國立清華大學 電機工程學系

EE2410 Data Structure

Chapter 7 Sorting

Outline

- **Insertion Sort**
- **Quick Sort**
- **How Fast Can We Sort ?**
- **Merge Sort**
- **Heap Sort**
- **Sorting On Several Keys**
- **List and Table Sorts**

Basics

- **Terminology**
  - List →a collection of records having one or more fields
  - 關鍵值 – Key → the field used to distinguish among the records
- **Example: telephone directory**
  - a record has three fields: name, address, phone number
  - key is usually a person's name
  - key could also be the phone number

```
class  Element
{
public
    int   getKey() const { return key; }
    void  setKey(int k) { key = k; }
private:
    int key;  // other fields not shown here
};
```

每一筆資料以一個或多個關鍵值做為索引標準其他資料省略未列出

Sequential Search

```
int  SeqSearch (Element  *f, const int n, const int k)
// Search a list f with key values f[1].key, ..., f[n].key
// Return i such that f[i].key == k. If there is no such record, return 0
{
  int   i = n;  // set pointer to the last element initially
  f[0].setKey(k); // set the key of the 0th-element to k
  while ( f[i].getKey() != k )  i--;  // sequential search from n to 1
  return i;
}
```

Time complexity is O(n)

Trick: introduce a dummy record 0 with f[0].key = k

→ simplifies the body in while loop

→ End-of-list test is avoided

→ can achieve significant speedup when n is large

Sorting Problem

- **Given**
  - A list of records (R_1, R_2,..., R_n)
  - Each record, R_i, has key value K_i.
  - We assume an ordering relation (<) on the keys
    (1) for any two key values x and y, x=y, or x<y, or x>y
    (2) the relation is transitive (i.e., x<y, y<z, then x<z)
- **Definition of a stable sorting method**
  - The sorting problem is to find a permutation σ, such that $K_{\sigma(i)} \leqq K_{\sigma(i+1)}$ $1 \leqq i \leqq n-1$
  - If $i<j$ and $K_i == K_j$ in the input list, then R_i precedes R_j in the sorted list

Example: Internal Revenue Service

- **Two lists of tax forms that come in at random**
  - One from the employee with m forms → F1
  - One from the employers with n forms → F2
  - Keys are the social security numbers
- **Problems**
  - To make sure that the two lists are consistent
- **Complexity comparison**
  - direct search: O(mn)
  - sort each list and then compare: $O(t_{sort}(n) + t_{sort}(m) + m + n)$
    where $t_{sort}(n)$ is the time to sort n records → $O(n \cdot \log n)$
    Therefore, the total time is $O(\max\{ n \cdot \log n, m \cdot \log m \})$

Sorting Methods Classification

- **Internal Sorting**
  - Methods to be used when the list to be sorted is small enough so that the entire sorting can be carried out in main memory
  - inserting sort, quick sort, merge sort, heap sort, and radix sort
- **External Sorting**
  - Methods to be used on larger lists

Insertion Sort

Initial list of records: $R_0\ R_1 \ldots R_i$ ($K_1 \leqq K_2 \leqq \ldots \leqq K_i$)

```
int insert(const Element e, Element *list, int i)
// Insert element e with key e.key into the ordered sequence list
// list[0], ..., list[i], such that the resulting sequence is also ordered.
// Assume that e.key ≧ list[0].key
// The array list must have space allocated for at least i+2 elements
{
  while ( e.getKey( ) < list[i].getKey( ) ) {
      list[i+1] = list[i];    i--;
  }
  list[i+1] = e;
}
```

The R_0 is an artificial record with key K_0=MININT (i.e., all keys are $\geqq K_0$)

```
int InsertionSort( Element *list, const int n)
// Sort list in nondecreasing order of key
{
  list[0].setKey(MININT);
  for ( int j=2; j<=n; j++){
      insert( list[j], list, j-1);
}
```

Example of Inserting Sort

Example:

	j	[1]	[2]	[3]	[4]	[5]
	-	5	4	3	2	1
list after insertion of 4	2	4	5	3	2	1
list after insertion of 3	3	3	4	5	2	1
list after insertion of 2	4	2	3	4	5	1
list after insertion of 1	5	1	2	3	4	5

- **Complexity**
  - $O(1+2+3+\cdots+n-1) = O(n^2)$
- **Relative disorder in the input list**
  - A record R_i is left out of order (LOO) iff $R_i < \max_{1 \leq j \leq i} \{R_j\}$
  - Let k be the number of LOO records, the computing time is $O(k \cdot n)$

Variations

- **Binary Search Sort**
  - The number of comparisons made is reduced
  - But the complexity remains unchanged, because the number of records moved is not changed
- **List Insertion Sort**
  - The elements of the list are represented as a linked list rather than an array
  - No record movement
  - However, the search is still sequential

