Algorithms

Chapter 6
Algorithms Involving Sequences & Sets
Part 2
(pp. 119~127)

Sorting

method	average	worst	stability	extra space
bucket	$O(n)$	$O(m)$	stable	$O(m)$
radix	$O(n\log_p k)$	$O(n\log_p k)$ ~$O(n)$	stable	$O(nxp)$
insertion	$O(n^2)$	$O(n^2)$	stable	$O(1)$
selection	$O(n^2)$	$O(n^2)$	unstable	$O(1)$
bubble	$O(n^2)$	$O(n^2)$	stable	$O(1)$
merge	$O(n\log n)$	$O(n\log n)$	stable	$O(n)$
quick	$O(n\log n)$	$O(n^2)$	unstable	$O(\log n)$~$O(n)$
heap	$O(n\log n)$	$O(n\log n)$	unstable	$O(1)$

Bucket Sort & Radix Sort

M. K. Shan, CS, NCCU

Bucket Sort

- Bucket sort
  - allocate sufficient number of buckets &
    - put element in corresponding buckets
    - at the end, scan the buckets in order & collect all elements
  - n elements, ranges from 1 to m $\rightarrow$ m buckets
  - O(m+n)
  - Best case: O(n), Worst case: O(m)

To sort 6, 2, 8, 3, 5

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

To sort 6, 2, 8, 3, 6, 5, 2, 6

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


Have you filled a bucket today?
Listen
Be kind
Show empathy
Share
Take turns
Play together
Include others
Smile
Be polite
Be an upstander
Forgive
Help
Compliment
Don Callejon Project Cornerstone
DC
Project Cornerstone

Radix Sort

- Drawback of bucket sort: waste buckets (space)
- Radix sort
  - use several passes of bucket sort
  - more than one number could fall into the same bucket
- Two approaches
  - most significant bit (MSB): radix-exchange sort
  - least significant bit (LSB): straight-radix sort

針對n筆資料(數值介於0~999)，如何利用10個Buckets，經三個回合來排序？先根據個位數，還是先根據百位數？

Radix-Exchange Sort

- Given n elements represented by k-digits
  - **hypothesis: we know how to sort elements with left k digits**
  - **induction**
    - **use bucket sort**

Radix-Exchange Sort (Cont.)

- Given (64, 8, 216, 512, 27, 729, 0, 1, 343, 125)

0	1	2	3	4	5	6	7	8	9
1 0 27 8 64	125	216	343		512		729		

Straight-Radix Sort

- Given n elements represented by k-digits
  - Hypothesis: sort elements with < k digits (right k-1 digits)
  - Induction
    - ignore the most significant bit & sort the n elements according to their k-1 least significant bits
    - scan all the elements & use bucket sort on the most significant bit
    - collect all the buckets in order

Straight-Radix Sort (Cont.)

- Given (64, 8, 216, 512, 27, 729, 0, 1, 343, 125)

0	1	512	343	64	125	216	27	8	729
0	1	2	3	4	5	6	7	8	9

$\Rightarrow$ (0, 1, 512, 343, 64, 125, 216, 27, 8, 729)

8 1 0	216 512	729 27 125		343		64			
0	1	2	3	4	5	6	7	8	9

$\Rightarrow$ (0, 1, 8, 512, 216, 125, 27, 729, 343, 64)

Straight-Radix Sort (Cont.)

⇒ (0, 1, 8, 512, 216, 125, 27, 729, 343, 64)

64									
27									
8									
1									
0	**125**	**216**	**343**		**512**		**729**		
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

⇒ (0, 1, 8, 27, 64, 125, 216, 343, 512, 729)

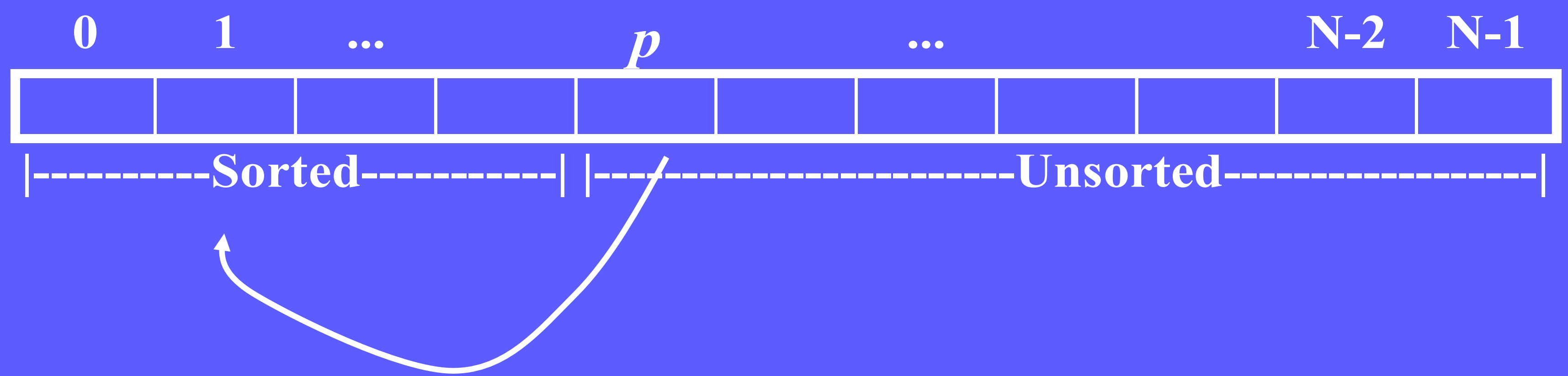
```
Algorithm Straight-Radix(X, n, k);
Input: X (array of n elements with k digits, base p)
Output:X
begin
    all elements are initially in a global queue GQ
    for i:=1 to d do
        initialize queue Q[i] to be empty;
    for i:=k  downto 1 do
        while GQ is not empty do
            dequeue x from GQ;
            d:= the i-th digit of x;
            enqueue x into Q[d];
        for t:=1 to p do
            enqueue Q[t] into GQ;
    for i:=1 to n do
        dequeue X[i] from GQ
end
```

針對n筆資料(數值介於0~999)，如何利用7個Buckets來做Radix Sort？共需幾個回合？

Insertion Sort

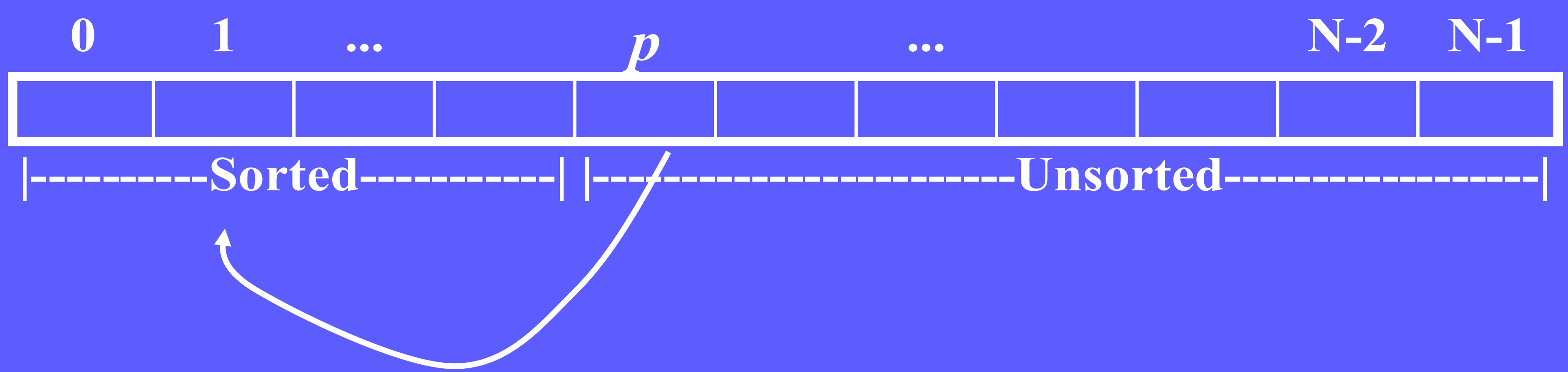
Insertion Sort

- Hypothesis: we know how to sort *n*-1 elements
- Induction on the *n*-th element
  1. **sort *n*-1 elements**
  2. **put the *n*-th element in its correct place by scanning the *n*-1 sorted elements**



Insertion Sort (cont.)

- For N elements A[0], …A[n-1], insertion sort consists of (n-1) passes (Pass 1 through n-1).
- In pass p, A[p] is moved left until its correct place is found among the first (p+1) elements, i.e., A[p] is inserted into the correct place among A[0],..A[p-1].
- movements: O(n^2), comparison: O(n^2)
- improvement
  - use binary search in finding correct place
  - comparison: O(nlogn), movement: O(n^2)



Example of Insertion Sort

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
Original	34	8	64	51	32	21
after pass 1	8	34	64	51	32	21
afetr pass 2	8	34	64	51	32	21
after pass 3	8	34	51	64	32	21
afetr pass 4	8	32	34	51	64	21
after pass 5	8	21	32	34	51	64

Example of Insertion Sort (Cont.)

