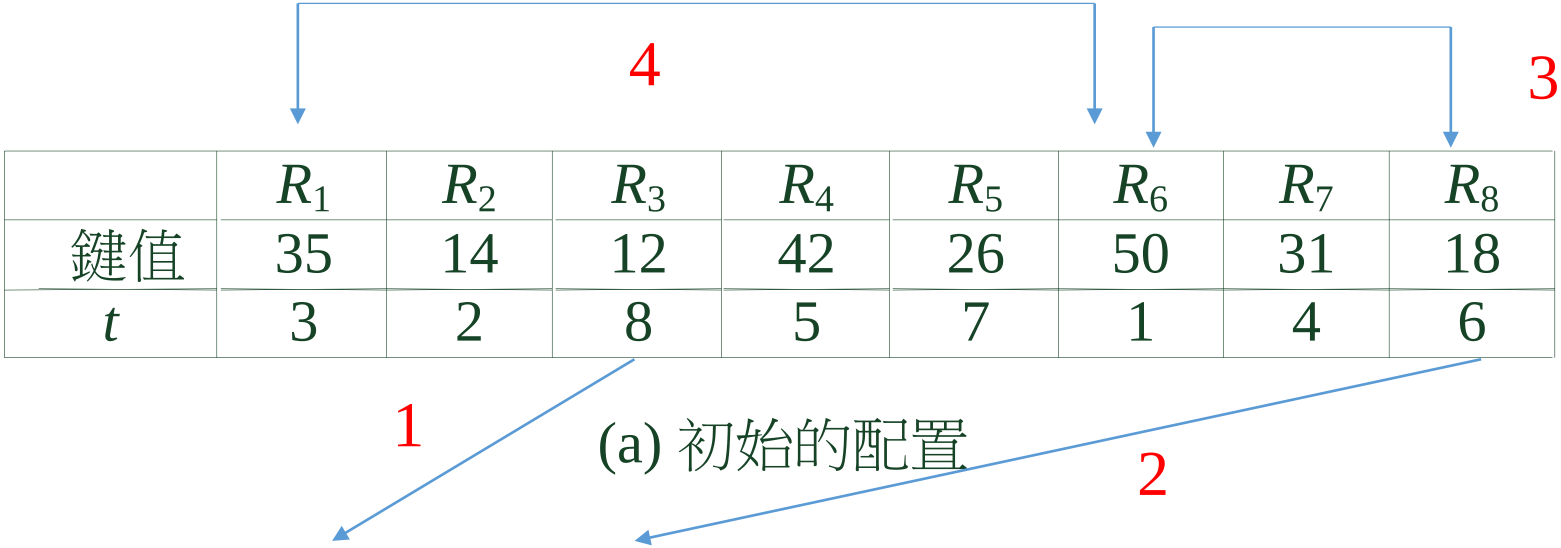


R4 R5 R7 R4

4 5 7

26 31 42

42 4



	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
鍵值	35	14	12	42	26	50	31	18
t	3	2	8	5	7	1	4	6

(a) 初始的配置

鍵值	12	14	18	42	26	35	31	50
t	1	2	3	5	7	6	4	8

(b) 第一個迴路重新排列後的配置

鍵值	12	14	18	26	31	35	42	50
t	1	2	3	4	5	6	7	8

(c) 第二個迴路重新排列後的配置

Table Sort

```
void tableSort(element a[], int n, int t[])
{ /* 重新排列a[1:n]成序列a[t[1]], …, a[t[n]] */
   int i, current, next;
   element temp;
   for (i = 1; i < n ; i++)
     if (t[i] != i)
      { /* 從i開始的非瑣碎迴路 */
        temp = a[i]; current = i;
        do {
          next = t[current];
          a[current] = a[next];
          t[current] = current;
          current = next;
        } while (t[current] != i);
        a[current] = temp;
        t[current] = current;
      }
}
```