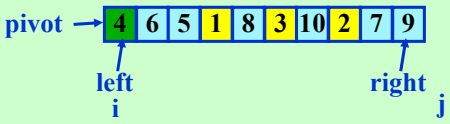
Outline

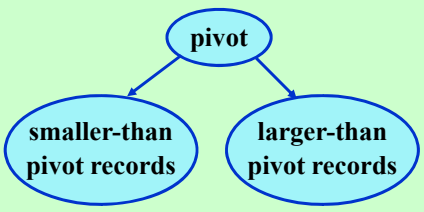
- Insertion Sort
- Quick Sort
- How Fast Can We Sort ?
- Merge Sort
- Heap Sort
- Sorting On Several Keys
- List and Table Sort

Quick Sort

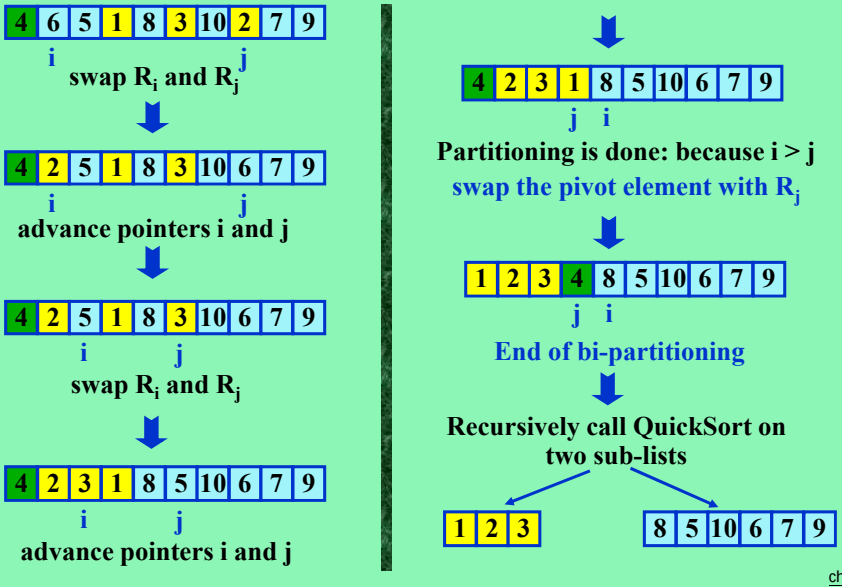
```
void  QuickSort( Element *list, const int left, const int right)
// Sort records list[left], · · ·, list[right] into nondecreasing order on field key
// Key pivot = list[left].key
{
   if ( left < right ) {
      int   i = left,
            j = right + 1,
            pivot = list[left].getKey();
      do {
            do i++; while ( list[i].getKey() < pivot );  // forward search for larger records
            do j--; while ( list[j].getKey() > pivot); // backward search for smaller records
            if ( i < j )  InterChange( list, i, j );
      } while ( i < j );
      InterChange(list, left, j);

      QuickSort( list, left, j-1);
      QuickSort( list, j+1, right);
   }
}
```



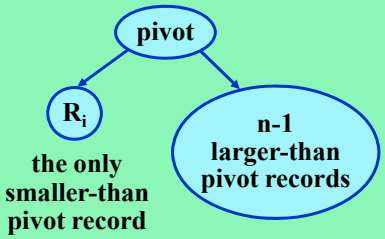


Example Of Quick Sort



How Fast is Quick Sort ?

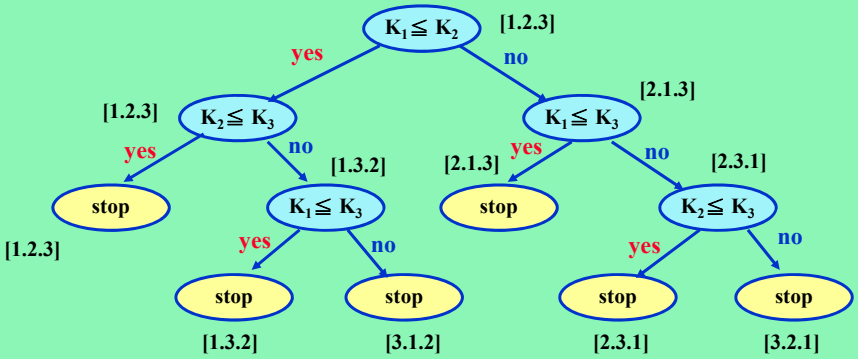
- **Average case:**
  - T(n) = cn + 2T(n/2)
  - T(n) = O(n · long n)
- **Worst case**
  - T(n) = cn + T(n-1)
  - $T(n) = O(n^2)$
- **Variation**
  - A better pivot selection: pivot= median $\{K_l, K_{(l+r)/2}, K_r\}$
- **It has been observed that**
  - quick sort is the fastest sorting algorithm on average



worst-case of quick sort

How Fast Can We Sort?

