Sorting Algorithms

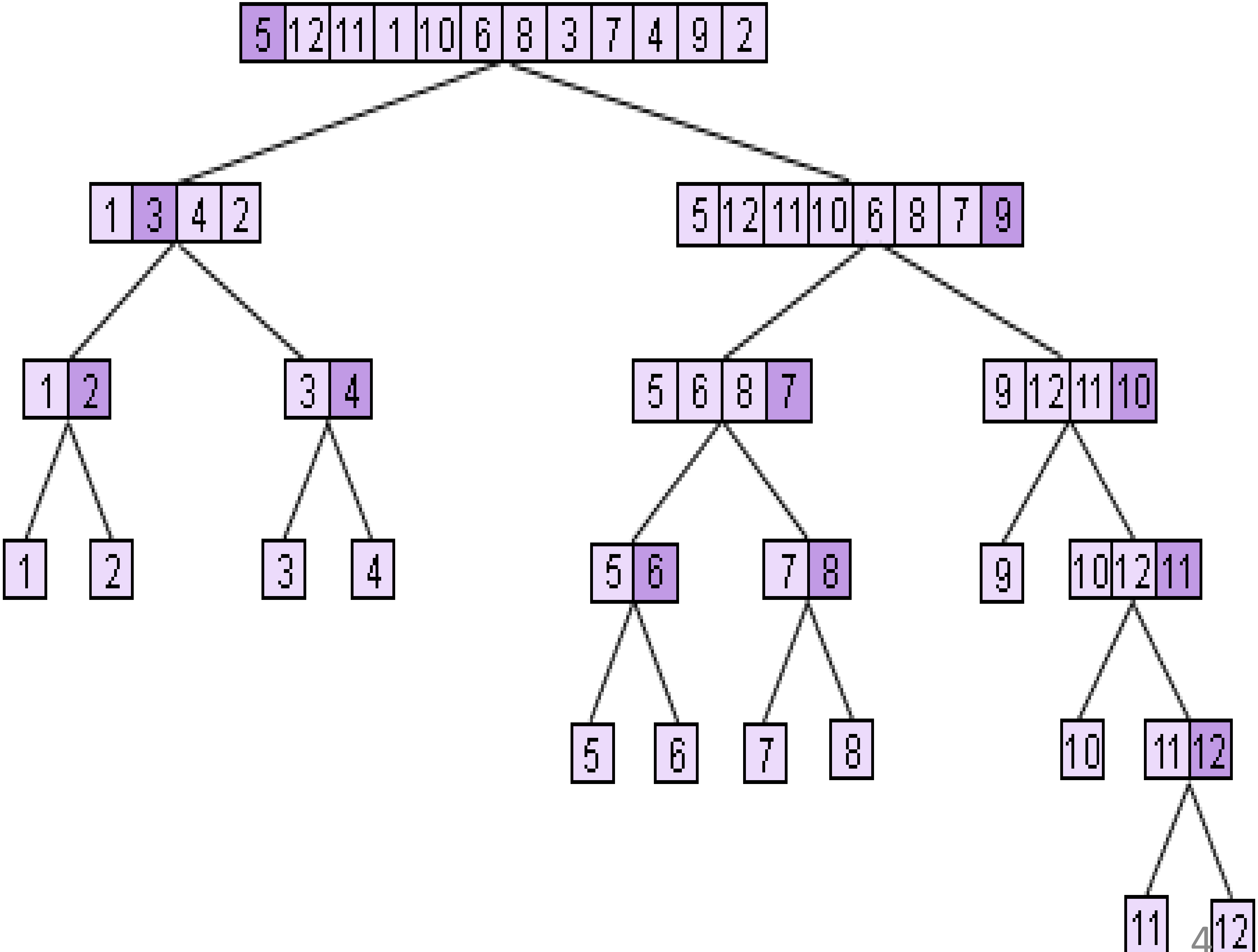
Dai Bo
ACM10
doubledaibo@gmail.com

Outline

- QuickSort
- Stable QuickSort
- Merge Sort
- In-place Merge Sort
- HeapSort
- Bucket Sort
- Counting Sort
- Radix Sort

QuickSort

- Top down
- Choose a pivot
- Divide into 2 parts
- Sorting the 2 parts recursively



5 12 11 1 10 6 8 3 7 4 9 2
1 3 4 2
5 12 11 10 6 8 7 9
1 2
3 4
5 6 8 7
9 12 11 10
1
2
3
4
5 6
7 8
9
10 12 11
5
6
7
8
10
11 12
11
12

Code

- **Time: O(nlogn) Extra Space: O(1)**

```
void QuickSort(int a[], int l, int r)
{
        int i, j, x;
        i = l; j = r; x = a[(l + r) / 2];
        while (i < j)
        {
                while (a[i] < x) ++i;
                while (a[j] > x) --j;
                if (i <= j)
                {
                        std::swap(a[i], a[j]);
                        ++i; --j;
                }
        }
        if (i < r) QuickSort(a, i, r);
        if (l < j) QuickSort(a, l, j);
}
```

Stable QuickSort

- Sorting with 2 keys
- (value, original position)

