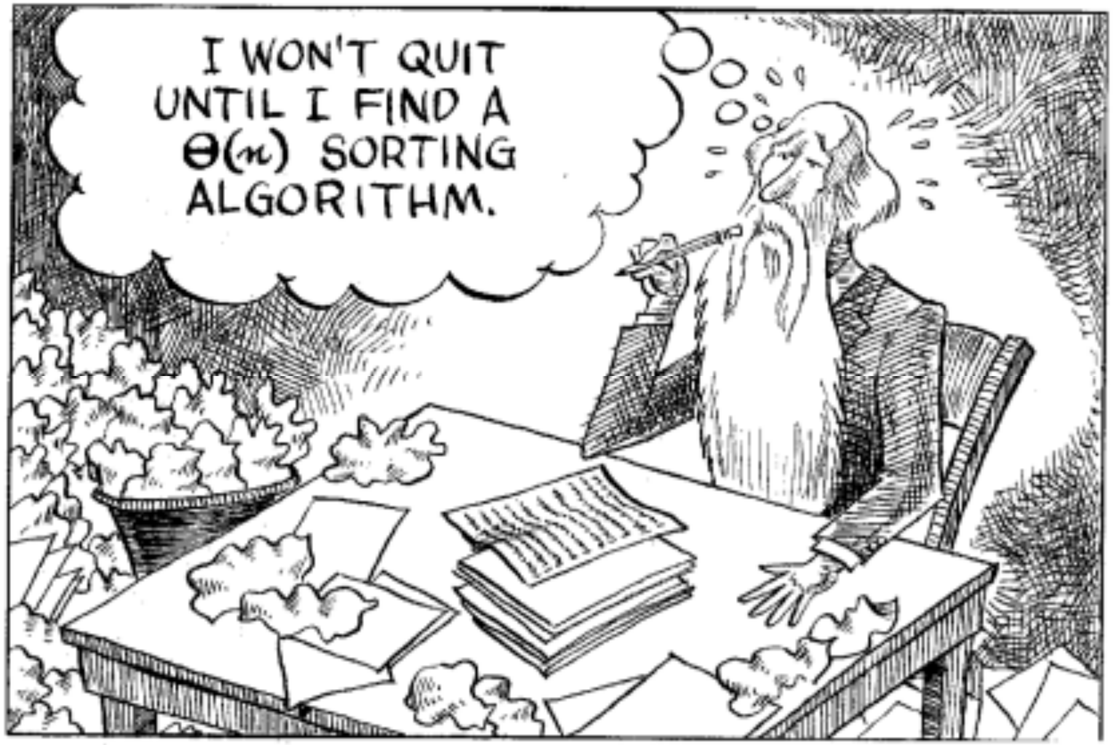
Chapter 7

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]

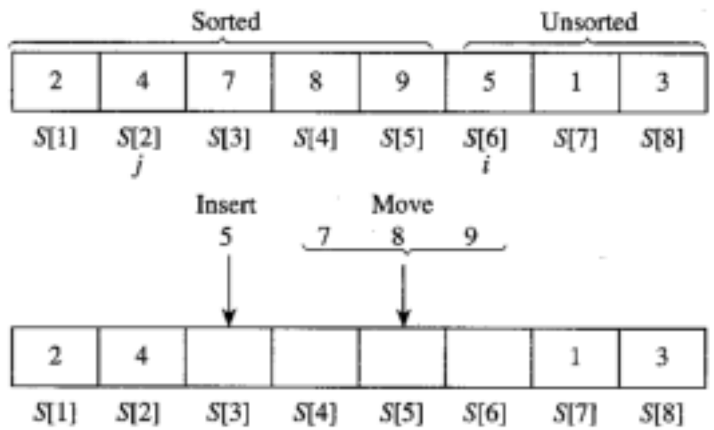
- Develop a lower bound of this problem
  - $\Omega(n^2)$: easy

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 $\Omega(n^{2.3456789})$.

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)

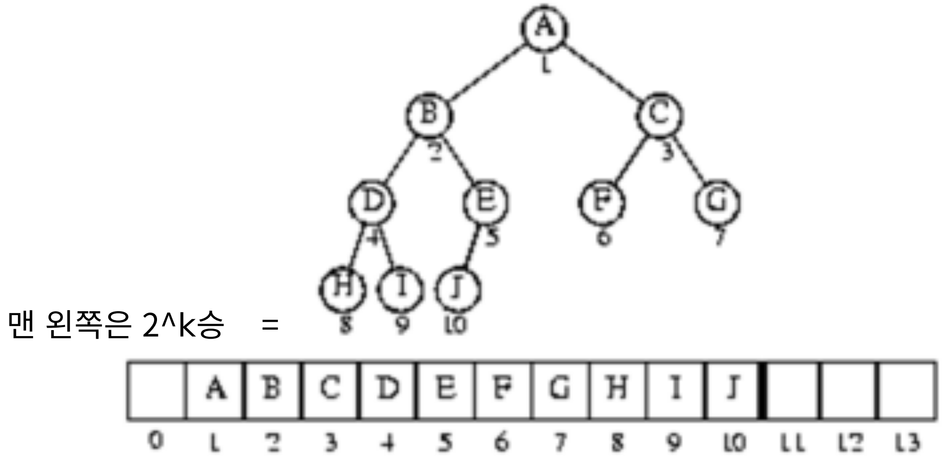
- Categories
  - A Dictionary:
    - Basic Operations
      - Insert
      - Delete
      - Search
  - Data Structures for Dictionary
    - Binary Search Tree,
    - Red-Black Tree,
    - Splay Tree, etc

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.