- **The Sorting Algorithm**
  - The best possible time is $O(n \cdot \log n)$
  - That is, the worst-case computing time $\Omega(n \cdot \log n)$



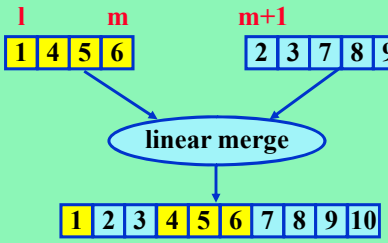
The height of this decision tree:
$\log_2(n!) + 1 \geqq \log_2((n/2)^{(n/2)}) = \Omega(n \cdot \log n)$

Outline

- Insertion Sort
- Quick Sort
- How Fast Can We Sort ?
- Merge Sort
- Heap Sort
- Sorting On Several Keys
- List and Table Sorts

Merging Of Two Sorted Lists

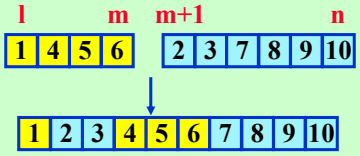
- **Given two sorted lists**
  - **List 1: (initList$_l$, · · · , initList$_m$)**
  - **List 2: (initList$_{m+1}$, · · · , initList$_n$)**
- **Output the merged list of the two lists**
  - **merged list: (mergedList$_l$, · · · , mergedList$_n$)**



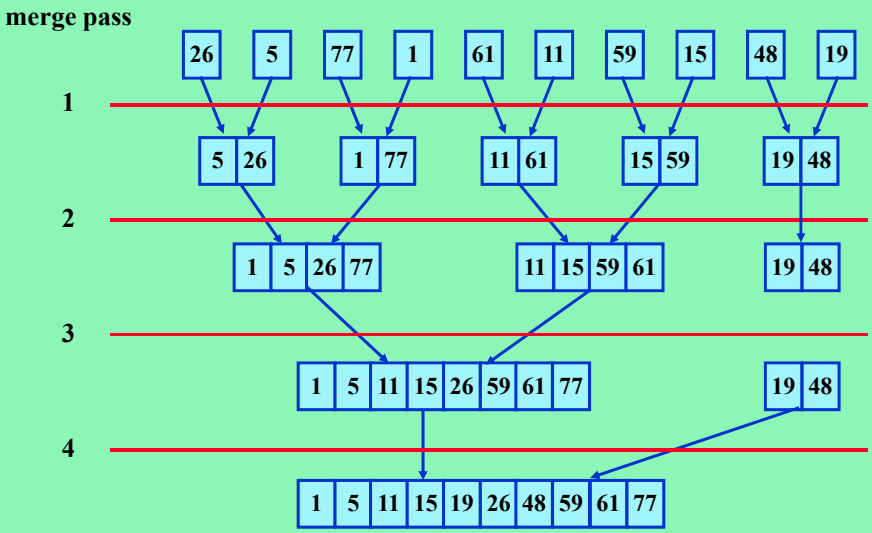
Time = O(n)
Space = O(n)
n is the total record number

Algorithm of Merging Two Lists

```
void  merge (Element *initList, Element *mergedList,
                  const int l, const int m, const int n)
{
   for ( int i1 = l, iResult = l, i2 = m+1; // i1, i2, and iResult are positions
         i1 <= m && i2 <=n; // both input lists not yet exhausted
         iResult++) {
        if ( initList[i1].getKey() <= initList[i2].getKey() ) {
             mergedList[iResult] = initList[i1];
             i1++;
        }
        else {
             mergedList[iResult] = initList[i2];
             i2++;
        }
   }
   if (i1 > m)   for ( t=i2; t<=n; t++) { mergedList[iResult+t-i2] = initList[t]; }
   else          for ( t=i1; t<=m; t++) { mergedList[iResult+t-i1] = initList[t]; }
}
```

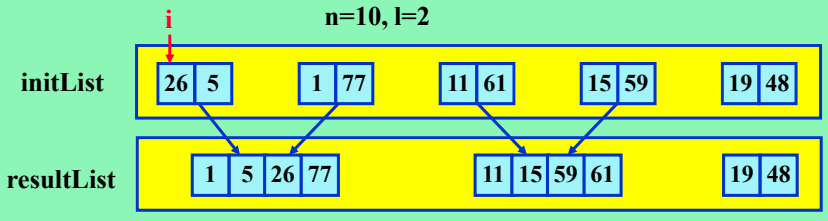


Iterative Merge Tree



MergePass Algorithm

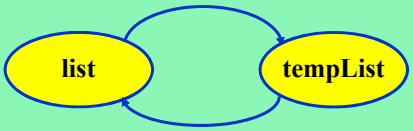
```
void  MergePass(Element *initList, Element *resultList, const int n, const int l)
// One pass of merge sort. Adjacent pairs of sublists of length l are merged
// from list initList to list resultList. n is the number of records in initList
{
  for (int  i=1; // i is the first position in the first of the two sublists being merged
        i <= n-2*l + 1;  // Are enough elements left to form two sublists of length l?
        i = i + 2*l ) {
          merge ( initList, resultList, i, i+l-1, i+2*l-1);
  }
  // merge remaining list of length < 2*l
  if ( ( i+l-1) < n)  merge ( initList, resultList, i, i+l-1, n);  // two sublists
  else  for ( int t=i; t<=n; t++)  resultList[t] = initList[t];  // one sublist
}
```



