CS235102
Data Structures

Chapter 7 Sorting
(Concentrating on Internal Sorting)

Introduction (9/9)

- Two important applications of sorting:
  - An aid to search
  - Matching entries in lists

Introduction (9/9)

- 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

Quick Sort (1/6)

- Let K_i denote a pivot key
- Let K_i be placed in position $s(i)$ after sorting then $K_j \le K_{s(i)}$ for $j < s(i)$,
- $K_j \ge K_{s(i)}$ for $j > s(i)$.

Quick Sort (2/6)

- 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

Quick Sort (2/6)

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
26	5	37	1	61	11	59	15	48	19

Quick Sort (2/6)

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
26	5	37	1	61	11	59	15	48	19
26	5	19	1	61	11	59	15	48	37

Quick Sort (2/6)

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
26	5	37	1	61	11	59	15	48	19
26	5	19	1	61	11	59	15	48	37
26	5	19	1	61	11	59	15	48	37

Quick Sort (2/6)

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
26	5	37	1	61	11	59	15	48	19
26	5	19	1	61	11	59	15	48	37
26	5	19	1	61	11	59	15	48	37
26	5	19	1	15	11	59	61	48	37

Quick Sort (2/6)

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
26	5	37	1	61	11	59	15	48	19
26	5	19	1	61	11	59	15	48	37
26	5	19	1	61	11	59	15	48	37
26	5	19	1	15	11	59	61	48	37
26	5	19	1	15	11	59	61	48	37

Quick Sort (2/6)

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
26	5	37	1	61	11	59	15	48	19
26	5	19	1	61	11	59	15	48	37
26	5	19	1	61	11	59	15	48	37
26	5	19	1	15	11	59	61	48	37
26	5	19	1	15	11	59	61	48	37
26	5	19	1	15	11	59	61	48	37

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
26	5	37	1	61	11	59	15	48	19
26	5	19	1	61	11	59	15	48	37
26	5	19	1	61	11	59	15	48	37
26	5	19	1	15	11	59	61	48	37
26	5	19	1	15	11	59	61	48	37
26	5	19	1	15	11	59	61	48	37
11	5	19	1	15	26	59	61	48	37

Quick Sort (2/6)

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

Quick Sort (2/6)

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

Quick Sort (2/6)

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
11	5	19	1	15	26	59	61	48	37
11	5	1	19	15	26	59	61	48	37
11	5	1	19	15	26	59	61	48	37
1	5	11	19	15	26	59	61	48	37

Quick Sort (2/6)

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

Quick Sort (2/6)

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

Quick Sort (2/6)

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

Quick Sort (2/6)

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

Quick Sort (2/6)

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

Quick Sort (2/6)

