Sorting(3)

College of Computer Science, CQU

Outline

- Heapsort
- Binsort and Radixsort
- Comparison of Sorting Algorithms

Heap Sorting

- Heapsort is based on the heap data structure presented in Section 5.5.
- Step 1: Build a heap;   O(n)
- Step 2: removefirst( );   O(logn)

```
template <typename E, typename Comp>
void heapsort(E A[], int n) { // Heapsort
  E maxval;
  heap<E,Comp> H(A, n, n);    // Build the heap
  for (int i=0; i<n; i++)        // Now sort
    maxval = H.removefirst();    // Place maxval at end
}
```

Build a heap: siftdown ()

```
// Helper function to put element in its correct place
void siftdown(int pos) {
  while (!isLeaf(pos)) { // Stop if pos is a leaf
    int j = leftchild(pos);  int rc = rightchild(pos);
    if ((rc < n) && Comp::prior(Heap[rc], Heap[j]))
      j = rc;                // Set j to greater child's value
    if (Comp::prior(Heap[pos], Heap[j])) return; // Done
    swap(Heap, pos, j);
    pos = j;                 // Move down
  }
}
```

```
void buildHeap()              // Heapify contents of Heap
  { for (int i=n/2-1; i>=0; i--) siftdown(i); }
```

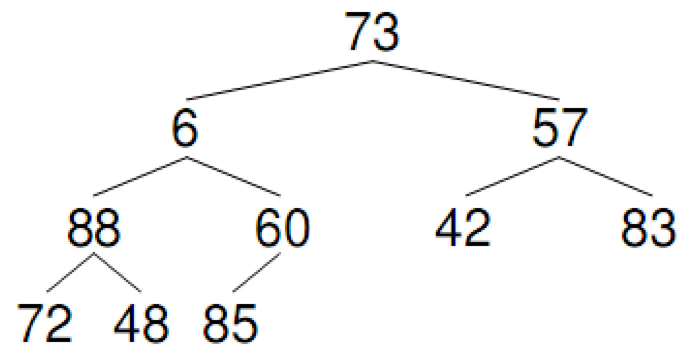
removefirst()& remove()

```
// Remove first value
E removefirst() {
  Assert (n > 0, "Heap is empty");
  swap(Heap, 0, --n);        // Swap first with last value
  if (n != 0) siftdown(0);  // Siftdown new root val
  return Heap[n];             // Return deleted value
}
```