Merge Sort

- Bottom up
- Equally divide into 2 parts
- Sorting the 2 parts recursively
- Merge them orderly

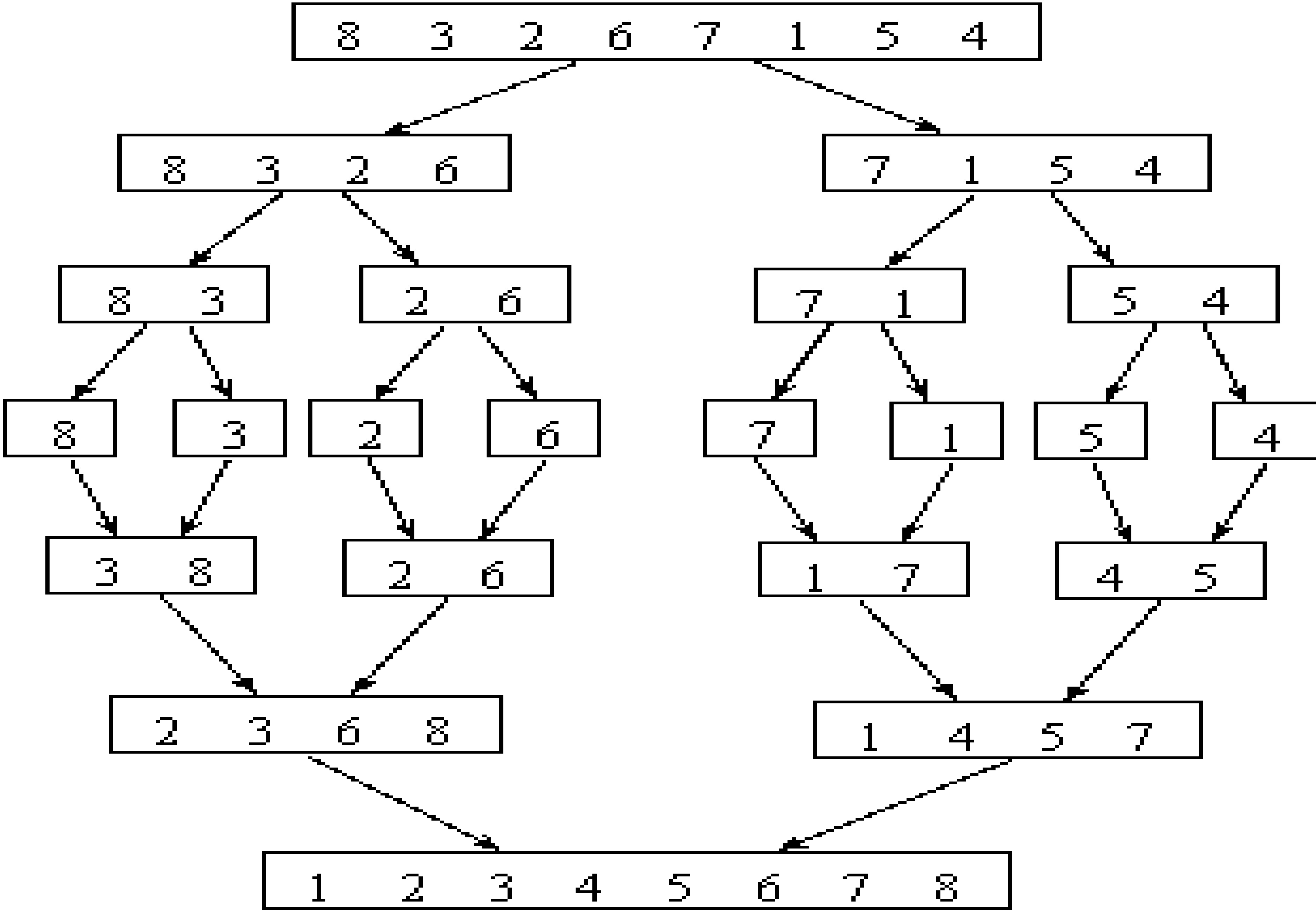
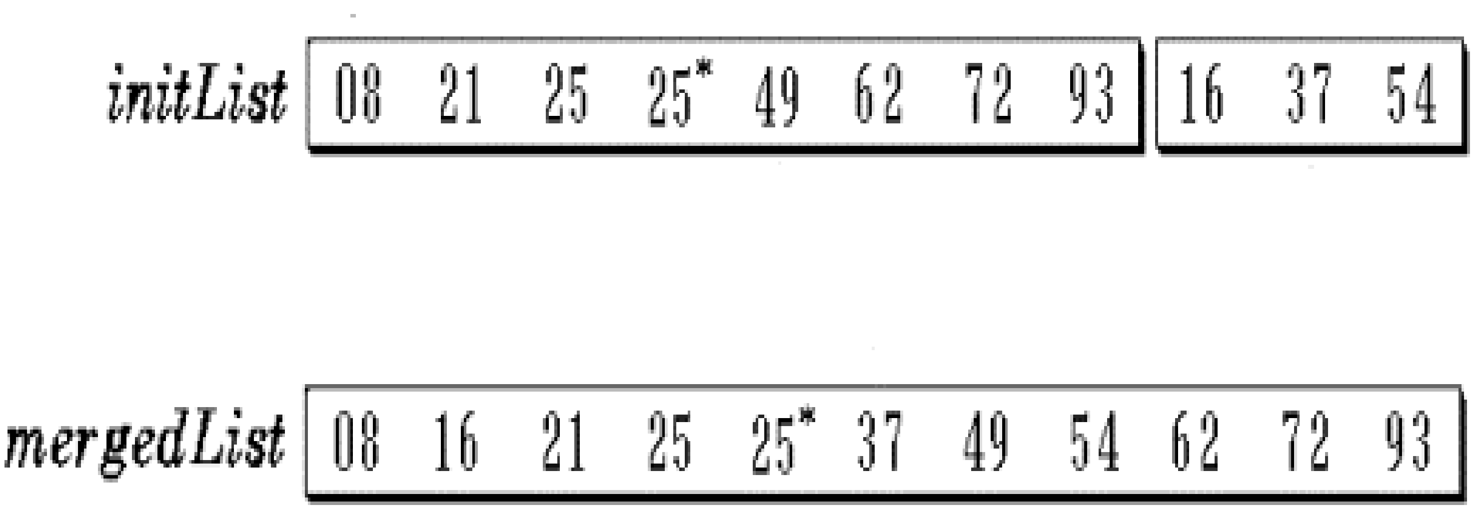


