Chapter 7
Sorting

Yi-Fen Liu

Department of IECS, FCU

References:
- E. Horowitz, S. Sahni and S. Anderson-Freed, *Fundamentals of Data Structures (2nd Edition)*
- Slides are credited from Prof. Chung, NTHU

Outline

- Introduction
  - Searching and List Verification
  - Definitions
- Insertion Sort
- Quick Sort
- Merge Sort
  - Merge
  - Iterative / Recursive Merge Sort
- Heap Sort
- Radix Sort
- Summary of Internal Sorting

INTRODUCTION

Introduction (1)

- Why efficient sorting methods are so important ?
- The efficiency of a searching strategy depends on the assumptions we make about the arrangement of records in the list
- No single sorting technique is the “best” for all initial orderings and sizes of the list being sorted.
- We examine several techniques, indicating when one is superior to the others.

Introduction (2)

- Sequential search
  - We search the list by examining the key values *list*[*0*].*key*, … , *list*[*n*-*1*].*key*.
    - Example: *List* has *n* records.
      4, 15, 17, 26, 30, 46, 48, 56, 58, 82, 90, 95
  - In that order, until the correct record is located, or we have examined all the records in the list
  - Unsuccessful search: $n+1 \Rightarrow O(n)$
  - Average successful search

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

Introduction (3)

- Binary search
  - Binary search assumes that the list is ordered on the key field such that *list*[0].*key* ≤ *list*[1]. *key* ≤ … ≤ *list*[*n*-1]. *key*.
  - This search begins by comparing *searchnum* (search key) and *list*[*middle*].*key* where *middle*=(*n*-1)/2

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

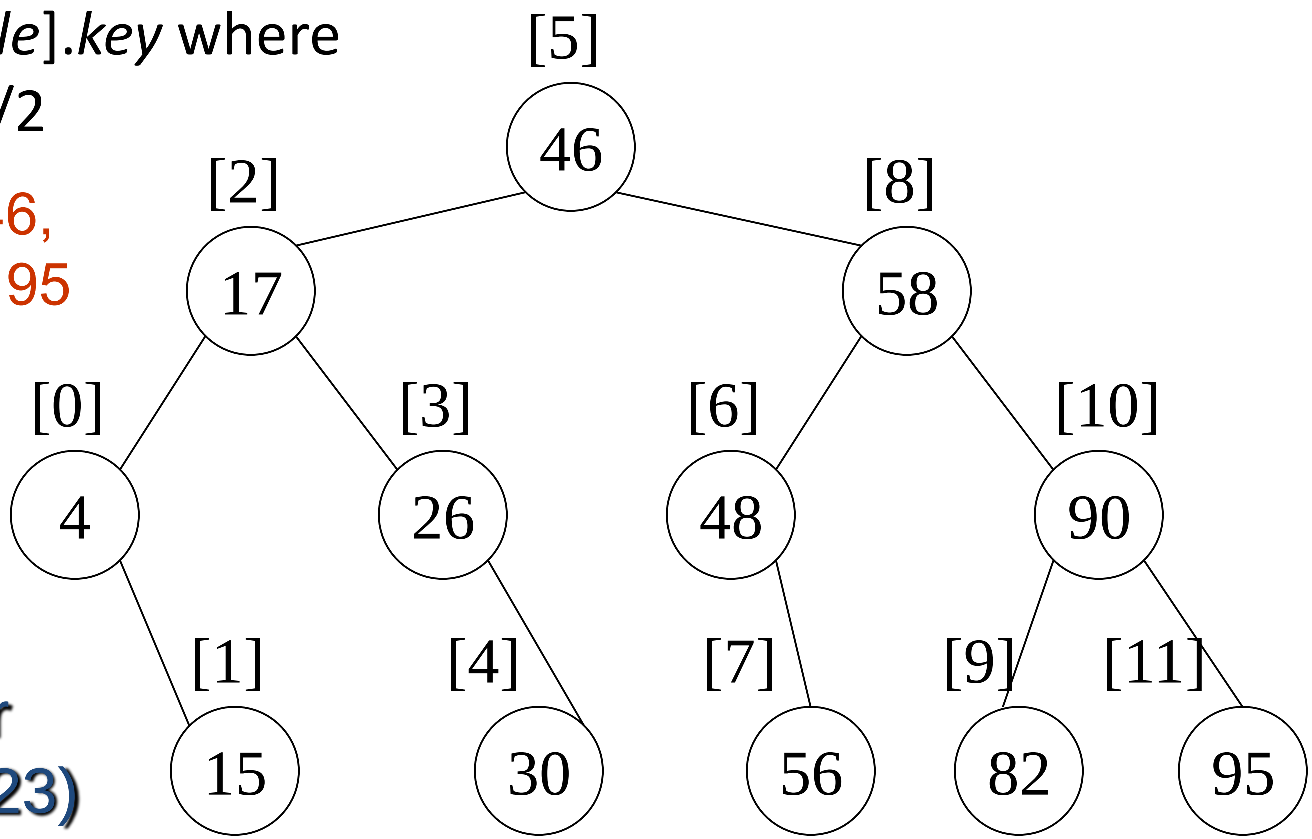


Figure 7.1: Decision tree for binary search (p.323)

Introduction (4)

- Binary search (cont'd)
  - **Analysis of *binsearch*:** makes no more than O(log *n*) comparisons

```
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;
                   break;
         case 0 : return middle;
         case 1 : right = middle - 1;
      }
   }
   return - 1;
}
```

Program 7.2: Binary search

Introduction (5)

- List Verification
  - Compare lists to verify that they are identical or identify the discrepancies.
- Example
  - International Revenue Service (IRS) (e.g., employee vs. employer)
- Reports three types of errors:
  - all records found in *list*1 but not in *list*2
  - all records found in *list*2 but not in *list*1
  - all records that are in *list*1 and *list*2 with the same key but have different values for different fields

Introduction (6)

- Verifying using a sequential search

Check whether the elements in list1 are also in list2

And the elements in list2 but not in list1 would be show up

```
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 */
         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: Verifying using a sequential search

Introduction (7)

- Fast verification of two lists

The element of list1 is not in list2

The element of two lists are matched

The element of list2 is not in list1

The remainder elements of a list is not a member of another list

```
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++;
     }
     else if (list1[i].key == list2[j].key) {
     /* compare list1[i] and list2[j] on each of the other
     fields 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);
}
```

Program 7.4: Fast verification of two lists

Introduction (8)

- Complexities
  - Assume the two lists are randomly arranged
  - Verify1: $O(mn)$
  - Verify2: sorts them before verification
    $O(tsort(n) + tsort(m) + m + n) \Rightarrow O(max[n\log n, m\log m])$
    - $tsort(n)$: the time needed to sort the n records in *list*1
    - $tsort(m)$: the time needed to sort the m records in *list*2
    - we will show it is possible to sort n records in $O(n\log n)$ time
- Definition
  - Given $(R_0, R_1, \ldots, R_{n-1})$, each R_i has a key value K_i
    find a permutation σ, such that $K_{\sigma(i-1)} \leq K_{\sigma(i)}$, $0<i \leq n-1$
  - σ denotes an unique permutation
  - 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

Introduction (9)

- Two important applications of sorting:
  - An aid to search
  - Matching entries in lists
- Internal sort
  - The list is small enough to sort entirely in main memory
- External sort
  - There is too much information to fit into main memory

INSERTION SORT

Insertion Sort (1)

- Concept:
  - The basic step in this method is to insert a record R into a sequence of ordered records, R_1, R_2, …, R_i ($K_1 \le K_2 \le$, …, $\le K_i$) in such a way that the resulting sequence of size i is also ordered
- Variation
  - Binary insertion sort
    - reduce search time
  - List insertion sort
    - reduce insert time

Insertion Sort (2)

- Insertion sort program

i = 4 next = 4

list	[0]	[1]	[2]	[3]	[4]
	1	2	3	4	5