- **Detail (for example, doing pass 4 after pass 3)**

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
Original	34	8	64	51	32	21
after pass 1	8	34	64	51	32	21
afetr pass 2	8	34	64	51	32	21
after pass 3	8	34	51	64	32	21
doing pass 4	8	34	51	64	32	21
					Tmp	
afetr pass 4	8	32	34	51	64	21
after pass 5	8	21	32	34	51	64

Algorithm of Insertion Sort

```
void InsertionSort(ElemenType A[ ], int N)
{    int j,p;
     Element Type Tmp;
     for (P=1; P < N; p++)
     {
          Tmp=A[P];
          for (j=P; j>0 && A[j-1] > Tmp; j--)
               A[j]=A[j-1];
          A[j]=Tmp;
      };
}
```

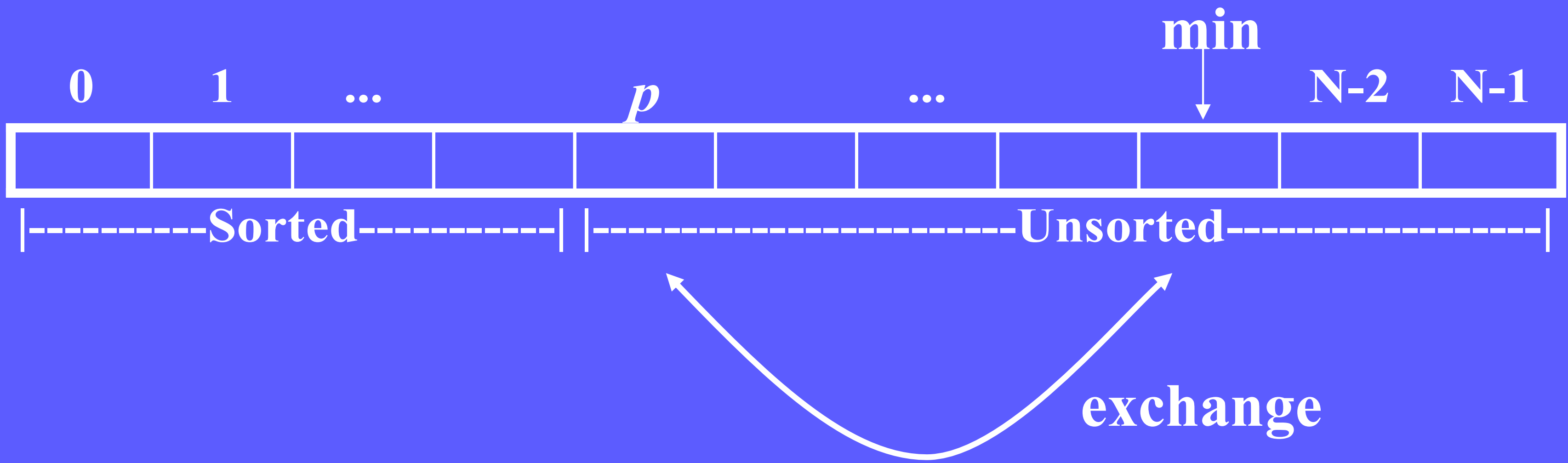
Selection Sort

Selection Sort

- Hypothesis: we know how to sort *n*-1 elements
- Induction on a special *n*-th numbers
  1. **sort *n*-1 elements**
  2. **select the minimal element from unsorted as the *n*-th element**
  3. **put in correct place by swapping**
- **movement: O(*n*-1), comparison: O(n^2)**

Selection Sort (cont.)

- For N elements A[0], …A[n-1], selection sort consists of (N-1) passes (Pass 0 through N-2).
- In pass P, select the smallest element from unsorted element (i.e., A[P],..A[N-1]), exchange with A[P]
- movement: O(n-1), comparison: O(n^2)



Example of Selection Sort

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
Original	34	8	64	51	32	21
after pass 0	8	34	64	51	32	21
afetr pass 1	8	21	64	51	32	34
after pass 2	8	21	32	51	64	34
afetr pass 3	8	21	32	34	64	51
after pass 4	8	21	32	34	51	64

Example of Selection Sort (Cont.)

- Detailed (for example, doing pass 3 after pass 2)

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	
Original	34	8	64	51	32	21	
after pass 0	8	34	64	51	32	21	
afetr pass 1	8	21	64	51	32	34	
after pass 2	8	21	32	51	64	34	
doing pass 2	8	21	32	51	64	34	minimun
					exchange		
afetr pass 3	8	21	32	34	64	51	
after pass 4	8	21	32	34	51	64	

Algorithm of Selection Sort

```
void SelectionSort(ElemenType A[ ], int N)
{    int j,p;
     Element Type Min;
     for (P=0; P <= N-2; P++)
     {
          Min=P;
          for (j=P+1; j <= N-1 ; j++)
               if (A[j] < A[Min])
                    Min=j;
          exchange (A[P], A[Min]);
      };
}
```

Bubble Sort

Bubble Sort

- For N elements A[0], …,A[N-1], Bubble sort consists of (N-1) passes (Pass 0 through N-2).
- In pass P, adjacent elements in A[P], ..,A[N-1] are compared & exchanging if necessary.
- After pass P, the first P elements have been in correct position.

Example of Bubble Sort

- Detailed (for example, doing pass 0)

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
Original	34	8	64	51	32	21
doing pass 0	34	8	64	51	21	32
	34	8	64	21	51	32
	34	8	21	64	51	32
	34	8	21	64	51	32
after pass 0	8	34	21	64	51	32
afetr pass 1	8	21	34	32	64	51
after pass 2	8	21	32	34	51	64
afetr pass 3	8	21	32	34	51	64
after pass 4	8	21	32	34	51	64

Algorithm of Bubble Sort

```
void BubbleSort(ElemenType A[ ], int N)
{     int j,p;
      Element Type Min;
      for (P=0; P <=N-2; P++)
      {
           for (j=N-1; j >= P ; j--)
                if (A[j+1] > A[j])
                     exchange (A[j+1], A[j]);
       };
}
```

有人說Bubble Sort
也是一種Selection Sort，
你認為呢？

Merge Sort

M. K. Shan, CS, NCCU

Merge Sort

- Hypothesis: we know how to sort n/2 elements
- Induction
  - **sort two n/2 elements**
  - **merge**
- **Merge sort**
  - **merge two sorted lists**
  - **recursive algorithm**
  - **Divide-and-conquer strategy**
  - **drawback: merging step requires additional storage**

Example of Merge Sort

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
Original	11	8	14	7	6	8	23	4
after pass 1	8	11	7	14	6	8	4	23
afetr pass 2	7	8	11	14	4	6	8	23
after pass 3	4	6	7	8	8	11	14	23

Example of Merge

0	1	2	3	4	5	6	7
7	8	11	14	4	6	8	23

4	**6**	**7**	**8**	**8**	**11**	**14**	**23**
0	1	2	3	4	5	6	7

Algorithm of Merge Sort

```
void MergeSort(ElemenType A[ ], ElementType Tmp[],
                              int Left, Right)
{    int Center;
     if (Left < Right)
     {
        Center = (Left + Right)/2;
        MergeSort(A, Tmp, Left, Center);
        MergeSort(A, Tmp, Center+1, Right);
        Merge(A, Tmp, Left, Center+1, Right);
}
```

Algorithm of Merge

```
void Merge( ElementType A[ ], TmpArray[ ],
                    int Lpos, int Rpos, int RightEnd )
{    int i, LeftEnd, NumElements, TmpPos;
     LeftEnd = Rpos - 1;        TmpPos = Lpos;
     NumElements = RightEnd - Lpos + 1;
      while ( Lpos <= LeftEnd && Rpos <= RightEnd )
           if  ( A[ Lpos ] <= A[ Rpos ] )
                    TmpArray[ TmpPos++ ] = A[ Lpos++ ];
           else    TmpArray[ TmpPos++ ] = A[ Rpos++ ];
      while ( Lpos <= LeftEnd )  /* Copy rest of first half */
               TmpArray[ TmpPos++ ] = A[ Lpos++ ];
      while( Rpos <= RightEnd ) /* Copy rest of second half */
               TmpArray[ TmpPos++ ] = A[ Rpos++ ];
     for( i = 0; i < NumElements; i++, RightEnd-- )
              A[ RightEnd ] = TmpArray[ RightEnd ];
}
```

Merge Sort時間複雜度？
如何分析？

Analysis of Merge Sort

- Recurrence Relation: T(1)=1, T(N)=2T(N/2)+N
- Solution 1

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + 1$$

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + 1$$

$$\frac{T(N/4)}{N/4} = \frac{T(N/8)}{N/8} + 1$$

$$\cdots$$

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

$$\Rightarrow \frac{T(N)}{N} = \frac{T(1)}{1} + \log N$$

$$\boxed{\Rightarrow T(N) = N \log N + N = O(N \log N)}$$

Analysis of Merge Sort (Cont.)

- Recurrence Relation: T(1)=1, T(N)=2T(N/2)+N
- Solution 2

$$T(N) = 2T(N/2) + N$$

$$\because 2T(N/2) = 2(2(T(N/4)) + N/2) = 4T(N/4) + N$$

$$\Rightarrow T(N) = 4T(N/4) + 2N$$

$$\because 4T(N/4) = 4(2(T(N/8)) + N/4) = 8T(N/8) + N$$

$$\Rightarrow T(N) = 8T(N/8) + 3N$$

…

$$\Rightarrow T(N) = 2^k T(N/2^k) + k * N$$

Using k =log$N$

$$\Rightarrow T(N) = N T(1) + N \log N = N \log N + N = O(N \log N)$$

Quick Sort

Quick Sort

- the fastest known sorting algorithm in practice
- divide-and-conquer recursive strategy
- basic algorithm