图 8-20 归并排序的递归执行过程

Merging Process



Code

- **Time: O(nlogn)   Extra Space: O(n)**

```
void MergeSort(int a[], int c[], int l, int r)
{
        if (l == r) return;
        int mid;
        mid = (l + r) / 2;
        MergeSort(a, c, l, mid);
        MergeSort(a, c, mid + 1, r);
        int i = l, j = mid + 1;
        for (int k = l; k <= r; ++k)
                if ((j > r) || (i <= mid && a[i] < a[j]))
                {
                        c[k] = a[i]; ++i;
                }
                else
                {
                        c[k] = a[j]; ++j;
                }
        for (int k = l; k <= r; ++k)
                a[k] = c[k];
}
```

In-place Merge Sort

- Extra Space O(1)
- Different in merging process

Merging Process

- Rotate sequence

  $e_0, e_1, \ldots, e_{i-1}, e_i, e_{i+1}, \ldots, e_{n-1}, e_n$

  To $e_i, e_{i+1}, \ldots, e_{n-1}, e_n, e_0, e_1, \ldots, e_{i-1}$

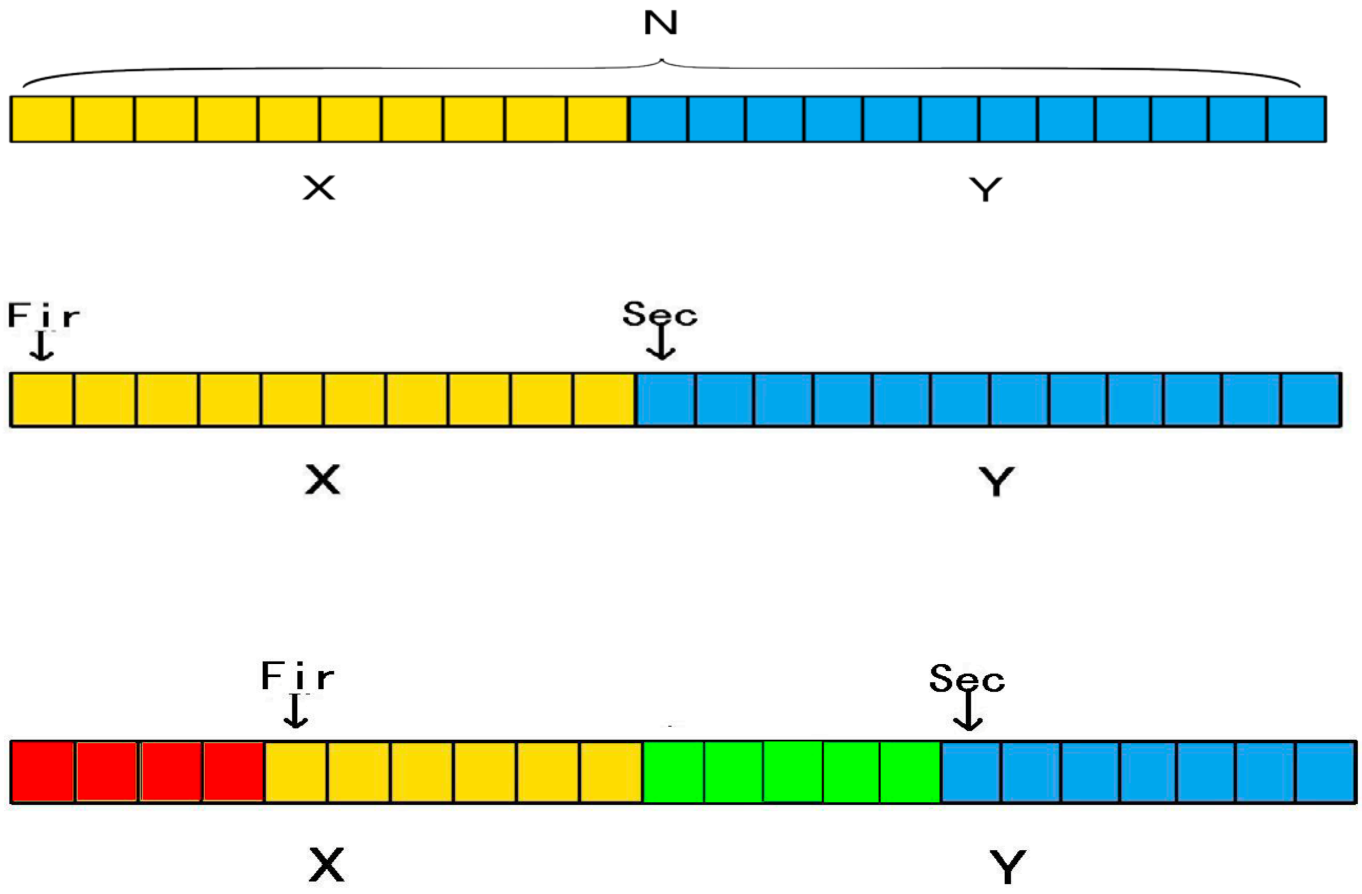
- Operation
  - Reverse 2 subsequence

    $e_{i-1}, \ldots, e_1, e_0$ && $e_n, e_{n-1}, \ldots, e_{i+1}, e_i$
  - Link them together

    $e_{i-1}, \ldots, e_1, e_0, e_n, e_{n-1}, \ldots, e_{i+1}, e_i$
  - Reverse the sequence

    $e_i, e_{i+1}, \ldots, e_{n-1}, e_n, e_0, e_1, \ldots, e_{i-1}$

Merging Process



Code

- **Time: O(nlogn) for most situation   Extra Space: O(1)**

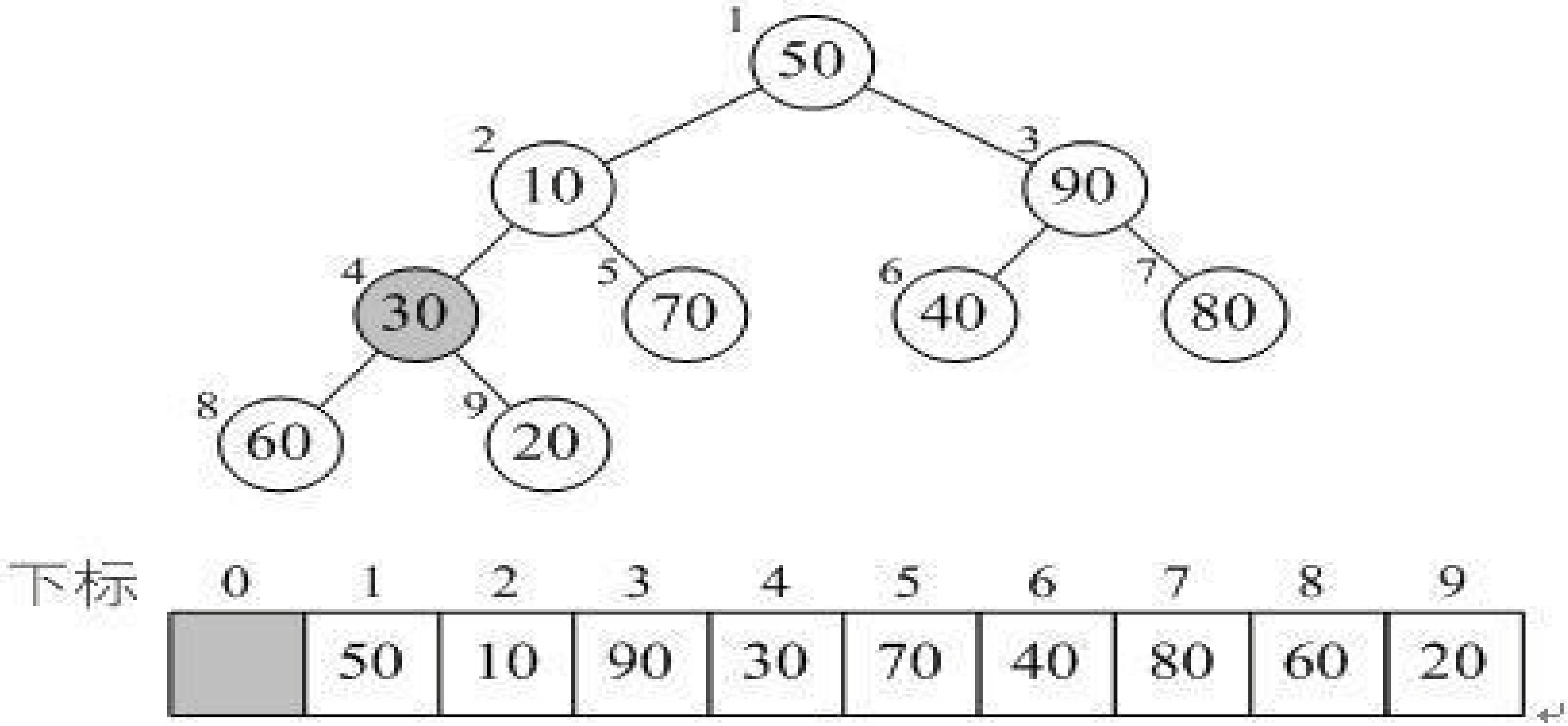
```
void MergeProcess(int v[], int size, int pos)
{
     int fir = 0, sec = pos;
     while (fir < sec && sec < size)
     {
          while (fir < sec && v[fir] <= v[sec]) ++fir;
          int maxMove = 0;
          while (sec < size && v[fir] > v[sec])
          {
               ++maxMove; ++sec;
          }
          Exchange(v + fir, sec - fir, sec - fir - maxMove);
          fir += maxMove;
     }
}
```