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	48	37	59	61

Quick Sort Program

```
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 Program

i: ↑ j: ↑

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
[illegible]	5	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

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 (4/6)

- 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

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

$$\ldots$$

$$\le cn\log_2 n + nT(1) = O(n\log n)$$

Quick Sort (4/6)

- Analysis for Quick Sort
- Time complexity
  - Average case and best case: $O(n\log n)$
  - Worst case: $O(n^2)$
  - Best internal sorting method considering the average case

Quick Sort (4/6)

- Quick sort is Unstable
  - A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in the sorted output as they appear in the unsorted input.
  - Whereas a sorting algorithm is said to be unstable if there are two or more objects with equal keys which don’t appear in same order before and after sorting.

Quick Sort (5/6)

- **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 $T_{avg}(n) \le kn\log_e n$ for $n \ge 2$

Quick Sort (6/6)

- 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 (1/13)

Basic idea

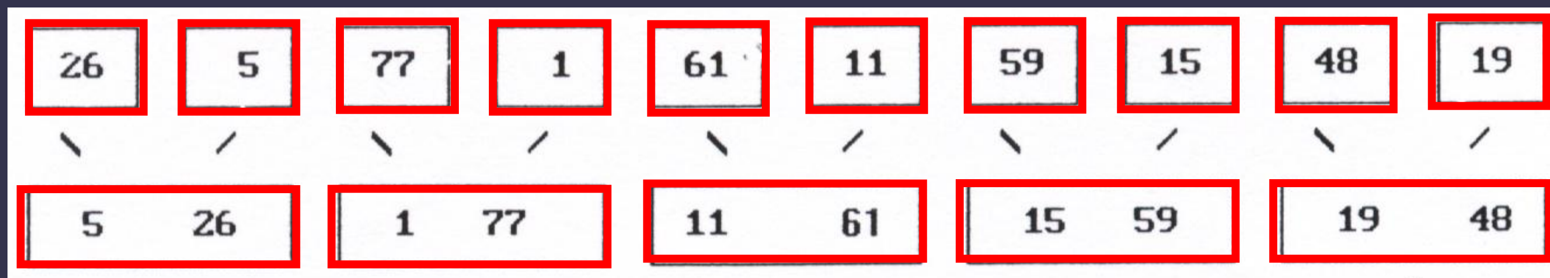
merges two sorted lists
($list[i]$, … , $list[m]$) and ($list[m+1]$, …, $list[n]$)
into a single sorted list, ($sorted[i]$, … , $sorted[n]$).

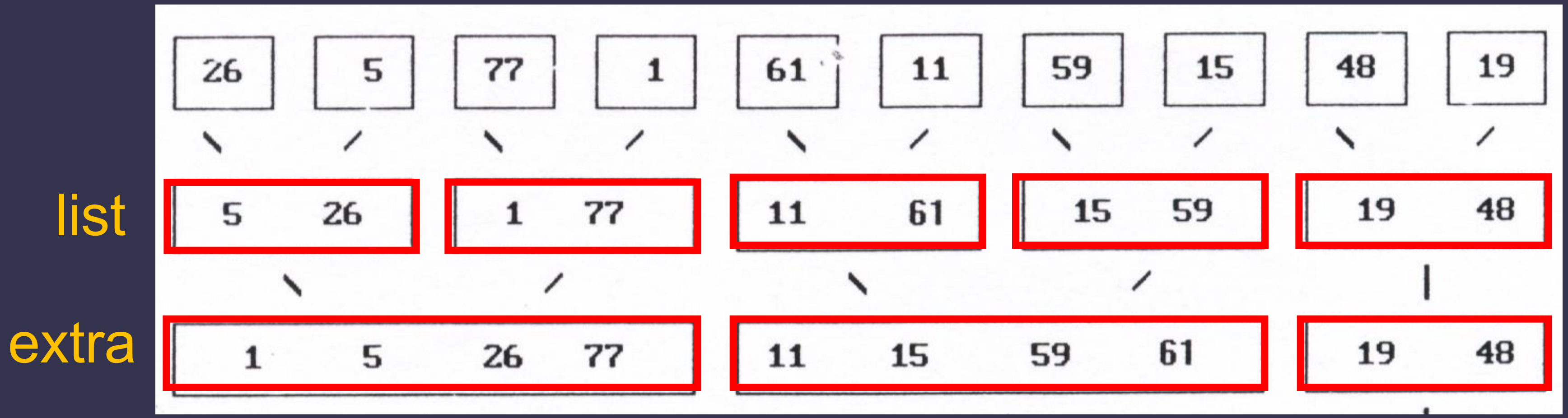
Merge Sort

- Iterative merge sort
  1. assume that the input sequence has *n* sorted lists, each of length 1.
  2. merge these lists pairwise to obtain *n*/2 lists of size 2.
  3. then merge the n/2 lists pairwise, and so on, until a single list remains.

list

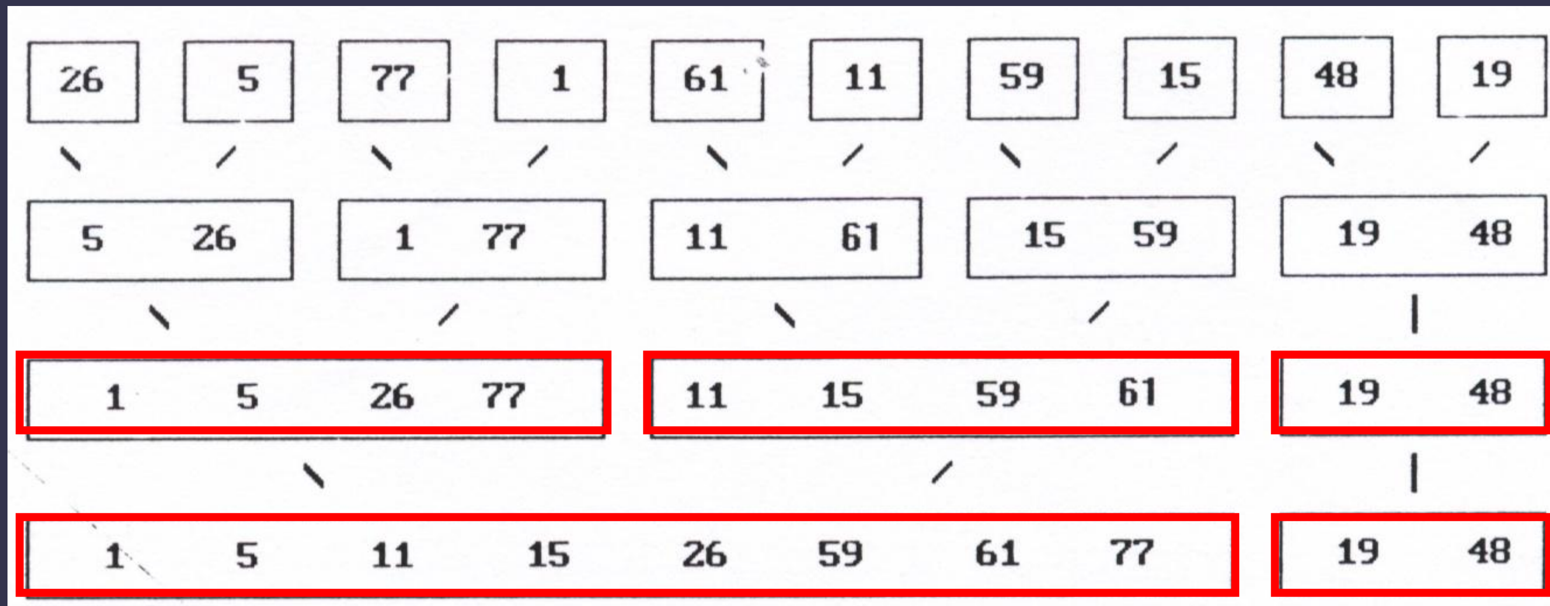
extra

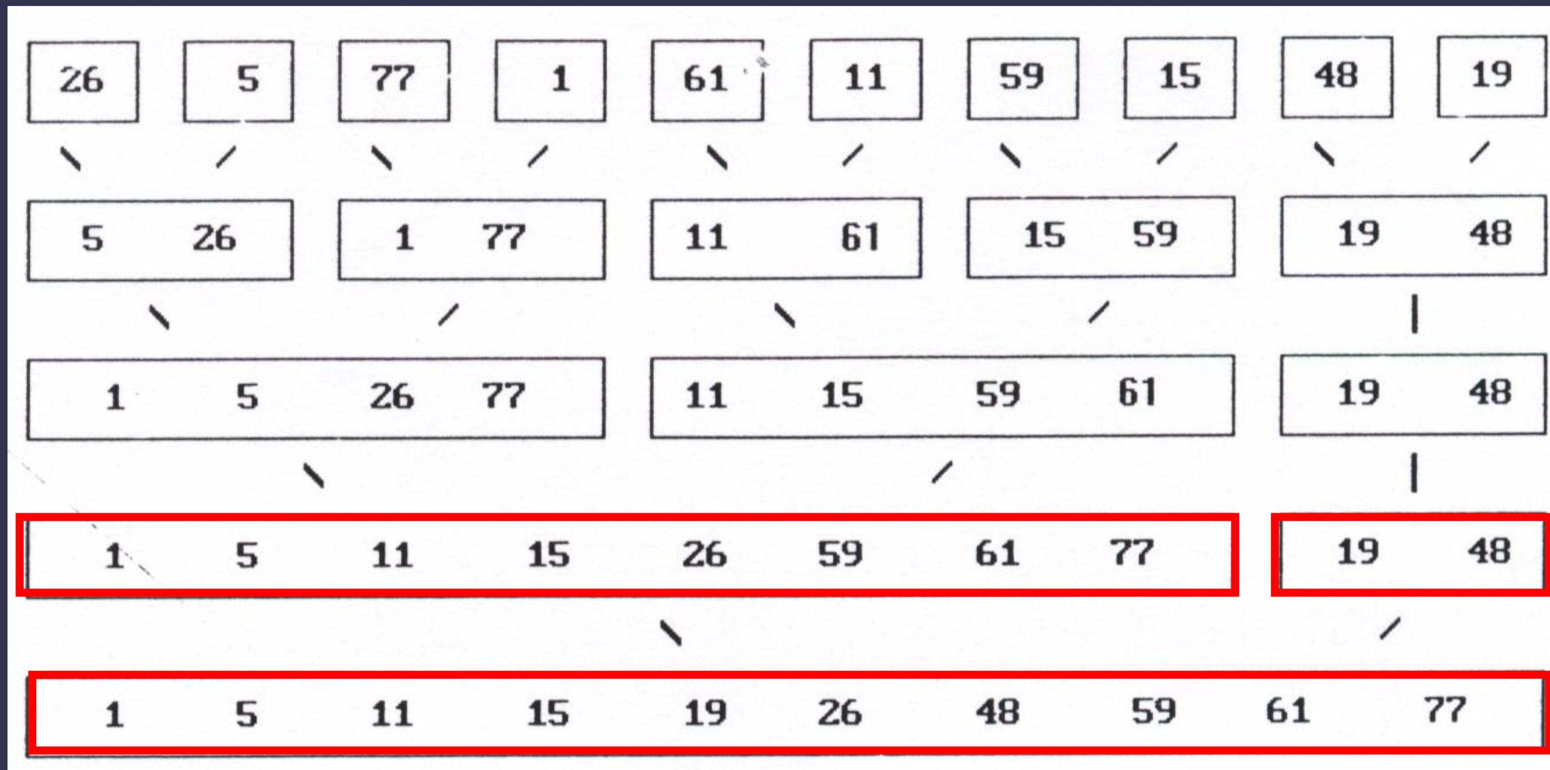



list
extra
26
5
77
1
61
11
59
15
48
19
5 26
1 77
11 61
15 59
19 48
1 5 26 77
11 15 59 61
19 48

list

extra


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


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

list

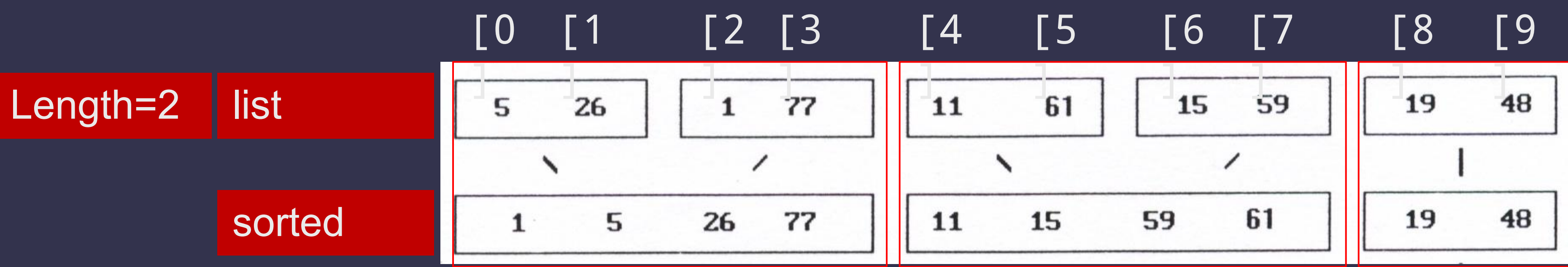
extra

Merge Sort

- Analysis
  - Total number of passes is the celling of $\log_2 n$
  - merge two sorted list in linear time: $O(n)$
  - The total computing time is $O(n \log n)$.

Merge Sort (8/13)

- *merge_pass*
  - Perform one pass of the merge sort. It merges adjacent pairs of subfiles from list into 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

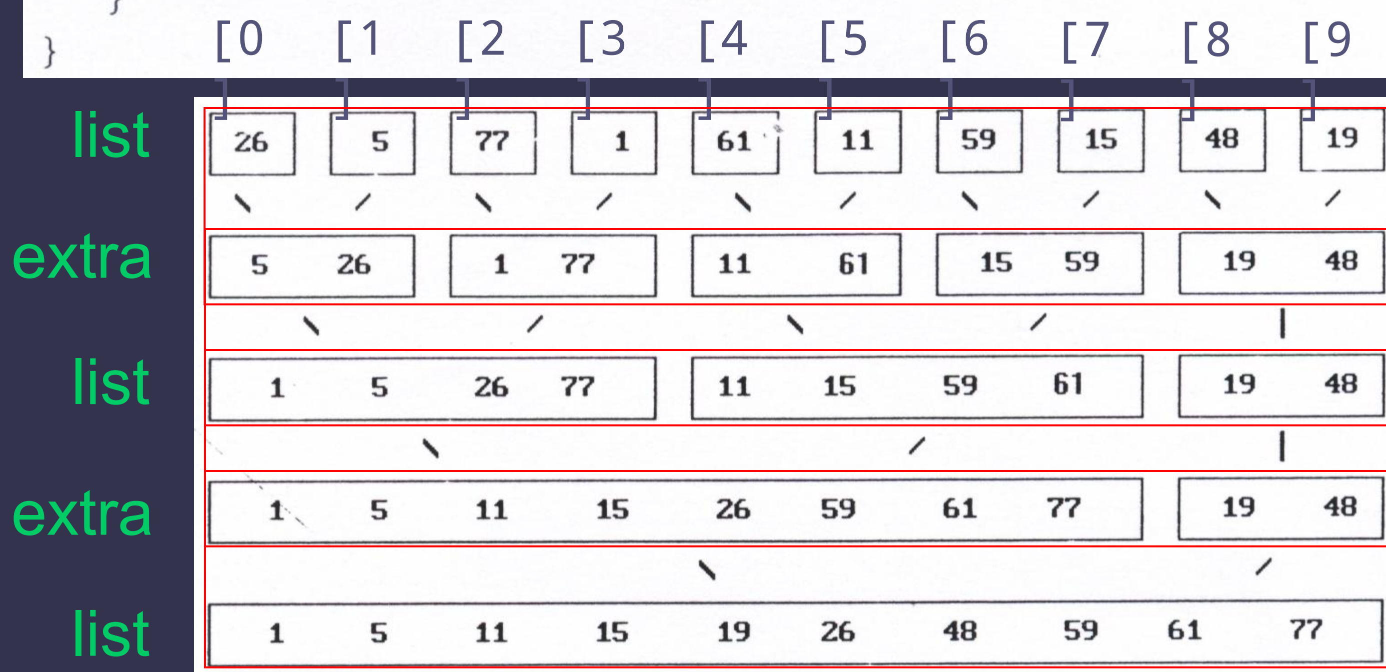
the number of elements in the list

- Iterative-Merge two sorted lists (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];
}
```