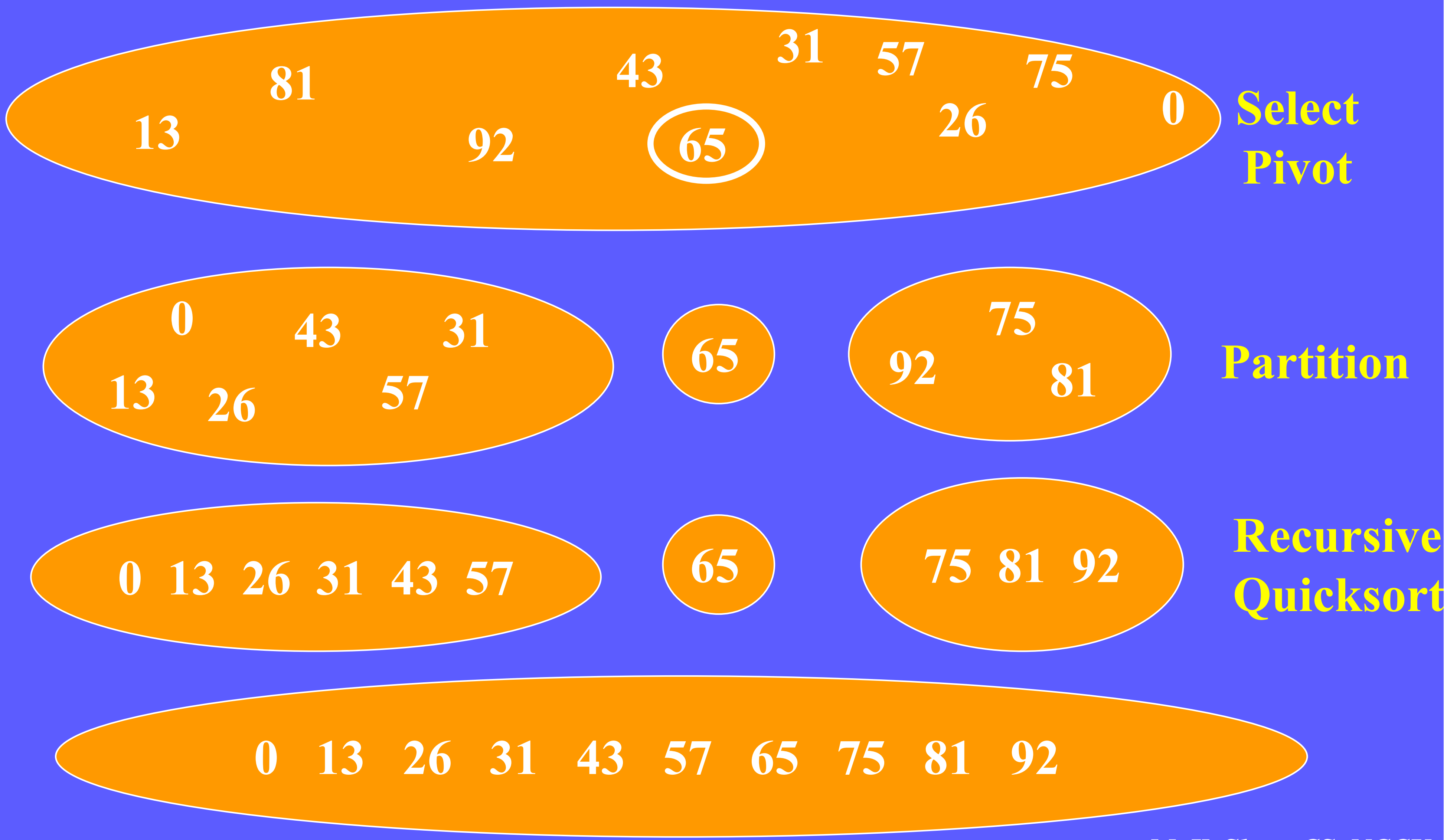
1. Pick an element *v* as pivot

2. Partition two groups S1 & S2,

$\forall x \in$ S1, x < *v*, $\forall y \in$ S2, *y* > *v*

3. Recursively quick sort S1 and S2

Quick Sort (Cont.)



Quick Sort (Cont.)

13	81	92	43	65	31	57	26	75	0
13	26	57	43	0	31	65	81	75	92
13	0	26	43	31	57	65	81	75	92
0	13	26	43	31	57	65	81	75	92
0	13	26	31	43	57	65	81	75	92
0	13	26	31	43	57	65	75	92	81
0	13	26	31	43	57	65	75	81	92

Pivot的選擇會影響 Quick Sort 的速度嗎？
什麼是好的 Pivot？
如何選擇好的 Pivot？
如何快速地找出好的 Pivot？

Picking the Pivot

- A poor way: the smallest or the largest element
- A safe maneuver: choose pivot randomly with the cost of random number generation.
- Median-of-Three Partitioning:
  - base choice: the median value (how to find?)
  - estimate:
    - pick three elements randomly and use the median.
    - use the median of the left, right, and center element.

Quick Sort (Cont.)

13	81	92	43	65	31	57	26	75	0
13	26	57	43	0	31	65	81	75	92
13	0	26	43	31	57	65	81	75	92
0	13	26	43	31	57	65	81	75	92
0	13	26	31	43	57	65	81	75	92
0	13	26	31	43	57	65	75	92	81
0	13	26	31	43	57	65	75	81	92

選出好的 Pivot後，如何做 Partition？
如何快速地找出好的 Pivot？

Partitioning Strategy

pivot=median(A[0], A[9], A[4])=A[4]=6

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
8	1	4	9	6	3	5	2	7	0
8	1	4	9	0	3	5	2	7	6
8	1	4	9	0	3	5	2	7	6
2	1	4	9	0	3	5	8	7	6
2	1	4	5	0	3	9	8	7	6
2	1	4	5	0	3	9	8	7	6
2	1	4	5	0	3	6	8	7	9

Improvement of Quicksort

- Quick sort doesn't perform well for small array
- Use insertion sort for small array when quick sort recursively.

```
ElementType  Median3( ElementType A[ ], int Left, int Right )
 {
        int Center = ( Left + Right ) / 2;
        if ( A[ Left ] > A[ Center ] )
            Swap( &A[ Left ], &A[ Center ] );
        if ( A[ Left ] > A[ Right ] )
           Swap( &A[ Left ], &A[ Right ] );
        if ( A[ Center ] > A[ Right ] )
           Swap( &A[ Center ], &A[ Right ] );
       Swap( &A[ Center ], &A[ Right - 1 ] );
       return A[ Right - 1 ];
}
```

```
void QuickSort( ElementType A[ ], int Left, int Right )
{   int i, j;      ElementType Pivot;
    if ( Left + Cutoff <= Right )
    {   Pivot = Median3( A, Left, Right );
        i = Left+1; j = Right - 2;
        for ( ; ; )
        {    while ( A[i] < Pivot ) i++;
             while ( A[j] > Pivot ) j--;
             if ( i < j )   Swap( &A[ i ], &A[ j ] );
             else   break;
        }
        Swap( &A[ i ], &A[ Right - 1 ] );
        QuickSort( A, Left, i - 1 );
        QuickSort( A, i + 1, Right );
    }
    else  InsertionSort( A + Left, Right - Left + 1 );
}
```

Quick Sort與Merge Sort都是Divide & Conquer，

為什麼Quick Sort不需要額外O(n)的空間，

而Merge Sort需要額外O(n)的空間?

Example of Merge Sort

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
Original	11	8	14	7	6	8	23	4
after pass 1	8	11	7	14	6	8	4	23
afetr pass 2	7	8	11	14	4	6	8	23
after pass 3	4	6	7	8	8	11	14	23

Quick Sort (Cont.)

13	81	92	43	65	31	57	26	75	0

13	26	57	43	0	31	65	81	75	92

13	0	26	43	31	57	65	81	75	92

0	13	26	43	31	57	65	81	75	92

0	13	26	31	43	57	65	81	75	92

0	13	26	31	43	57	65	75	92	81

0	13	26	31	43	57	65	75	81	92

Quick Sort時間複雜度？
如何分析？

Analysis of Quick Sort

- **Recurrence Relation: T(N)=T(i)+T(N-i-1)+cN**
  - **Worst case(O(N^2)): T(0)=1, T(N)=T(N-1)+cN, N > 1**
  - **Best case(O(N log$N$)): T(N)=2T(N/2)+cN**
  - **Average case(O(N log$N$)):**

    T(N)= T(i)*Avg(T(i))+T(N-i-1)* Avg(T(N-i-1))+cN

$$\Rightarrow T(N) = \frac{2}{N}\left[\sum_{j=0}^{N-1} T(j)\right] + cN$$

Lower Bound for Sorting (Complexity of Sorting Problem)

Lower Bound for Sorting:

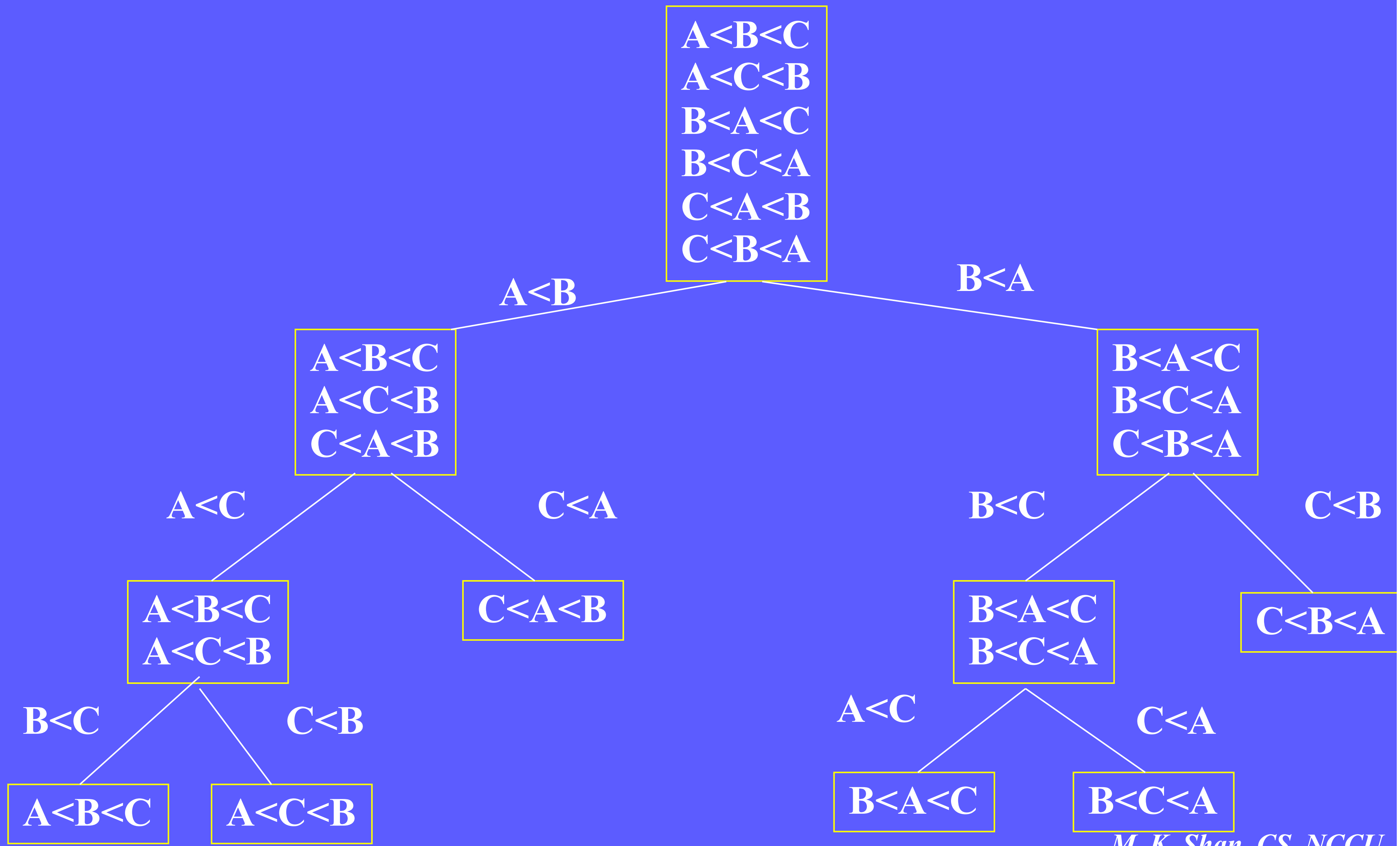
有沒有比O(nlogn)還要快的Sorting演算法?

