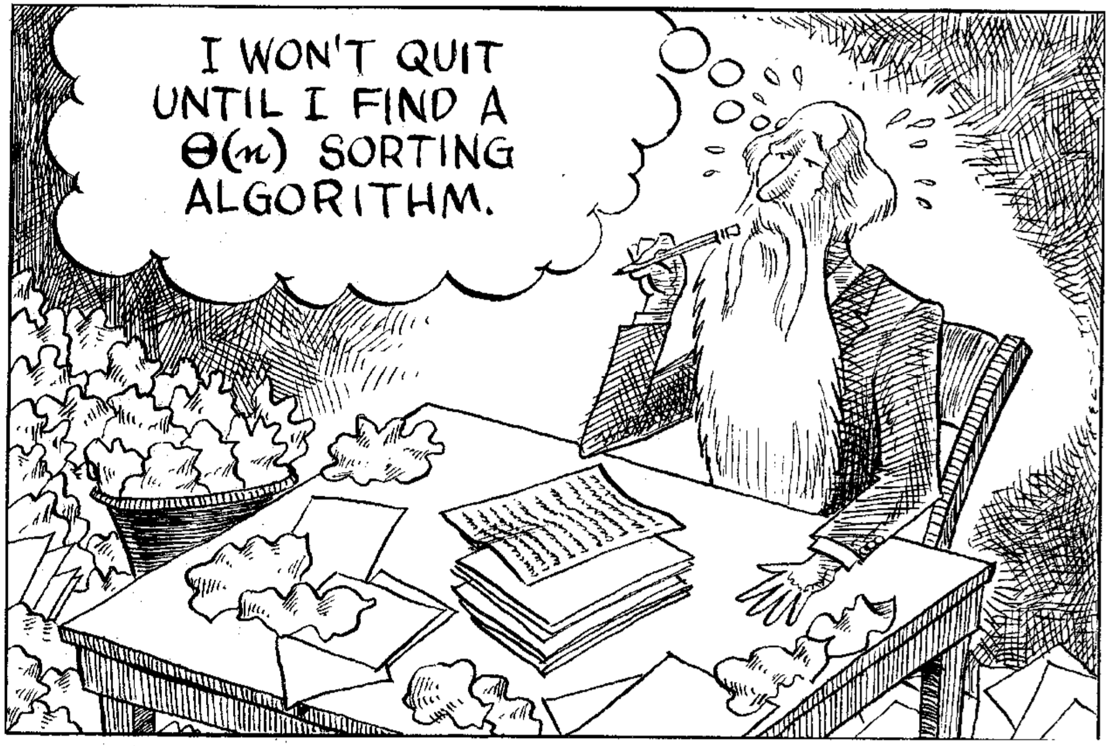
Chapter 7

Introduction to Computational Complexity: The Sorting Problem

Computational Complexity



- Can we find a new sorting algorithm better than $O(n\log n)$?

Computational Complexity

- Computational complexity
  - A field of computer science studying a problem itself, not developing efficient algorithms solving the problem.
    - Prove that some problems cannot be solved by computers : halting problem
    - Prove a lower bound of problems
  - A computational complexity analysis tries to determine a ***lower bound*** on the efficiency of all algorithms for a given problem.

Computational Complexity

- For example, the lower bound of sorting problem is $\Omega(n\log n)$.
- This implies that it is ***impossible*** to develop an algorithm better than $O(n\log n)$.
- Therefore, merge-sort, quick-sort algorithms are the best algorithms solving the sort problem.

Computational Complexity

❑ Example: Matrix Multiplication Problem

- How fast can we multiply two matrices of size $n \times n$?

- Design an efficient algorithm: basic operation is multiplication of two numbers
  - $O(n^3)$: simple
  - $O(n^{2.81})$: Strassen [1969]
  - $O(n^{2.38})$: Coppersmith, Winograd [1969]

- Develop a lower bound of this problem
  - $\Omega(n^2)$: easy

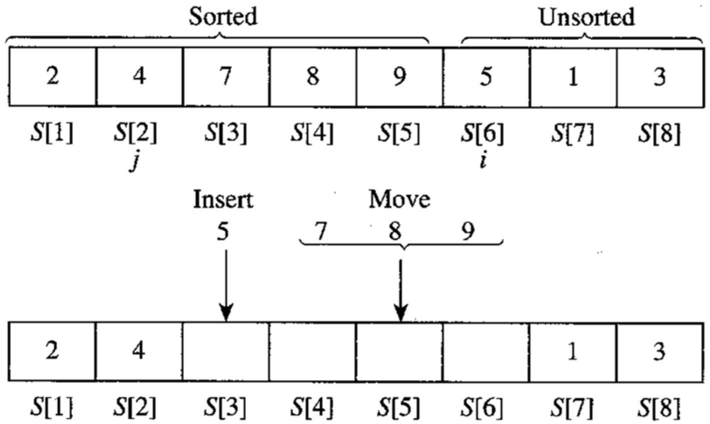
Computational Complexity

- What do we do next?
  - Fill the gap between the lower bound $\Omega(n^2)$ and the best upper bound $O(n^{2.38})$.
    - Develop a new algorithm better than $O(n^{2.38})$.
    - Prove a new lower bound of the problem better than $\Omega(n^2)$, for example $\Omega(n^{2.3456789})$.

KMU KOOKMIN UNIVERSITY SINCE 1946 국민대학교

Insertion Sort

- Insertion sorting




```
void insertionsort (int n, keytype S[ ])
{
   index i, j;
   keytype x;

   for (i = 2; i <= n; i++) {
      x = S[i];
      j = i − 1;
      while (j > 0 && S[j] > x) {
         S[j + 1] = S[j];
         j− −;
      }
      S[j + 1] = x;
   }
}
```

Insertion Sort

Analysis

- Worst-case Time complexity Analysis of ***Number of Comparisons***
  - Basic operation : the comparison of *S[j]* with *x*.
  - Input size : *n*, the number of keys to be sorted

$$T(n) = \sum_{i=2}^{n} (i-1) = \frac{n(n-1)}{2}$$

Selection Sort

❑ Selection sorting

```
void selectionsort (int n, keytype S[ ])
{
   index i, j, smallest;

   for (i = 1; i <= n − 1; i++) {
      smallest = i;
      for (j = i + 1; j <= n; j++)
         if (S[j] < S[smallest])
            smallest = j;
      exchange S[i] and S[smallest];
   }
}
```

- Selection sorting is an $O(n\log n)$ algorithm.