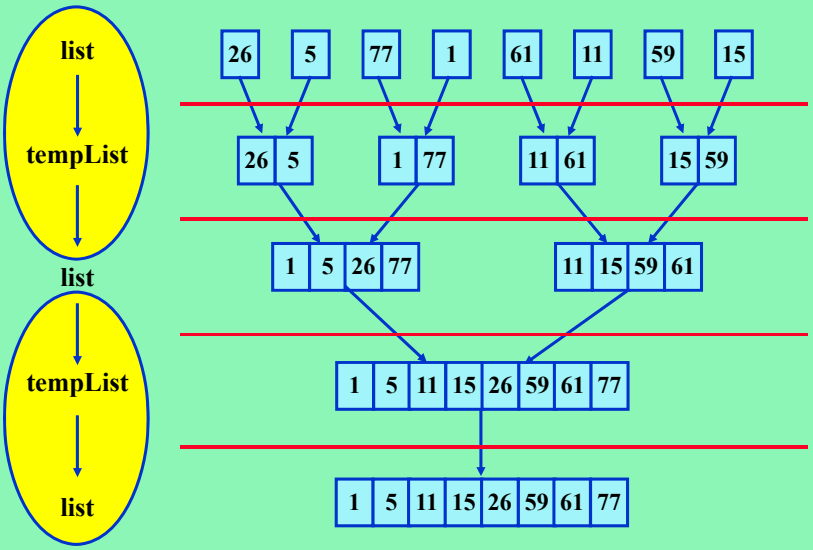
Merge Sort Algorithm

```
void  MergeSort( Element *list, const int n)
// Sort a list into nondecreasing order of the keys list[1].key, ..., list[n].key
{
   Element  *tempList = new Element[n+1];
   //  l is the length of the sublist currently being merged
   for ( int  l = 1; l < n;  l = l *2 )
   {
        MergePass ( list, tempList, n, l);
        l = l * 2;
        MergePass ( tempList, list, n, l);  // interchange role of list and tempList
   }
   delete  [ ]  tempList;
}
```

list and tempList hold the partially sorted list alternatively



Two Alternating Lists

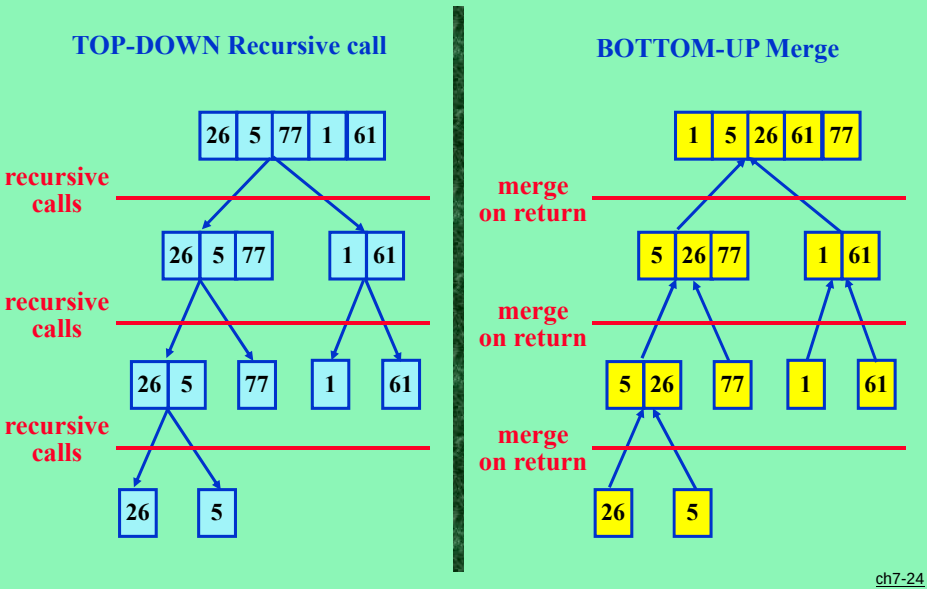


Recursive Merge Sort Algorithm

```
class  Element
{
private:
   int  key;  Field other;  int  link;
public:  Element() { link = 0; }
};
```

```
int  rMergeSort ( Element *list, const int left, const int right )
// List = ( list[left], ..., list[right] ) is to be sorted on the field key
// link is a field in each record that is initially 0
// rMergeSort returns the index of the first element in the sorted chain
// list[0] is a record for intermediate results used only in ListMerge
{
   if ( left >= right) return left;
   int  mid  = (left + right) /2;
   return ListMerge( list,
                     rMergeSort(list, left, mid), // sort left half
                     rMergeSort(list, mid+1, right); // sort right half
}
```


