Data Structure

Lecture#18: Internal Sorting 3 (Chapter 7)

U Kang
Seoul National University

In This Lecture

- Learn the main idea and advantage of Heapsort
- Learn the main idea and advantage/disadvantage of Binsort
- Learn the main idea, efficient algorithm, and cost of radix sort

Heapsort (1)

- Use Heap data structure
- Given an input array of size *n*
  - Build a max-heap from the array
  - Remove max element *n* times

Heapsort (2)

```
static <E extends Comparable<? super E>>
void heapsort(E[] A) { // Heapsort
  MaxHeap<E> H = new MaxHeap<E>(A, A.length,
                                A.length);
  for (int i=0; i<A.length; i++)  // Now sort
    H.removemax(); // Put max at end of heap
}
```

# of items

max size

Reminder: removemax() of MaxHeap class

```
public E removemax() {
  assert n > 0 : "Removing from empty heap";
  DSutil.swap(Heap, 0, --n);
  if (n != 0) siftdown(0);
  return Heap[n];
}
```

Heapsort (3)

```
static <E extends Comparable<? super E>>
void heapsort(E[] A) { // Heapsort
  MaxHeap<E> H = new MaxHeap<E>(A, A.length,
                                A.length);
  for (int i=0; i<A.length; i++)  // Now sort
    H.removemax(); // Put max at end of heap
}
```

- Use a max-heap, so that elements end up sorted within the array.
- Cost of heapsort:
- Cost of finding *k* largest elements:

Heapsort (4)

```
static <E extends Comparable<? super E>>
void heapsort(E[] A) { // Heapsort
  MaxHeap<E> H = new MaxHeap<E>(A, A.length,
                                A.length);
  for (int i=0; i<A.length; i++)  // Now sort
    H.removemax(); // Put max at end of heap
}
```

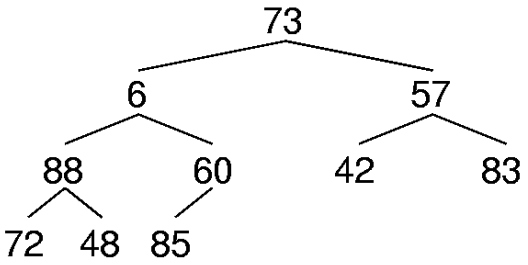
- Use a max-heap, so that elements end up sorted within the array.
- Cost of heapsort: $\Theta(n + n \log n) = \Theta(n \log n)$
- Cost of finding k largest elements: $\Theta(n + k \log n)$

If k is small, Heapsort is very fast

Heapsort Example (1)

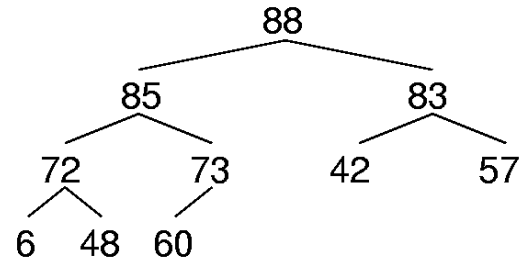
Original Numbers

73	6	57	88	60	42	83	72	48	85



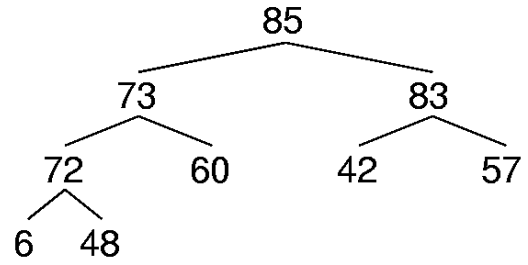
Build Heap

88	85	83	72	73	42	57	6	48	60



Remove 88

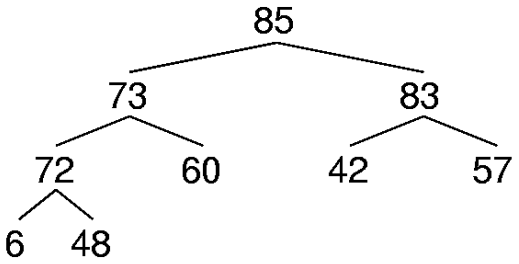
85	73	83	72	60	42	57	6	48	88



Heapsort Example (2)