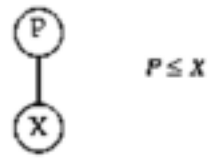
Complete Binary Tree

- 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*

N개 h 는??? =>
2^k<= N <2^(k+1) 로그 씨우면 k<=log2.N<k+1 .... k=log2.N의 floor

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

부모는 자식보다 작거나 같다
(가장작은값이 루트)

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

Binary Heap

- Example:

6때문에 힙아님!!!!!

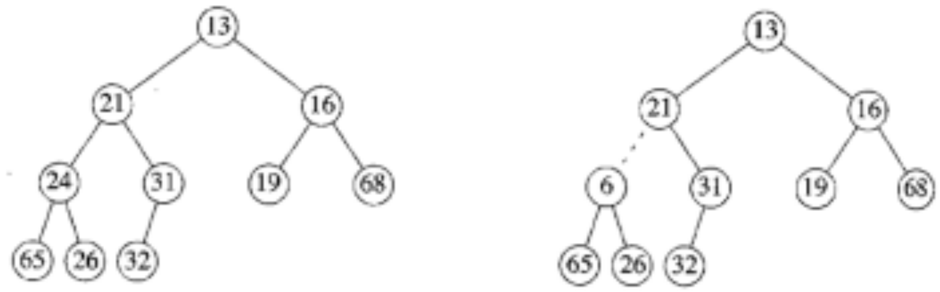


Figure 20.3 Two complete trees (only the left tree is a heap)

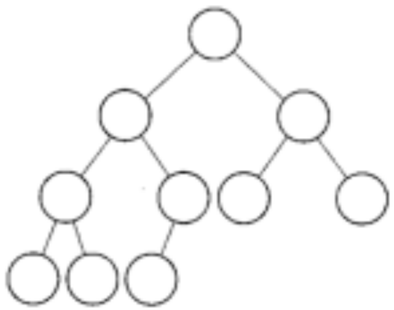


Figure 7.4 An essentially complete binary tree.

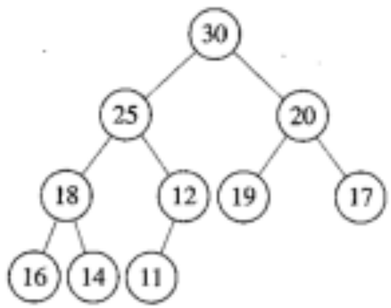
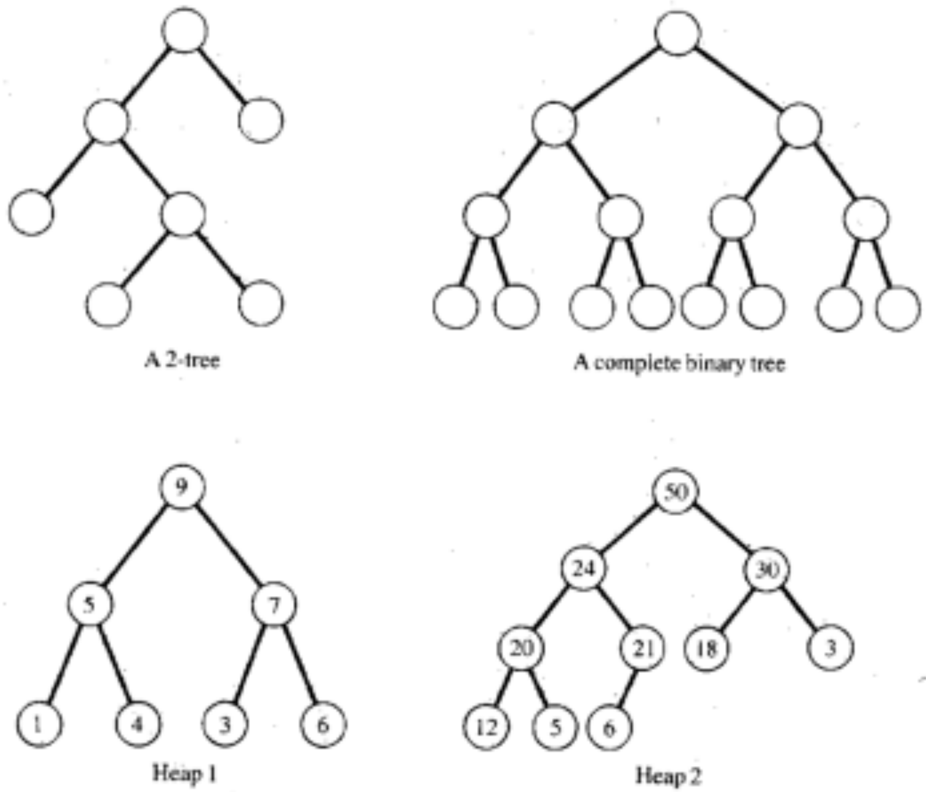


Figure 7.5 A heap.

max힙!!

Binary Heap

- Example:



Binary Heap

루트만 빼고 삽일 할수 있다(?)