形成cycle

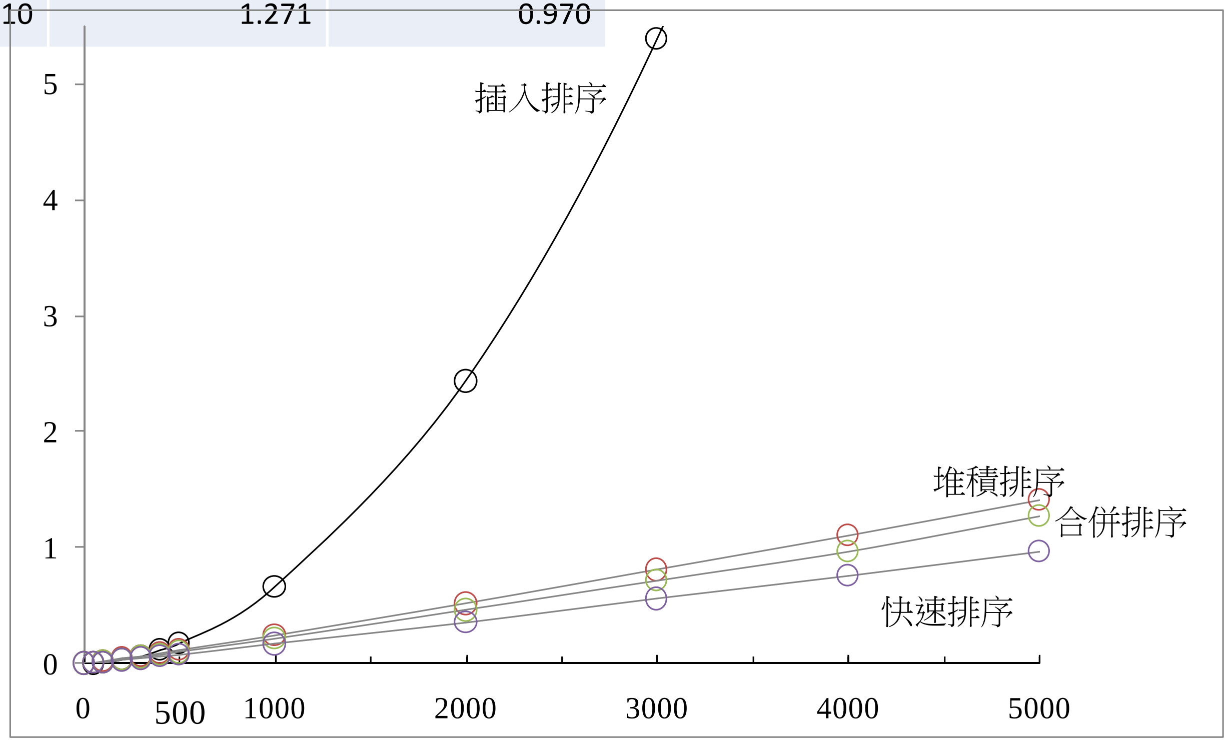
Summary of Internal Sorting

- No one method is best under all circumstances.
  - Insertion sort is good when the list is already partially ordered. And it is the best for small *n*.
  - Merge sort has the best worst-case behavior but needs more storage than heap sort.
  - Quick sort has the best average behavior, but its worst-case behavior is O(n^2).
  - The behavior of Radix sort depends on the size of the keys and the choice of *r*.

Complexity of Sort

	stability	space	time		
			best	average	worst
Bubble Sort	stable	little	O(n)	O(n2)	O(n2)
Insertion Sort	stable	little	O(n)	O(n2)	O(n2)
Quick Sort	untable	O(logn)	O(nlogn)	O(nlogn)	O(n2)
Merge Sort	stable	O(n)	O(nlogn)	O(nlogn)	O(nlogn)
Heap Sort	untable	little	O(nlogn)	O(nlogn)	O(nlogn)
Radix Sort	stable	O(np)	O(nlogn)	O(nlogn)	O(nlogn)
List Sort	?	O(n)	O(1)	O(n)	O(n)
Table Sort	?	O(n)	O(1)	O(n)	O(n)

n	插入	堆積	合併	快速
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



External Sorting

- The lists to be sorted are too large to be contained totally in the internal memory.
  - So internal sorting is impossible.
- The list (or file) to be sorted resides on a disk.
- Block: unit of data read from or written to a disk at one time. A block generally consists of several records.
- Read/write time of disks:
  - seek time (搜尋時間)：把讀寫頭移到正確磁軌(track, cylinder)
  - latency time (延遲時間)：把正確的磁區(sector)轉到讀寫頭下
  - transmission time (傳輸時間)：把資料區塊傳入/讀出磁碟

Merge Sort as External Sorting

- The most popular method for sorting on external storage devices is merge sort.
- Phase 1: Obtain sorted runs (segments) by internal sorting methods, such as heap sort, merge sort, quick sort or radix sort. These sorted runs are stored in external storage.
- Phase 2: Merge the sorted runs into one run with the merge sort method.