- **Better solution see:**

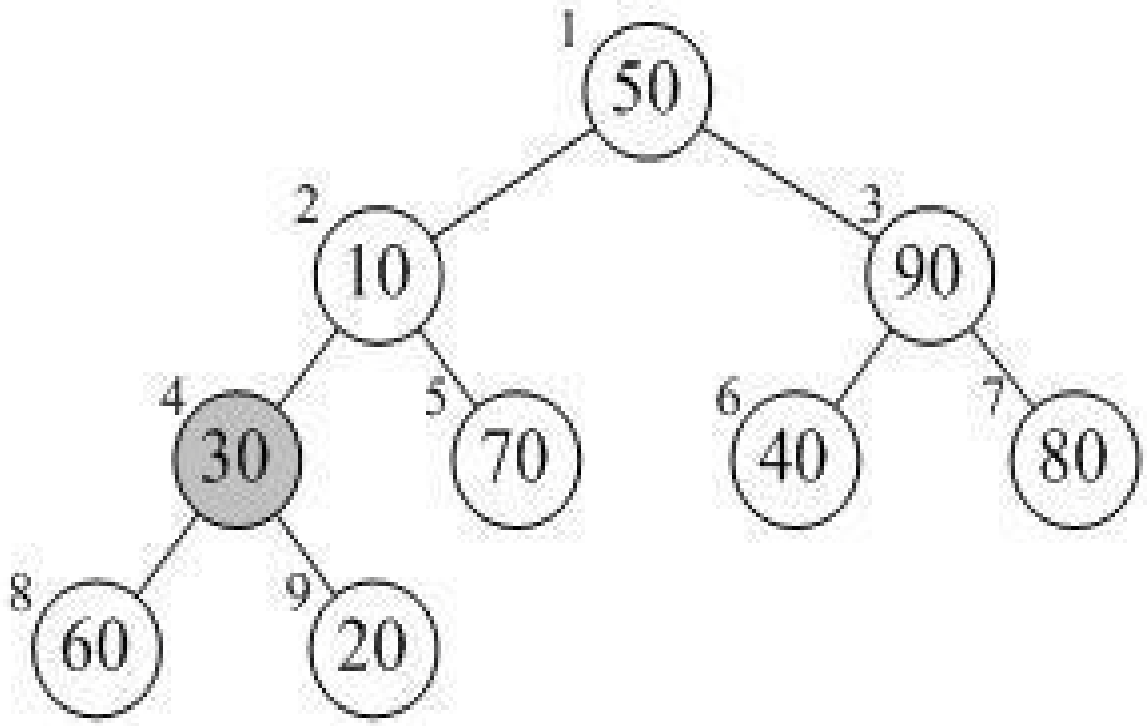
  http://blog.csdn.net/zentropy/article/details/6863051