Heap Sorting

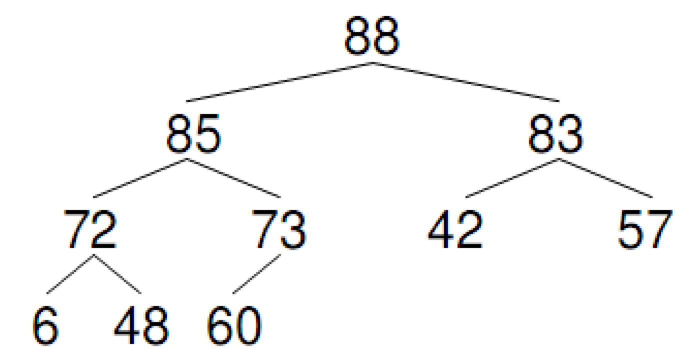
Original Numbers

73	6	57	88	60	42	83	72	48	85



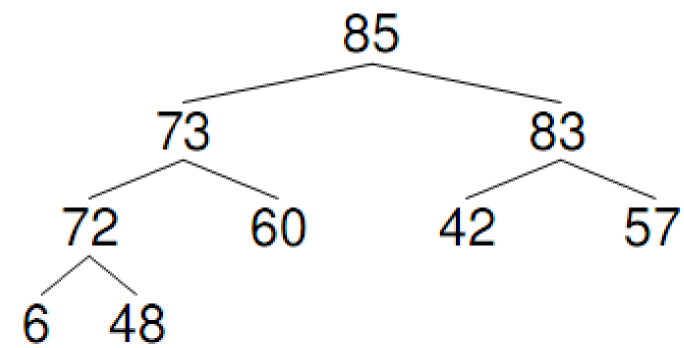
Build Heap

88	85	83	72	73	42	57	6	48	60



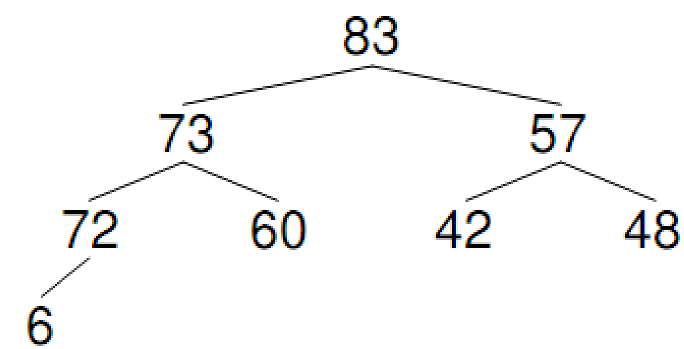
Remove 88

85	73	83	72	60	42	57	6	48	88



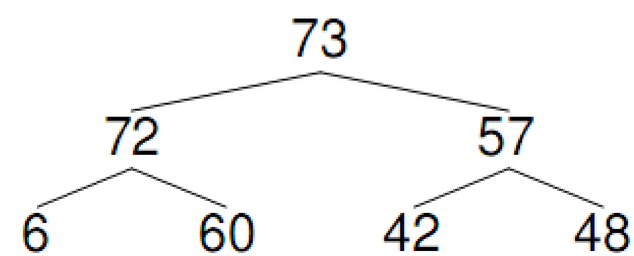
Remove 85

83	73	57	72	60	42	48	6	85	88



Remove 83

73	72	57	6	60	42	48	83	85	88



Binsort

The most basic example :

•to sort a permutation of the numbers 0 through n-1,

•key values are used to assign records to bins.

```
for (i=0; i<n; i++)
   B[A[i]] = A[i];
```

- extremely efficient algorithm, taking $\Theta(n)$ time regardless of the initial ordering of the keys.
- far better than the performance of any sorting algorithm that we have seen so far.
- The only problem is that this algorithm has limited use because it works only for a permutation of the numbers from 0 to n-1.
- Each bin contains one key value.

The extended Binsort

- We can extend this simple Binsort algorithm to be more useful.
- The simplest extension is to allow for duplicate values among the keys. This can be done by turning array slots into arbitrary-length bins by turning **B** into an array of linked lists. In this way, all records with key value *i* can be placed in bin **B[i]**.
- A second extension allows for a key range greater than *n*. The only requirement is that each possible key value have a corresponding bin in **B**. This version of Binsort can sort any collection of records whose key values fall in the range from 0 to **MaxKeyValue**-1.

The extended Binsort

```
template <typename E, class getKey>
void binsort(E A[], int n) {
  List<E> B[MaxKeyValue];
  E item;
  for (int i=0; i<n; i++) B[A[i]].append(getKey::key(A[i]));
  for (int i=0; i<MaxKeyValue; i++)
    for (B[i].setStart(); B[i].getValue(item); B[i].next())
      output(item);
}
```