(Sorting Problem有多難？)

General Lower Bound for Sorting

- Comparison-based Sorting Algorithm
- $O(N \log N)$ $\Theta(N \log N)$ $\Omega(N \log N)$
- Proved by decision tree

Decision Tree for Sorting Three Element



Proof of Lower Bound of Sorting

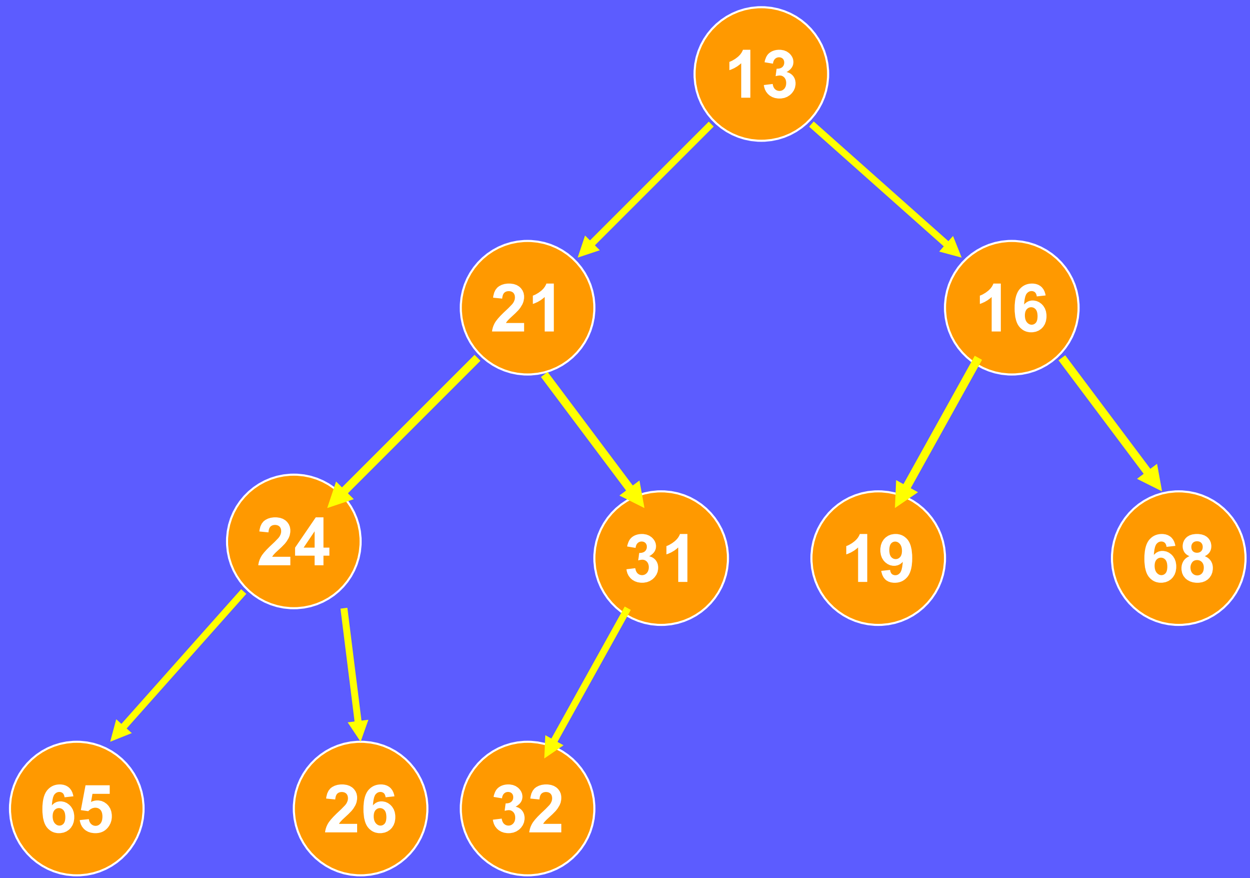
- **A binary tree T of depth *d* has at most 2^d leaves**
- **A binary tree with *L* leaves have depth at least $\lceil \log L \rceil$**
- **Any comparison-based sorting algorithm requires at least $\lceil \log(N!) \rceil$ comparisons in the worst case.**
- **Any comparison-based sorting algorithm requires $\Omega(N\log N)$ comparisons.**

  (log(N!) >= (N/2)*log(N/2))

Heap Sort

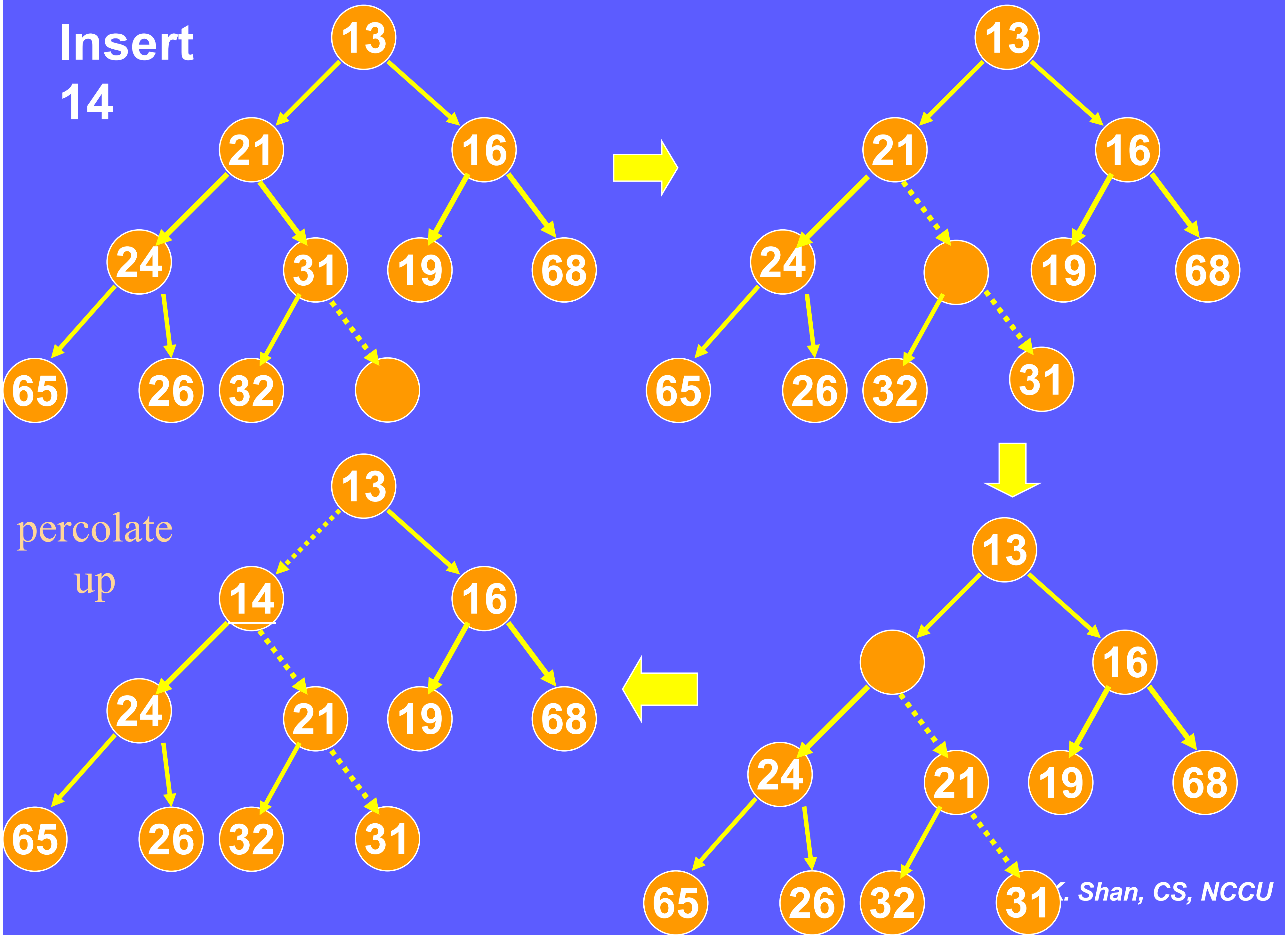
Heaps

- **Heaps (Priority Queue)**
  - **Structure property: complete binary tree**
    - *** complete binary tree: height $\lfloor \log N \rfloor$**
    - *** array implementation of heap**
  - **Order property: for each node X, the key in the parent of X is smaller than or equal to the key in X.**



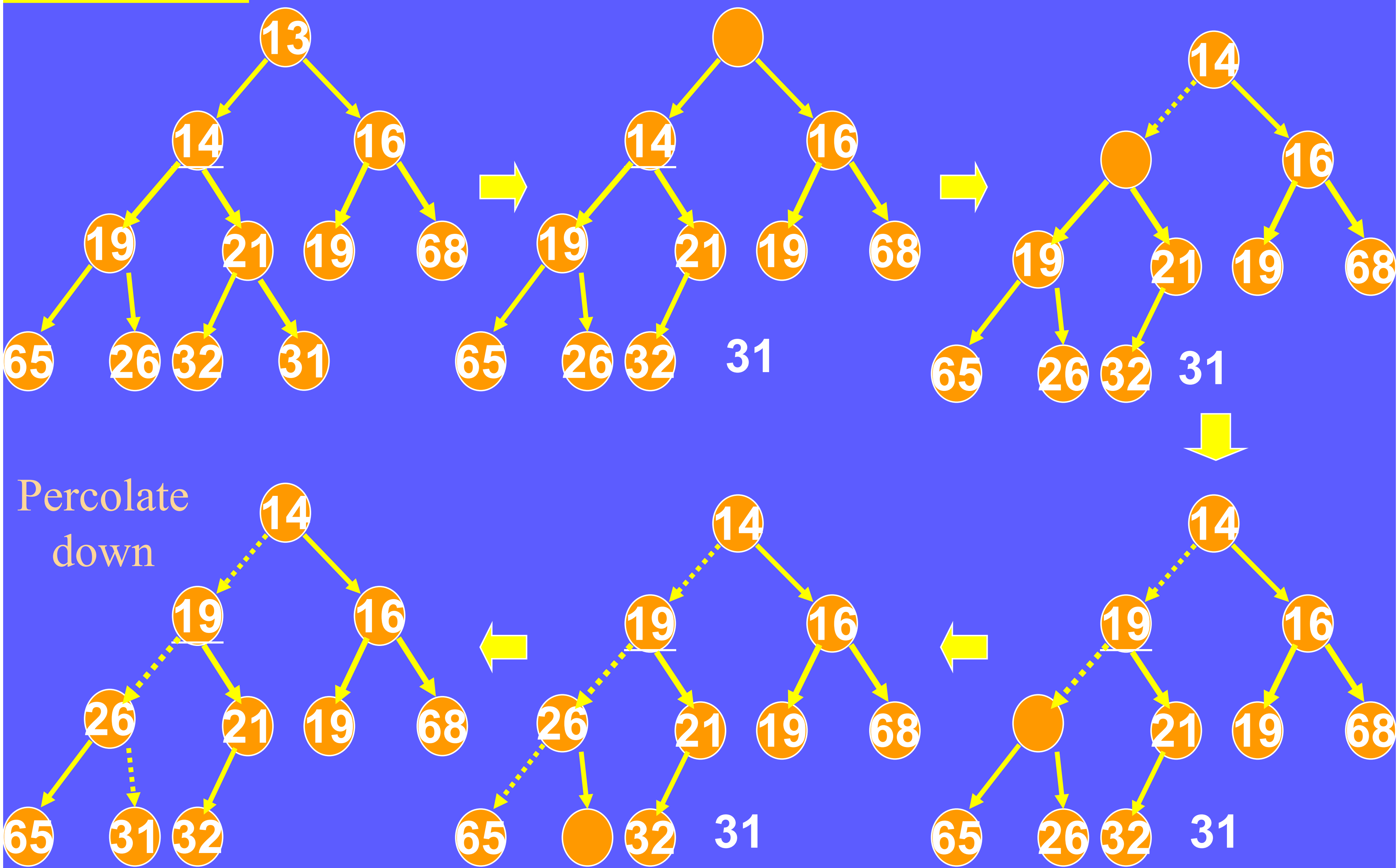
	13	21	16	24	31	19	68	65	26	32	
0	1	2	3	4	5	6	7	8	9	10	11