HeapSort

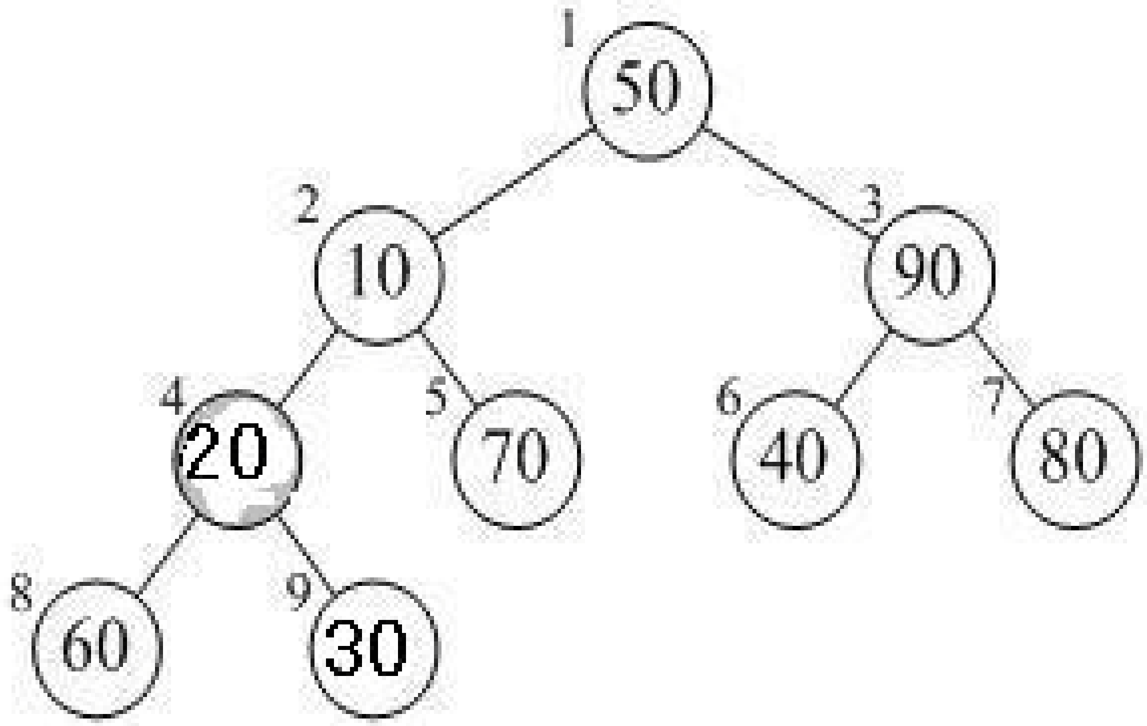
- Binary tree
- Every subtree T, root of T is the minimal (maximal) one in T
- Bottom up



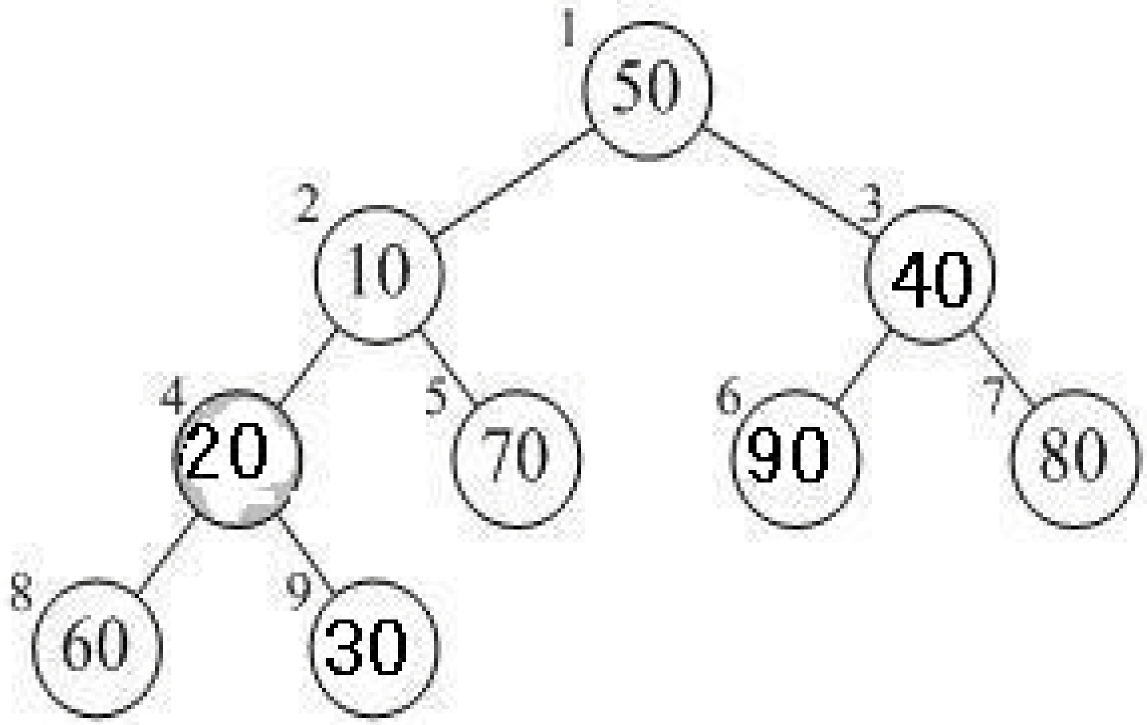
- n’s left child: n * 2; n’s right child: n * 2 + 1