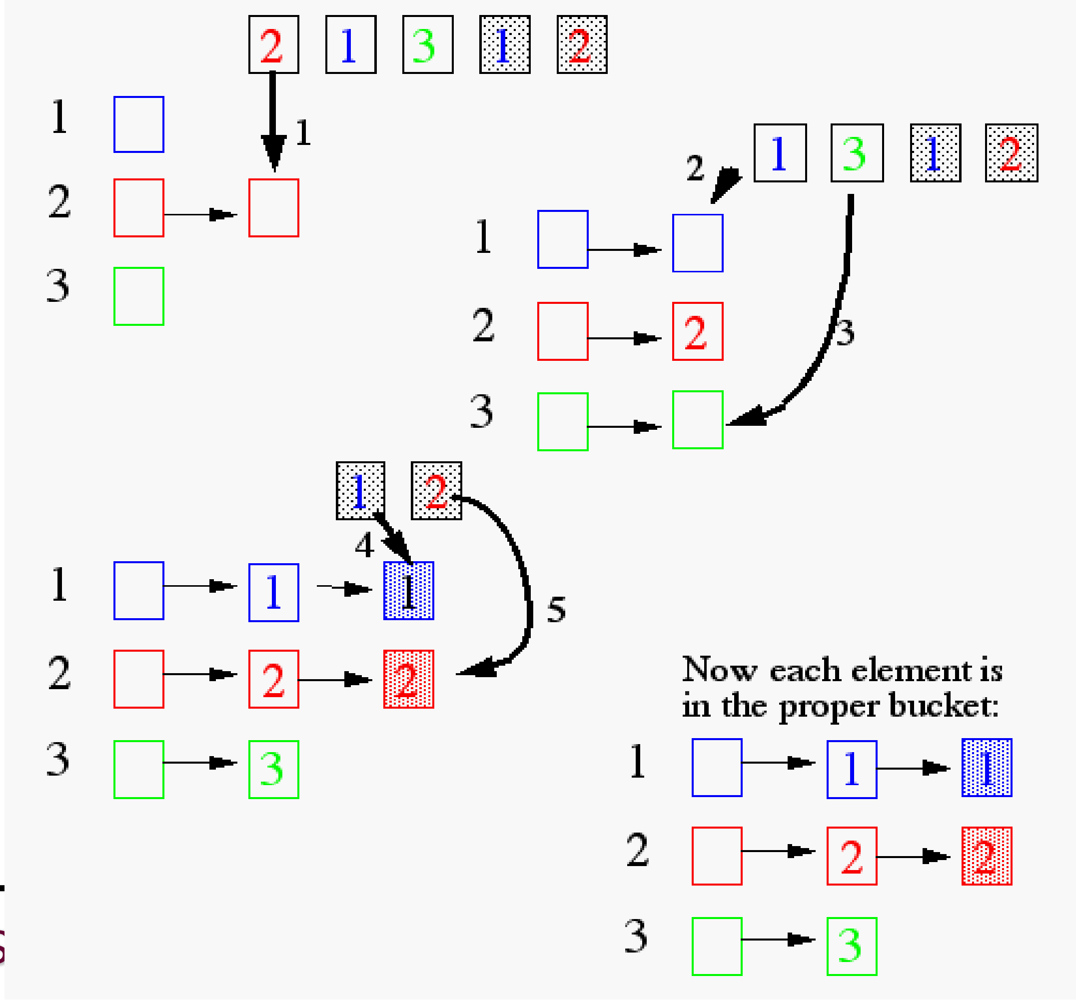
QUESTION?

Bucket sort: A generalization to Binsort

- **Each bucket contains multiple key values**
  - Assumption: the keys are in the range [0, N)
  - Basic idea:
    1. Create *N* linked lists (*buckets*) to divide interval [0,N) into subintervals of size 1
    2. Add each input element to appropriate bucket
    3. Concatenate the buckets
  - Expected total time is Θ(n + N), with n = size of original sequence
    - if N is $\Theta(n)$ → sorting algorithm in $\Theta(n)$!