Heapsort (Binary Heap)

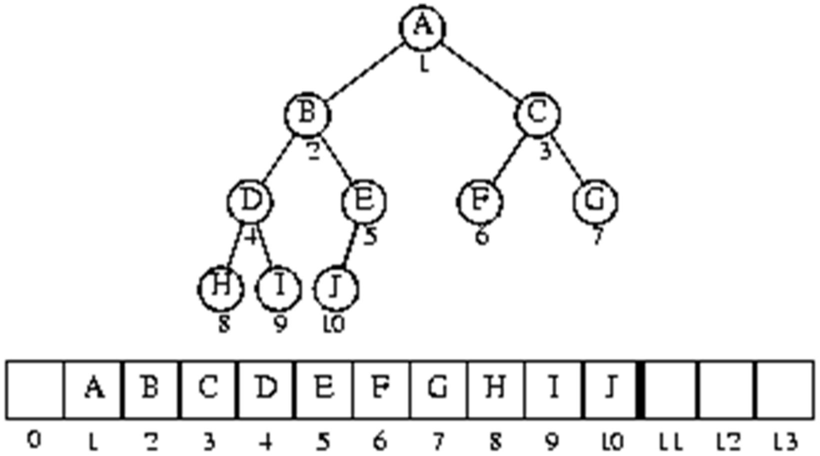
- Categories
  - A Dictionary:
    - Basic Operations
      - Insert
      - Delete
      - Search
  - Data Structures for Dictionary
    - Binary Search Tree,
    - Red-Black Tree,
    - Splay Tree, etc

Heapsort (Binary Heap)

- A Priority Queue
  - Basic Operations
    - Insert
    - Delete Min (or Delete Max)
- Data Structures for Priority Queue
  - The Binary Heap

Complete Binary Tree

- The complete binary tree
  - A tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right.
  - An array of size *N* can represent a complete binary tree with *N* elements.

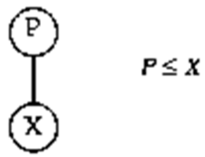


Complete Binary Tree

- Lemma:
  - The height (longest path length from the root) of a complete binary tree is $\lfloor \log N \rfloor$.
  - A complete binary tree of height H has between 2^H and $2^{H+1}-1$ nodes.
  - In an array representation of a complete binary tree, for a node of position *k*,
    - the parent is in position $\lfloor k/2 \rfloor$.
    - the left child is in *2k*
    - the right child is in *2k+1*

Binary Heap

- The binary heap
  - The binary heap has the following properties:
    - It is a complete binary tree
    - (**heap order property**) In a heap, for every node X with parent P, the key in P is smaller than or equal to the key in X.



Heap order property

  - In this case, the heap is called a *min heap*.
  - *Max heaps* have the heap order property in the other way.

Binary Heap

- Example:

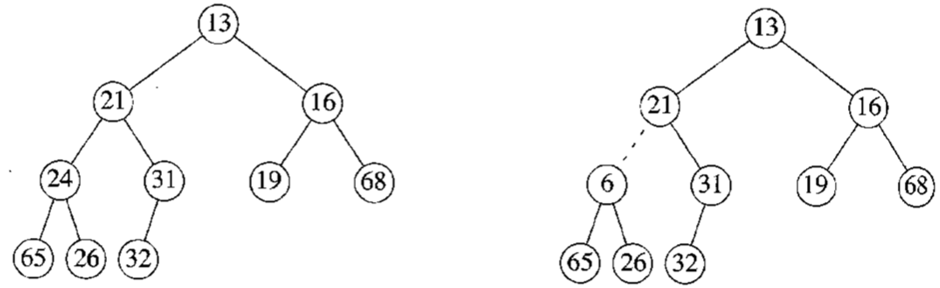


Figure 20.3 Two complete trees (only the left tree is a heap)

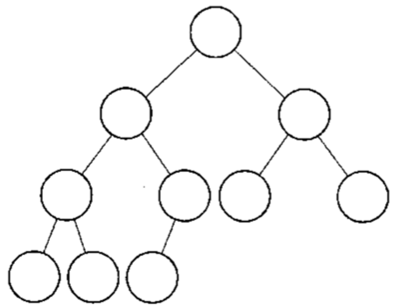


Figure 7.4 An essentially complete binary tree.

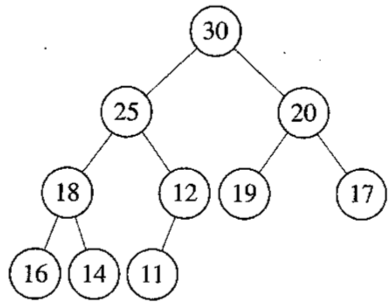
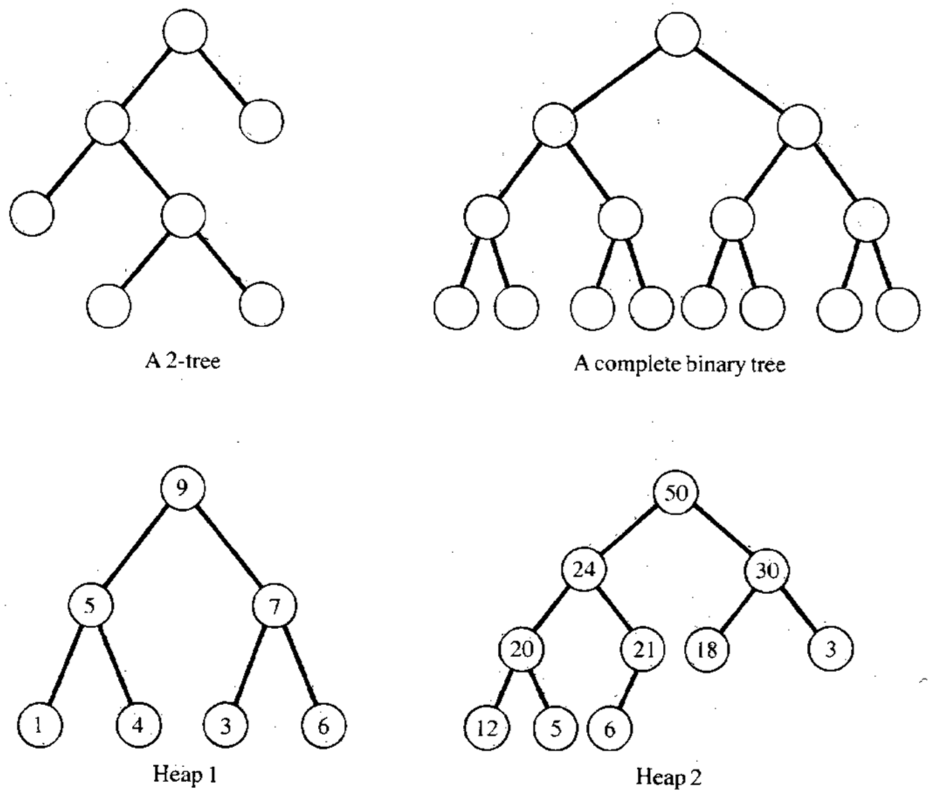


Figure 7.5 A heap.