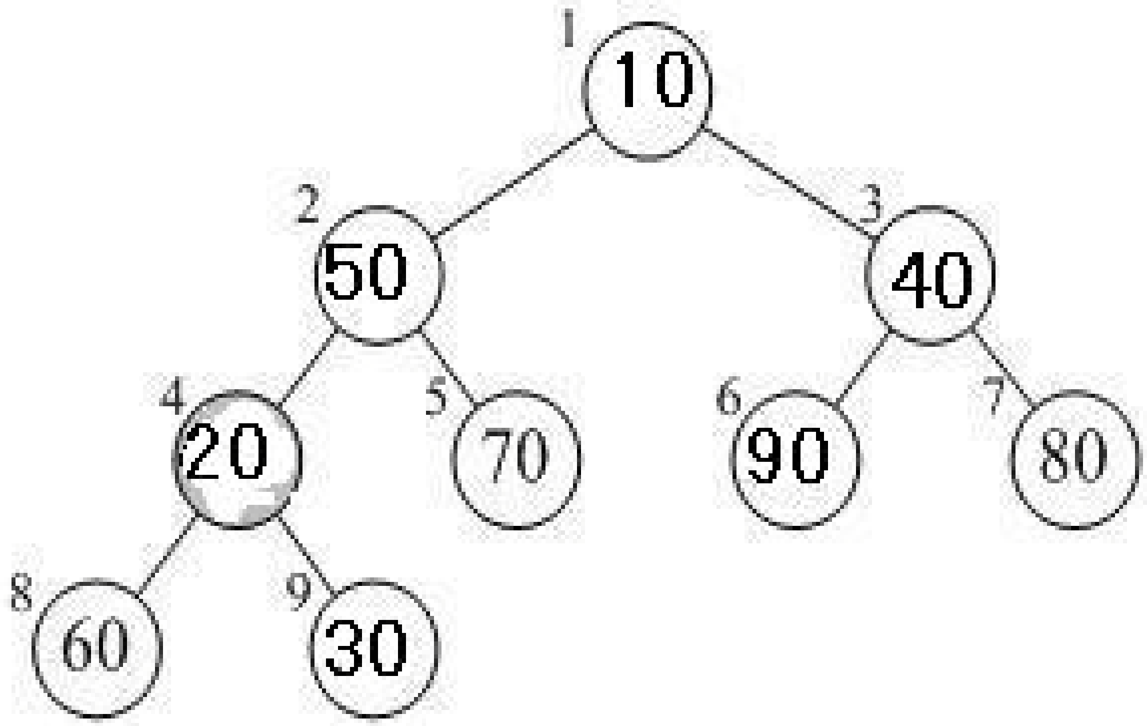
下标	0	1	2	3	4	5	6	7	8	9
		50	10	90	30	70	40	80	60	20



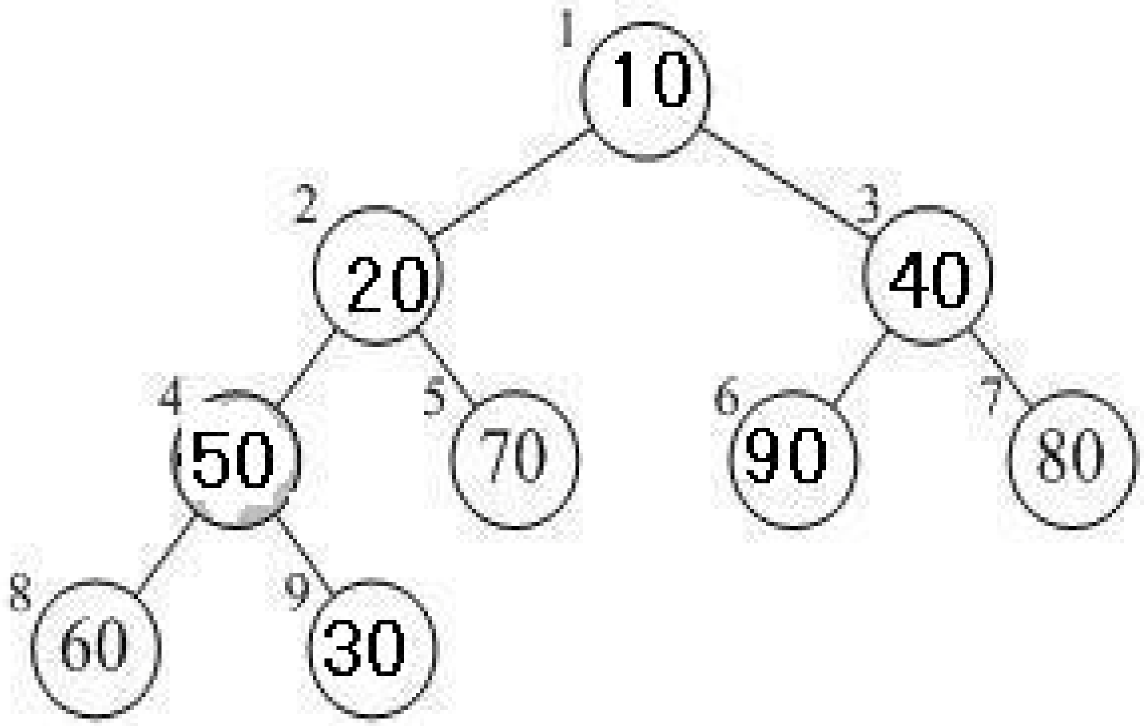
下标	0	1	2	3	4	5	6	7	8	9
		50	10	90	20	70	40	80	60	30