❑ Allowed operations in heap

- Insert an element in a heap

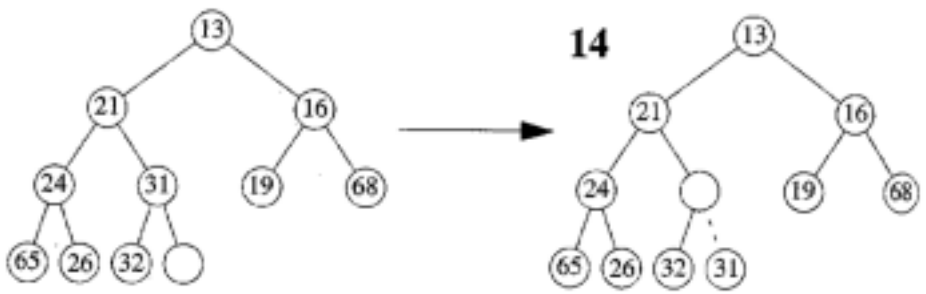


Figure 20.9 Attempt to insert 14, creating the hole and bubbling the hole up

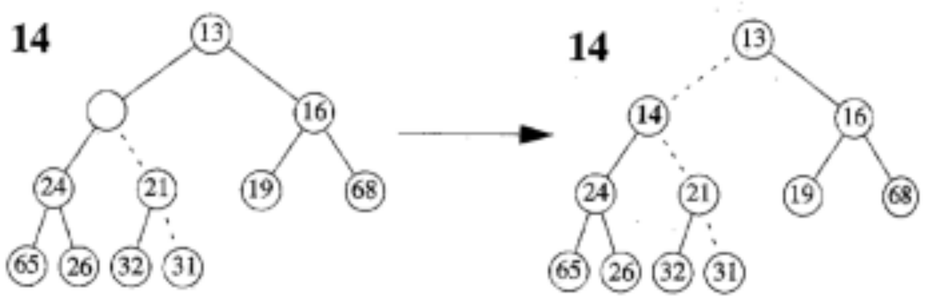


Figure 20.10 The remaining two steps to insert 14 in previous heap

(10 삽입시)

10

13

16 68

19

Binary Heap

- Delete a minimum element from a heap

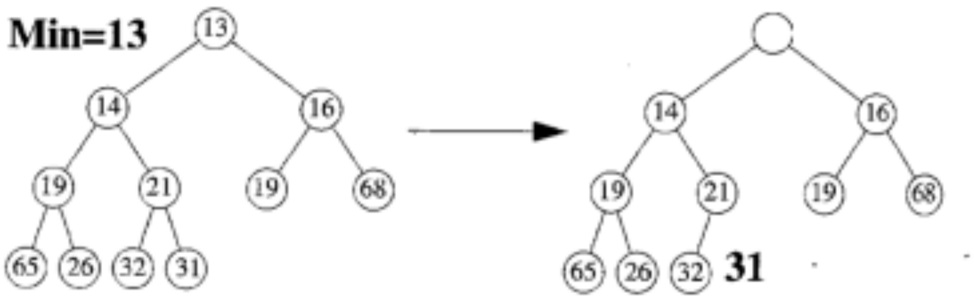


Figure 20.13 Creation of the hole at the root

31을 루투에 올린후 정렬

Binary Heap

- Delete a minimum element from a heap

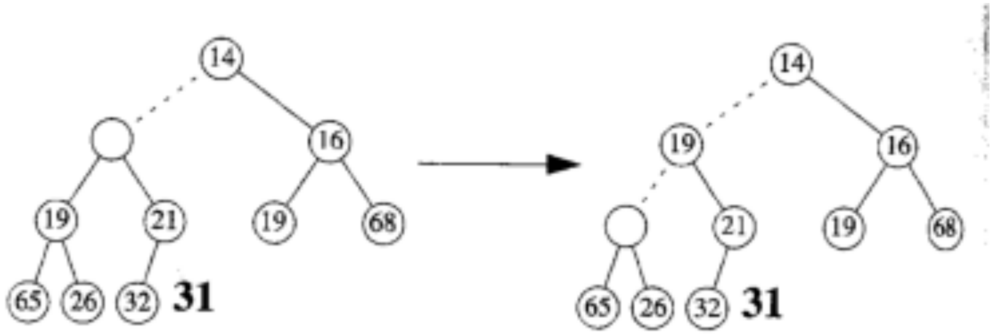
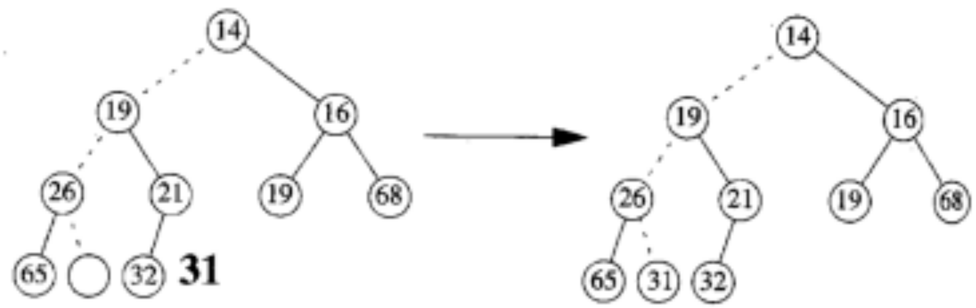


Figure 20.14 Next two steps in `DeleteMin`



delete min =14
16
19
32 68

Figure 20.15 Last two steps in `DeleteMin`

Bina

- Delete a maximum element from a heap.