```
void insertion_sort(element list[], int n)
/* perform a insertion sort on the list */
{
    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;
    }
}
```

Program 7.5: *insertion_sort*

Insertion Sort (3)

- Analysis of Insertion Sort:
  - If k is the number of records LOO, then the computing time is $O((k+1)n)$
  - The worst-case time is $O(n^2)$.
  - The average time is $O(n^2)$.
  - The best time is $O(n)$.

    left out of order (LOO)

$$R_i \text{ is LOO } iff \ R_i < \max_{0 \le j < i}\{R_j\}$$

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

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

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

$O(n)$

QUICK SORT

Quick Sort (1)

- The quick sort scheme developed by C. A. R. Hoare has the best average behavior among all the sorting methods we shall be studying
- Given $(R_0, R_1, \ldots, R_{n-1})$ and K_i denote a pivot key
- If K_i is placed in position $s(i)$,
  then $K_j \leq K_{s(i)}$ for $j < s(i)$, $K_j \geq K_{s(i)}$ for $j > s(i)$.
- After a positioning has been made, the original file is partitioned into two subfiles, $\{R_0, \ldots, R_{s(i)-1}\}$, $R_{s(i)}$, $\{R_{s(i)+1}, \ldots, R_{s(n-1)}\}$, and they will be sorted independently

Quick Sort (2)

- Quick Sort Concept
  - select a pivot key
  - interchange the elements to their correct positions according to the pivot
  - the original file is partitioned into two subfiles and they will be sorted independently

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
26	5	37	1	61	11	59	15	48	19
11	5	19	1	15	26	59	61	48	37
1	5	11	19	15	26	59	61	48	37
1	5	11	19	15	26	59	61	48	37
1	5	11	15	19	26	59	61	48	37
1	5	11	15	19	26	48	37	59	61
1	5	11	15	19	26	37	48	59	61
1	5	11	15	19	26	37	48	59	61

- Quick Sort Program

i: ↑ j: ↑

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
1	5	11	15	19	26	37	48	59	61

pivot= [illegible]

left [illegible]

right [illegible]

```
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);
      SWAP(list[left],list[j],temp);
      quicksort(list,left,j-1);
      quicksort(list,j+1,right);
    }
}
```

Quick Sort (4)

- Analysis for Quick Sort
  - 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
  - $T(n) \le cn + 2T(n/2)$ for some c

    $\le cn + 2(cn/2 + 2T(n/4))$

    ...

    $\le cn\log_2 n + nT(1) = O(n\log n)$
- Time complexity
  - Average case and best case: $O(n\log n)$
  - Worst case: $O(n^2)$
  - Best internal sorting method considering the average case
- Unstable

Quick Sort (5)

- **Lemma 7.1:**
  - Let $T_{avg}(n)$ be the expected time for quicksort to sort a file with n records. Then there exists a constant k such that $T_{avg}(n) \leq kn\log_e n$ for $n \geq 2$
- Space for Quick Sort
  - The smaller of the two subarrays is always stored first, the maximum stack space is less than $2\log n$
- Stack space complexity:
  - Average case and best case: $O(\log n)$
  - Worst case: $O(n)$

Quick Sort (6)

- Quick Sort Variations
  - Quick sort using a median of three: Pick the median of the first, middle, and last keys in the current sublist as the pivot. Thus, pivot = median$\{K_l, K_{(l+r)/2}, K_r\}$.

MERGE SORT

Merge Sort (1)

- Before looking at the merge sort algorithm to sort *n* records, let us see how one may merge two sorted lists to get a single sorted list.
- Merging
  - The first one, Program 7.7, uses O(*n*) additional space.
  - It merges the sorted lists (*list*[*i*], … , *list*[*m*]) and (*list*[*m+1*], …, *list*[*n*]), into a single sorted list, (*sorted*[*i*], … , *sorted*[*n*]).

• Merge (using O(n) space)

```
void merge(element list[], element sorted[], int i, int m,
                                              int n)
/* merge two sorted files: list[i],...,list[m], and
list[m+1],..., list[n]. These files are sorted to
obtain a sorted list: sorted[i],..., sorted[n] */
{
  int j,k,t;
  j = m+1;          /* index for the second sublist */
  k = i;            /* index for the sorted list */

  while (i <= m && j <= n) {
    if (list[i].key <= list[j].key)
      sorted[k++] = list[i++];
    else
      sorted[k++] = list[j++];
  }
  if (i > 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 Sort (3)

- Iterative merge sort
  1. We assume that the input sequence has *n* sorted lists, each of length 1.
  2. We merge these lists pairwise to obtain *n*/2 lists of size 2.
  3. We then merge the n/2 lists pairwise, and so on, until a single list remains.
- Analysis
  - Total number of passes is the ceiling of $\log_2 n$
  - merge two sorted list in linear time: O(*n*)
  - The total computing time is O(*n* log *n*).

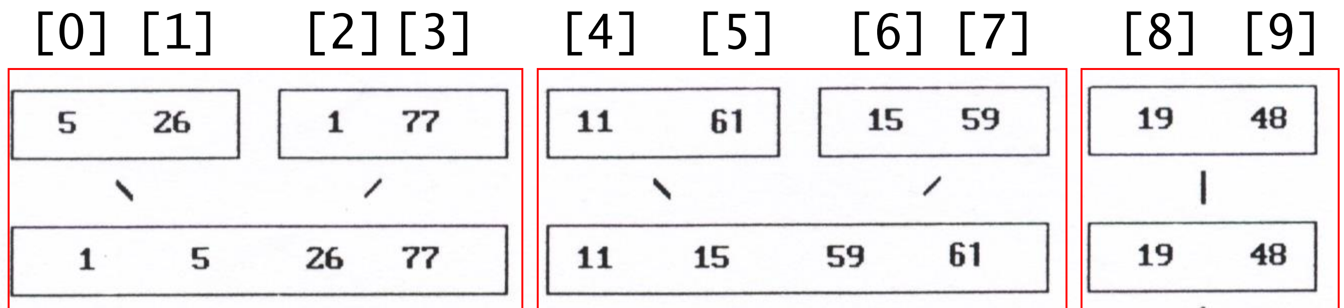
Merge Sort (4)

- *merge_pass*
  - Invokes *merge* (Program 7.7) to merge the sorted sublists
  - Perform one pass of the merge sort. It merges adjacent pairs of subfiles from list into sorted.