Insert 14


13
21
16
24
31
19
68
65
26
32
13
21
16
24
19
68
65
26
32
31
percolate up
13
14
16
24
21
19
68
65
26
32
31
13
16
24
21
19
68
65
26
32
31

```
void Insert( ElementType X, PriorityQueue H )
{
    int i;
    if( IsFull( H ) )
    {
      Error( "Priority queue is full" );
      return;
    }
    for  ( i = ++H->Size; H->Elements[ i / 2 ] > X; i /= 2 )
         H->Elements[ i ] = H->Elements[ i / 2 ];
    H->Elements[ i ] = X;
}
```

DeleteMin



```
 ElementType  DeleteMin( PriorityQueue H )
{     int i, Child;
      ElementType MinElement, LastElement;
      if   ( IsEmpty( H ) )  {
           Error( "Priority queue is empty" );
           return H->Elements[ 0 ];
      };
     MinElement = H->Elements[ 1 ];
     LastElement = H->Elements[ H->Size-- ];
     for( i = 1; i * 2 <= H->Size; i = Child )  {
           Child = i * 2;
           if  ( Child != H->Size && H->Elements[ Child + 1]
                                        < H->Elements[ Child ] )
                Child++;
           if  ( LastElement > H->Elements[ Child ] )
                H->Elements[ i ] = H->Elements[ Child ];
           else
               break;
     };
     H->Elements[ i ] = LastElement;
     return MinElement;
}
```

Complexity of Heap Operations

- Insertion: precolate up, O(logN)
- DeleteMin: precolate down, O(logN)
- Can heap be used in sorting? Complexity?

Heap 是找最小(或最大)，

可以運用來做Sorting嗎？

Heap Sort

- Algorithm (Increasing order)

  Step 1: Build heap (Min-heap)

  Step 2: for (i=0; i <N; i++)

  DeleteMin;

Heap Sort有可能
不需額外O(n)的空間嗎 ？
(hint: 利用Max-Heap由小排到大)

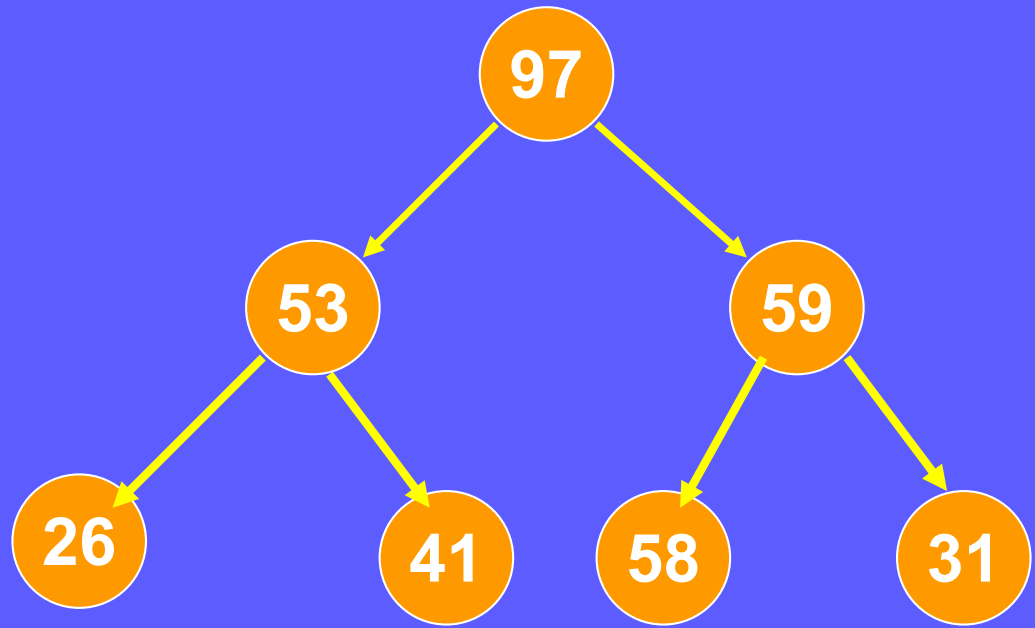
Improvement of Heap Sort

- Drawback of previous heap sort:
  extra array to store output
- Algorithm (Increasing order)

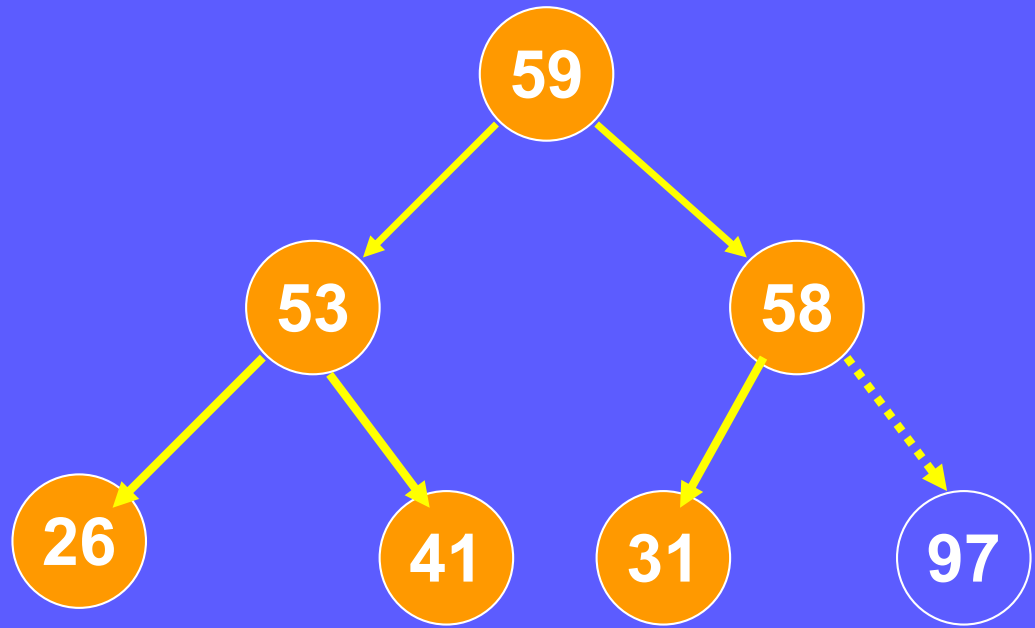
  Step 1: Build heap (Max-heap)

  Step 2: for (i=0; i <N; i++)

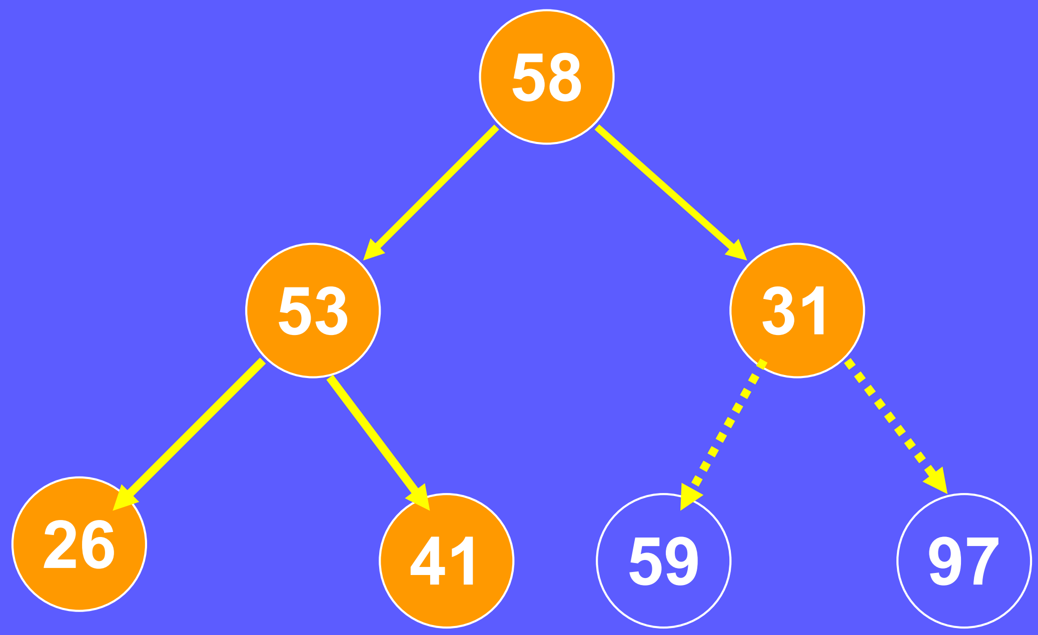
  DeleteMax;