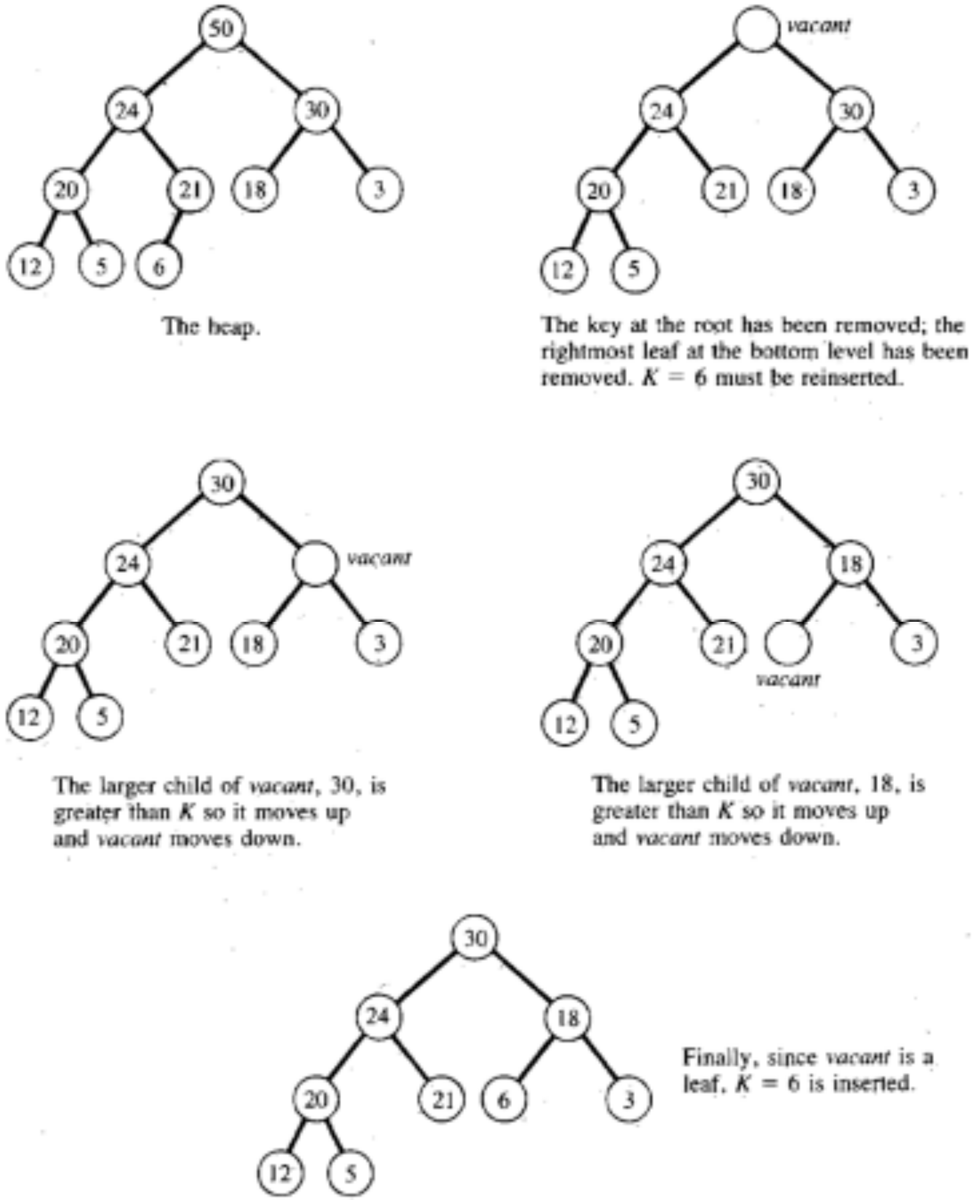


Figure 2.15 Deleting the key at the root and reestablishing the heap property.

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:
    - We are given a complete tree that only the root violates the heap order property.

트리 -> 힙트리로 만들고 -> 소팅

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

fix heap = shift down operation

Heap Construction

- Shift down operation
  - shiftdown() in the text book is the same as FixHeap() .

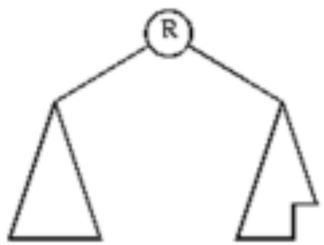
```
void siftdown (heap& H)                        // H starts out having the
{                                              // heap property for all
   node parent, largerchild;                   // nodes except the root.
                                               // H ends up a heap.
   parent = 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;
   }
}
```

자기 위치를 찾아감... 한번 뿐아니라 계속 내려갈수있어!!!!!!! 조건에 맞을 때까지.