Execution of Recursive Merge Sort

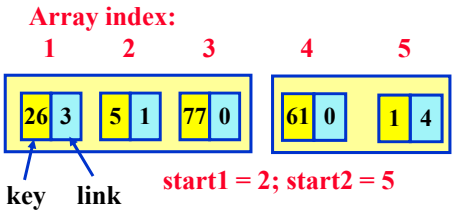


List Merge Algorithm

```
int ListMerge(Element *list, const int start1, const int start2)
// The sorted linked lists whose first elements are indexed by start1 and start2,
// respectively, are merged to obtain the sorted linked list. The index of the first element of the
// sorted list is returned. Integer links are used.
{
   int iResult = 0;
   for (int i1 = start1, i2 = start2; i1 && i2; )
      if (list [i1] . key <= list [i2] . key) {
         list [iResult ] . link = i1;
         iResult = i1; i1 = list [i1] . link;
      }
      else {
         list [iResult ] . link = i2;
         iResult = i2; i2 = list [i2] . link;
      }

   // move remainder
   if (i1 == 0) list [iResult ] . link = i2;
   else list [iResult ] . link = i1;
   return list [0] . link;
}
```

list[0] is the header element



Outline

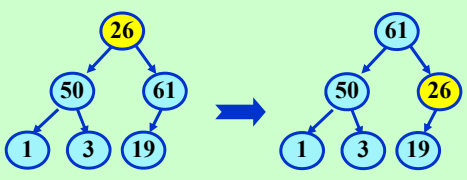
- Insertion Sort
- Quick Sort
- How Fast Can We Sort ?
- Merge Sort
- Heap Sort
- Sorting On Several Keys
- List and Table Sorts

Basics of Heap Sort

- **Procedure**
  - Step 1: build the given list as a max heap
  - Step 2: extract one record at a time from the heap
- **Time Complexity**
  - worst case: $O(n \cdot \log n)$
  - average case: $O(n \cdot \log n)$
- **Space Complexity**
  - only a fixed amount of additional storage is needed: $O(1)$
- **Heap Sort is not stable**

Adjust Routine

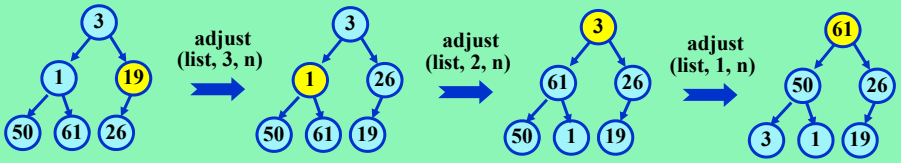
```
void  adjust ( Element *tree, const  int  root, const  int n)
// Adjust the binary tree with root to satisfy the heap property. The left and right
// subtrees of root already satisfy the heap property.
// No node has index greater than n
{
   Element  e = tree[root];
   int  k = e.getKey();
   for ( int j=2*root; j<=n; j*=2)
   {  // first find max of left and right child
        if ( j < n )
            if ( tree[j].getKey() < tree[j+1].getKey() )  j++;  // select larger child to j
        // compare max child with k. If k is max, then done
        if ( k >= tree[j].getKey() )  break;
        tree[j/2] = tree[j];  // move jth record up the tree
    }
    tree[j/2] = e;
}
```



Heap Sort

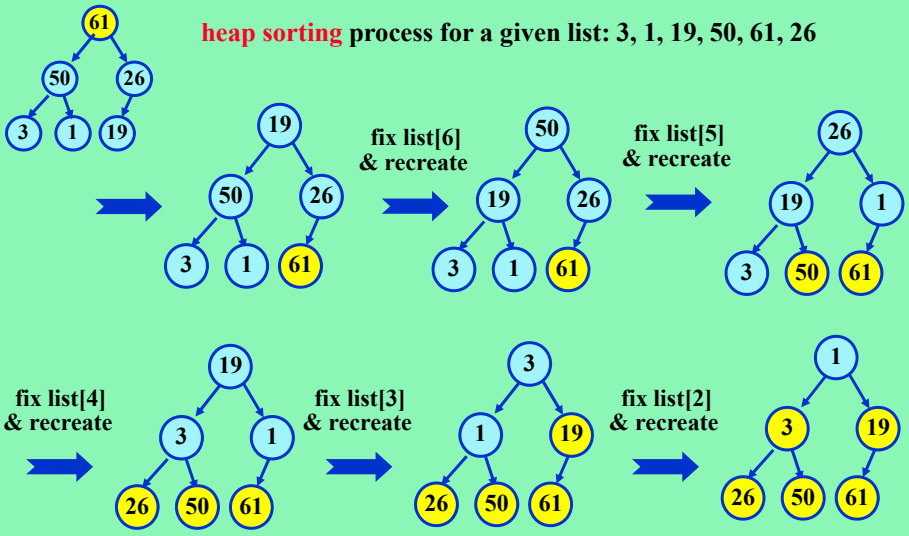
```
void  HeapSort ( Element *list, const int n)
//  The list = (list[1], ..., list[n]) is sorted into nondecreasing order of the field key
{
   for ( int  i = n/2 ;  i>=1; i--) // convert list into a heap
         adjust ( list, i, n);
   for ( i = n-1;  i>=1;  i--) // sort list
   {
        Element t = list[i+1]; // interchange list1 and listi+1
        list[i+1] = list[1];
        list[1] = t;
        adjust(list, 1, i);  // recreate the heap
   }
}
```

heap construction process for a given list: 3, 1, 19, 50, 61, 26



Step 2 of Heap Sorting

heap sorting process for a given list: 3, 1, 19, 50, 61, 26



Outline

- Insertion Sort
- Quick Sort
- How Fast Can We Sort ?
- Merge Sort
- Heap Sort
- Sorting On Several Keys
- List and Table Sorts