Binary Heap

- Example:



Binary Heap

- Allowed operations in heap
  - Insert an element in a heap

배열로 그려서도 할줄 알아야함

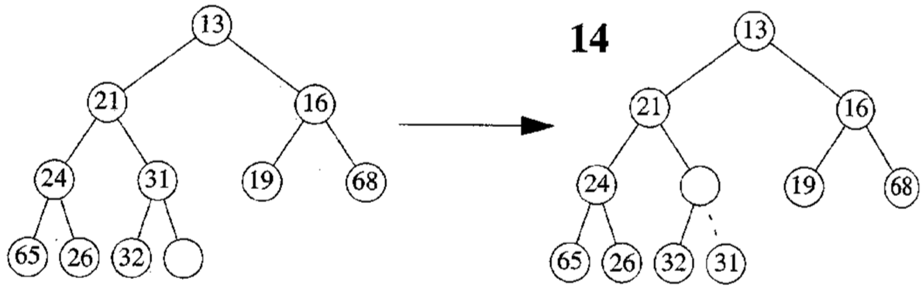


Figure 20.9 Attempt to insert 14, creating the hole and bubbling the hole up

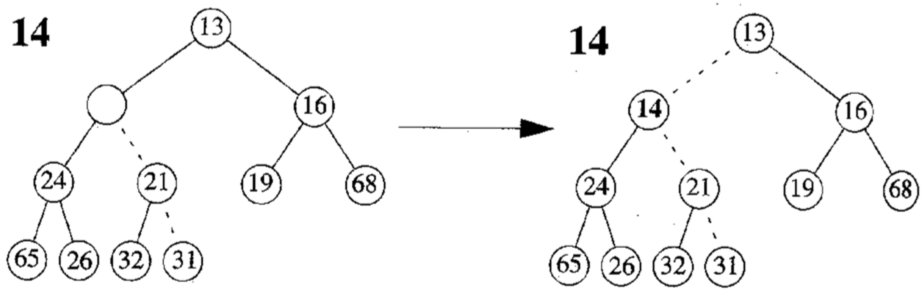


Figure 20.10 The remaining two steps to insert 14 in previous heap

Binary Heap

- Delete a minimum element from a heap

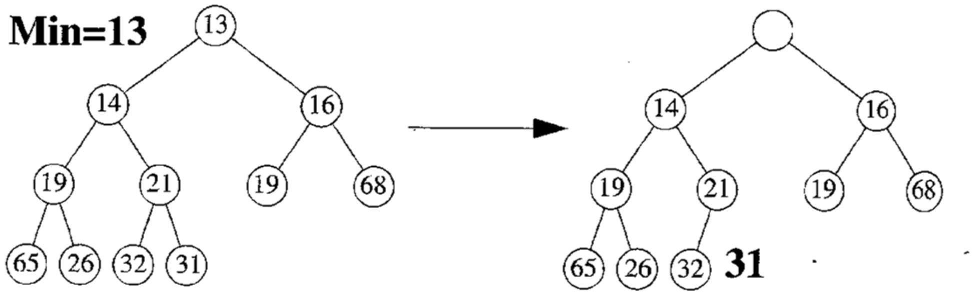


Figure 20.13 Creation of the hole at the root

Binary Heap

- Delete a minimum element from a heap

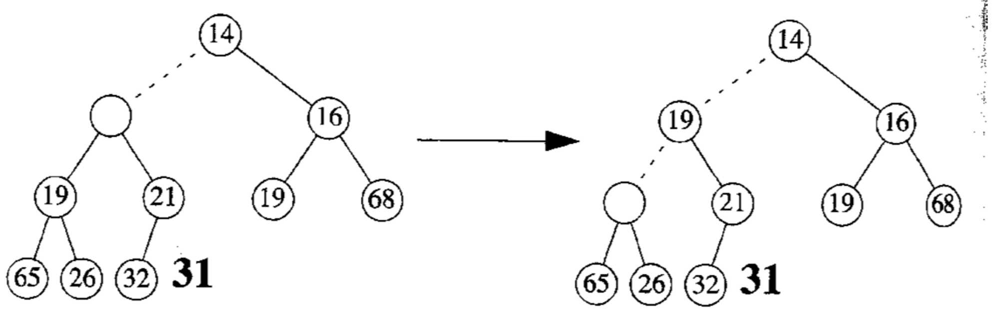


Figure 20.14 Next two steps in `DeleteMin`

delete min 할때 root 를 지우고 맨 오른쪽꺼를 root로 옮겨서 child2개중 작은거와 swap를 반복

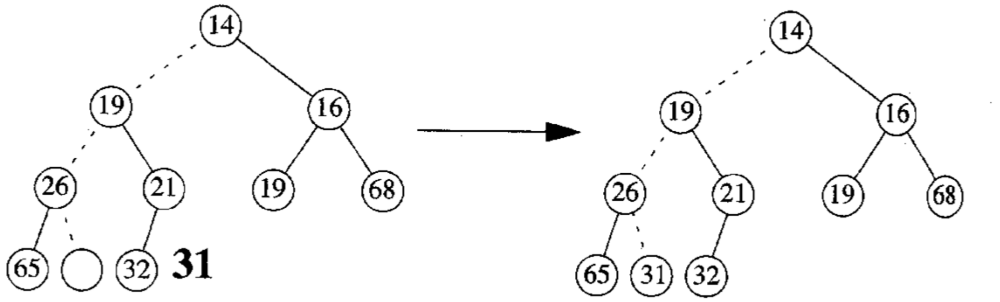


Figure 20.15 Last two steps in `DeleteMin`

Bina

- Delete a maximum element from a heap.

max heap 이므로 max 지운곳에
맨오른쪽꺼 6을 root에 임시로 저장시킨 후
max heap이 맞을 때 까지 child와 비교한 후
swap 한다 (max head에서는 큰거랑 swap)