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 }
```



Recursive view of the heap

1 째방법:
류트 기준으로 왼,오 리컬시브하게
힙으로 만들고
마지막에 픽스힙
(픽스힙은 하나하나 비교하면서
밑으로 보냄)

2 째방법:
자식노드를 갖는 마지막 노드의 수 k
floor(k/2)부터 루트까지 while돌리면
힙 가능

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:

루트2= 래벨별로-1개 ,
래벨안에있는 각자의 노드-1개

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 }

마지막에서 인덱스를 하나씩 줄여서 픽스힙하면 루프 1개로 가능.

Heap Construction

- Iterative version of Heap construction:

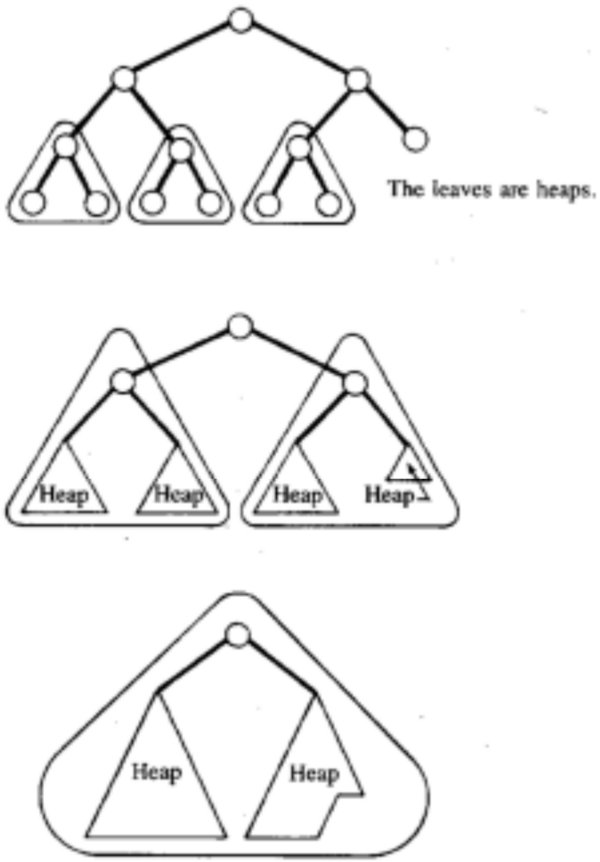


Figure 2.16 Constructing the heap. (*FixHeap* is called for each circled subtree.)

Heap Construction

❑ Example

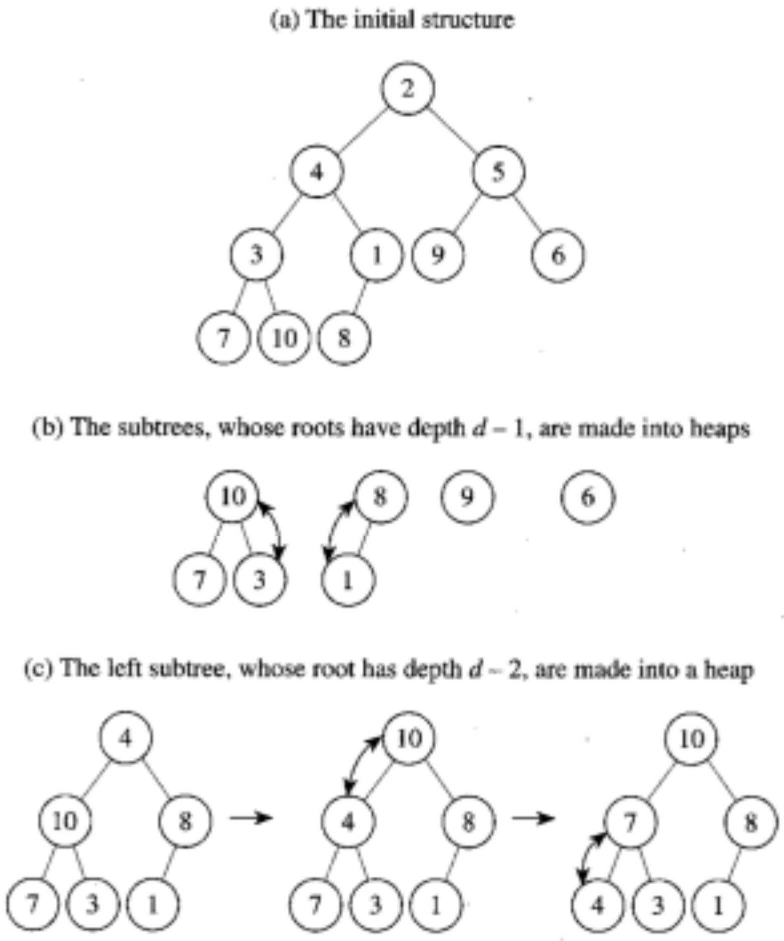


Figure 7.7 Using *siftdown* to make a heap from an essentially complete binary tree. After the steps shown, the right subtree, whose root has depth d − 2, must be made into a heap, and finally the entire tree must be made into a heap.

Heap Construction

- Example

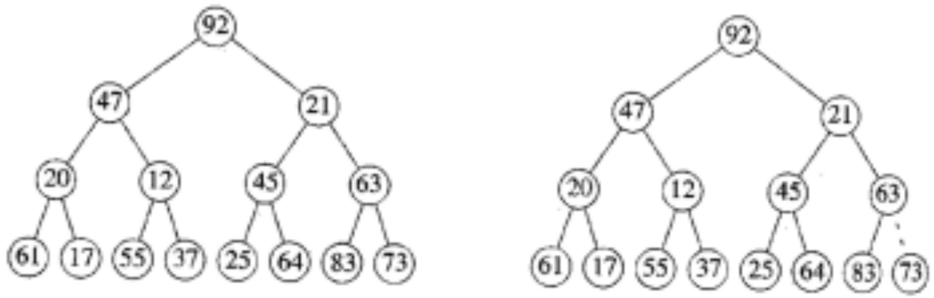


Figure 20.20 Initial heap (left); after `PercolateDown(7)` (right)

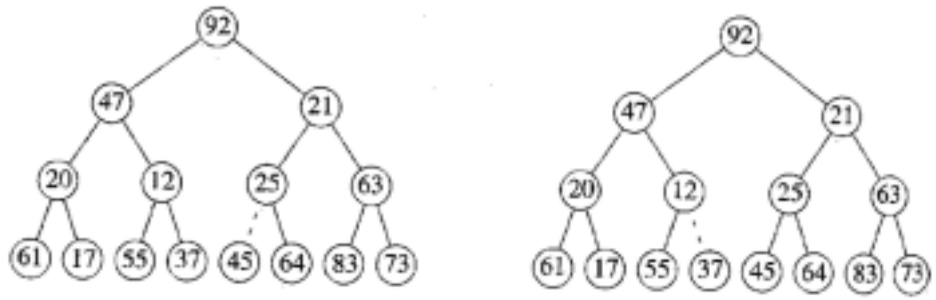


Figure 20.21 After `PercolateDown(6)` (left); after `PercolateDown(5)` (right)

Heap Construction XXXXXXXXXXXXXXXXXXXXXXXXXx

❑ Example

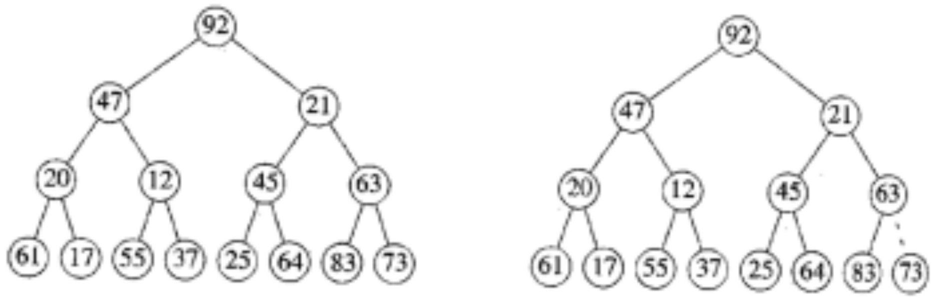


Figure 20.20 Initial heap (left); after `PercolateDown(7)` (right)

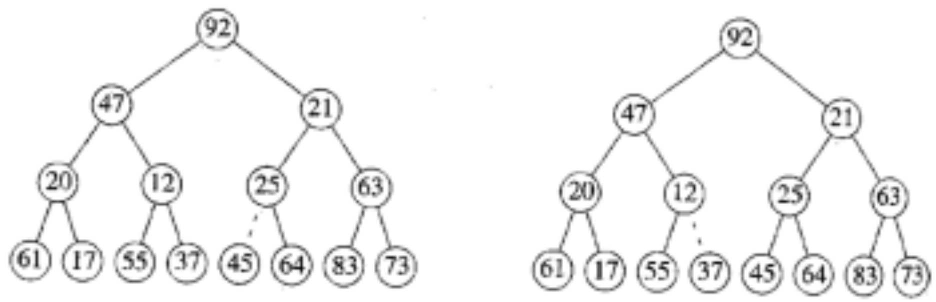


Figure 20.21 After `PercolateDown(6)` (left); after `PercolateDown(5)` (right)

Heap Construction

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

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

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

insert 보다 컴플릿트 트리를 construct 가 더 빠름
nlogn n

밑에 삽입되고 위로 올라가야되니깐
한래벨에 내려갈것이 별로 없어....

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.

방법1)delete 민해민서 다른 배열에 넣어서 쏘팅.....(결국 배열 두개가 필요)
방법2) delete 맥스해서 다른 배열에 넣지말고 마지막을 루트로 올렸으므로 마지막에 넣어...

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 }
```