Basics of Sorting On Several Keys

- **Terminology**
  - Keys: $K^1, K^2, ..., K^r$
  - K^1 is the most significant key
  - K^r is the least significant key
- **Comparison of multiple key**
  - The r-tuple $(x^1, x^2, ..., x^r)$ is less than or equal to the r-tuple $(y^1, y^2, ..., y^r)$ iff either one of the following two conditions is satisfied

    (1) $x^i = y^i$ for $1 \leqq i \leqq r$,

    (2) $x^i = y^i$ for $1 \leqq i < \alpha$, and $x^\alpha < y^\alpha$ for some $1 \leqq \alpha \leqq r$

    E.g., (1, 2, 3) < (1, 2, 5) → $\alpha = 3$

Most Significant Digit First Sorting

- **Example**
  - Sorting a deck of cards
  - The first key K^1: suit (spade, heart, diamond, club)
  - The second key K^2: face value (2, 3, ..., J, Q, K, A)
- **Most Significant Digit (MSD) First Sorting**
  - Sort the cards into 4 piles using K^1, one for each suit
  - Sort each of the 4 piles using K^2
  - Cascade the sorted 4 piles with the order of (spade, heart, diamond, club)

Least Significant Key Sorting

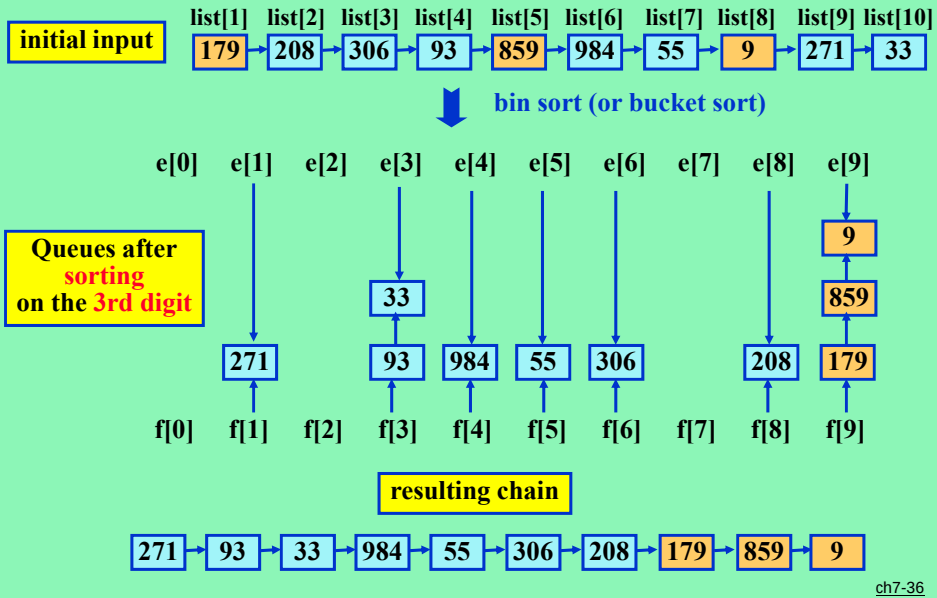
- **Example**
  - Sorting a deck of cards
  - The first key K^1: suit (spade, heart, diamond, club)
  - The second key K^2: face value (2, 3, ..., J, Q, K, A)
- **Least Significant Digit (LSD) First Sorting**
  - Sort the cards into 13 piles using K^2
  - Then, cascade the 13 piles into a big pile with the order of 2, 3, 4, ..., J, Q, K, A
  - Finally, sort the big pile using a stable sorting algorithm