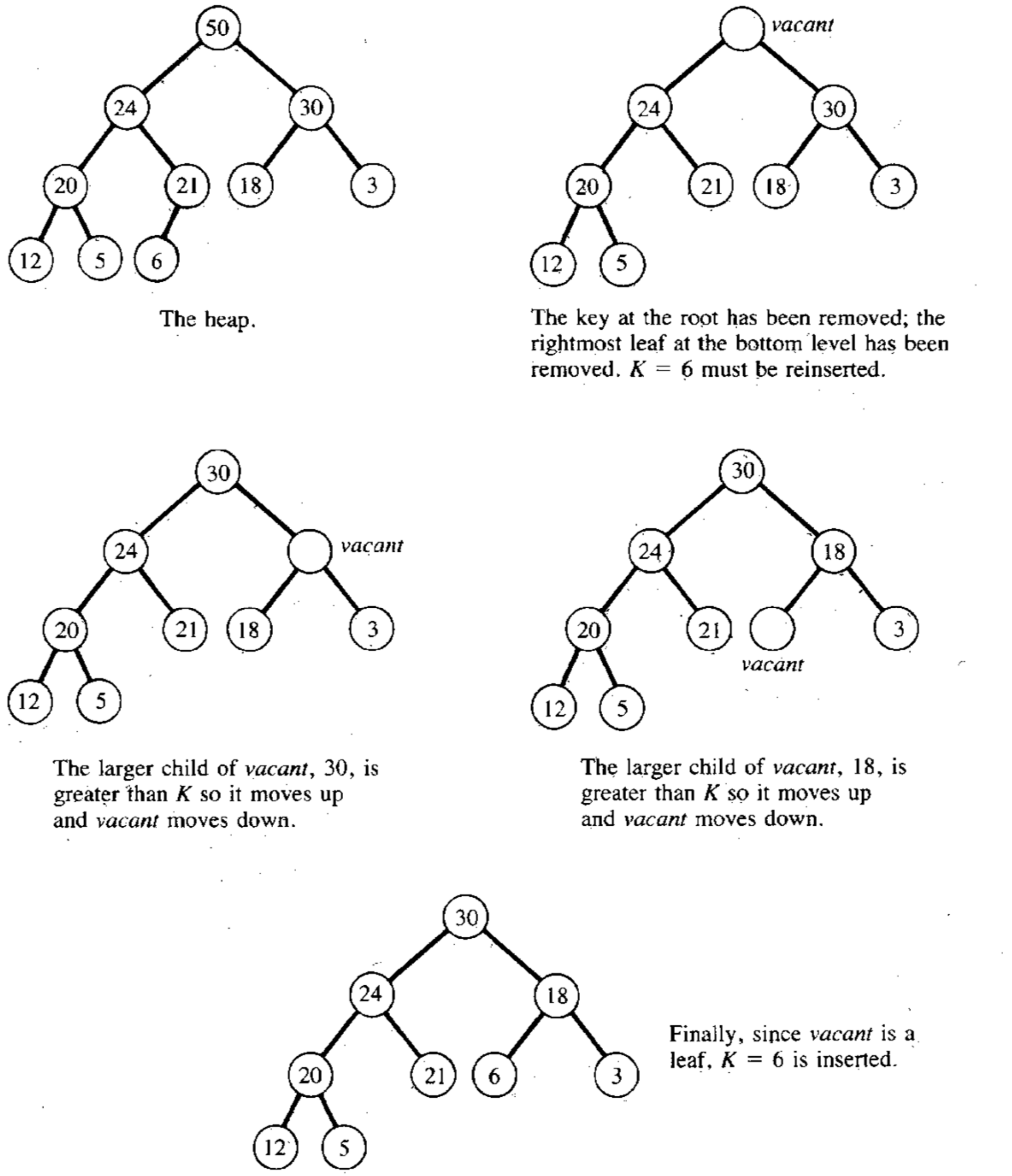


Figure 2.15 Deleting the key at the root and reestablishing the heap property.

Heap Construction

- Heap construction
  - If we are given a complete tree that does not have heap order, we are going to construct a heap.
  - FixHeap Operation:아까 6을 root로 보냈을 때 Head이 될때까지 알맞은 위치로 보내는과정
    - We are given a complete tree that only the root violates the heap order property.

Heap Construction

- Fix heap operation
  - Example for a max heap

Algorithm 2.8 FixHeap

Input: The *root* of a heap and a key *K* to be inserted.

Output: The heap with keys properly rearranged.

```
procedure FixHeap (root: Node; K: Key);
var
    vacant, largerChild : Node;
begin
    vacant := root;
    while vacant is not a leaf do
        largerChild := the child of vacant with the larger key;
        if K < largerChild's key
            then
                copy largerChild's key to vacant;
                vacant := largerChild
            else exitloop
        end { if }
    end { while };
    put K in vacant
end { FixHeap }
```

Heap Construction

- Fix heap operation

최악의 경우 leaf node까지 가야하므로

  - The FixHeap operation takes $2\lfloor \log N \rfloor$ time, if there are N elements in the heap.

Heap Construction

- **Shift down operation** fix heap이랑 의미는 같음
  - shiftdown() in the text book is the same as FixHeap() .

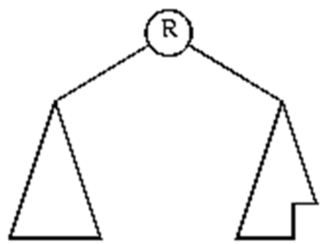
```
void siftdown (heap& H)                         // H starts out having the
{                                               // heap property for all
   node parent, largerchild;                    // nodes except the root.
                                                // H ends up a heap.
   parent = root of H;
   largerchild = parent's child containing larger key;
   while (key at parent is smaller than key at largerchild) {
      exchange key at parent and key at largerchild;
      parent = largerchild;
      largerchild = parent's child containing larger key;
   }
}
```

Heap Construction

❑ Heap construction by divide and conquer

```
procedure ConstructHeap (root: Node);
begin
    if root is not a leaf then
        ConstructHeap (left child of root);
        ConstructHeap (right child of root);
        FixHeap (root, key in root)
    end { if }
end { ConstructHeap }
```

왼쪽에 대해서 recursive
오른쪽에 대해서 recursive
root때문에 heap 아니므로
FixHeap 한번실행



Recursive view of the heap

Heap Construction

- Heap construction by divide and conquer
  - Analysis:

$$T(N) = \begin{cases} 1 & \text{if } N = 1 \\ 2T(N/2) + \log N & \text{otherwise} \end{cases}$$

  - Can you represent the recurrence equation in closed form? (In this case, we cannot apply the master's theorem. Why?)
  - Next time we will show that *T(N)=O(N)*.