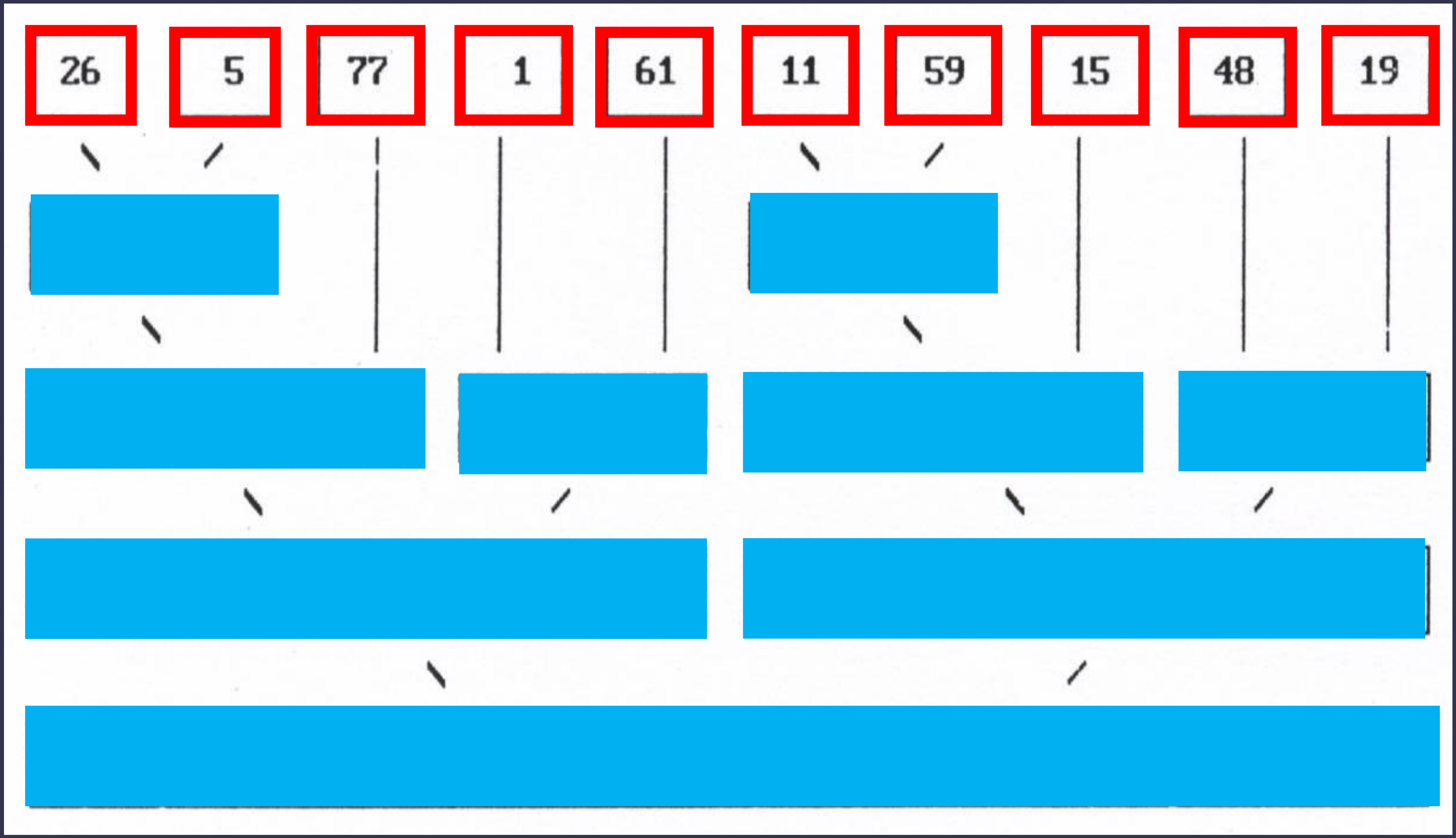
- *merge_sort*: Perform a merge sort on the file

```
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;
    }
}
```