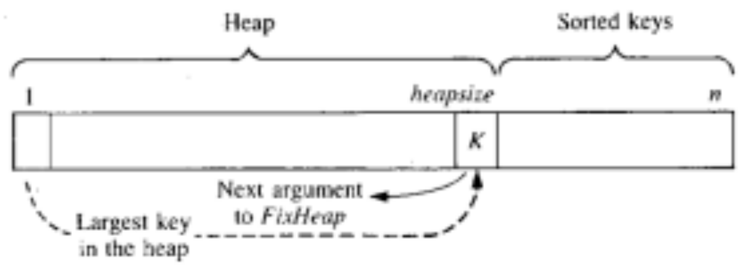


Figure 2.18 The heap and sorted keys in the array.

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.

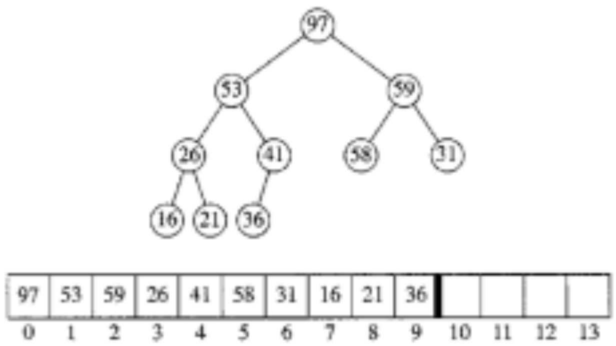


Figure 20.28 (Max) Heap after `FixHeap` phase

Heap Sorting

- Save the array space:

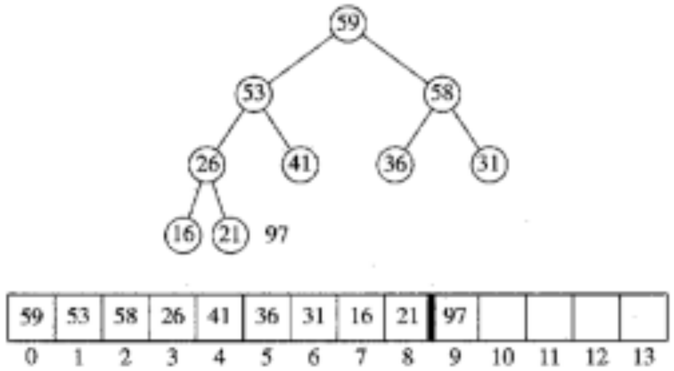
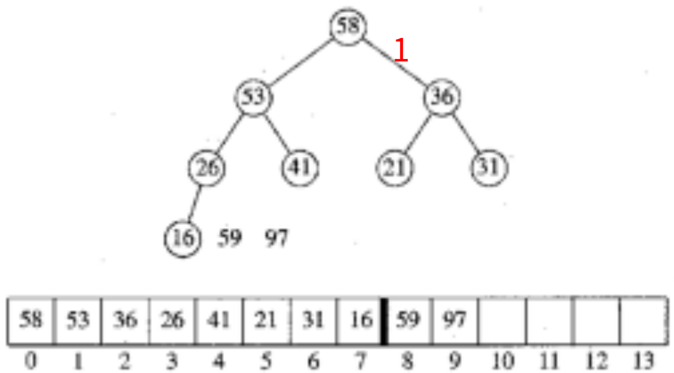
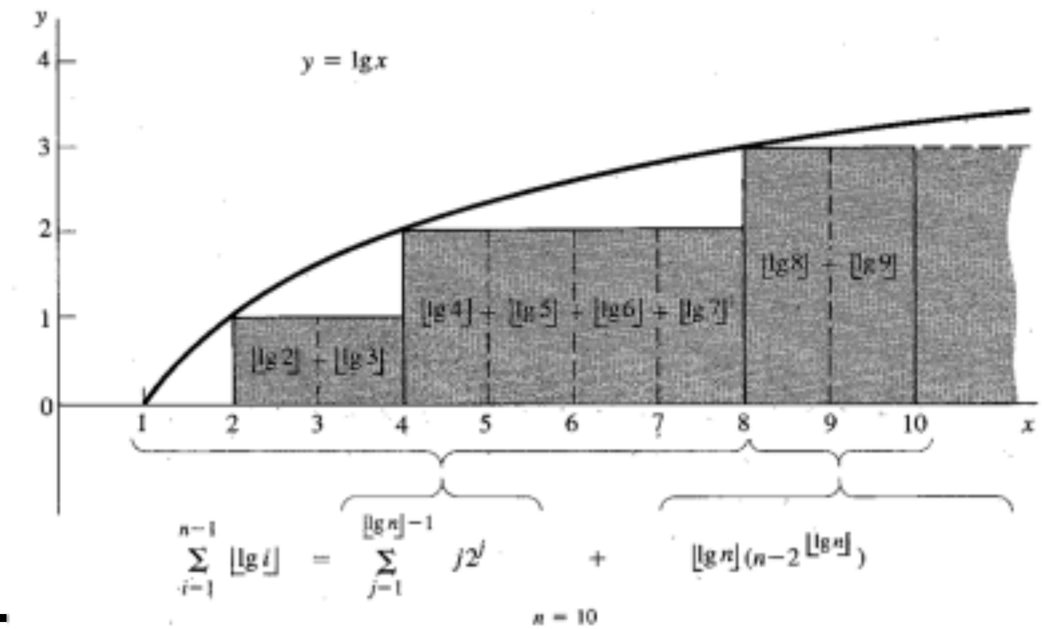


Figure 20.29 Heap after first `DeleteMax`



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

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

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

힙소팅
배열 하나로 소팅한다. 배열을 맥스힙으로 -> 맥스 지우기-> 맨뒤로 넣고 -> 맨 뒤를 루트로 하고 픽스힙 -> 반복
(오름차순으로 소팅됨)