length=2
n=10
i= 8

list

sorted




```
void merge_pass(element list[], element sorted[], int n,
                                          int length)
{
  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)
    merge(list,sorted,i,i + length - 1,n - 1);
  else
    for (j = i; j < n; j++)
      sorted[j] = list[j];
}
```

the length of the subfile

the number of elements in the list

4 5 7

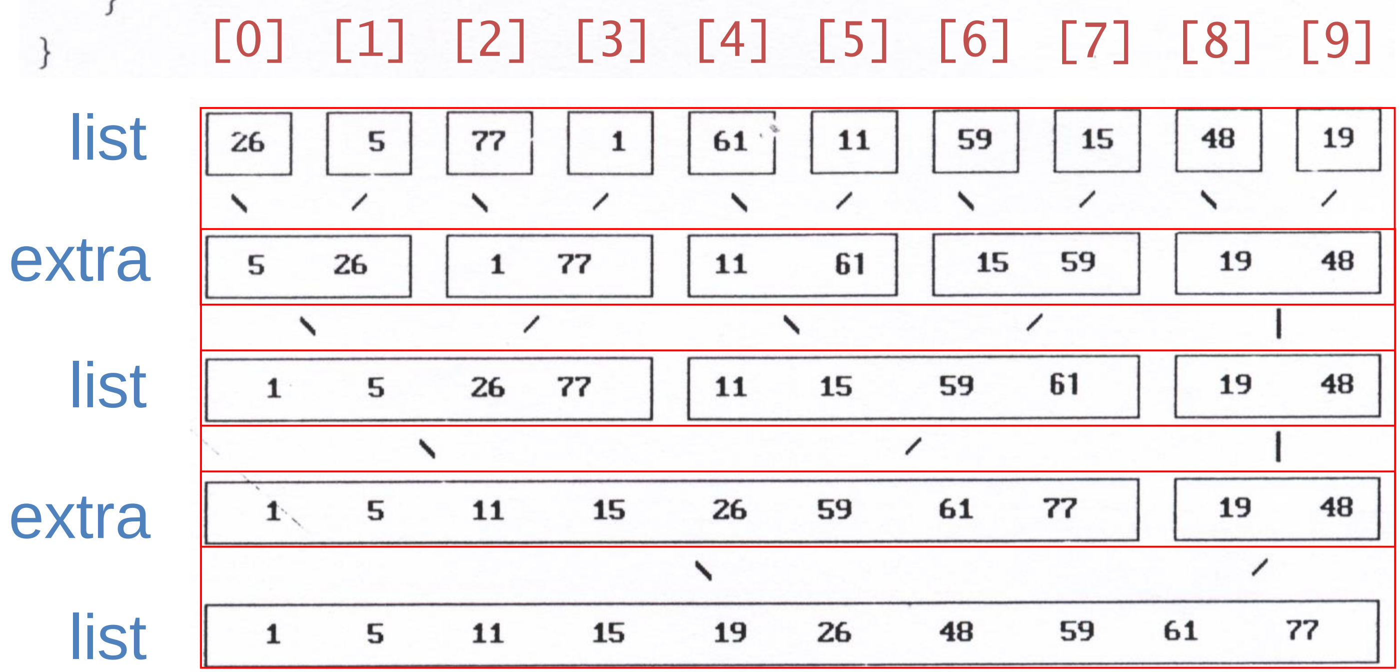
- *merge_sort*: Perform a merge sort on the file

length=16

n=10

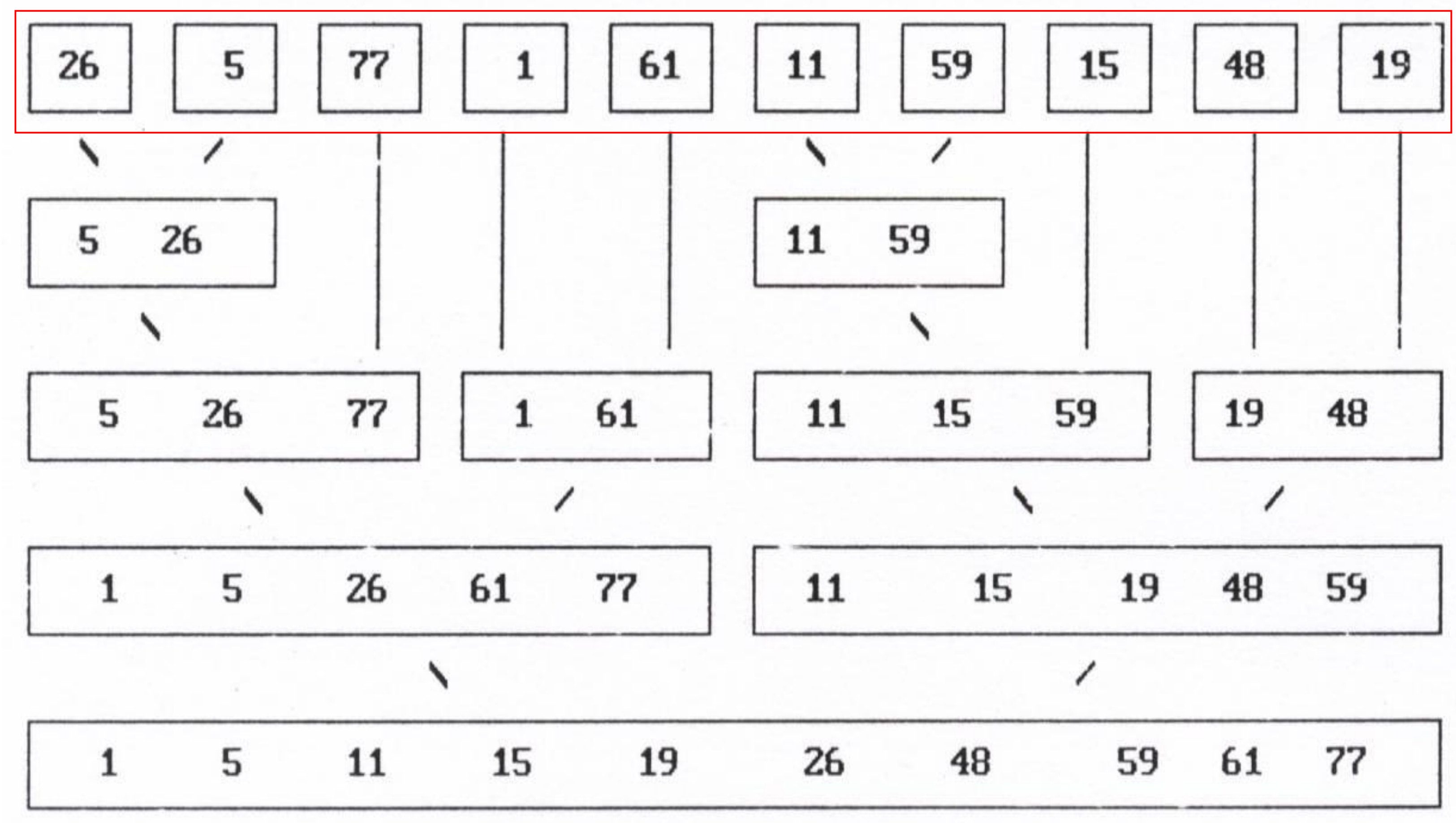
```
void merge_sort(element list[], int n)
{
    int length = 1; /* current length being merged */
    element extra[MAX_SIZE];

    while (length < n) {
        merge_pass(list,extra,n,length);
        length *= 2;
        merge_pass(extra,list,n,length);
        length *= 2;
    }
}
```