Heap Construction

- Iterative version of Heap construction:

Algorithm 2.9 Heap Construction

Input: A heap structure (Property (1)) with keys in arbitrary nodes.

Output: The same structure satisfying the heap-ordering property (Property (2)).

for *level* := *depth*−1 **to** 0 **by** −1 **do**
  for each non-leaf *node* at level *level* **do**
    K := the key at *node*;
    FixHeap(*node*, *K*)
  end { for }
end { for }

for 2개의 loop를 한개로 줄일 수 있다.
실제로 constuct하는 거는 맨 마지막 child의 parent라는걸 이용

Heap Construction

- Iterative version of Heap construction:

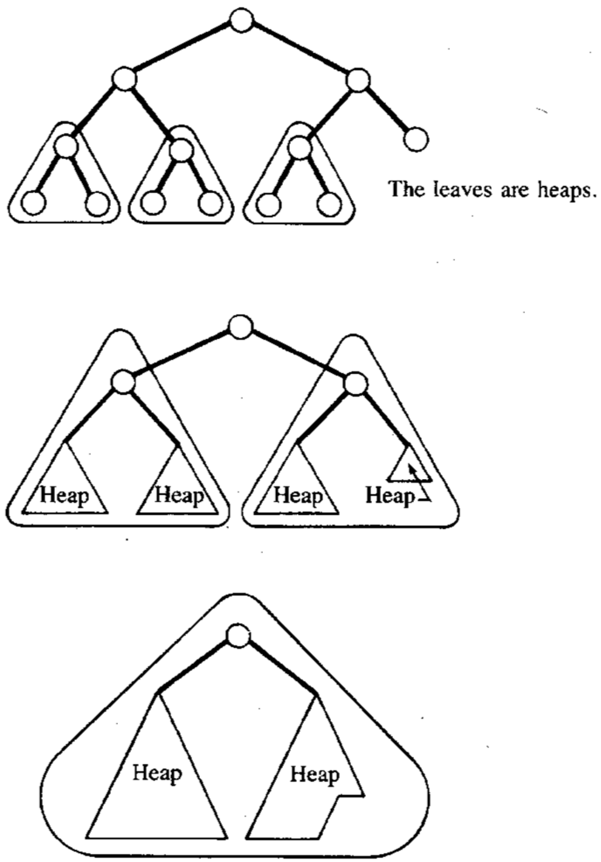


Figure 2.16 Constructing the heap. (*FixHeap* is called for each circled subtree.)

Heap Construction

❑ Example

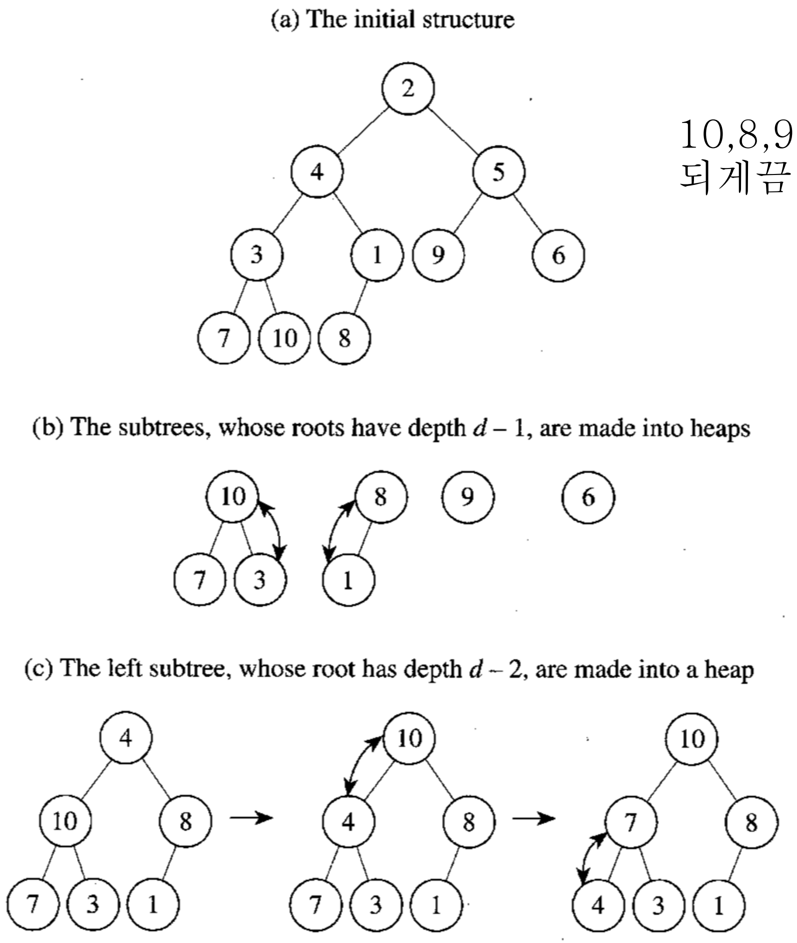


Figure 7.7 Using *siftdown* to make a heap from an essentially complete binary tree. After the steps shown, the right subtree, whose root has depth $d-2$, must be made into a heap, and finally the entire tree must be made into a heap.

10,8,9,7,2,5,6,4,3,1(정답)
되게끔 한번 그려서 해보삼

Heap Construction

- Example

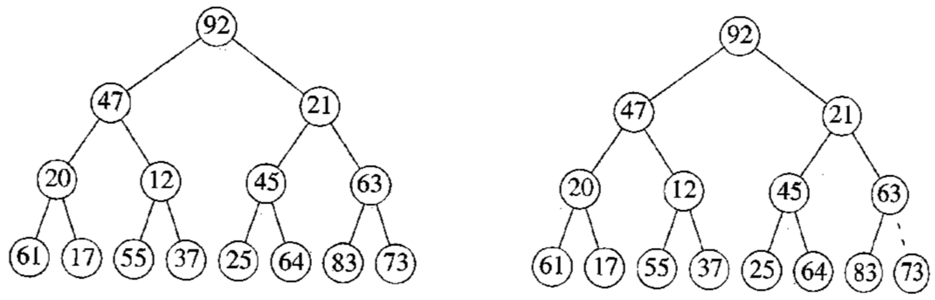


Figure 20.20 Initial heap (left); after `PercolateDown(7)` (right)