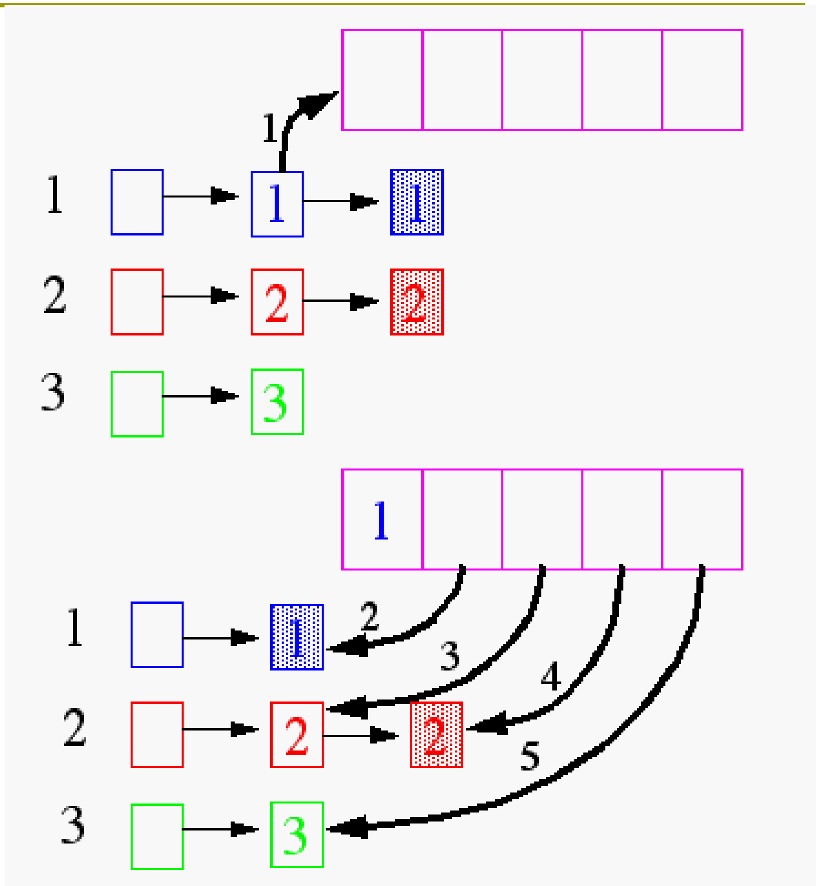
Bucket Sort

Each element of the array is put in one of the N “buckets”

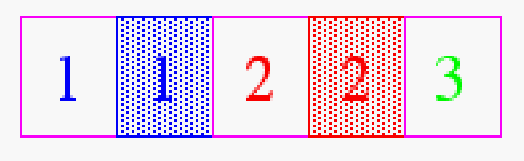


Bucket Sort

Now, pull the elements from the buckets into the array



At last, the sorted array (sorted in a stable way):



Radix Sort

- ***How did IBM get rich originally?***
- **Answer: punched card readers for census tabulation in early 1900's.**
  - In particular, a *card sorter* that could sort cards into different bins
    - Each column can be punched in 12 places
    - (Decimal digits use only 10 places!)
  - Problem: only one column can be sorted on at a time

Radix Sort

- Intuitively, you might sort on the most significant digit, then the second most significant, etc.
- Problem: lots of intermediate piles of cards to keep track of
- Key idea: sort the *least* significant digit first

```
RadixSort(A, d)
    for i=1 to d
        StableSort(A) on digit i
```