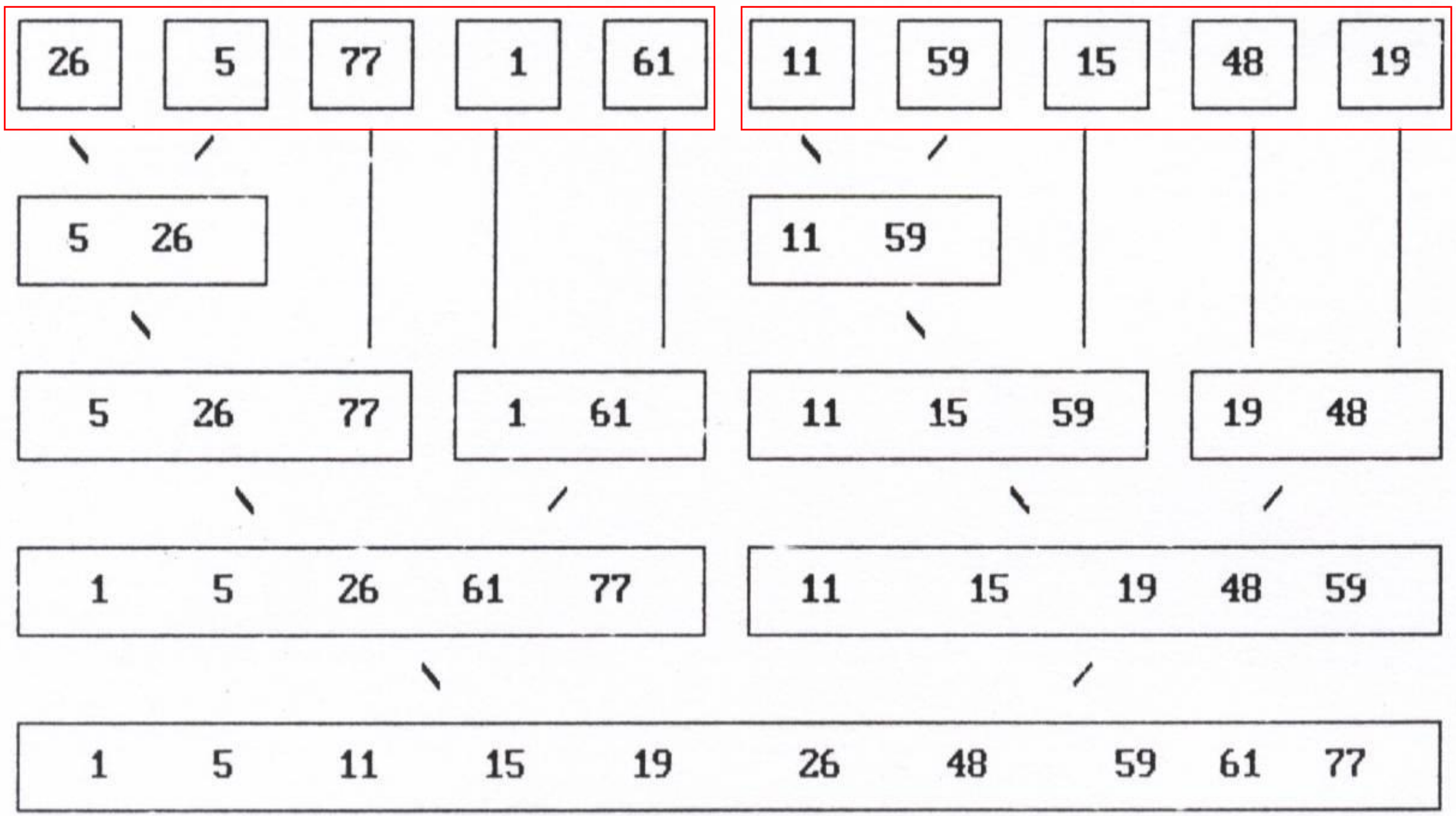
Merge Sort (6)

- Recursive merge sort concept



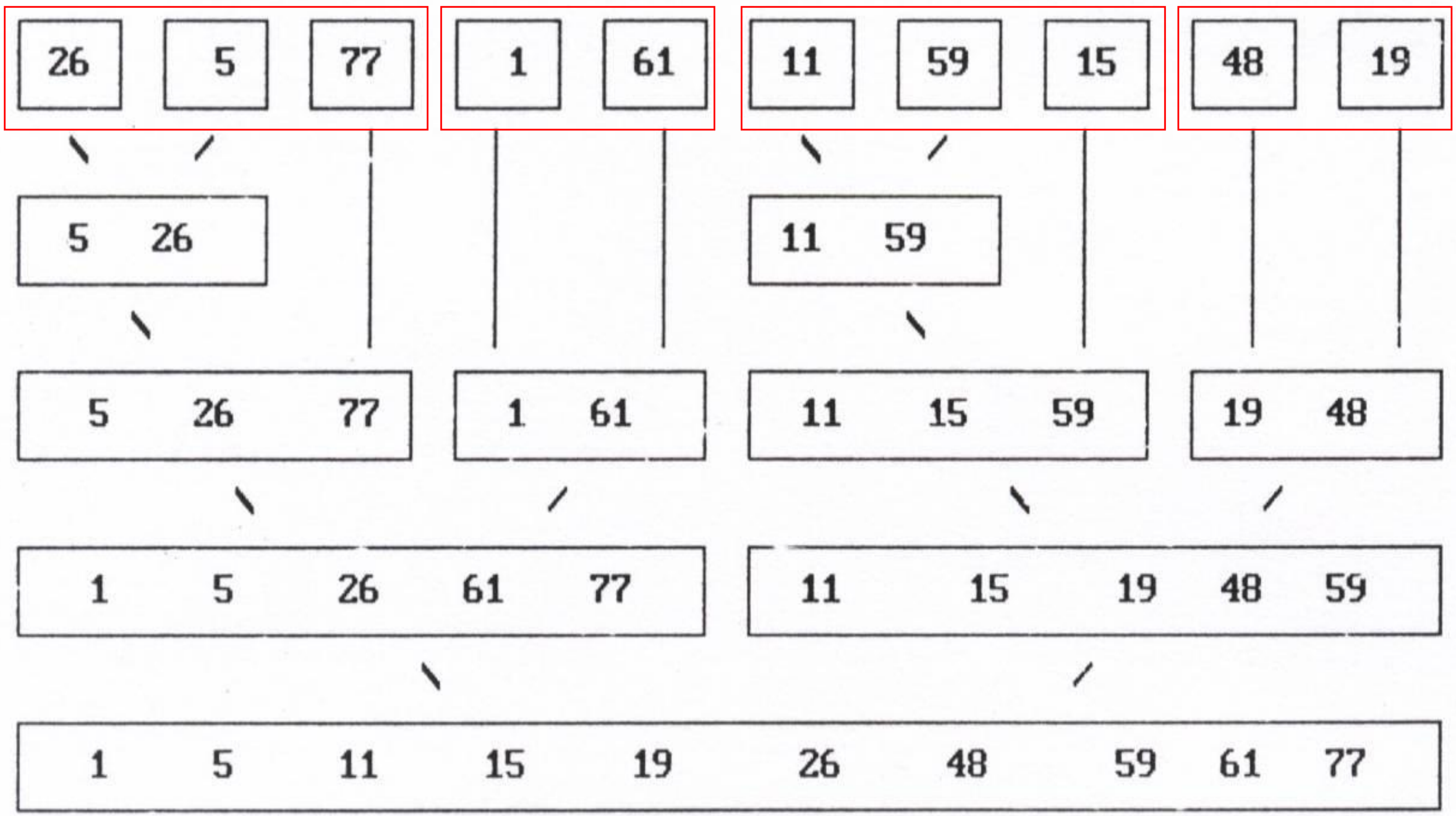
Merge Sort (6)