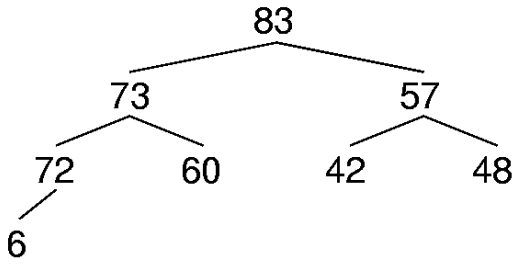
Remove 88

85	73	83	72	60	42	57	6	48	88



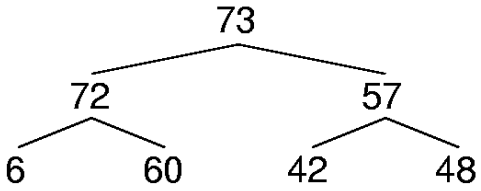
Remove 85

83	73	57	72	60	42	48	6	85	88



Remove 83

73	72	57	6	60	42	48	83	85	88



Binsort (1)

- A simple and efficient algorithm for sorting a permutation of the numbers from 0 ~ n-1:

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

- How can we extend it to
  - allow duplicate values, and
  - allow more keys than records ?

Binsort (2)

- A simple and efficient algorithm for sorting a permutation of the numbers from 0 ~ n-1:

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

- How can we extend it to
  - allow duplicate values, and
  - allow more keys than records ?

- Main Idea:
  - Use array of linked lists (each bin position contains a linked list)
  - Use larger array for B (proportional to MaxKeyValue)

max key value from the input

Binsort (3)

```
static void binsort(Integer A[]) {
  List<Integer>[] B =
        (LList<Integer>[])new LList[MaxKey];
  Integer item;
  for (int i=0; i<MaxKey; i++)
    B[i] = new LList<Integer>();
  for (int i=0; i<A.length; i++)
    B[A[i]].append(A[i]);
  for (int i=0; i<MaxKey; i++)
    for (B[i].moveToStart();
         (item = B[i].getValue()) != null;
         B[i].next())
      output(item);
}
```

Cost: $\Theta(n + MaxKeyValue)$

Binsort (4)