Radix Sort

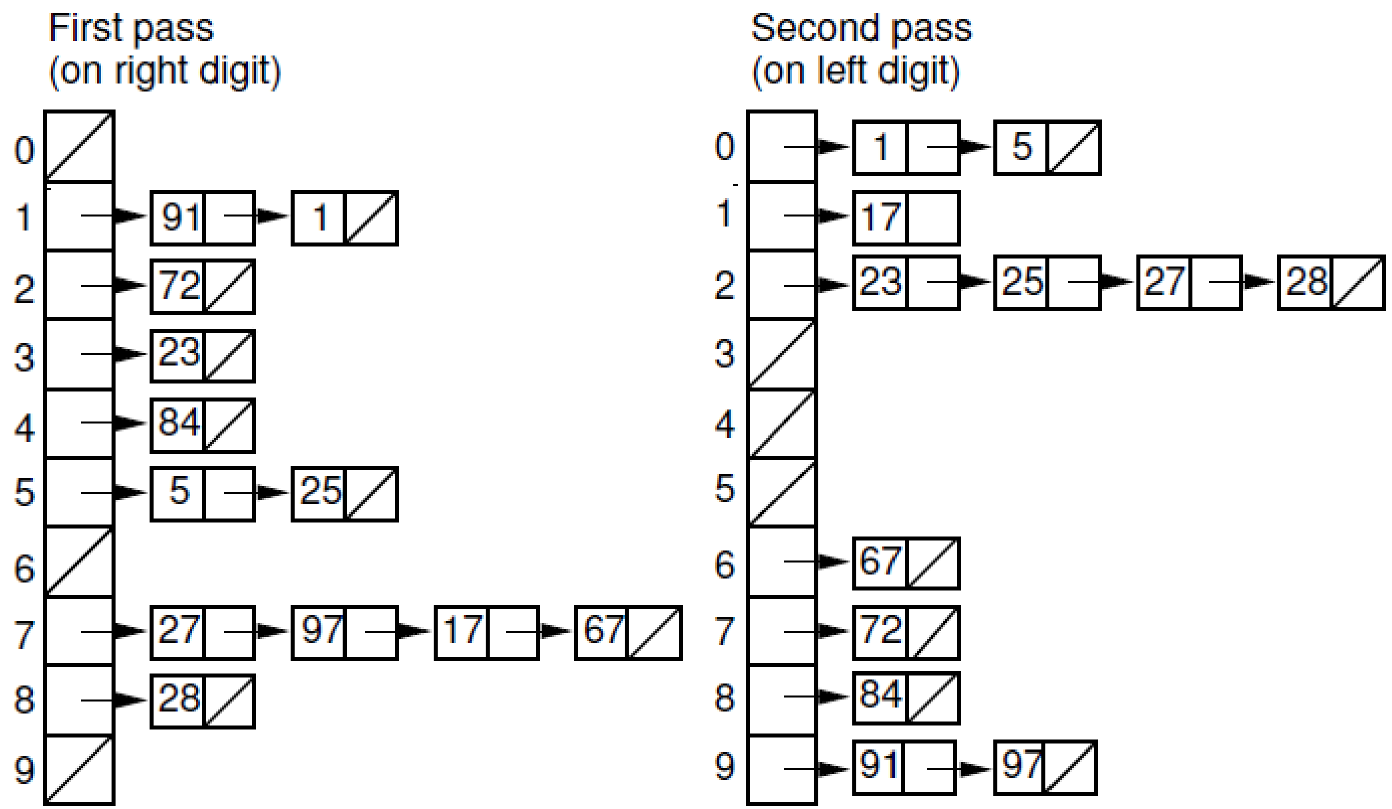
- ***What sort will we use to sort on digits?***
- **Bucket sort is a good choice:**
  - Sort n numbers on digits that range from 1..N
  - Time: O(n + N)
- **Each pass over n numbers with k digits takes time** $\Theta(n+r)$
- **so total time** $\Theta(nk+rk)$
  - When r is constant and k=O(1), takes $\Theta(n)$ time

Radix Sort Example

- **Problem: sort 1 million 64-bit numbers**
  - Treat as four-digit radix 2^{16} numbers
  - Can sort in just four passes with radix sort!
  - Running time: 4(1 million + 2^{16}) ≈ 4 million operations
- **Compare with typical O(*n* log *n*) comparison sort**
  - Requires approx log *n* = 20 operations per number being sorted
  - Total running time ≈ 20 million operations

Radix Sort Example

Initial List: 27 91 1 97 17 23 84 28 72 5 67 25



Result of first pass: 91 1 72 23 84 5 25 27 97 17 67 28
Result of second pass: 1 5 17 23 25 27 28 67 72 84 91 97

链式基数排序

假如多关键字的记录序列中，每个关键字的取值范围相同，则按LSD法进行排序时，可以采用**“分配-收集”**的方法，其好处是不需要进行关键字间的比较。

对于数字型或字符型的**单关键字**，可以看成是由多个数位或多个字符构成的**多关键字**，此时可以**采用**这种**“分配-收集”**的办法**进行排序，称作基数排序法**。