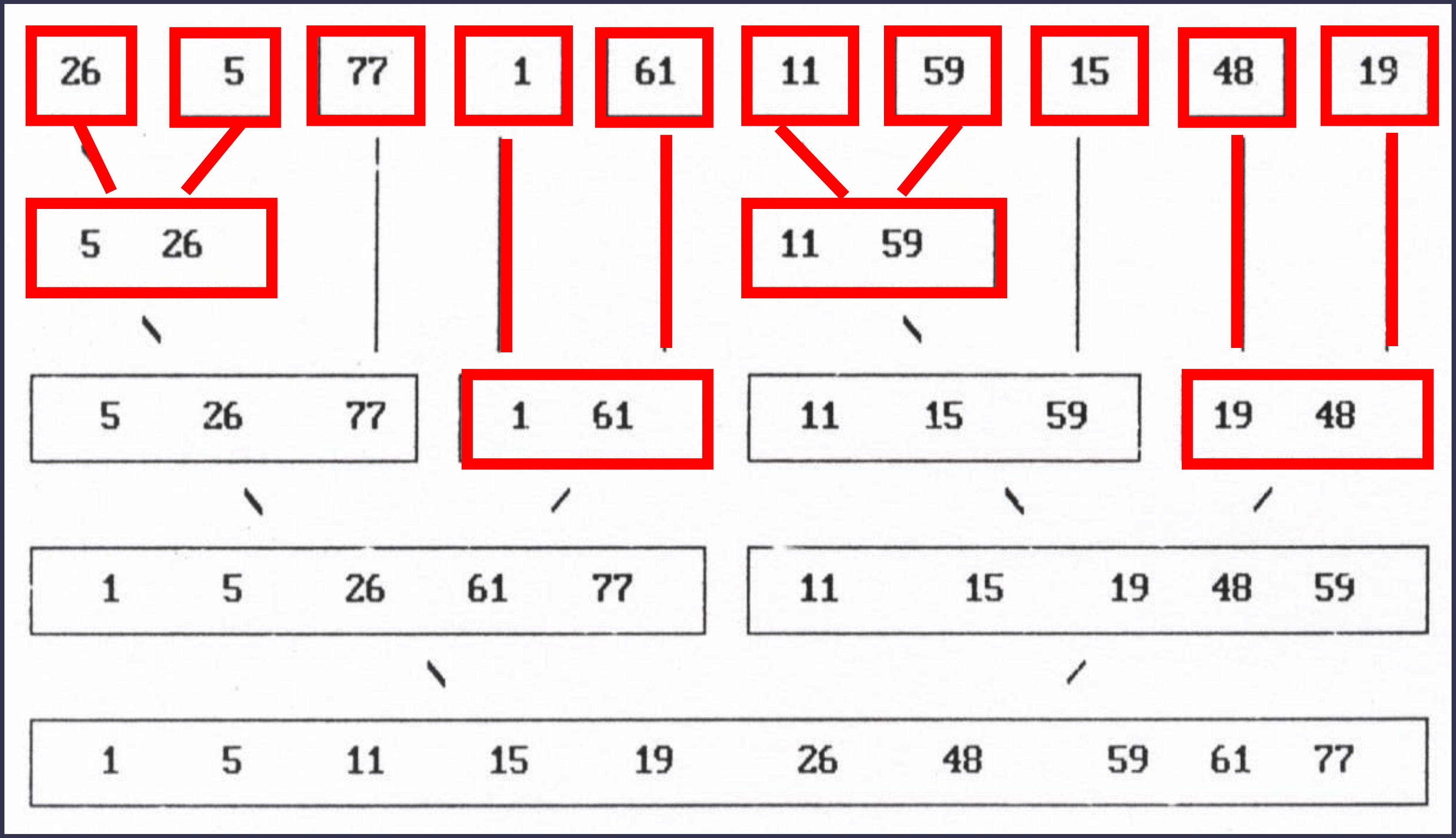
Merge Sort (10/13)

- Recursive merge sort concept



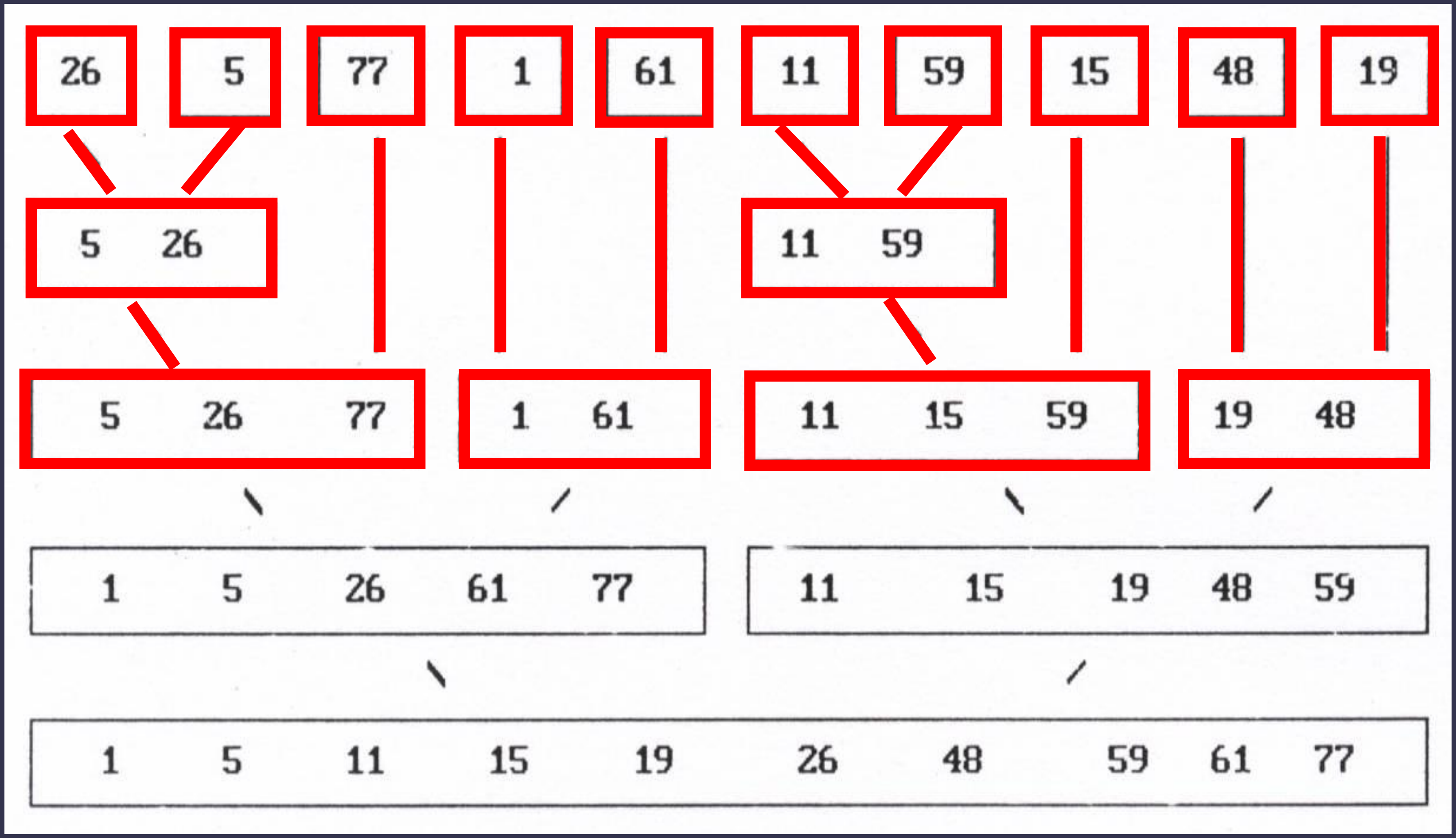
Merge Sort (10/13)