- Recursive merge sort concept



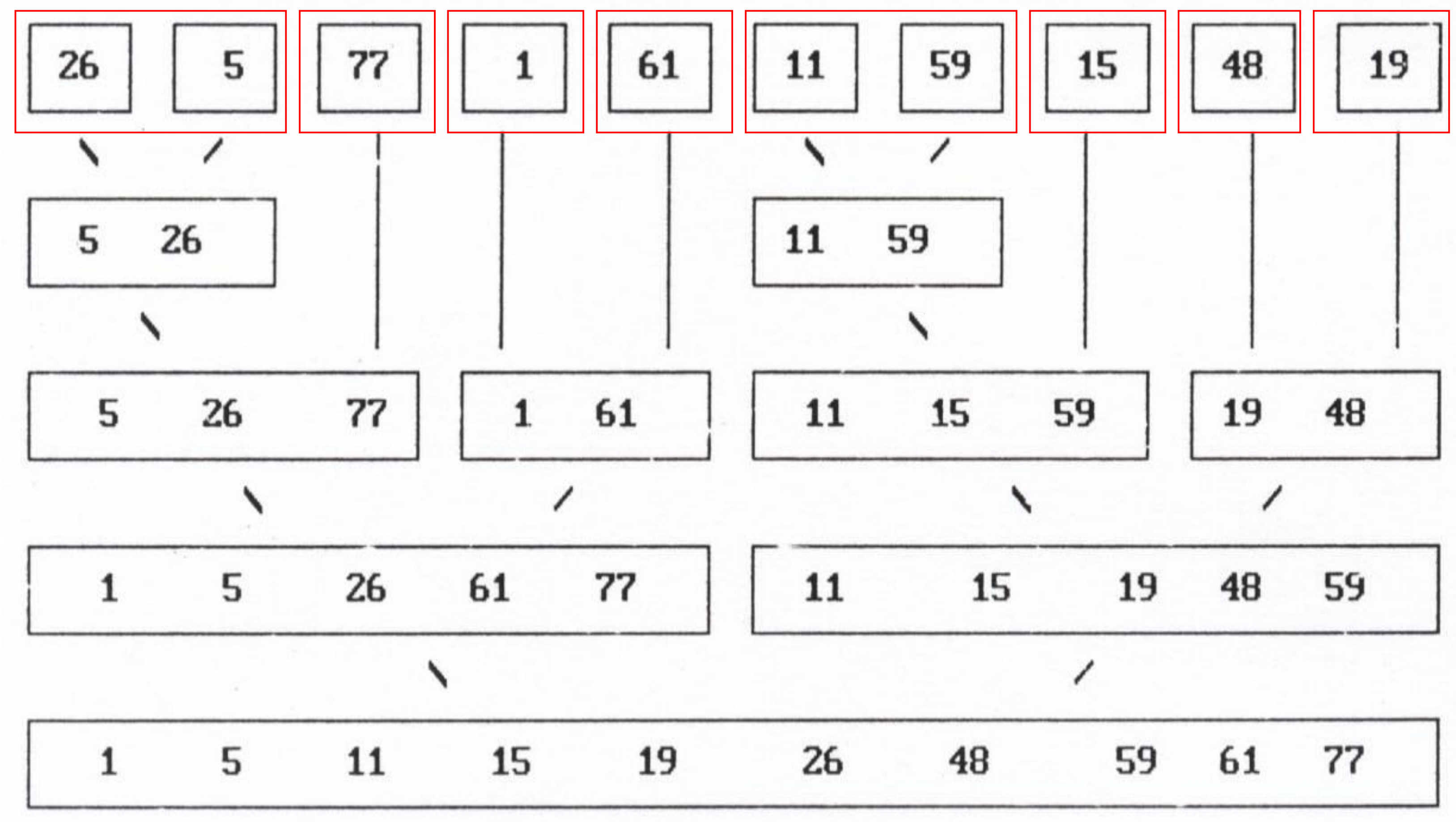
Merge Sort (6)

- Recursive merge sort concept



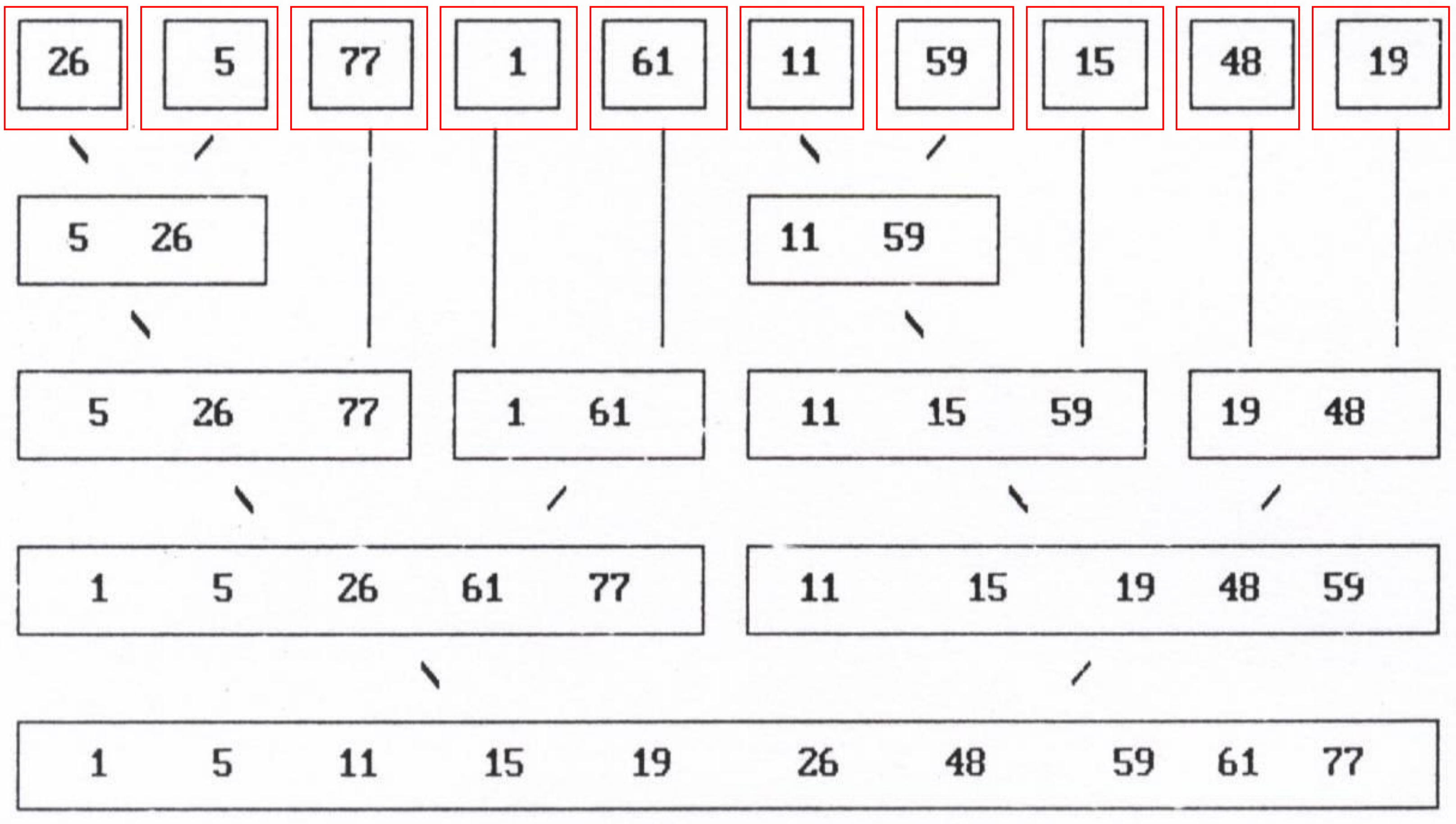
Merge Sort (6)

- Recursive merge sort concept



Merge Sort (6)

- Recursive merge sort concept



- *listmerge*:
  - Takes two sorted chains and returns an integer that points to the start of the sorted list

The link field in each record is initially set to -1

Since the elements were numbered from 0 to n-1, we use list[n] to store the start pointer

```
int listmerge(element list[], int first, int second)
/* merge lists pointed to by first and second */
{
  int start = n;
  while (first != -1 && second != -1)
    if (list[first].key <= list[second].key) {
    /* key in first list is lower, link this element to
    start and change start to point to first */
      list[start].link = first;
      start = first;
      first = list[first].link;
    }
    else {
    /* key second list is lower, link this element into
    the partially sorted list */
      list[start].link = second;
      start = second;
      second = list[second].link;
    }
  /* move remainder */
  if (first == -1)
    list[start].link = second;
  else
    list[start].link = first;
  return list[n].link; /* start of the new list */
}
```