File: 4500 records, A1, …, A4500

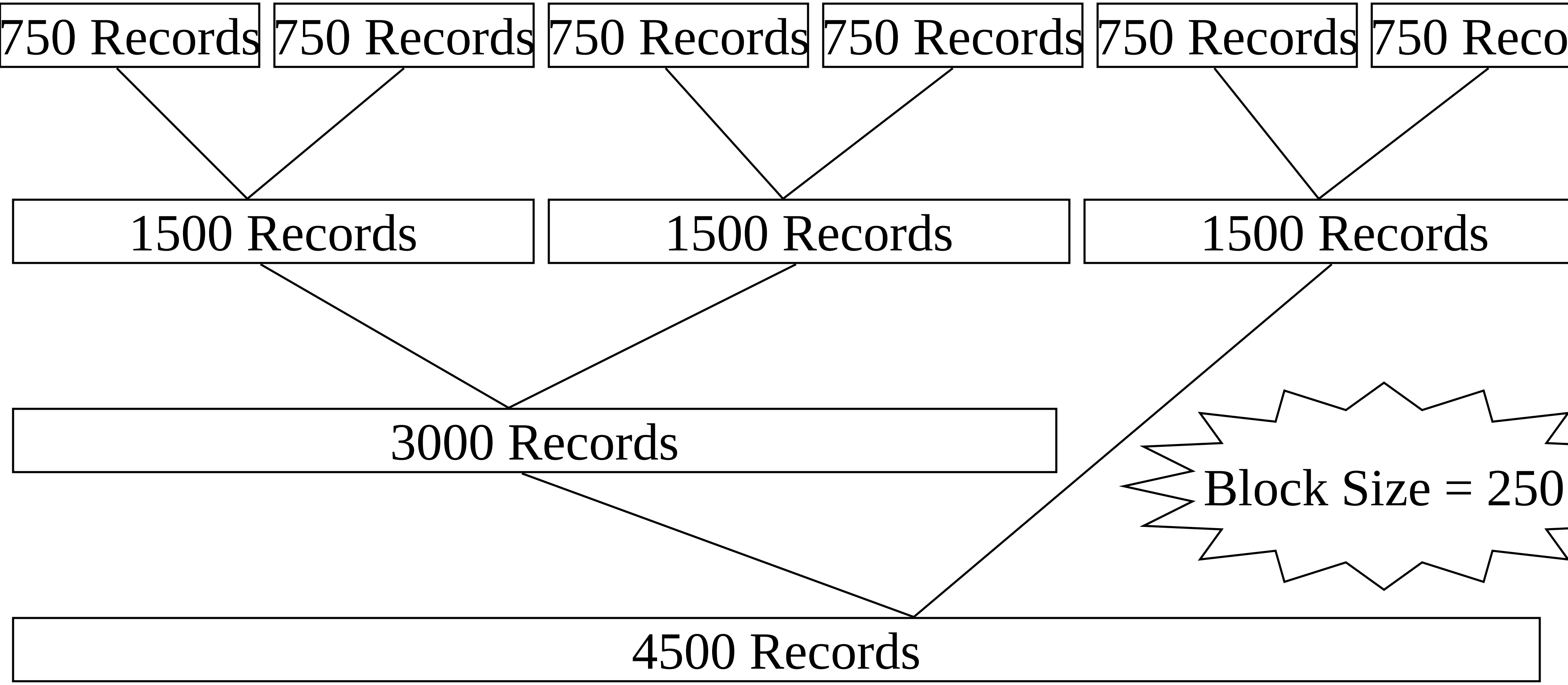
internal memory: 750 records (3 blocks)

block length: 250 records

input disk vs. scratch pad (disk)

(1) sort three blocks at a time and write them out onto scratch pad

(2) three blocks: two input buffers & one output buffer



750 Records 750 Records 750 Records 750 Records 750 Records 750 Reco

1500 Records 1500 Records 1500 Records

3000 Records

Block Size = 250

4500 Records

2 2/3 passes

Time Complexity of External Sort

input/output time

- t_s = maximum seek time
- t_l = maximum latency time
- t_{rw} = time to read/write one block of 250 records