- Recursive merge sort concept



Merge Sort (10/13)

- Recursive merge sort concept



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

lower= [illegible]

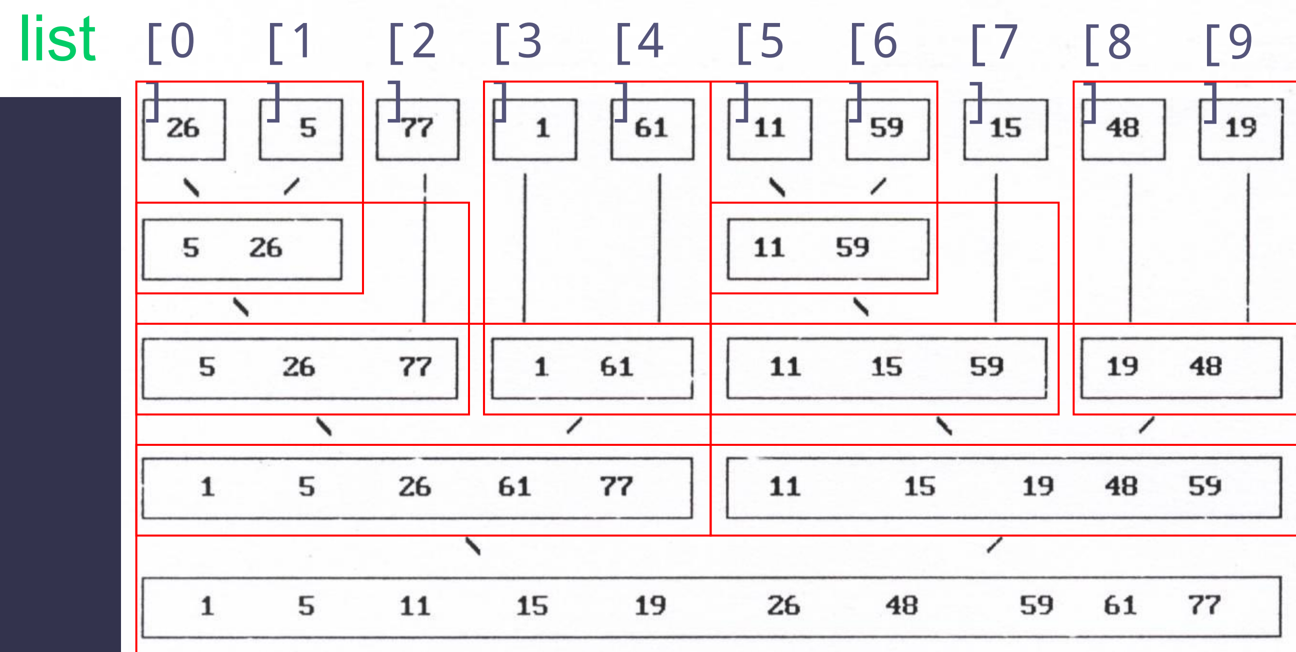
upper= [illegible]

middle= [illegible]

```
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



Heap Sort (1/3)

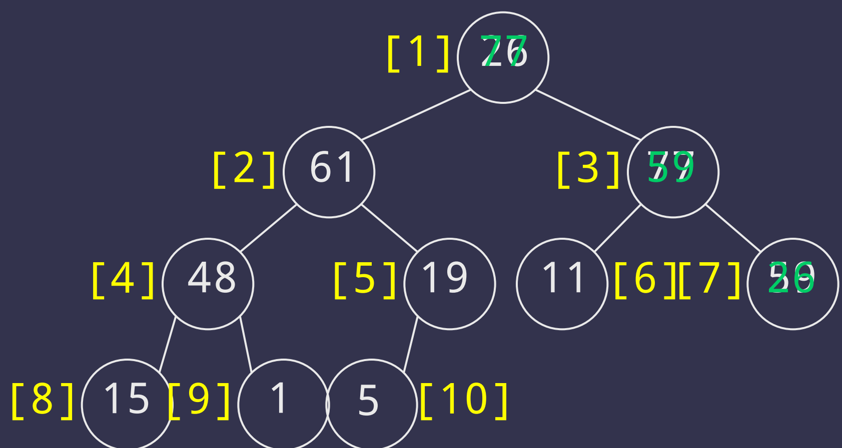
- The challenges of merge sort
  - The merge sort **requires additional storage** proportional to the number of records in the file being sorted.

Heap Sort (1/3)

- 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*

- Suppose that subtrees of a binary tree are max heaps, but the binary tree is not a max heap.



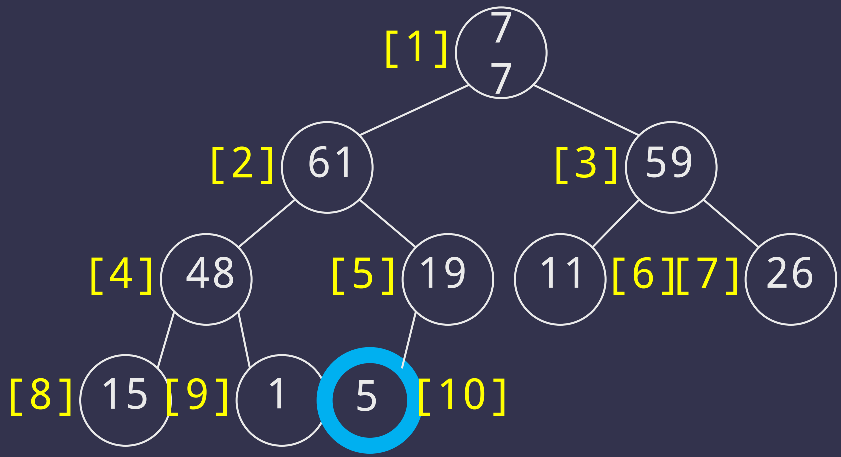
root = 1

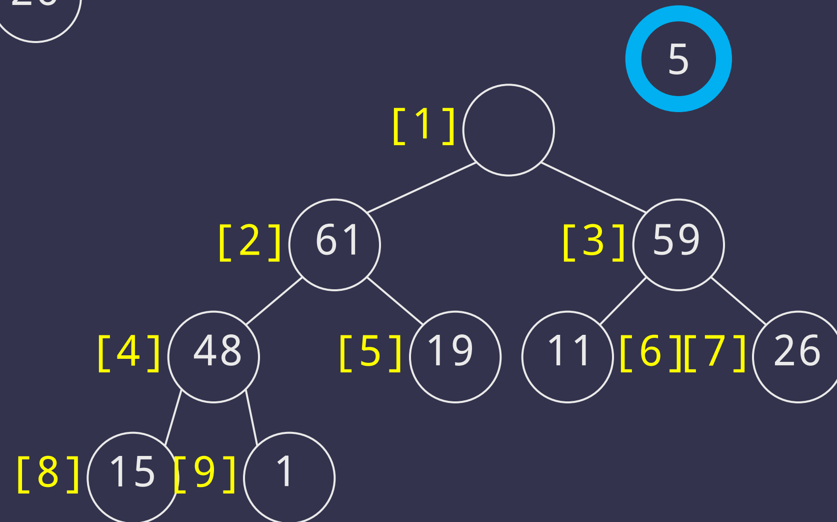
n = 10

rootkey = 26

child =

- ***adjust***
  - Adjust a max heap without root.





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

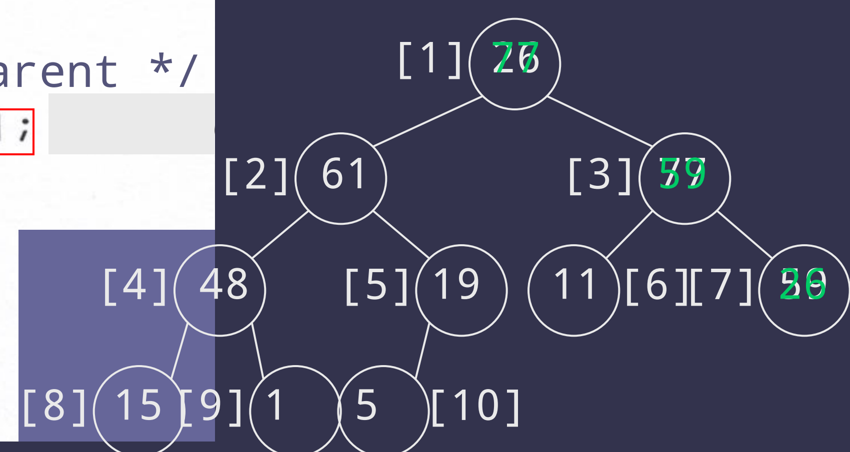
```
void adjust(element list[], int root, int n)
{
  int child,rootkey;
  element temp;
  temp = list[root];
  rootkey = list[root].key;
  child = 2 * root;          /* left child */
  while (child <= n) {
    if ((child < n) &&
    (list[child].key < list[child+1].key))
      child++;
    if (rootkey > 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 (3/3)