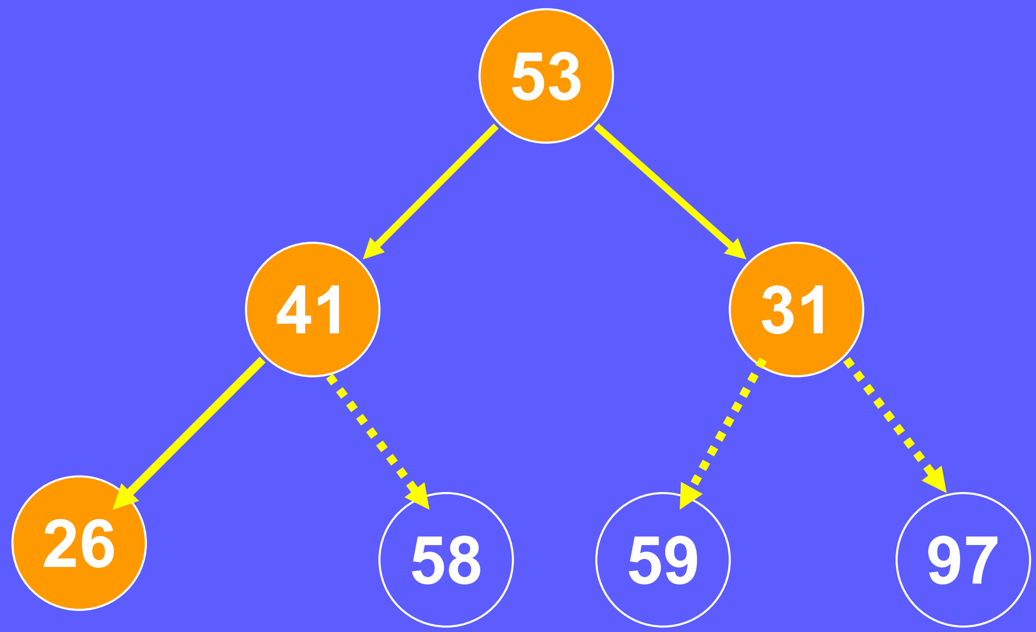
	97	53	59	26	41	58	31				
0	1	2	3	4	5	6	7	8	9	10	11



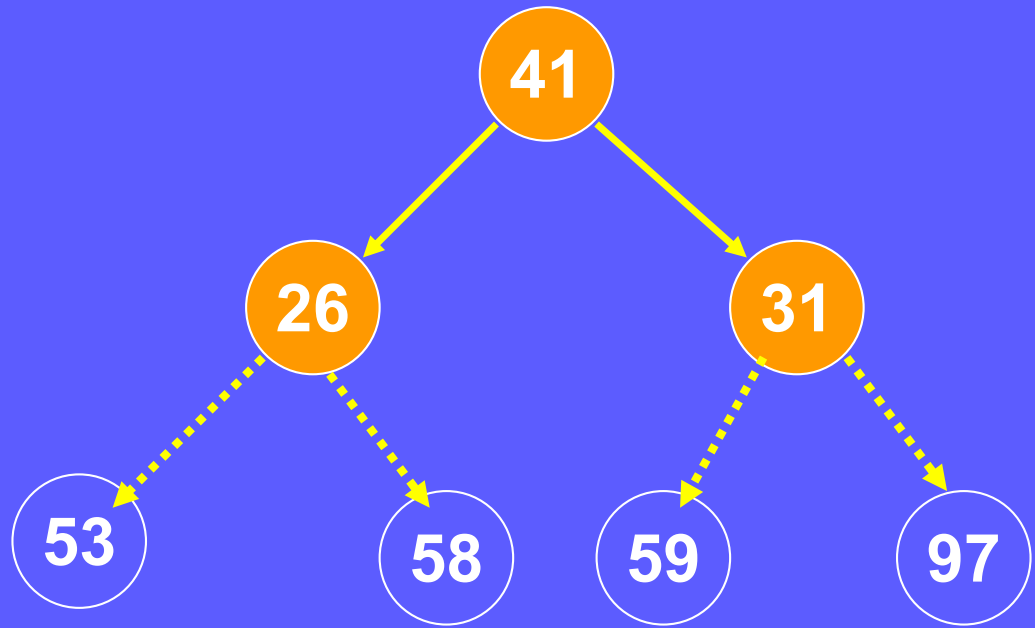
	59	53	58	26	41	31	97				
0	1	2	3	4	5	6	7	8	9	10	11



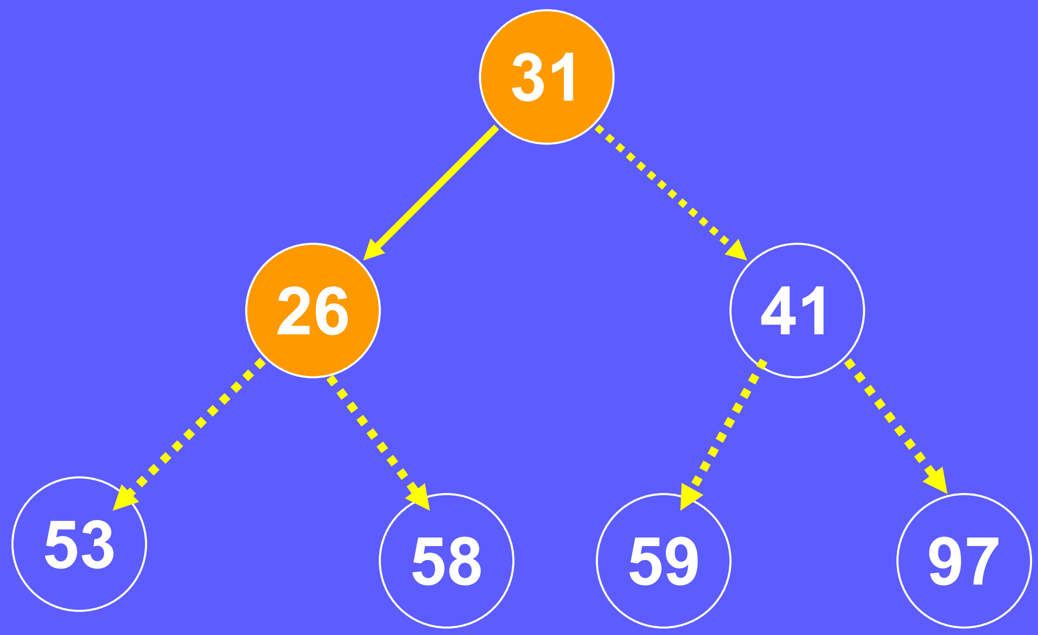
	58	53	31	26	41	59	97				
0	1	2	3	4	5	6	7	8	9	10	11



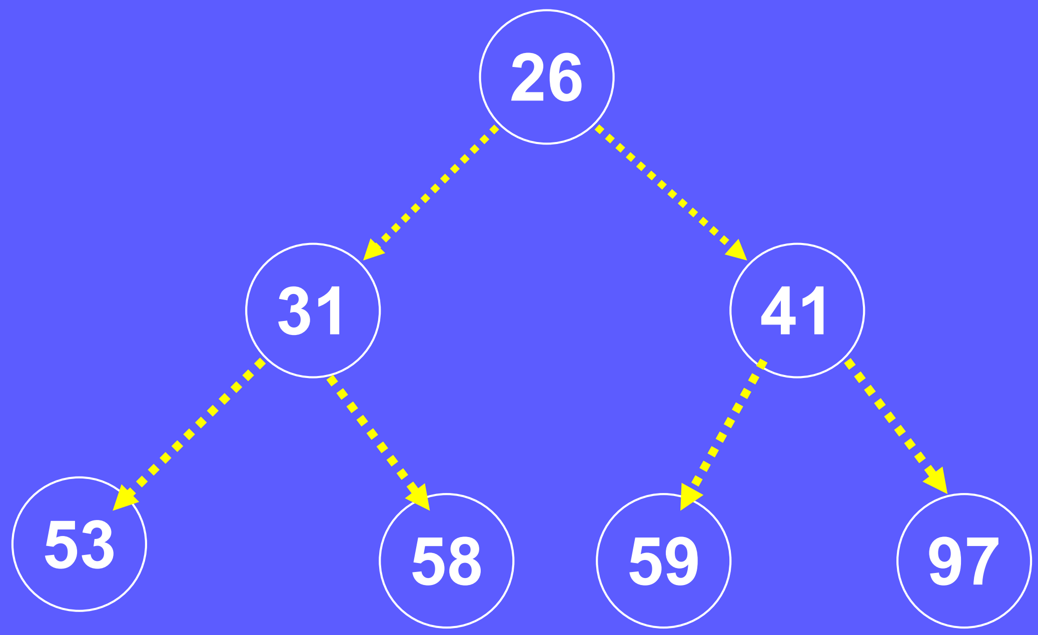
	53	41	31	26	58	59	97				
0	1	2	3	4	5	6	7	8	9	10	11