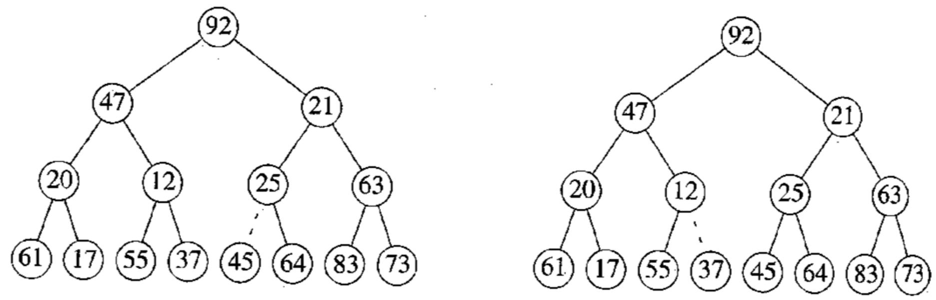


Figure 20.21 After `PercolateDown(6)` (left); after `PercolateDown(5)` (right)

Heap Construction

❑ Example

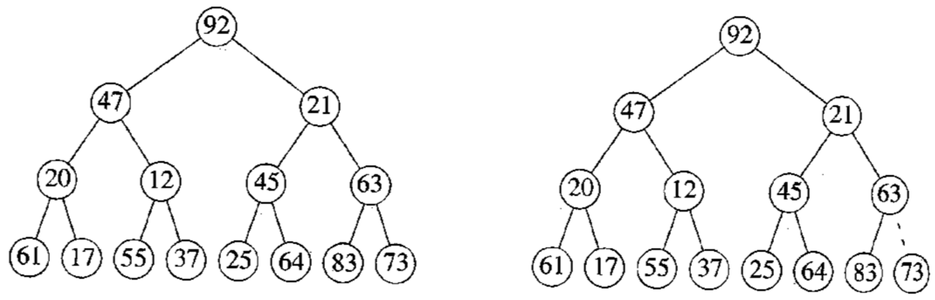


Figure 20.20 Initial heap (left); after `PercolateDown(7)` (right)

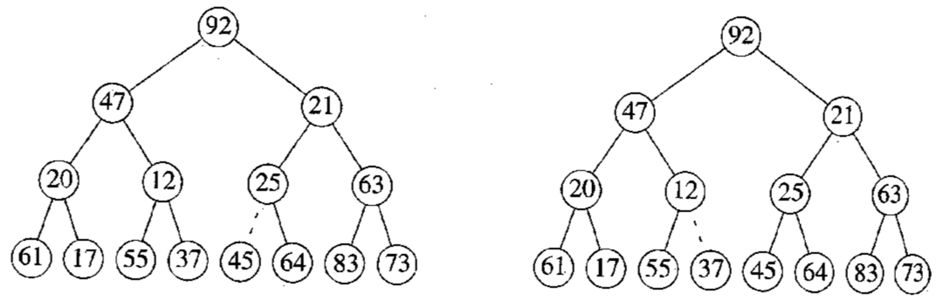


Figure 20.21 After `PercolateDown(6)` (left); after `PercolateDown(5)` (right)

Heap Construction

- Analysis of the heap construction:
  - Let $d = \lfloor \log N \rfloor$
  - Then,

$$
\begin{aligned}
T(N) &= \sum_{k=0}^{d-1} 2(d-k)(\text{the number of nodes at level } k) \\
&= 2\sum_{k=0}^{d-1}(d-k)2^k \\
&= 2^{d+2} - 2d - 4 \\
&= 4N - 2\log N - 4
\end{aligned}
$$

Heap Construction 방법엔 2가지가있는데
하나는 complete binary tree가있는데
헝크러져있어서
fix heap하는건 logn
empty에서 채우는건
nlogn 으로 훨씬 더 시간이 많이 걸린다

  - Thus the heap is constructed in $T(N) = O(N)$, linear time!

Heap Sorting

❑ Heapsort:

- The priority queue can be used to sort *N* items as follows:
  - Put all the elements in an array of size *N*.
  - Construct a heap
  - Extract every item by calling DeleteMin *N* times. The result is sorted.

insert 해서 min힙 만들고, delete min 하면 소팅 할수 있음

위의 방법은
일단 heap 쑤셔넣고
그다음 construct a heap (logN) 하고
그다음 delete min 한다.

Heap Sorting

- Heapsort:

Algorithm 2.10 Heapsort

Input: *L*, an unsorted array, and $n \geq 1$, the number of keys.

Output: *L*, with keys in nondecreasing order.

```
procedure Heapsort (var L: Array; n: integer);
var
    i, heapsize : Index;
    max : Key;
begin
    { Heap Construction }
    for i := ⌊n/2⌋ to 1 by −1 do
        FixHeap (i, L[i], n)
    end { for };

    { Repeatedly remove the key at the root and rearrange the heap. }
    for heapsize := n to 2 by −1 do
        max := L[1];
        FixHeap (1, L[heapsize], heapsize−1);
        L[heapsize] := max
    end { for }
end { Heapsort }
```

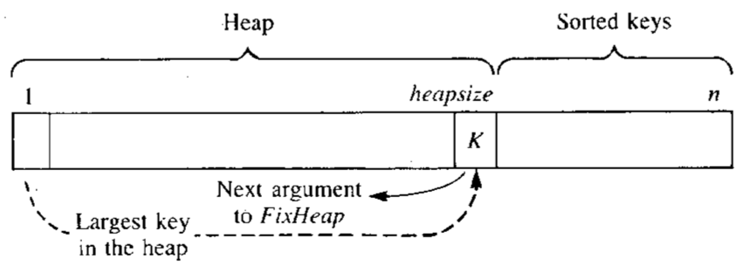


Figure 2.18 The heap and sorted keys in the array.

Heap Sorting

- Save the array space:
  - In heapsort, we construct a max heap, and retract a max value from the heap and put it in the end of the heap. Then we sort elements in increasing order.