- ***heapsort***

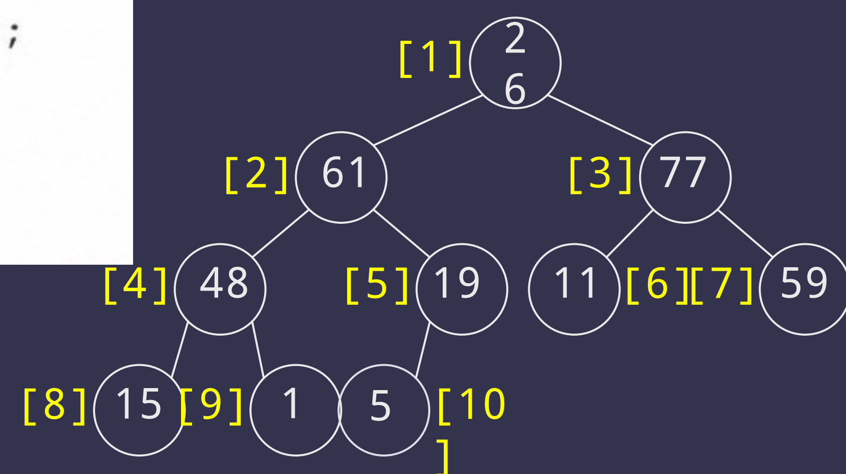
```
void heapsort(element list[], int n)
/* perform a heapsort on the array */
{
   int i,j;
   element temp;

   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);
   }
}
```

Make a heap

Sort

n = 10

i =



Heap Sort (3/3)