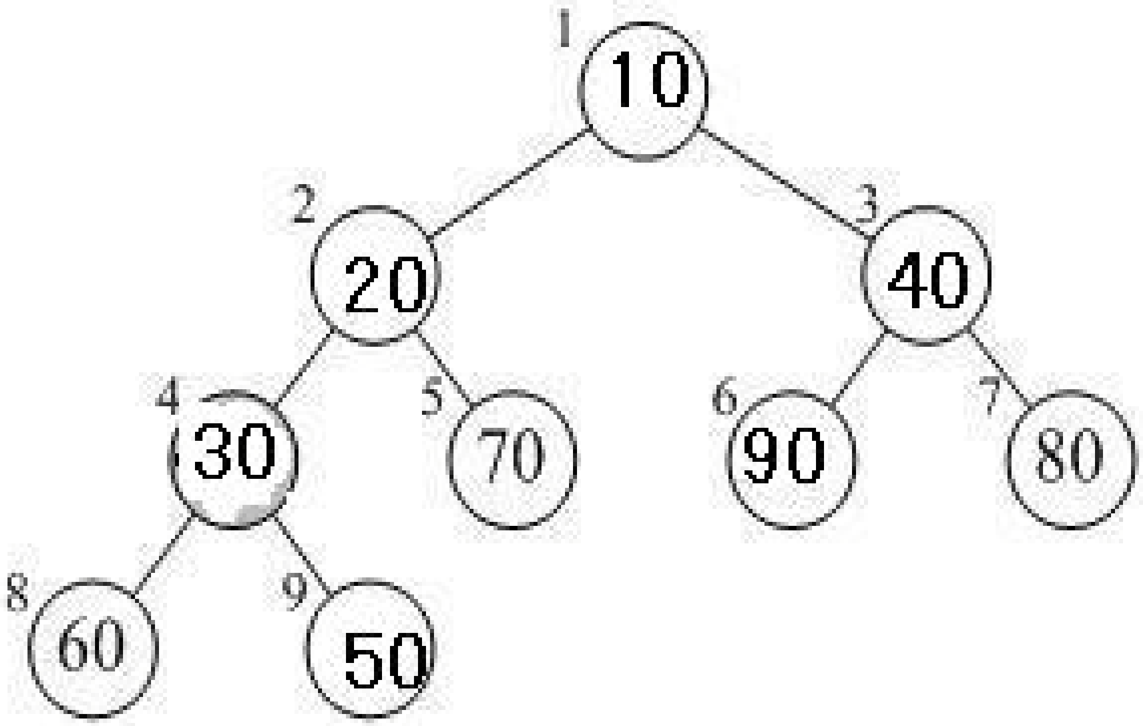
下标	0	1	2	3	4	5	6	7	8	9
		50	10	40	20	70	90	80	60	30



下标	0	1	2	3	4	5	6	7	8	9
		10	50	40	20	70	90	80	60	30



下标	0	1	2	3	4	5	6	7	8	9
		10	20	40	50	70	90	80	60	30



下标	0	1	2	3	4	5	6	7	8	9
		10	20	40	30	70	90	80	60	50

Code

- **O(nlogn)**

```
void HeapSort(int a[], int c[], int n)
{
        for (int i = n; i > n / 2; --i)
                GoDown(a, i, n);
        for (int i = n; i; --i)
        {
                c[n - i + 1] = a[1];
                a[1] = a[i];
                GoDown(a, 1, i - 1);
        }
}
```

```
void GoDown(int a[], int k, int m)
{
        int i, j, t;
        i = k; j = k * 2; t = a[k];
        while (j <= m)
        {
                if (j < m && a[j] > a[j + 1])
                        ++j;
                if (t > a[j])
                {
                        a[i] = a[j]; i = j; j = i * 2;
                }
                else break;
        }
        a[i] = t;
}
```

Bucket Sort

- Dividing sorting range [1..M] into k subranges [Li..Ri]
- Using other sorting algorithms(like Bubble Sort) sort subranges
- Time complexity depends on k
- Space: Extra O(M)

Counting sort

- Bucket Sort with k = M
- Applied when M is small
- Time: O(M + n) Extra Space: O(M)

Radix sort

- Using Bucket Sort several times
- Dividing data based on digit
- Time: O(n) Extra Space: O(10)

Based on units

73, 22, 93, 43, 55, 14, 28, 65, 39, 81

0
1 81
2 22
3 73 93 43
4 14
5 55 65
6
7
8 28
9 39

Based on tens

81, 22, 73, 93, 43, 14, 55, 65, 28, 39

0
1 14
2 22 28
3 39
4 43
5 55
6 65
7 73
8 81
9 93

14, 22, 28, 39, 43, 55, 65, 73, 81, 93
Done

Algorithm	Average time	Worst	Stable	Extra Space
BubbleSort	$O(n^2)$	$O(n^2)$	√	O(1)
SelectionSort	$O(n^2)$	$O(n^2)$	√	O(1)
QuickSort	O(nlogn)	$O(n^2)$		O(1)
MergeSort	O(nlogn)	O(nlogn)	√	O(n)
In-place MergeSort	O(nlogn)	O(nlogn)	√	O(1)
HeapSort	O(nlogn)	O(nlogn)		O(1)
BucketSort	O(nklogk)	O(nklogk)		O(k)
CountingSort	O(M)	O(M)	√	O(M)
RadixSort	$O(\log_R B)$	$O(\log_R B)$	√	O(n)