- *rmerge*: sort the list, list[lower], …, list[upper]. The link field in each record is initially set to -1

lower=

upper=

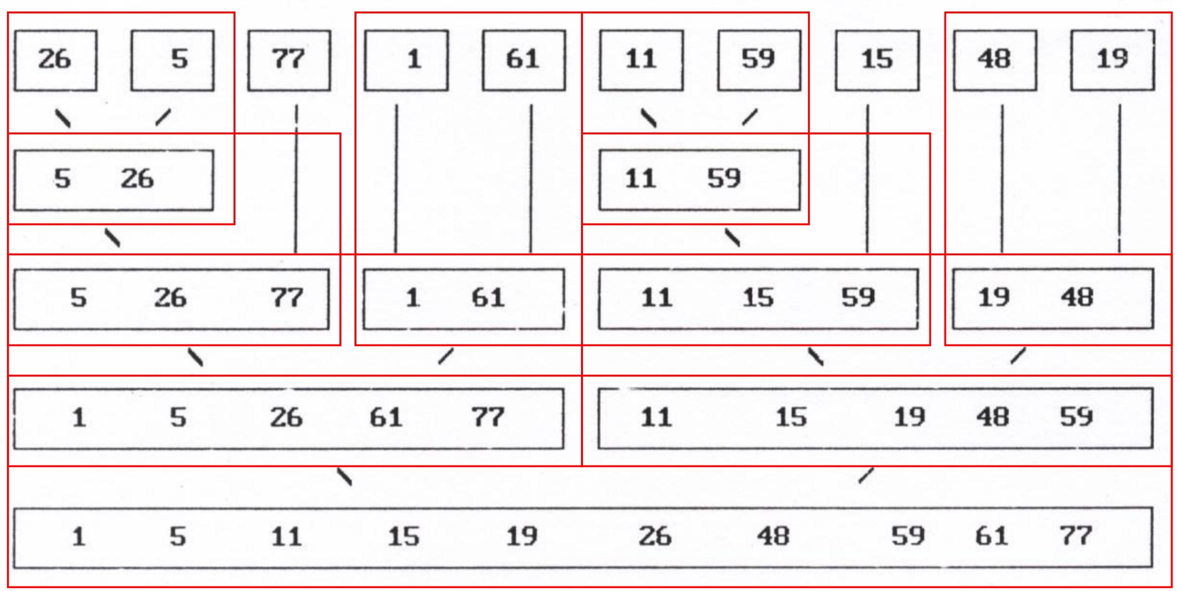
middle=

```
int rmerge(element list[], int lower, int upper)
{
   int middle;
   if (lower >= upper)
      return lower;
   else {
      middle = (lower + upper) / 2;
      return listmerge(list,rmerge(list,lower,middle),
                            rmerge(list,middle+1,upper));
   }
}
```

start = rmerge(list, 0, n-1);

= 0

list [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]



Merge Sort (10)

- **Variation: *Natural merge sort* :**
  - We can modify *merge_sort* 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 sequences of records that are in order.
  - The merge sort then uses these initially ordered sublists for the remainder of the passes.

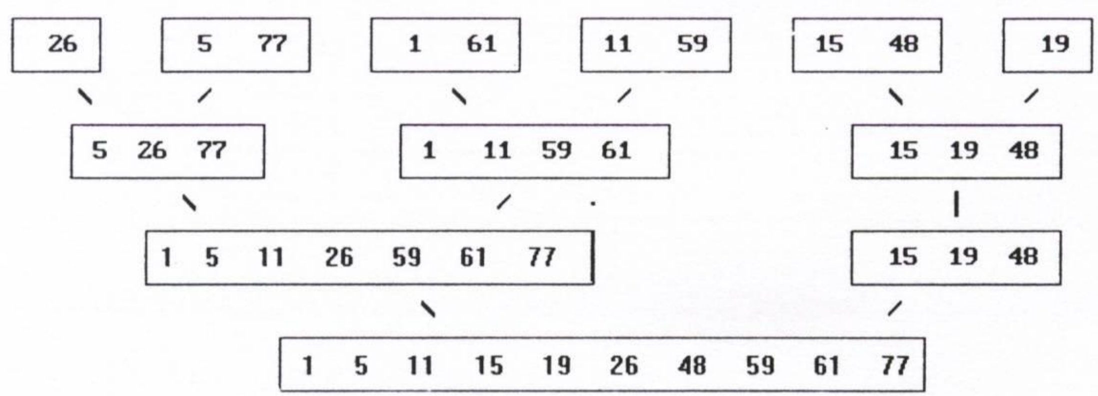


Figure 7.10: Merge sort starting with sorted sublists

HEAP SORT

Heap Sort (1)

- The challenges of merge sort
  - The merge sort requires additional storage proportional to the number of records in the file being sorted.
  - By using the $O(1)$ space merge algorithm, the space requirements can be reduced to $O(1)$, but significantly slower than the original one.
- Heap sort
  - Require only a fixed amount of additional storage
  - Slightly slower than merge sort using $O(n)$ additional space
  - Faster than merge sort using $O(1)$ additional space.
  - The worst case and average computing time is $O(n \log n)$, same as merge sort
  - Unstable

- ***adjust***
  - adjust the binary tree to establish the heap

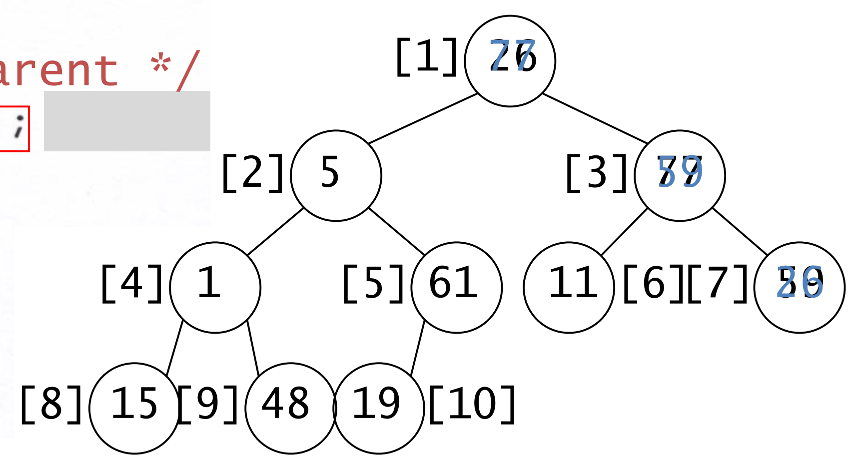
```
void adjust(element list[], int root, int n)
{
   int child,rootkey;
   element temp;
   temp = list[root];
   rootkey = list[root].key;
   child = 2 * root;          /* left child */
   while (child <= n) {
      if ((child < n) &&
      (list[child].key < list[child+1].key))
         child++;
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

root = 1

n = 10

rootkey = 26

child = 4



Heap Sort (3)

- ***heapsort***