	41	26	31	53	58	59	97				
0	1	2	3	4	5	6	7	8	9	10	11



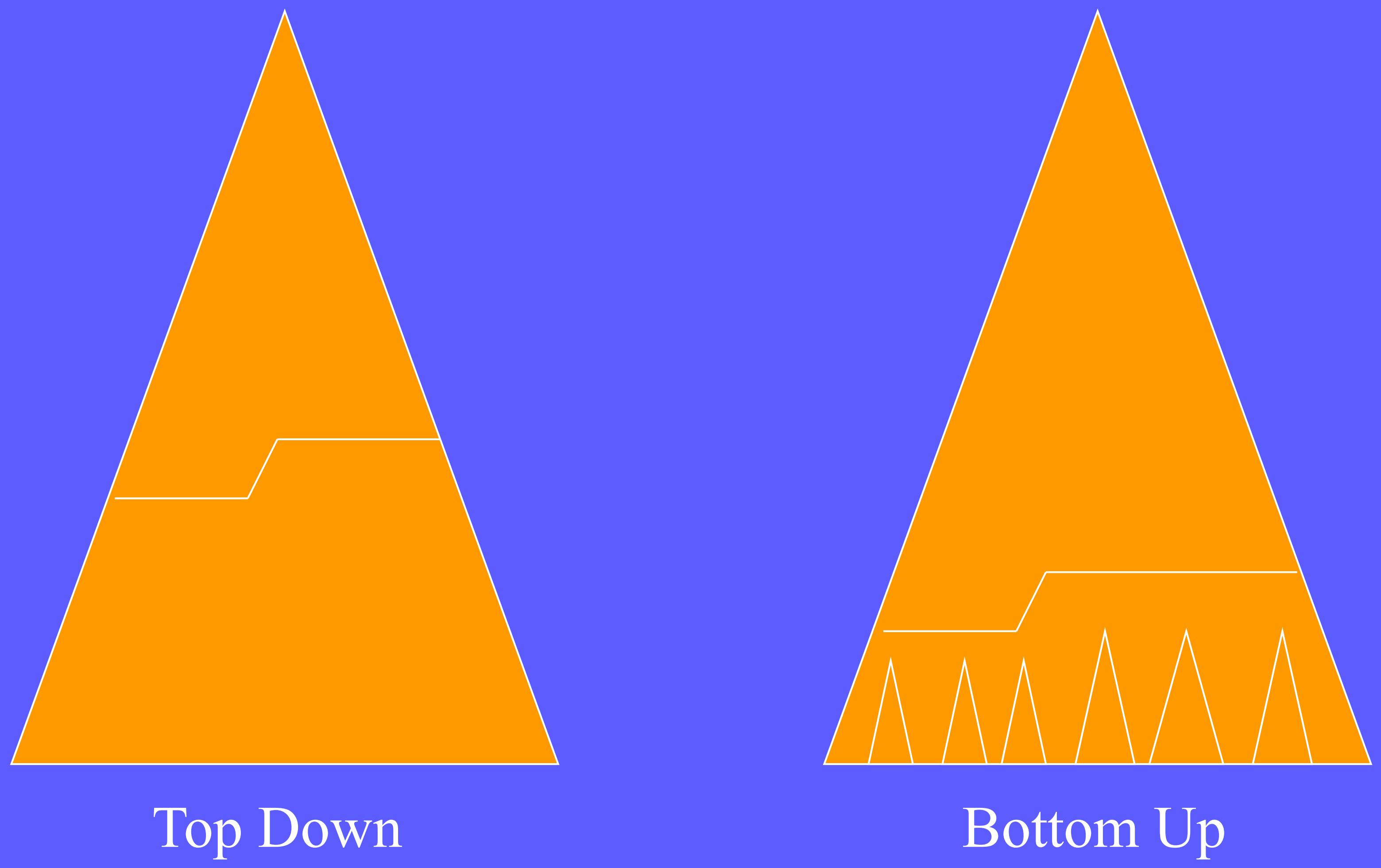
	31	26	41	53	58	59	97				
0	1	2	3	4	5	6	7	8	9	10	11



	26	31	41	53	58	59	97				
0	1	2	3	4	5	6	7	8	9	10	11

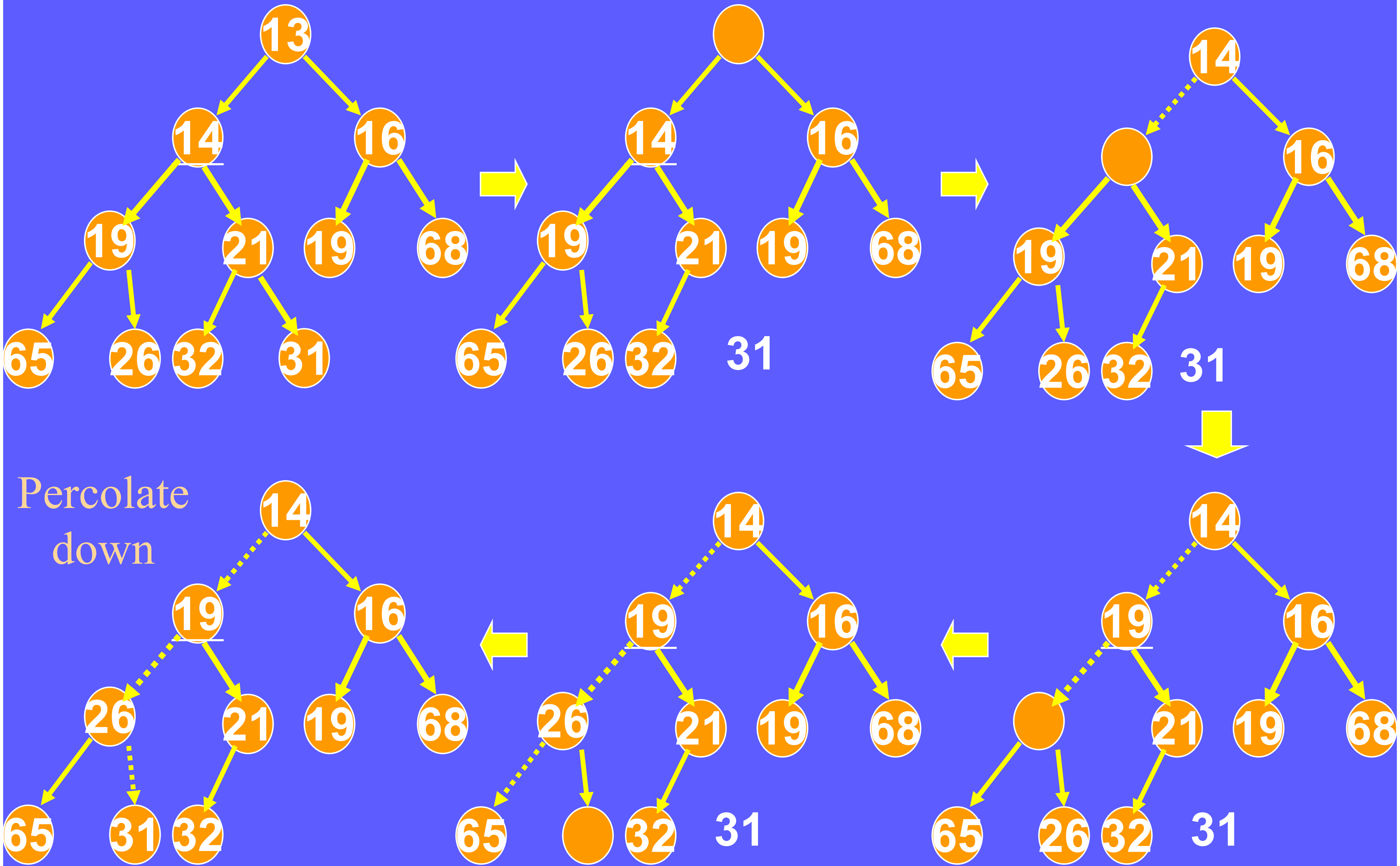
Building Heap

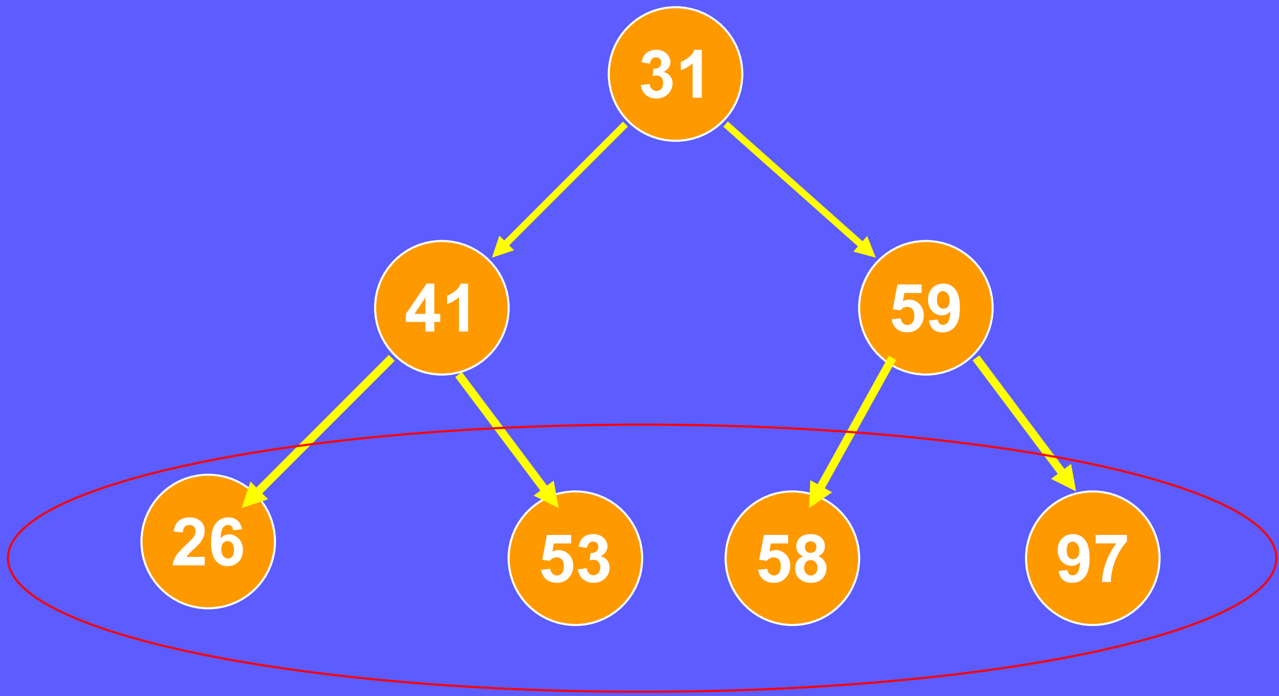
- Approaches
  - **Top down**
    - **Hypothesis: array [1..i] is a heap**
  - **Bottom up**
    - **Hypothesis: all trees represented by array A[i+1..n] satisfy the heap condition**


Top Down
Bottom Up

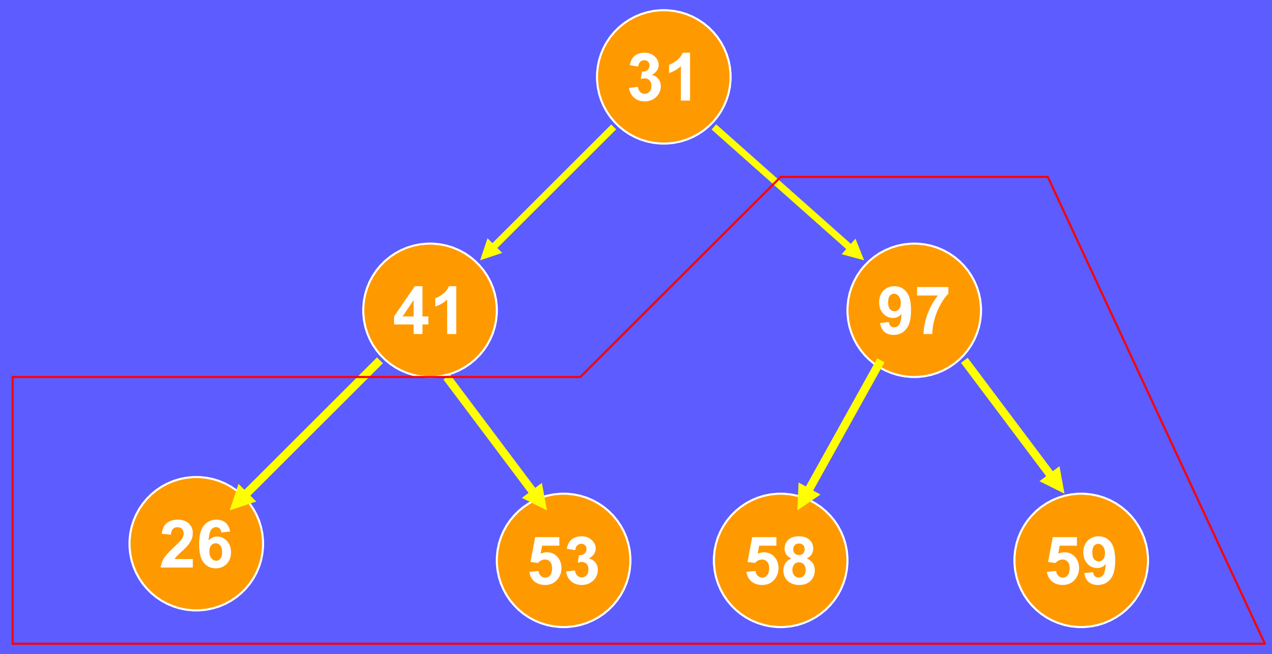
```
Building Heap (ElementType A[], int N)
{
  int i;
  for (i=N/2;  i >=0;  i--)
        PercDown(A, i, N)
}
```