$t_{IO} = t_s + t_l + t_{rw}$

cpu processing time

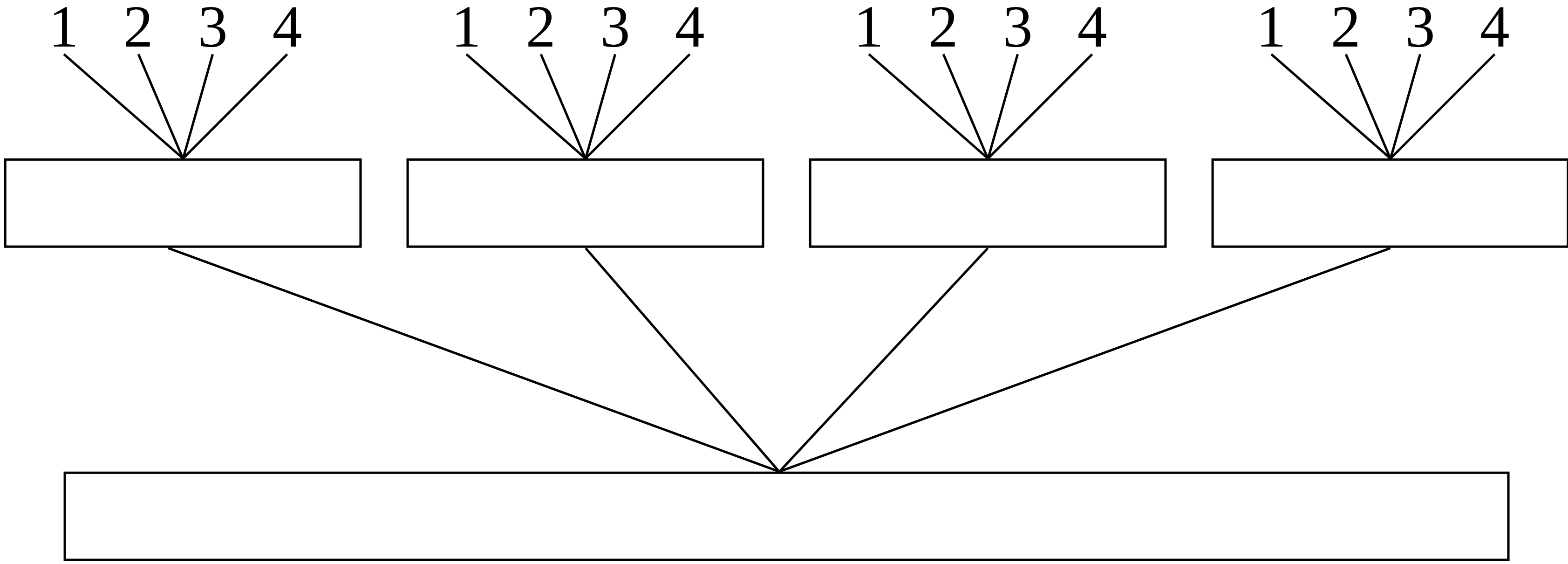
- t_{IS} = time to internally sort 750 records
- nt_m = time to merge n records from input buffers to the output buffer

運算	時間
(1)從輸入讀取18個區塊18t_{IO}，內部排序 6t_{IS}，寫入18個區塊18t_{IO}。	$36t_{IO} + 6t_{IS}$
(2) 兩兩合併串連1到6 (總共18個區塊, 4500記錄)	$36t_{IO} + 4500t_m$
(3) 合併兩個1500筆記錄，共12個區塊	$24t_{IO} + 3000t_m$
(4) 把一個有3000筆記錄的串連和一個有1500筆記錄的串連合併在一起	$36t_{IO} + 4500t_m$
總時間	$132t_{IO} + 12{,}000t_m + 6t_{IS}$

Consider Parallelism

- Carry out the CPU operation and I/O operation in parallel
- 132 t_{IO} = 12000 t_m + 6 t_{IS}
- Two disks: 132 t_{IO} is reduced to 66 t_{IO}

K-Way Merging



A 4-way merge on 16 runs

2 passes (4-way) vs. 4 passes (2-way)

Analysis

- $\lceil \log_k m \rceil$ passes
- 1 pass: $\lceil n\log_2 k \rceil$
- I/O time vs. CPU time
  - reduction of the number of passes being made over the data
  - efficient utilization of program buffers so that input, output and CPU processing is overlapped as much as possible
  - run generation
  - run merging

$O(n\log_2 k*\log_k m)$