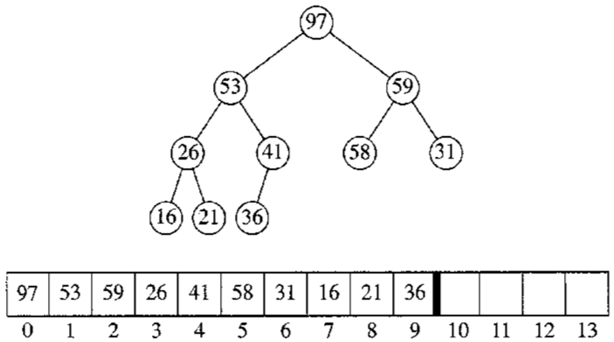


Figure 20.28 (Max) Heap after `FixHeap` phase

Heap Sorting

- Save the array space:

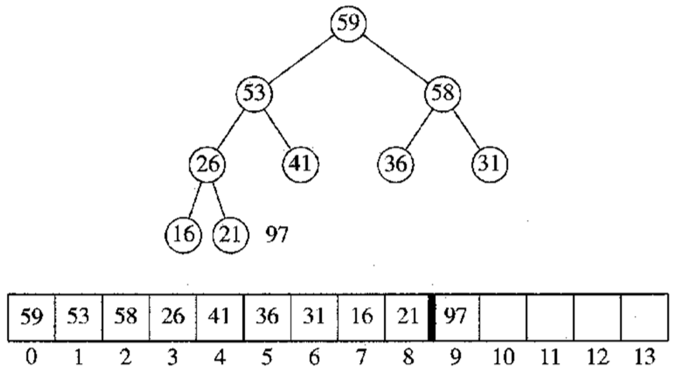
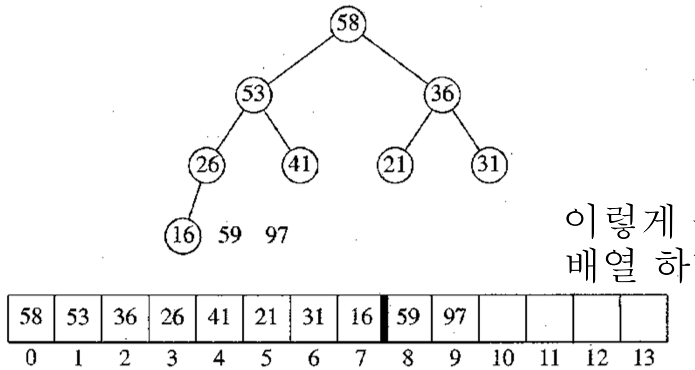


Figure 20.29 Heap after first `DeleteMax`

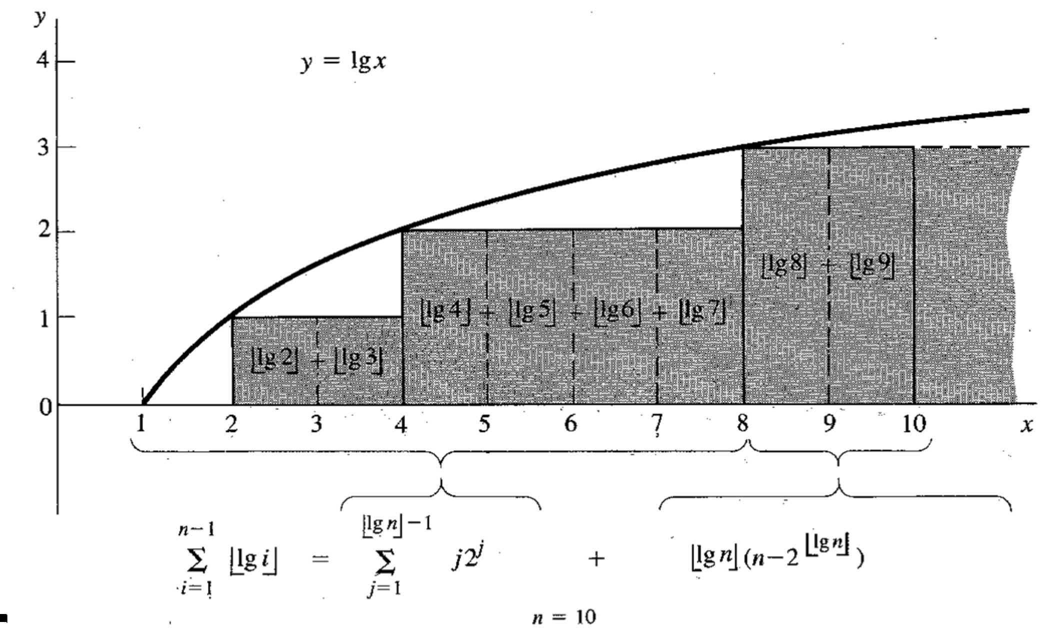


이렇게 복잡한 작업 하는이유?
배열 하나에 힙과 소팅 두마리 토끼를 잡을수 있음

왼쪽은 max 힙 // 오른쪽은 sorting된거

Heap Sorting

- **Analysis of heapsort:**
  - Since the number of comparison done by FixHeap on a heap with *k* elements is at most $2\lfloor \log k \rfloor$, so the total for all deletions is at most $2\sum_{k=1}^{n-1}\lfloor \log k \rfloor$



Heap Sorting

- Analysis of heapsort:
  - Let $d = \lfloor \log N \rfloor$
  - The sum is

$$\sum_{k=1}^{d-1} k2^k + d(N-2^d)$$
$$= 2(d2^{d+1} - 2^d + 1) + d(N-2^d)$$
$$= Nd - 2^{d+1} + 2$$
$$= O(N \log N)$$

  - Therefore heapsort takes *O(N log N)* time to sort *N* elements!

Lower Bounds for Sorting

- Lower bounds for sorting only by comparisons of keys
- Decision trees for sorting algorithms
  - An algorithm for sorting three distinct numbers:

```
void sortthree (keytype S[ ])      // S is indexed from 1 to 3.
{
   keytype a, b, c;

   a = S[1]; b = S[2]; c = S[3];
   if (a < b)
      if (b < c)
         S = a, b, c;              // This means S[1] = a; S[2] = b; S[3] = c;
      else if (a < c)
         S = a, c, b;
      else
         S = c, a, b;

   else if (b < c)
      if (a < c)
         S = b, a, c;
      else
         S = b, c, a;
   else
      S = c, b, a;
}
```

Lower Bounds for Sorting

- Lemma 7.1
  - To every algorithm for sorting distinct numbers, there corresponds a decision tree containing exactly *n*! keys.
- Example:
  - The decision tree corresponding to exchange sort when sorting three numbers.