DeleteMin

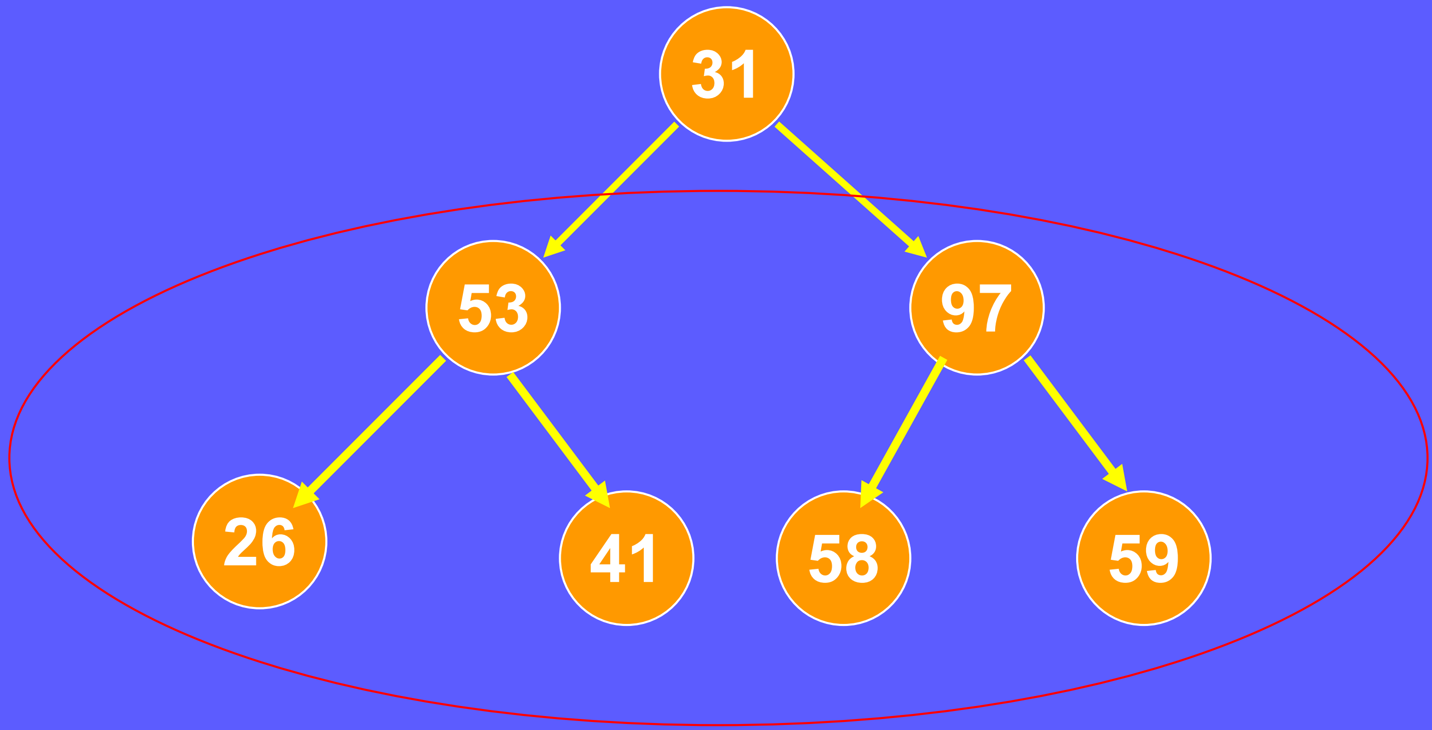




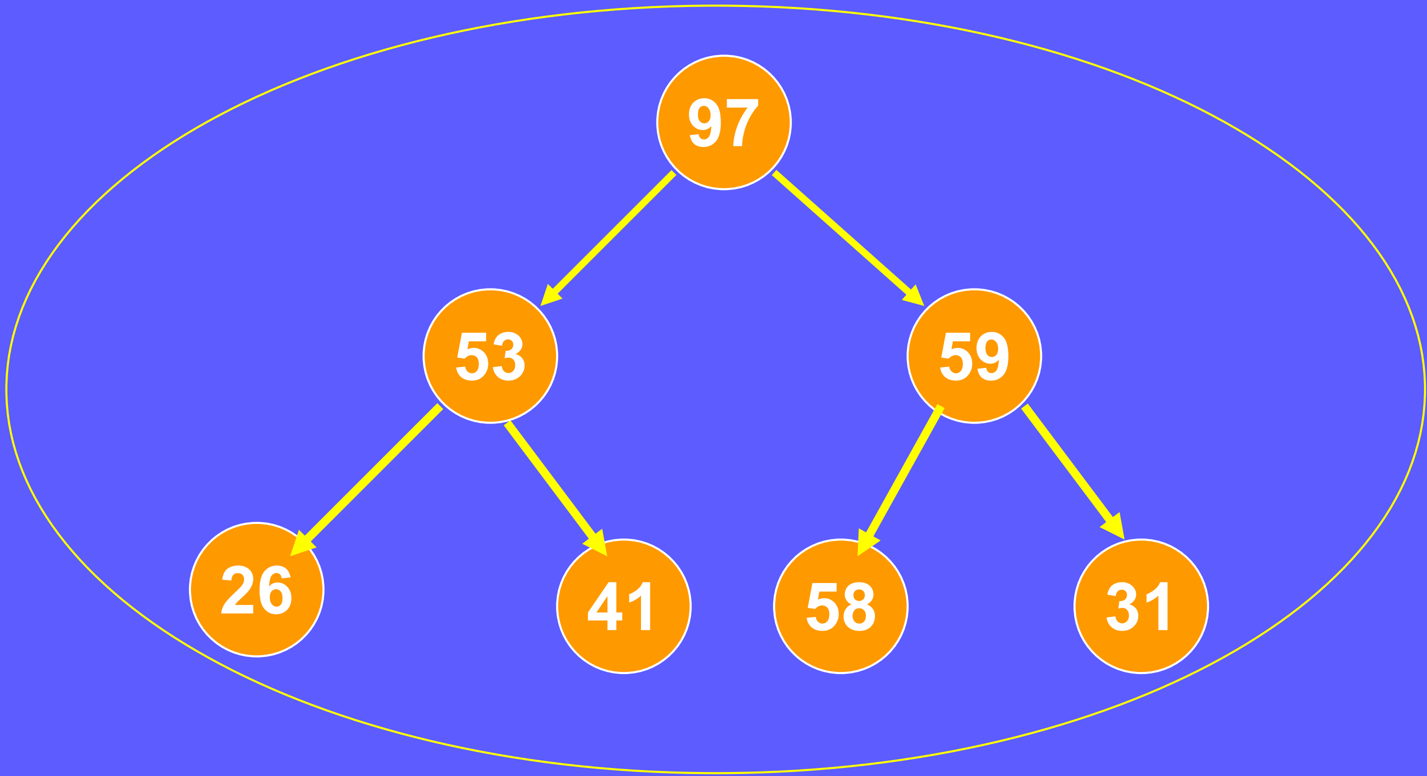
	31	41	59	26	53	58	97				
0	1	2	3	4	5	6	7	8	9	10	11



	31	41	97	26	53	58	59				
0	1	2	3	4	5	6	7	8	9	10	11



	31	53	97	26	41	58	59				
0	1	2	3	4	5	6	7	8	9	10	11



	97	53	59	26	41	58	31				
0	1	2	3	4	5	6	7	8	9	10	11

針對Large Structure Data
(例如500萬筆資料，
每筆資料有10000個欄位)，
如何減少排序時，
10000個欄位資料交換的時間？

Sorting Large Structures

Sorting Large Structures

- Sorting large structure
  - swapping two structures can be a very expensive operation

	No	Color	Size	A1	A2	…	A1000
1	103	Red	10	…	…	…	…
2	302	Blue	5	…	…	…	…
3	205	Blue	30	…	…	…	…
4	236	Green	90	…	…	…	…
5	282	White	20	…	…	…	…

Sorting Large Structures (cont.)

- Solution: indirect sorting
  - To have the input array contain pointers to the structures
  - Sort by comparing the keys the pointers point to, swapping pointers when necessary.

	No	Color	Size	A1	A2	…	A1000
1	103	Red	10	…	…	…	…
2	302	Blue	5	…	…	…	…
3	205	Blue	30	…	…	…	…
4	236	Green	90	…	…	…	…
5	282	White	20	…	…	…	…

	Size
2	5
1	10
5	20
3	30
4	90

Sorting

method	average	worst	stability	extra space
bucket	O(n)	O(m)	stable	O(m)
radix	$O(n\log_p k)$	$O(n\log_p k)$ ~O(n)	stable	O(nxp)
insertion	$O(n^2)$	$O(n^2)$	stable	O(1)
selection	$O(n^2)$	$O(n^2)$	unstable	O(1)
bubble	$O(n^2)$	$O(n^2)$	stable	O(1)
merge	O(nlogn)	O(nlogn)	stable	O(n)
quick	O(nlogn)	$O(n^2)$	unstable	O(logn)~O(n)
heap	O(nlogn)	O(nlogn)	unstable	O(1)