LSD Radix Sort

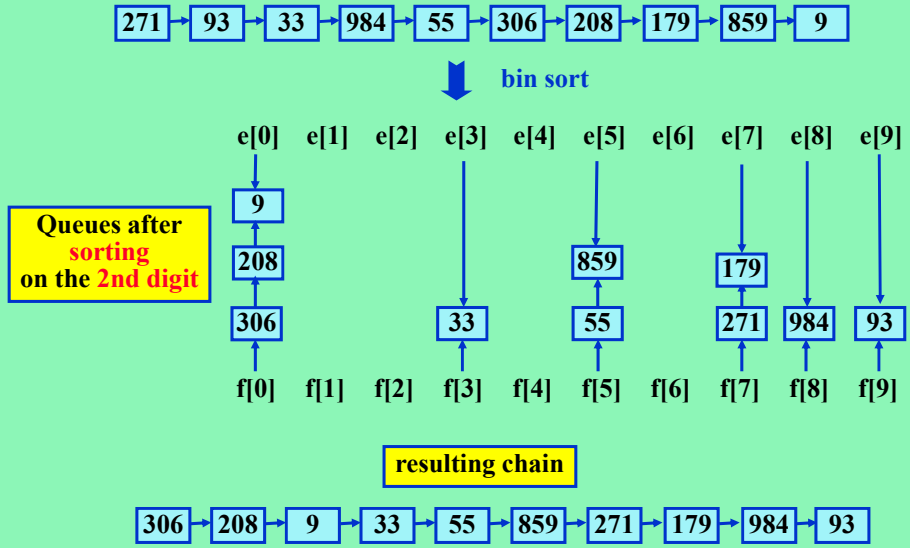
```
void RadixSort(Element *list, const int d, const int n)
// Records list = (list[1], ···, list[n]) are sorted on the keys key[0], ···, key[d−1].
// The range of each key is 0 ≤ key[i] < radix. radix is a constant.
// Sorting within a key is done using a bin sort.
{
   int e[radix], f[radix]; // queue pointers
   for (int i = 1; i <= n; i++) list[i].link = i + 1; // link into a chain starting at current
   list[n].link = 0; int current = 1;
   for (i = d − 1; i >= 0; i−−) // sort on key key[i]
   {
      for (int j = 0; j < radix; j++) f[j] = 0; // initialize bins to empty queues
      for ( ; current; current = list[current].link) { // put records into queues
         int k = list[current].key[i];
         if (f[k] == 0) f[k] = current;
         else list[e[k]].link = current;
         e[k] = current;
      }
      for (j = 0; f[j] == 0; j++); // find first nonempty queue
      current = f[j]; int last = e[j];
      for (int k = j + 1; k < radix; k++) // concatenate remaining queues
         if (f[k]) {
            list[last].link = f[k];
            last = e[k];
         }
      list[last].link = 0;
   } // end of for (i = d − 1; i >= 0; i−−)
}
```

Complexity = O(d · n)

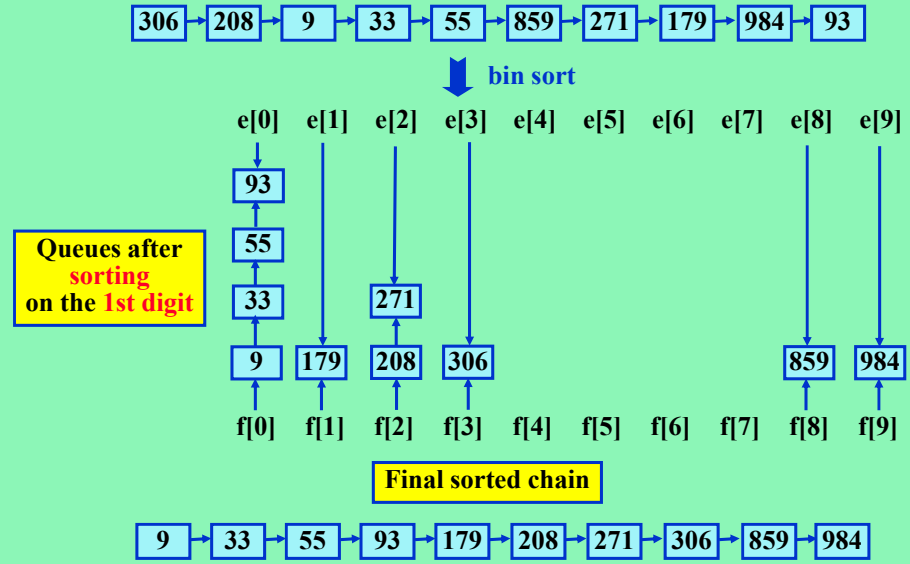
Radix Sort Example (I)



Radix Sort Example (II)



Radix Sort Example (III)



Performance Consideration

- **Excessive Data Movement**
  - tends to slow down the sorting process
- **Avoid data movement as much as possible**
  - Modifications can be made to insertion sort or merge sort
- **In-place post-sorting rearrangement**
  - may be needed to convert a sorted list to an array

名次與陣列註標一致

sorted list　　head

陣列註標	i	R1	R2	R3	R4
關鍵值	key	26	5	77	1
串列鏈結	link	3	1	0	2

sorted array

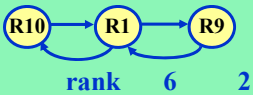
i	R1	R2	R3	R4
key	1	5	26	77
link	2	1	3	0