```
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);
   }
}
```

Program 7.14: Heap sort

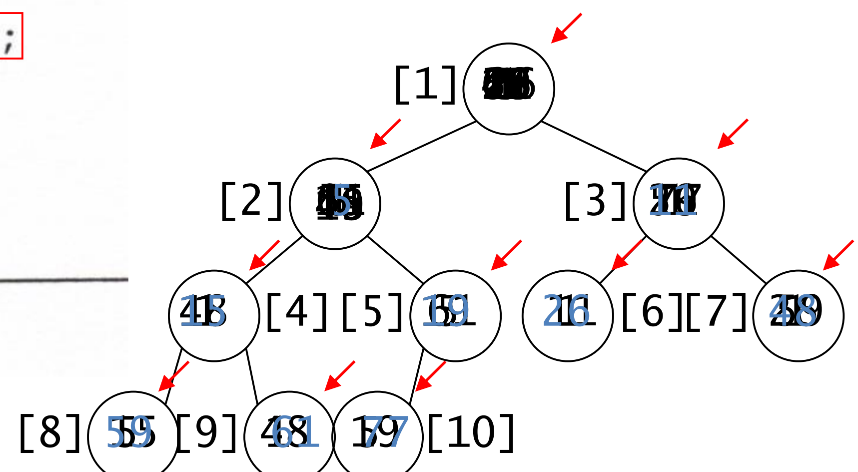
bottom-up

top-down

n = 10

i =

ascending order
(max heap)



RADIX SORT

Radix Sort (1)

- We considers the problem of sorting records that have several keys
  - These keys are labeled K^0 (most significant key), K^1, … , K^{r-1} (least significant key).
  - Let K_i^j denote key K^j of record R_i.
  - A list of records R_0, … , R_{n-1}, is lexically sorted with respect to the keys K^0, K^1, … , K^{r-1} *iff* $(K_i^0, K_i^1, \ldots, K_i^{r-1}) \leq (K^0_{i+1}, K^1_{i+1}, \ldots, K^{r-1}_{i+1})$, $0 \leq i < n-1$

Radix Sort (2)

- Example
  - sorting a deck of cards on two keys, suit and face value, in which the keys have the ordering relation:
    K^0 [Suit]: ♣ < ♦ < ♥ < ♠
    K^1 [Face value]: 2 < 3 < 4 < … < 10 < J < Q < K < A
  - Thus, a sorted deck of cards has the ordering:
    2♣, …, A♣, … , 2♠, … , A♠
  - Two approaches to sort:
    1. MSD (Most Significant Digit) first: sort on K^0, then K^1, ...
    2. LSD (Least Significant Digit) first: sort on K^{r-1}, then K^{r-2}, ...

Radix Sort (3)

- MSD first
  1. MSD sort first, e.g., bin sort, four bins ♣ ♦ ♥ ♠
  2. LSD sort second
  – Result: 2♣, …, A♣, … , 2♠, … , A♠

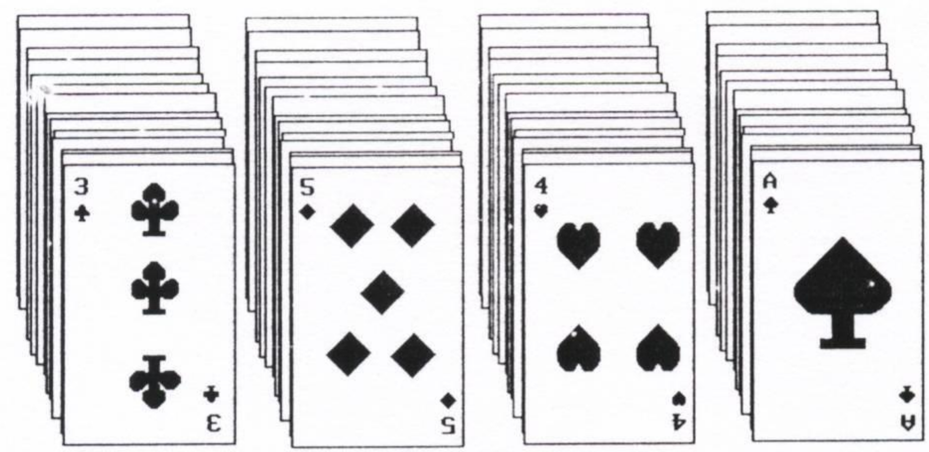


Figure 7.14: Arrangement of cards after first pass of an MSD sort

Radix Sort (4)

- LSD first
  1. LSD sort first, e.g., face sort,
     13 bins 2, 3, 4, …, 10, J, Q, K, A
  2. MSD sort second (may not needed, we can just classify these 13 piles into 4 separated piles by considering them from face 2 to face A)
  - Simpler than the MSD one because we do not have to sort the subpiles independently

Result:
2♣, …, A♣, … ,
2♠, …, A♠

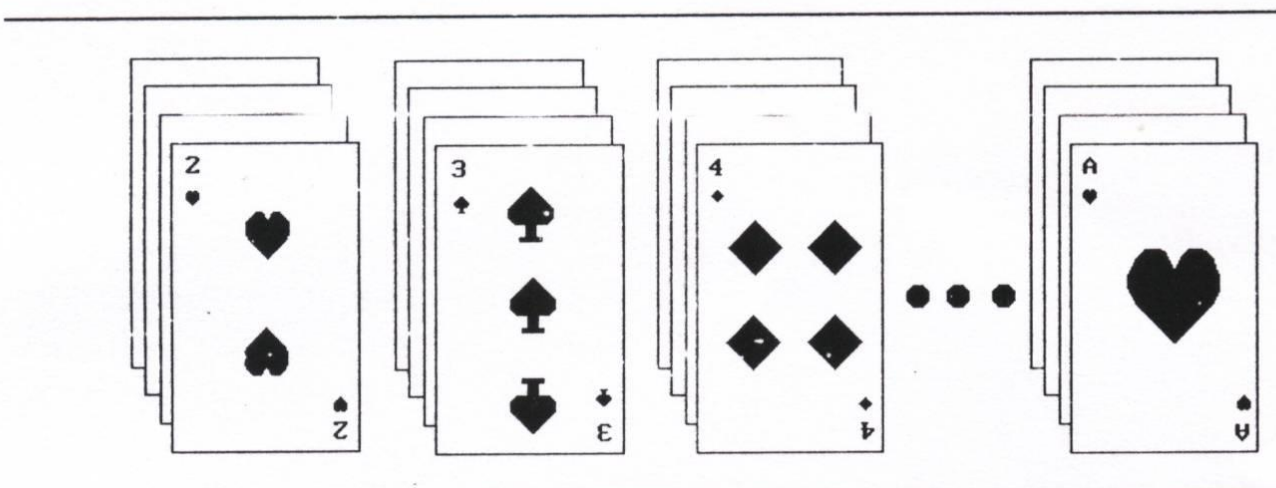


Figure 7.15: Arrangement of cards after first pass of LSD sort

Radix Sort (5)

- We also can use an LSD or MSD sort when we have only one logical key, if we interpret this key as a composite of several keys.
- Example:
  - integer: the digit in the far right position is the least significant and the most significant for the far left position
  - range: $0 \leq K \leq 999$
  - using LSD or MSD sort for three keys (K^0, K^1, K^2)
  - since an LSD sort does not require the maintainence of independent subpiles, it is easier to implement

MSD LSD

0-9 0-9 0-9

Radix Sort (6)

- *radix sort*
  - decompose the sort key into digits using a radix r.
  - Ex: When r =10, we get the common base 10 or decimal decomposition of the key

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

- LSD radix r sort
  - The records, $R_0, \ldots, R_{n-1}$
  - Keys: d-tuples $(x_0, x_1, \ldots, x_{d-1})$ and that $0 \leq x_i < r$.
  - Each record has a link field, and that the input list is stored as a dynamically linked list.
  - We implement the bins as queues
    - *front*[i], $0 \leq i < r$, pointing to the first record in bin i
    - *rear*[i], $0 \leq i < r$, pointing to the last record in bin i

- LSD Radix Sort

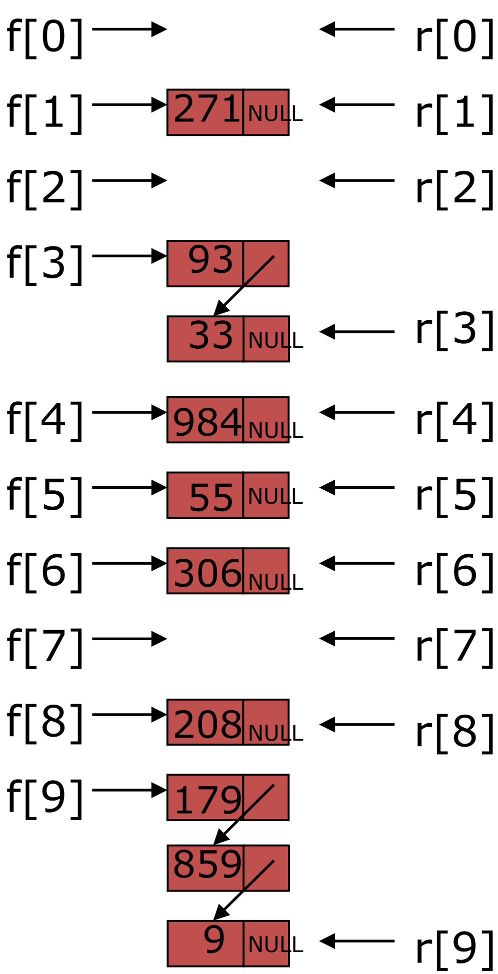
Time complexity: O(MAX_DIGIT(RADIX_SIZE+n))

RADIX_SIZE = 10

MAX_DIGIT = 3

Initial input:

179→208→306→93→859→984→55→9→271→33