- ***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);
   }
}
```

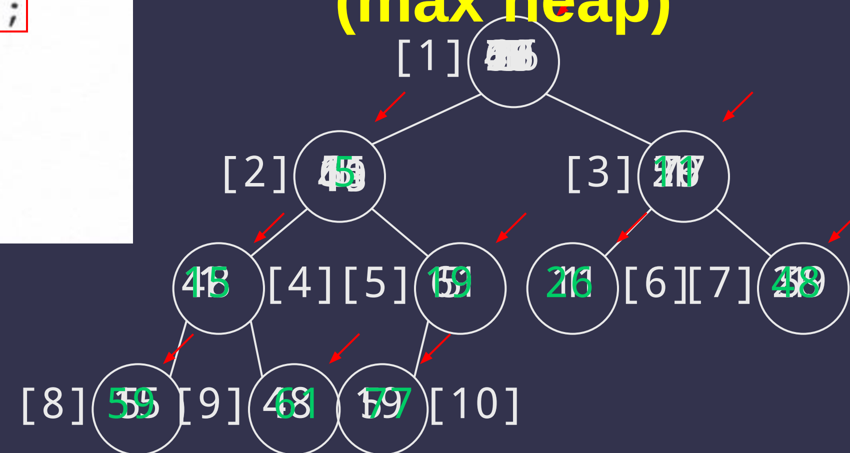
Make a heap

Sort

n = 10

i =

ascending order (max heap)



Radix Sort (1/8)

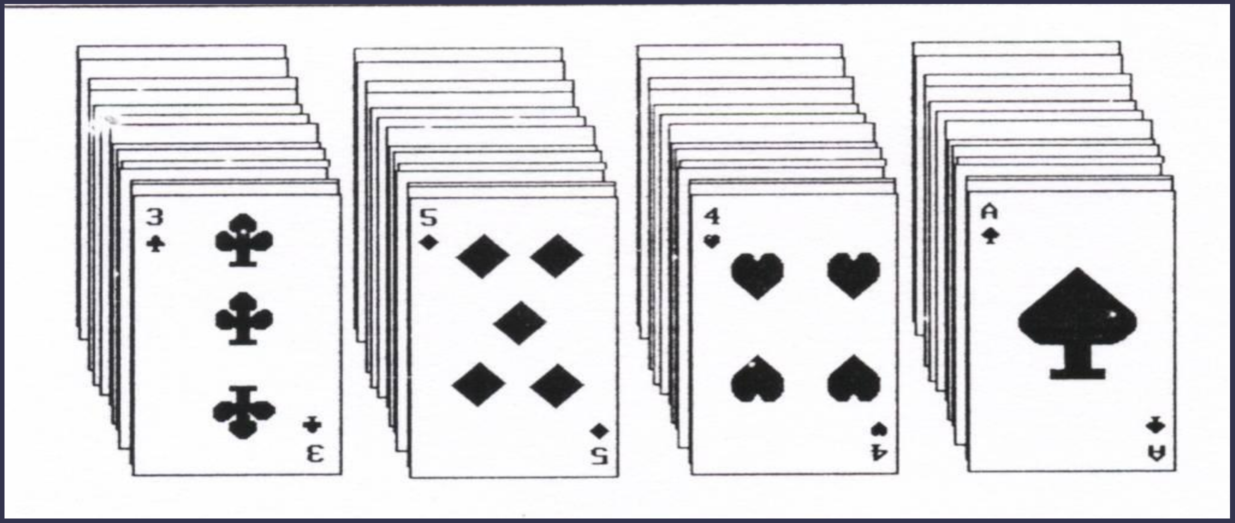
- We considers the problem of sorting records that have several keys
  - These keys are labeled K^0 (most significant key), K^1, … , K^{r-1} (least significant key).
  - Let K_i^j denote key K^j of record R_i.
  - A list of records R_0, … , R_{n-1}, is lexically sorted with respect to the keys K^0, K^1, … , 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 (2/8)

- 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 (3/8)

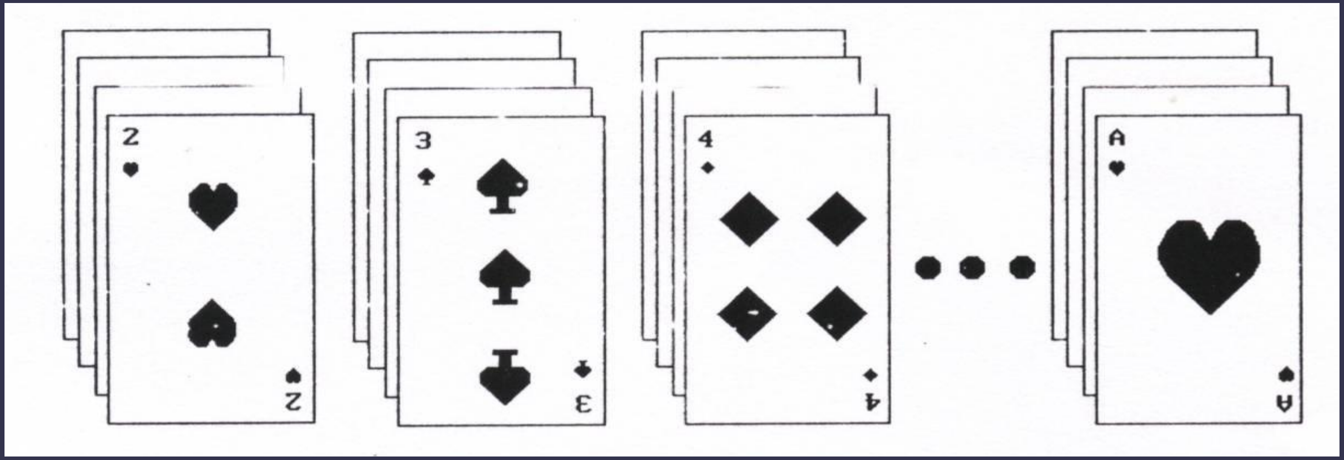
- MSD first
  1. MSD sort first, e.g., bin sort, four bins ♣ ♦ ♥ ♠
  2. LSD sort second
  - Result: 2♣, …, A♣, … , 2♠, … , A♠



Radix Sort (4/8)

- LSD first
  1. LSD sort first, e.g., face sort,
     13 bins 2, 3, 4, …, 10, J, Q, K, A
  2. MSD sort second (we can just classify these 13 piles into 4 separated piles by considering them from face 2 to face A)

Result:
2♣, …, A♣, … ,
2♠, …, A♠



Radix Sort (5/8)

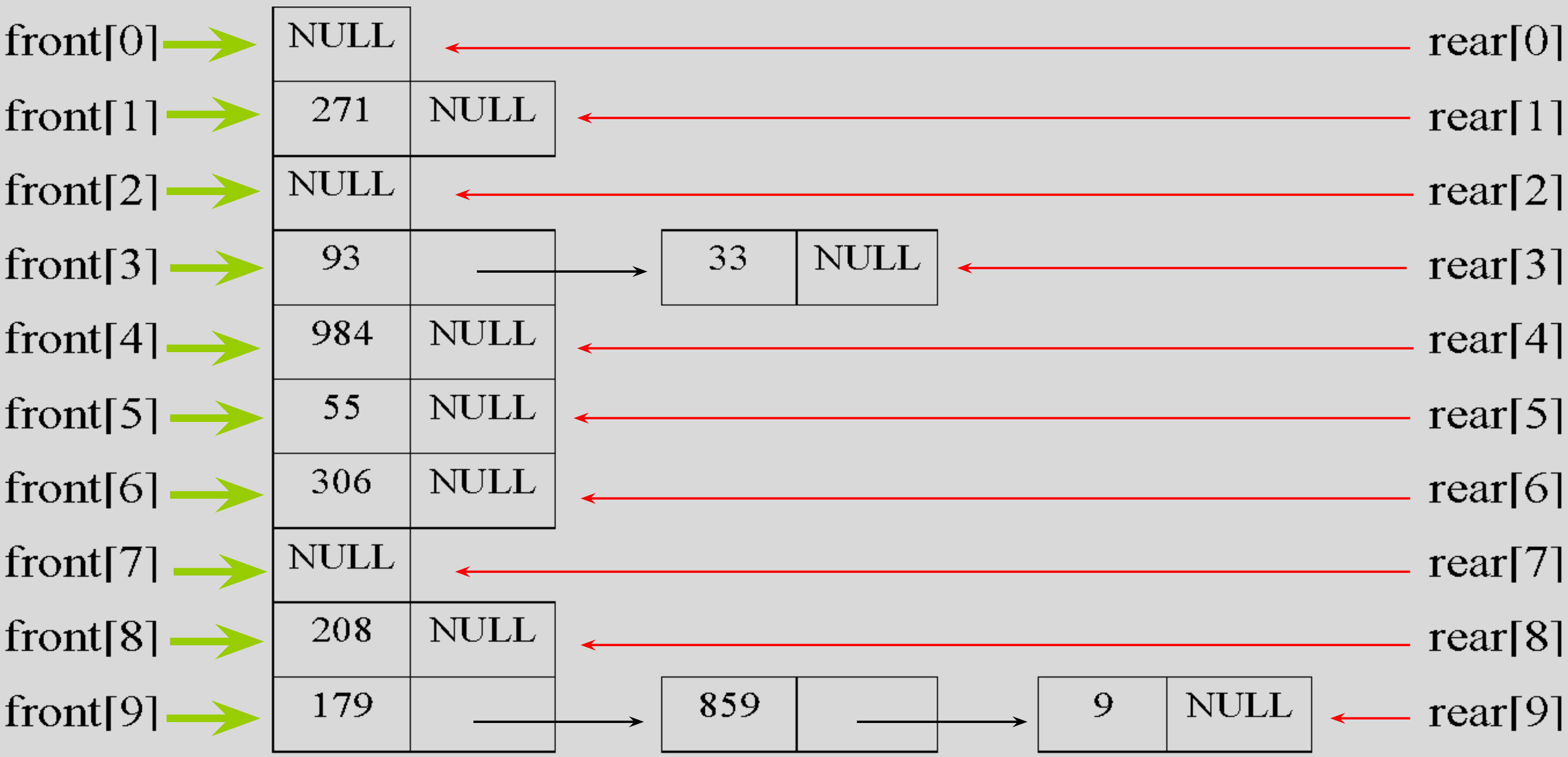
- We also can use an LSD or MSD sort when we have only one logical key, if we interpret this key as a composite of several keys.
- Example:
  - integer: the digit in the far right position is the least significant and the most significant for the far left position