在计算机上实现基数排序时，为减少所需辅助存储空间，应采用链表作存储结构，即链式基数排序，具体作法为：

1．待排序记录以指针相链，构成一个链表；

2．“分配” 时，按当前“关键字位”所取值，将记录分配到不同的 “链队列” 中，每个队列中记录的 “关键字位” 相同；

3．“收集”时，按当前关键字位取值从小到大将各队列首尾相链成一个链表；

4．对每个关键字位均重复 2) 和 3) 两步。

例如：对下列这组关键字

{209, 386, 768, 185, 247, 606, 230, 834, 539 }

首先按其“个位数”取值分别为 0, 1, …, 9 “分配”成 10 组，之后按从 0 至 9 的顺序将它们“收集”在一起；

然后按其“十位数”取值分别为 0, 1, …, 9 “分配”成 10 组，之后再按从 0 至 9 的顺序将它们“收集”在一起；

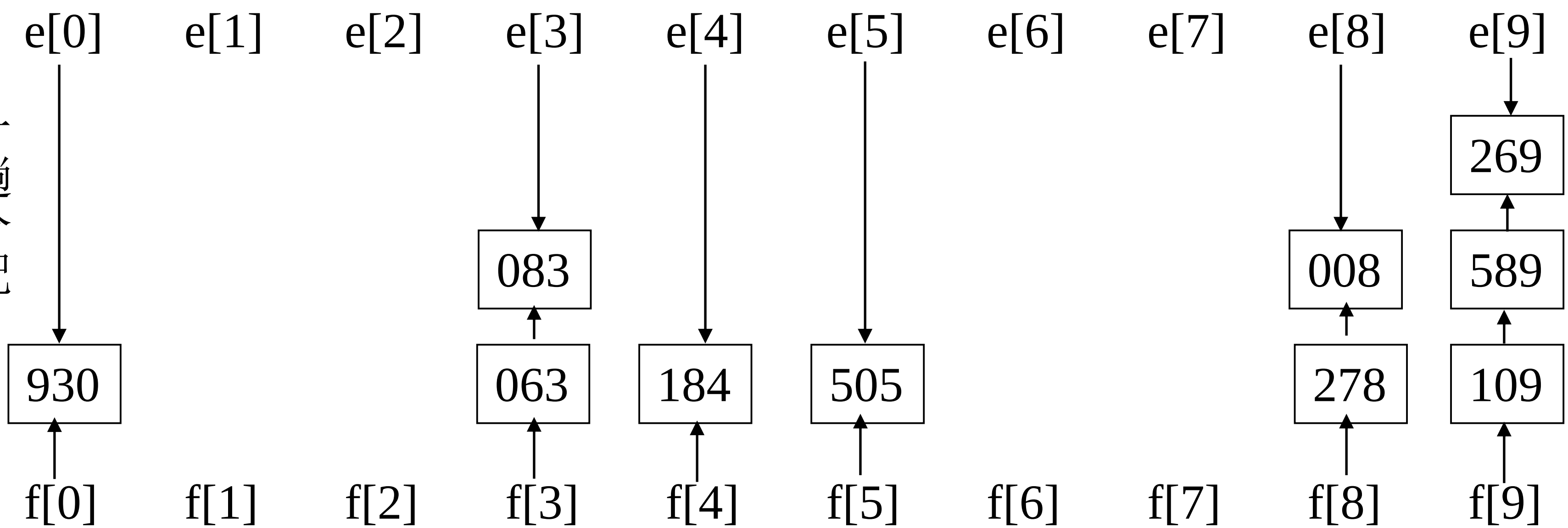
最后按其“百位数”重复一遍上述操作。

例

初始状态：

→ 278 → 109 → 063 → 930 → 589 → 184 → 505 → 269 → 008 → 083

一趟分配



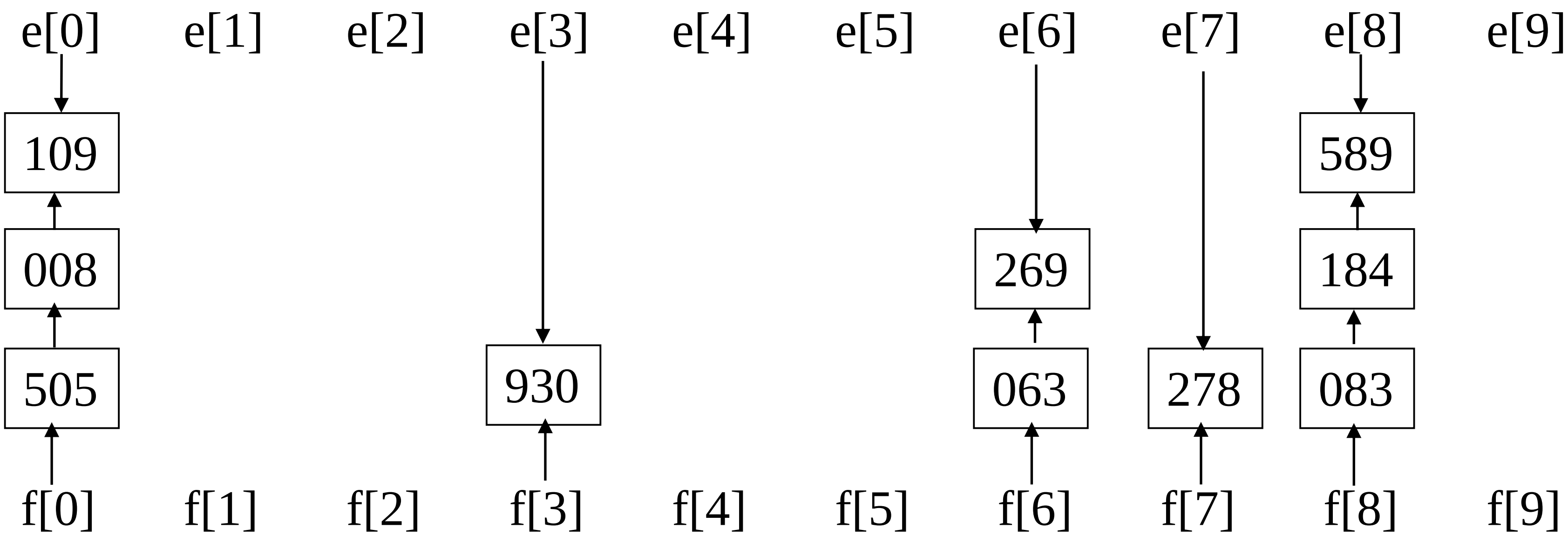
一趟收集：

→ 930 → 063 → 083 → 184 → 505 → 278 → 008 → 109 → 589 → 269

一趟收集：

→ 930 → 063 → 083 → 184 → 505 → 278 → 008 → 109 → 589 → 269

二趟分配

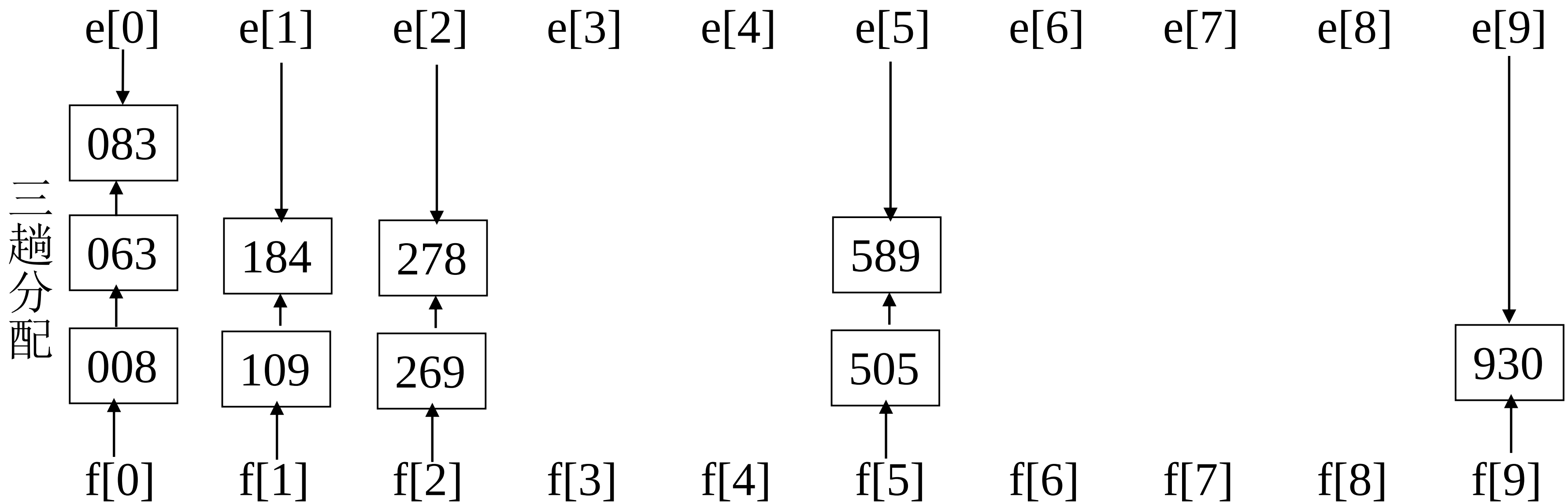


二趟收集：

→ 505 → 008 → 109 → 930 → 063 → 269 → 278 → 083 → 184 → 589

二趟收集：

→ 505 → 008 → 109 → 930 → 063 → 269 → 278 → 083 → 184 → 589



三趟收集：

→ 008 → 063 → 083 → 109 → 184 → 269 → 278 → 505 → 589 → 930

Radix Sort Implementation

```
template <typename E, typename getKey>
void radix(E A[], E B[],
           int n, int k, int r, int cnt[]) {
  // cnt[i] stores number of records in bin[i]
  int j;

  for (int i=0, rtoi=1; i<k; i++, rtoi*=r) { // For k digits
    for (j=0; j<r; j++) cnt[j] = 0;        // Initialize cnt