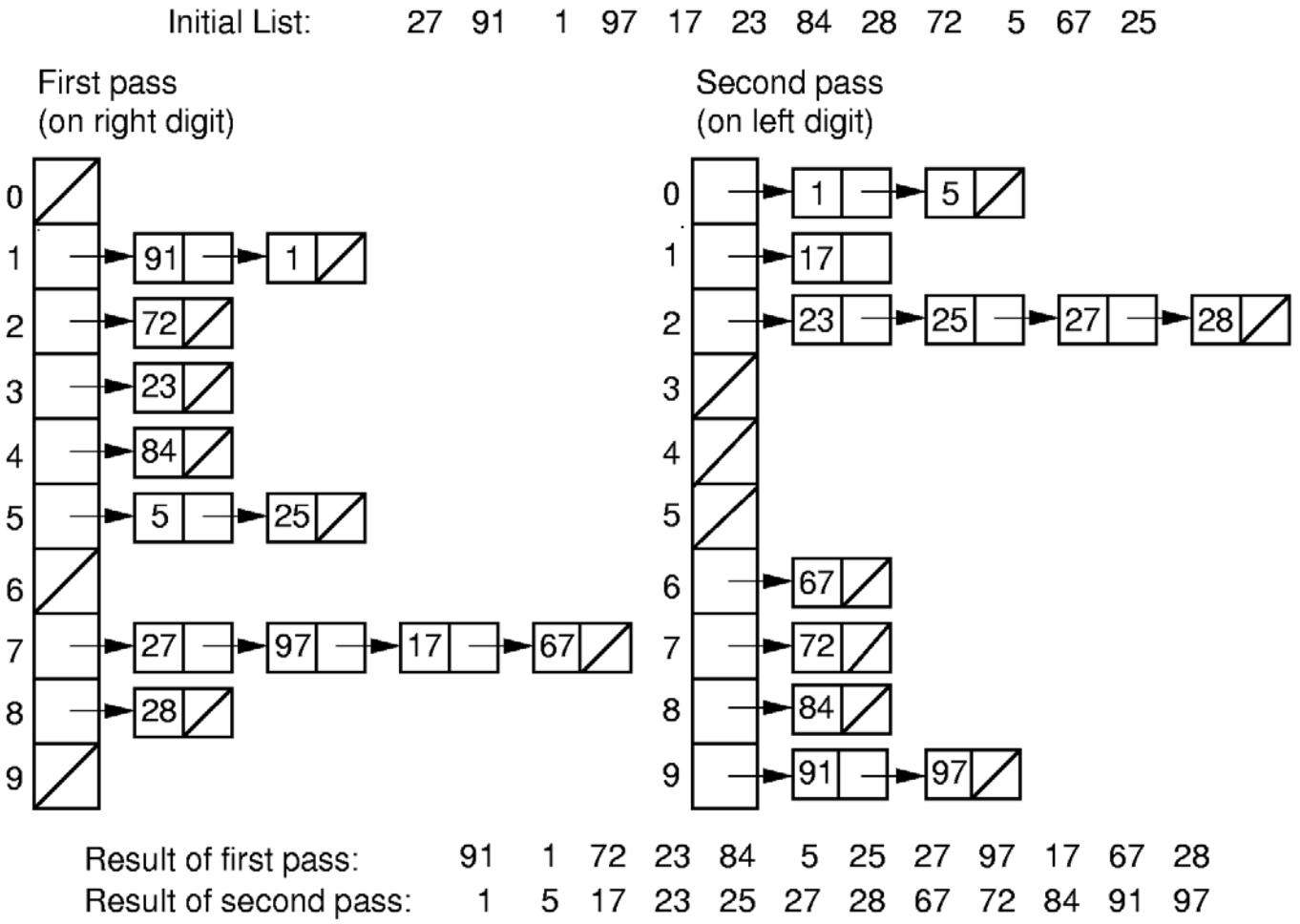
- Strength of Binsort
  - If MaxKeyValue is small, then the running time is fast, and the space requirement is small
- Weakness of Binsort
  - If MaxKeyValue is large, then the running time is slow, and the space requirement is large

Radix Sort (1)

- Motivated from Binsort
  - But, radix sort allocates less space than Binsort

- Works only for sorting numbers.
- Requires *k* passes for sorting *k*-digit numbers
  - *i*-th pass performs bin-sort based on *i*-th digit of each number (1st pass: least significant digit)
  - At each pass, access each element from the bins *in order*

Radix Sort (2)


Initial List: 27 91 1 97 17 23 84 28 72 5 67 25
First pass
(on right digit)
Second pass
(on left digit)
0
1
2
3
4
5
6
7
8
9
91
1
72
23
84
5
25
27
97
17
67
28
1
5
17
23
25
27
28
67
72
84
91
97
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

Radix Sort (3)