best case : 2번
worst case : 3번

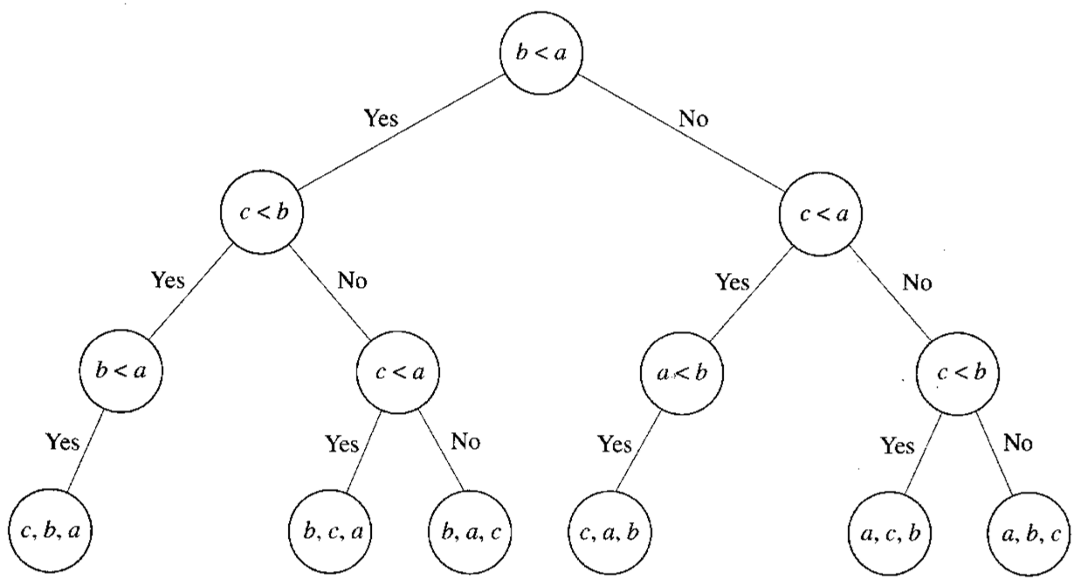


Figure 7.12 The decision tree corresponding to Exchange Sort when sorting three keys.

Lower Bounds for Sorting

COMPLETE BT 이상적 이유 말하고 (논리1)
모든 것은 디시젼 트리로 묶을수 있습니다. 가장 worst case가 적은거 complete BT입니다.
아래 식 (논리2) 식 다 외워야함 ㅠ

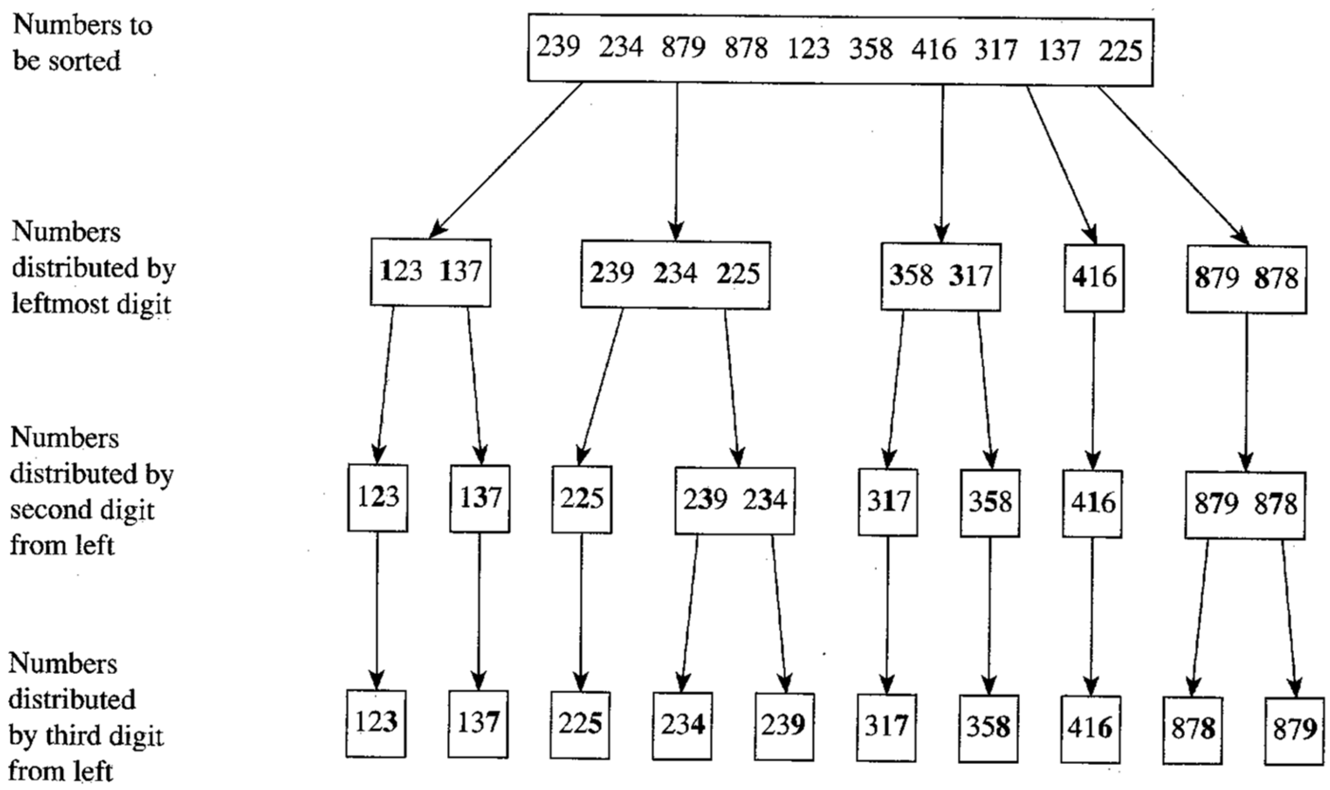
- **Theorem 7.2** 이거 시험!!!
  - Any algorithm that sorts distinct numbers only by comparison of numbers must in the worst case do at least $\lceil \lg(n!) \rceil = O(n\log n)$ comparison of numbers.
- **Proof:**
  - By lemma 7.1 the decision tree has $n!$ leaf nodes
  - Then the depth of the tree is greater than or equal to $\lceil \lg(n!) \rceil$.
  - Note that

$$\begin{aligned}
\lg(n!) &= \lg[n(n-1)(n-1)\cdots(2)1] \\
&= \sum_{i=1}^{n} \lg i \\
&\geq \int_1^n \lg x\,dx = \frac{1}{\ln 2}(n\ln n - n + 1) \\
&\geq n\lg n - 1.45n
\end{aligned}$$

Sorting by Distribution (Radix Sort)

- Method 1

Figure 7.14 Sorting by distribution while inspecting the digits from left to right.



Sorting by Distribution (Radix Sort)

- Method 2

Figure 7.15 Sorting by distribution while inspecting the digits from right to left.