    // Count the number of records for each bin on this pass
    for (j=0; j<n; j++) cnt[(getKey::key(A[j])/rtoi)%r]++;

    // Index B: cnt[j] will be index for last slot of bin j.
    for (j=1; j<r; j++) cnt[j] = cnt[j-1] + cnt[j];

    // Put records into bins, work from bottom of each bin.
    // Since bins fill from bottom, j counts downwards
    for (j=n-1; j>=0; j--)
      B[--cnt[(getKey::key(A[j])/rtoi)%r]] = A[j];

    for (j=0; j<n; j++) A[j] = B[j];    // Copy B back to A
  }
}
```

Radix()

Initial Input: Array A

27	91	1	97	17	23	84	28	72	5	67	25

First pass values for Count.
rtoi = 1.

0	1	2	3	4	5	6	7	8	9
0	2	1	1	1	2	0	4	1	0

Count array:
Index positions for Array B.

0	1	2	3	4	5	6	7	8	9
0	2	3	4	5	7	7	11	12	12

End of Pass 1: Array A.

91	1	72	23	84	5	25	27	97	17	67	28
0	1	2	3	4	5	6	7	8	9	10	11

Second pass values for Count.
rtoi = 10.

0	1	2	3	4	5	6	7	8	9
2	1	4	0	0	0	1	1	1	2

Count array:
Index positions for Array B.

0	1	2	3	4	5	6	7	8	9
2	3	7	7	7	7	8	9	10	12

End of Pass 2: Array A.

1	5	17	23	25	27	28	67	72	84	91	97
0	1	2	3	4	5	6	7	8	9	10	11

Radix Sort

- **In general, radix sort based on bucket sort is**
  - Asymptotically fast (i.e., O(n))
  - Simple to code
  - A good choice
- ***Can radix sort be used on floating-point numbers?***

Comparison of Sorting Algorithms

Performance factors:

⑴**Running time;**

⑵**Space；**

⑶**Stability；**

⑷**Simple；**

Comparison of Running Time

Sorting Algorithms	Average	Best	Worst
Insertion sort	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$
Shellsort	$\Theta(n \log n)$	$O(n^{1.5})$	$\Theta(n^2)$
Bubblesort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Quicksort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$
Selection sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Heapsort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Radixsort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Comparison of Space

Sorting Algorithms	Auxiliary space
Insertionsort	$O(1)$
Shellsort	$O(1)$
Bubblesort	$O(1)$
Quicksort	$O(\log n)$ ~$O(n)$
Selectionsort	$O(1)$
Heapsort	$O(1)$
Mergesort	$O(n)$
Radixsort	O(n)

Comparison of Stability

（1）Stable Algorithms

- Insertion sort
- Bubble sort
- Selection sort
- Merge sort

（2）Unstable Algorithms:

- Shell sort
- Quick sort
- Heap sort

Comparison of Simpleness

（1）Simple Algorithms:

- **Insertion sort**
- **Selection sort**
- **Bubble sort**
- **Radix sort**

（2）Not simple Algorithms:

- **Shell sort**
- **Heap sort**
- **Quick sort**

References

- **Data Structures and Algorithm Analysis Edition 3.2 (C++ Version)**
  - P.178-185, 251-259

Thank you for listening!