  - range: $0 \le K \le 999$

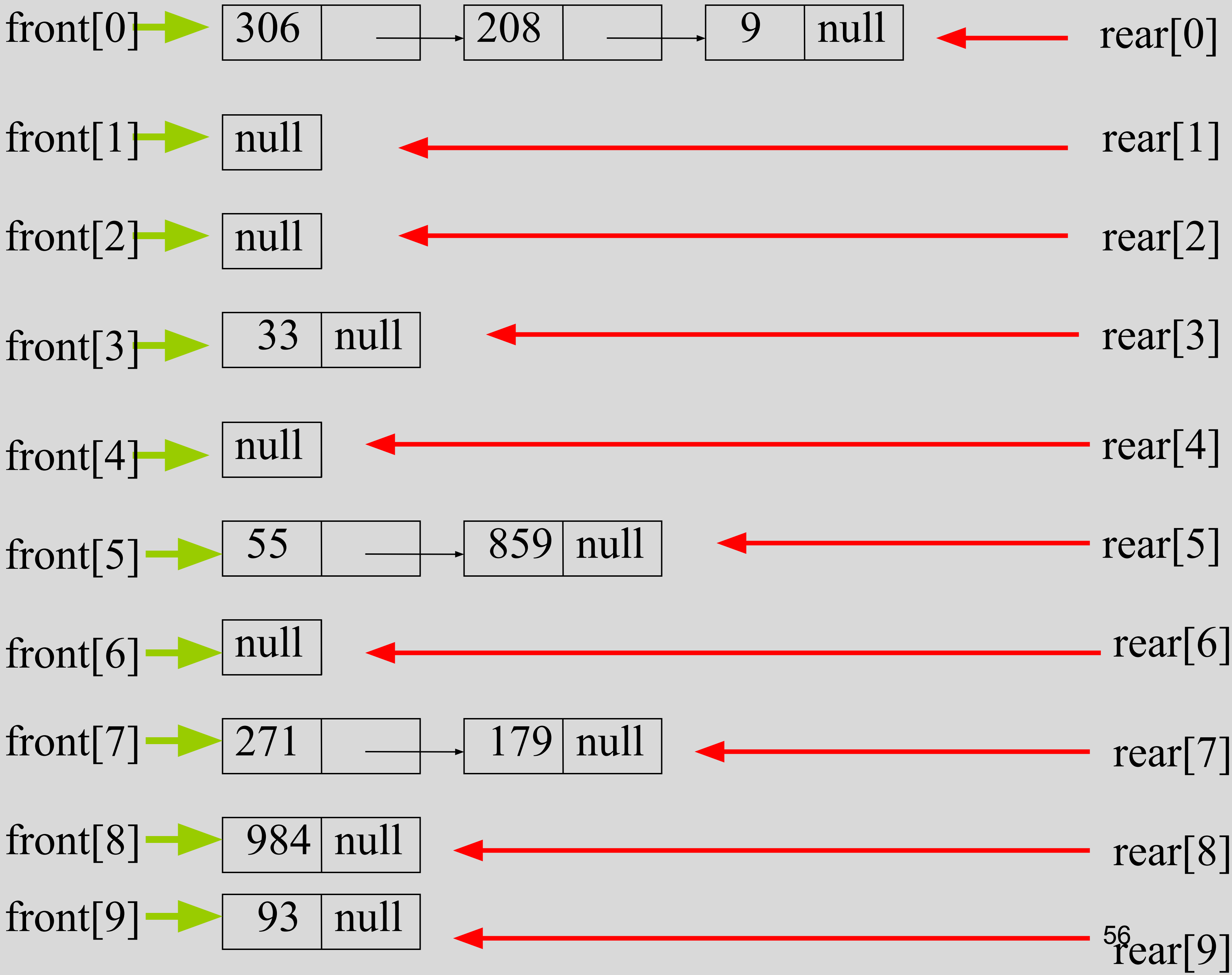
MSD		LSD
0-9	0-9	0-9

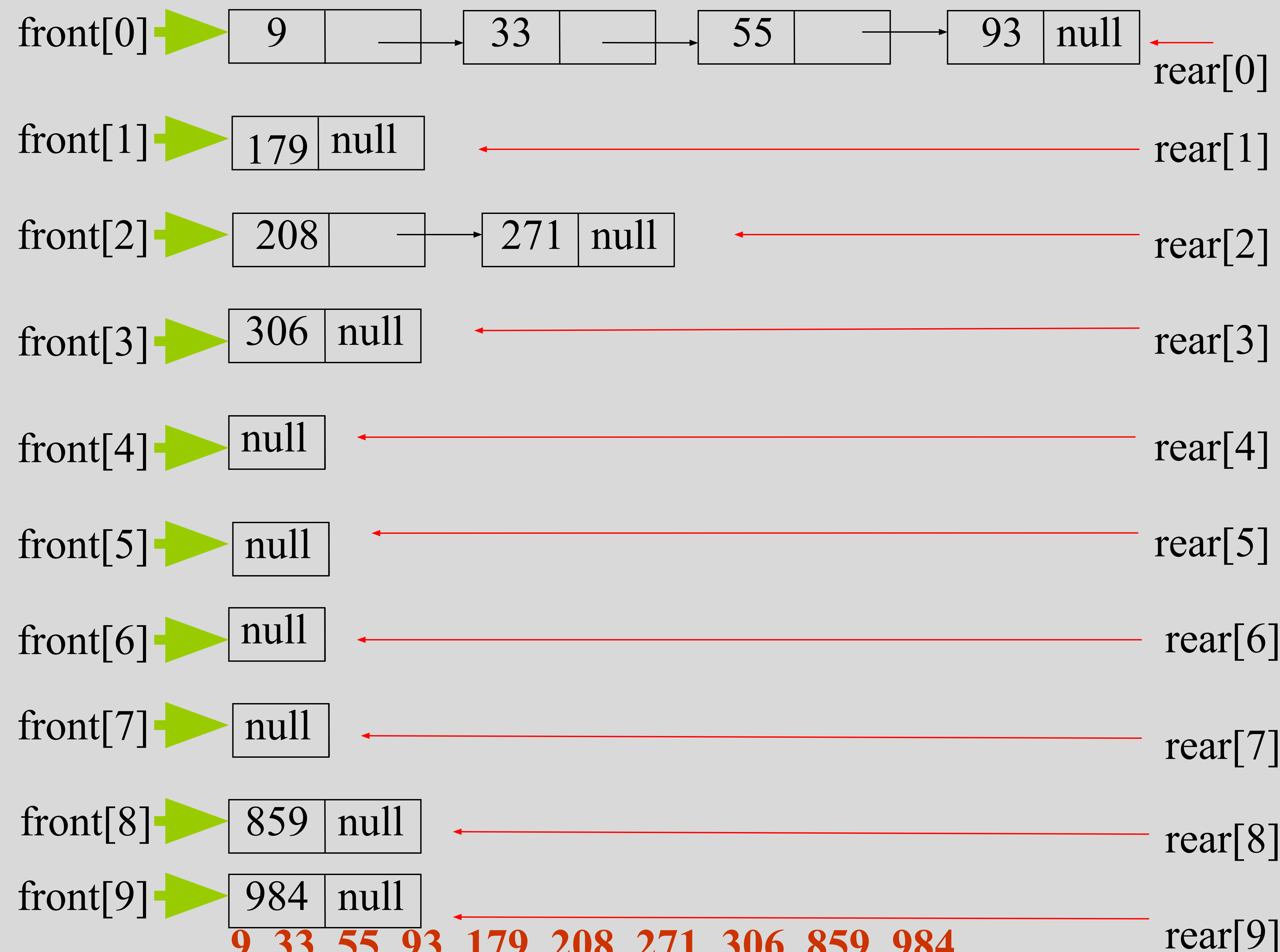
  - using LSD or MSD sort for three keys (K^0, K^1, K^2)
  - since an LSD sort does not require the maintainence of independent subpiles, it is easier to implement

Example for LSD Radix Sort

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




front[0]
306
208
9
null
rear[0]
front[1]
null
rear[1]
front[2]
null
rear[2]
front[3]
33
null
rear[3]
front[4]
null
rear[4]
front[5]
55
859
null
rear[5]
front[6]
null
rear[6]
front[7]
271
179
null
rear[7]
front[8]
984
null
rear[8]
front[9]
93
null
rear[9]



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

List and Table Sorts

- Many sorting algorithms require excessive data movement since we must physically move records following some comparisons
- We can reduce data movement by using a linked list representation

List and Table Sorts

- 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.

In Place Sorting

i	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9
key	**26**	**5**	**77**	**1**	**61**	**11**	**59**	**15**	**48**	**19**
link	**8**	**5**	**-1**	**1**	**2**	**7**	**4**	**9**	**6**	**0**

Summary of Internal Sorting (1/2)

- 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 (2/2)

- 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