```
list_pointer radix_sort(list_pointer ptr)
/*Radix Sort using a linked list */
{
   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])
         front[digit] = ptr;
       else
         rear[digit]->link = ptr;
       rear[digit] = ptr;
       ptr = ptr->link;
     }
     /* reestablish the linked list for the next pass
     ptr = NULL;
     for (j = RADIX_SIZE-1; j >= 0; j--)
       if (front[j]) {
         rear[j]->link = ptr;  ptr = front[j];
       }
   }
   return ptr;
}
```

MAX_DIGIT passes

O(RADIX_SIZE)

O(n)

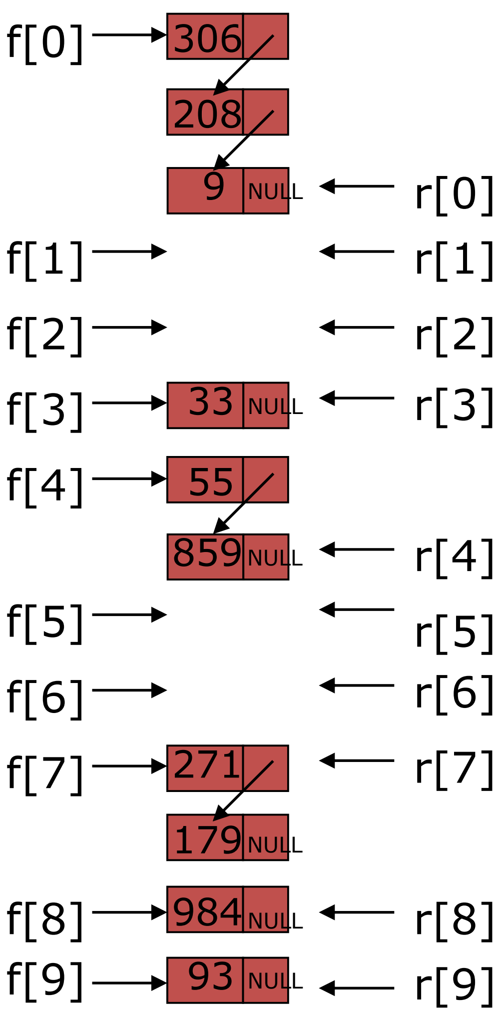
O(RADIX_SIZE)

Chain after first pass, *i*=2:

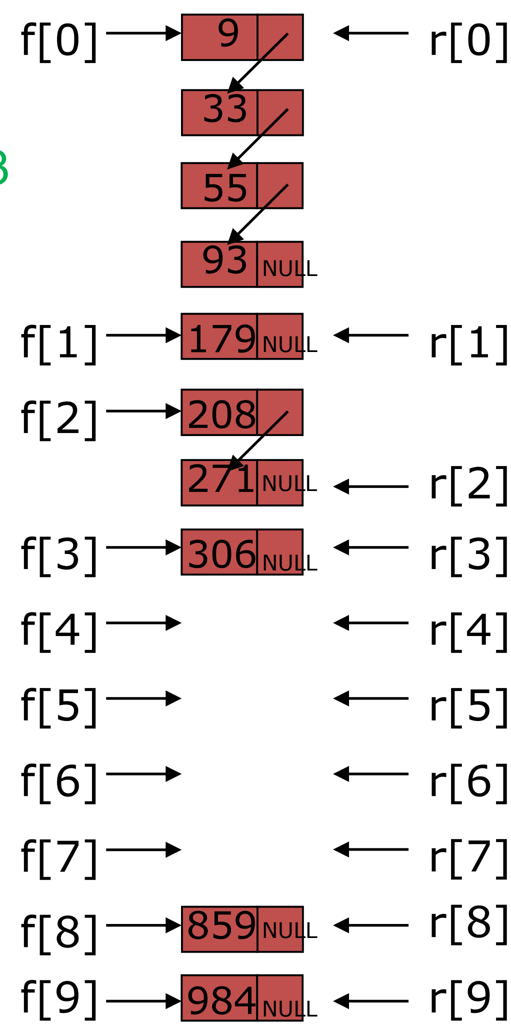
271→93→33→984→55→306→208→179→859→9

Radix Sort (8)

- Simulation of *radix_sort*



Chain after second pass, *i*=1:
306→208→9→33→55→859→271→179→984→93



Chain after third pass, *i*=0:
9→33→55→93→179→208→271→306→859→984

SUMMARY OF INTERNAL SORTING

Summary of Internal Sorting (1)

- Insertion Sort
  - Works well when the list is already partially ordered
  - The best sorting method for small *n*
- Merge Sort
  - The best/worst case ($O(n\log n)$)
  - Require more storage than a heap sort
  - Slightly more overhead than quick sort
- Quick Sort
  - The best average behavior
  - The worst complexity in worst case ($O(n^2)$)
- Radix Sort
  - Depend on the size of the keys and the choice of the radix

Summary of Internal Sorting (2)

- Analysis of the average running times

Times in hundredths of a second

n	quick	merge	heap	insert
0	0.041	0.027	0.034	0.032
10	1.064	1.524	1.482	0.775
20	2.343	3.700	3.680	2.253
30	3.700	5.587	6.153	4.430
40	5.085	7.800	8.815	7.275
50	6.542	9.892	11.583	10.892
60	7.987	11.947	14.427	15.013
70	9.587	15.893	17.427	20.000
80	11.167	18.217	20.517	25.450
90	12.633	20.417	23.717	31.767
100	14.275	22.950	26.775	38.325
200	30.775	48.475	60.550	148.300
300	48.171	81.600	96.657	319.657
400	65.914	109.829	134.971	567.629
500	84.400	138.033	174.100	874.600
600	102.900	171.167	214.400	
700	122.400	199.240	255.760	
800	142.160	230.480	297.480	
900	160.400	260.100	340.000	
1000	181.000	289.450	382.250