Example: In-Place Rearrangement

rank	6	2	10	1	9	3	8	4	7	5

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
linkb	10	4	5	0	7	2	9	6	1	8

first　　to be changed



fix the position of R4

rank	6	2	10	1	9	3	8	4	7	5

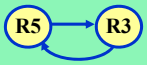
i	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
key	1	5	77	26	61	11	59	15	48	19
link	2	6	0	9	3	8	5	10	7	4
linkb	0	4	5	10	7	2	9	6	4	8

first　　{ R2, R3, ..., R10 } still forms a sorted list

Example: In-Place Rearrangement

rank	6	2	10	1	9	3	8	4	7	5
i	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
key	1	5	77	26	61	11	59	15	48	19
link	2	6	0	9	3	8	5	10	7	4
linkb	0	4	5	10	7	2	9	6	4	8

first



fix the position of R6

rank	6	2	10	1	9	3	8	4	7	5
i	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
key	1	5	11	26	61	77	59	15	48	19
link	2	6	8	9	6	0	5	10	7	4
linkb	0	4	2	10	7	5	9	6	4	8

first

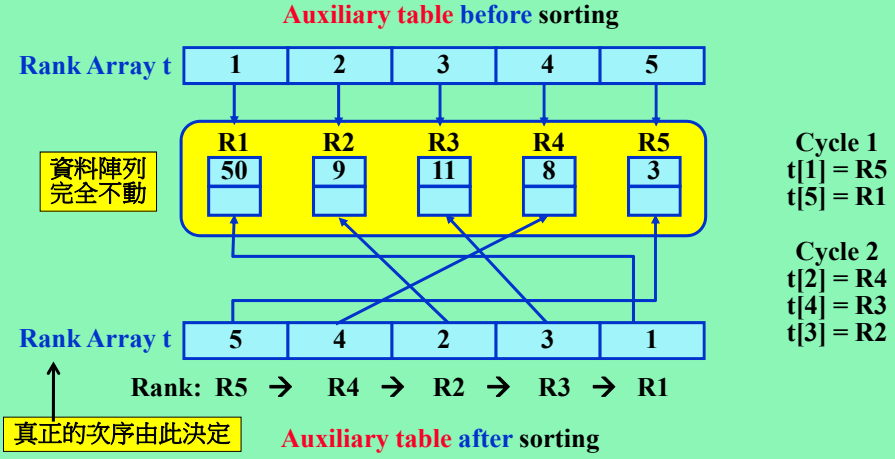
List2Array Algorithm

```
void list1(Element *list, const int n, int first)
// Rearrange the sorted chain first so that the records list [1], ··· , list [n ]
// are in sorted order. Each record has an additional link field linkb.
{
   int prev = 0;
   for (int current = first; current; current = list [current ] . link)
   // convert chain into a doubly linked list
   {
      list [current ] . linkb = prev;
      prev = current;
   }

   for (int i = 1; i < n ; i ++) // move list_first to position i while
   {                             // maintaining the list
      if (first != i) {
         if (list [i ] . link) list [list [i ] . link ] . linkb = first;
         list [list [i ] . linkb ] . link = first;
         Element a = list [first]; list [first] = list [i ]; list [i ] = a;
      }
      first = list [i ] . link;
   }
}
```

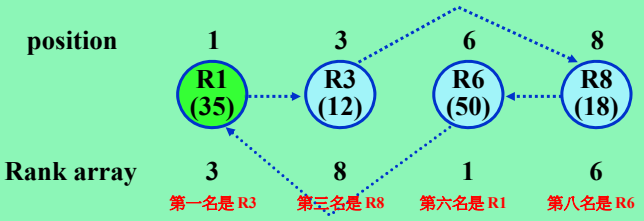
List-Based Array for Quick-Sort - To avoid data movement

We can use the following rank array for providing *array_based list* for Quick-Sort or Heap Sort → The i-th record is R[t[i]].



Re-Arrangement Within One Cycle

Before Re-Arrangement



After Re-Arrangement

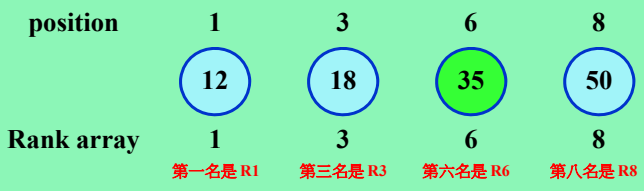


Table Sort Example

	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

t: rank array

(a) Initial configuration

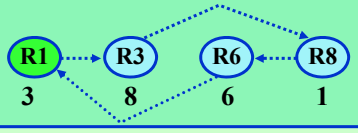
key	12	14	18	42	26	35	31	50
t	1	2	3	5	7	6	4	8

(b) Configuration after rearrangement of first cycle

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

(c) Configuration after rearrangement of second cycle

Table Sort




```
void table_sort ( Element *list, const int n, int *t )
// Rearrange list[1], ..., list[n] to correspond to the sequence
// list[ t[1]], ..., list[ t[n] ], n≧1
{
   for ( int i=1; i<n; i++) {
        if ( t[i] != i ) {  // There is a non-trivial cycle starting at i
             Element p = list[i];  int j = i;  // remember first record p=list[i]
             do { // Move record list[k] to position j
                  int k = t[j]; list[j] = list[k];  t[j] = j; j = k;
             } while ( t[j] != i );
             list[j] = p;   // Move record p to the last position in the cycle
             t[j] = j;
        } // end of if statement
   }
}
```