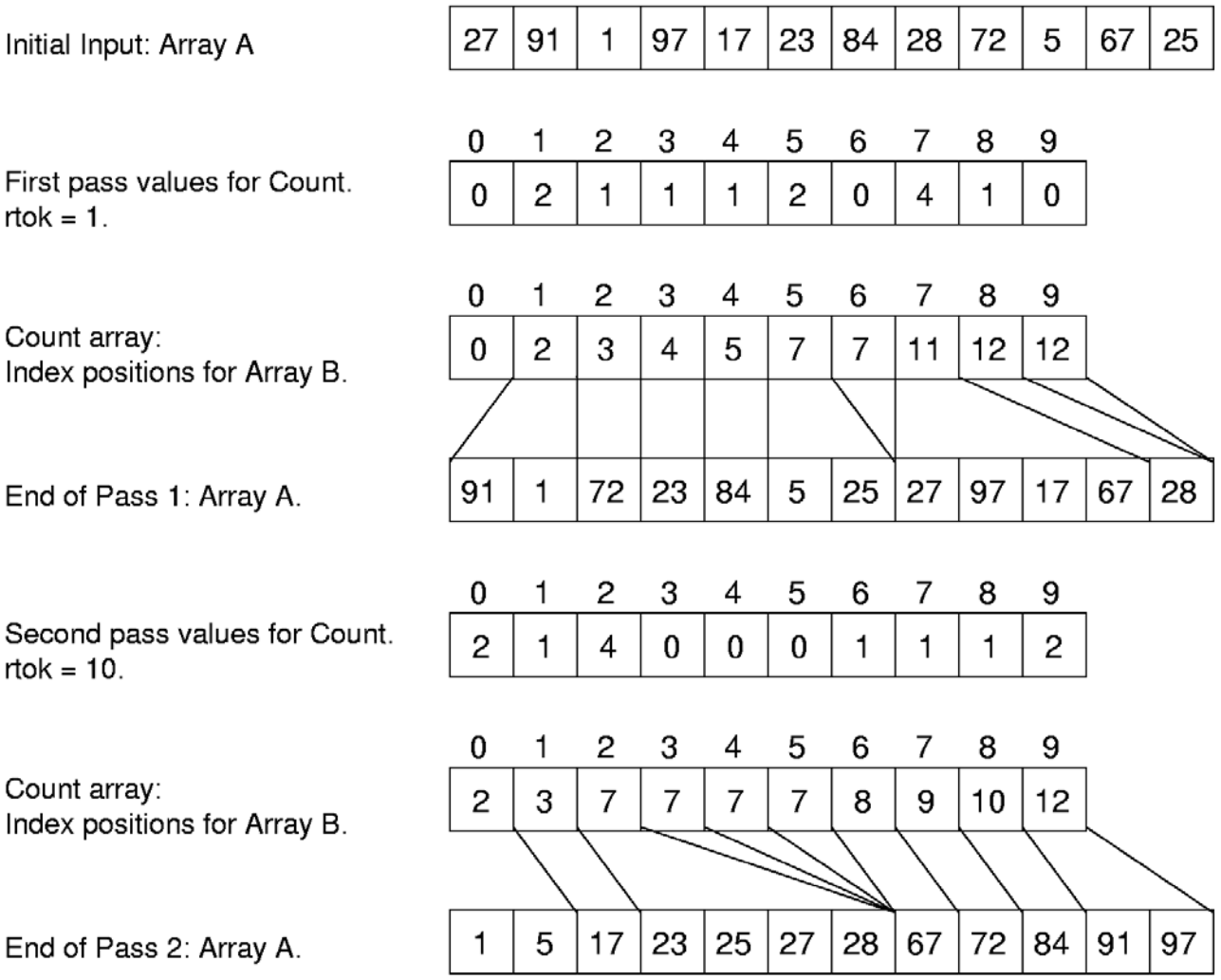
- The previous slide requires a data structure using *array* and *linked list*
- Can we implement radix sort using an *array* of size *n*?
  - With some additional arrays
- Yes!
  - Main idea: 2-pass algorithm on the input data
    - Pass 1: count the number of items for each bin
    - Pass 2: put each input element in a correct position using the counts from pass 1

Radix Sort (4)

```
static void radix(Integer[] A, Integer[] B,
                  int k, int r, int[] count) {
  int i, j, rtok;

  for (i=0, rtok=1; i<k; i++, rtok*=r) {
    for (j=0; j<r; j++) count[j] = 0;
    // Count # of recs for each bin on this pass
    for (j=0; j<A.length; j++)
      count[(A[j]/rtok)%r]++;
    // count[j] is index in B for last slot of j
    for (j=1; j<r; j++)
      count[j] = count[j-1] + count[j];
    for (j=A.length-1; j>=0; j--)
      B[--count[(A[j]/rtok)%r]] = A[j];
    for (j=0; j<A.length; j++) A[j] = B[j];
  }
}
```

Radix Sort Example



Radix Sort Cost

- Time complexity of Radix Sort
  - $\Theta(nk + rk)$

- If there are n distinct keys, what is the minimum length k of a key?
  - At least $\log_r n$.
  - Thus, Radix Sort is $\Theta(n \log n)$ in general case

What you need to know

- Heapsort: main idea and advantage
  - Useful for finding k largest(smallest) items
- Binsort: main idea and advantage/disadvantage
  - Depends on MaxKeyValue
- Radix sort: main idea, efficient algorithm, and cost
  - How to implement it with array

Questions?