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;
}
```

2번 비교해서 3개 소팅??? => 3 개 -> 3! = 리프노드가 최소6개가 된다...

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.

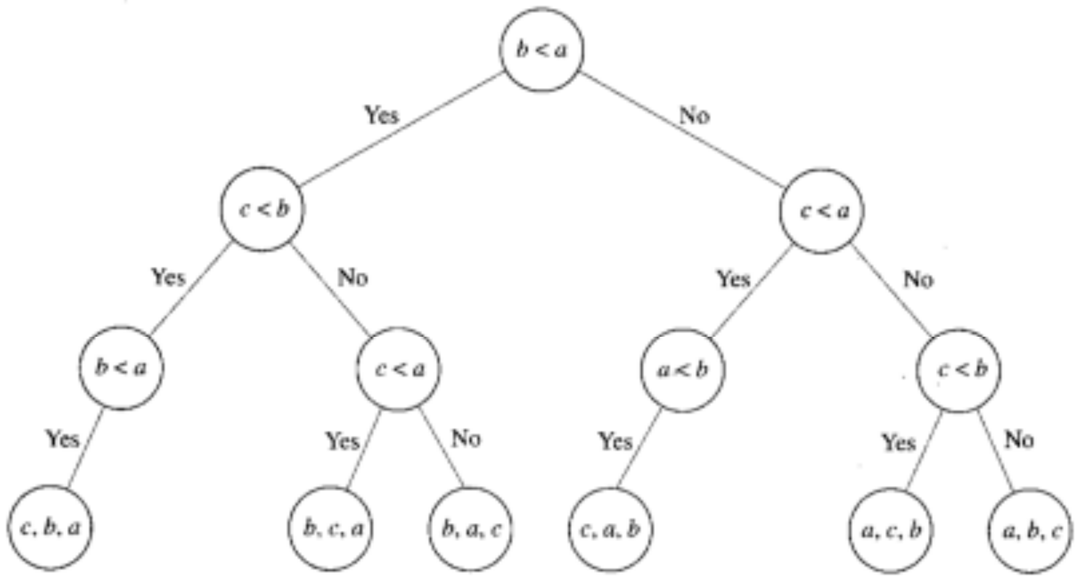


Figure 7.12 The decision tree corresponding to Exchange Sort when sorting three keys.

Lower Bounds for Sorting

- **Theorem 7.2**
  - 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

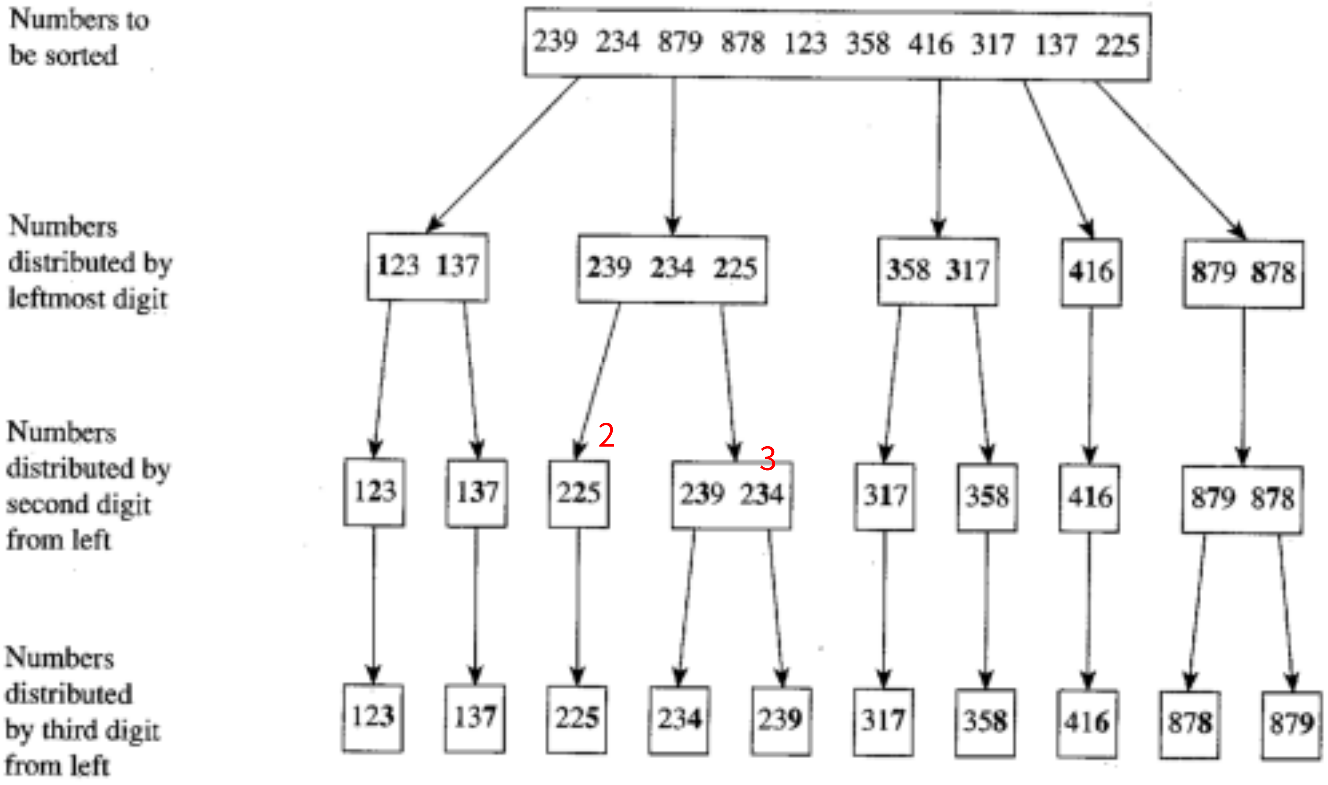
$$
\begin{aligned}
\lg(n!) &= \lg[n(n-1)(n-1)\cdots(2)1] \\
&= \sum_{i=1}^{n} \lg i \\
&\geq \int_1^n \lg x\,dx = \frac{1}{\ln 2}(n\ln n - n + 1) \\
&\geq n\lg n - 1.45n
\end{aligned}
$$

비교를 안하고 소팅!!!!! radix sort : 자리수를 비교하여....(숫자뿐아니랑 무늬+숫자같은 것도 소팅할수 있다.)

Sorting by Distribution (Radix Sort)

❑ Method 1

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



맨 앞자리 같은 것끼리 분류...

Sorting by Distribution (Radix Sort)

❑ Method 2

낮은 자리부터 하는 것이 더욱 굳. (버켓을 줄일수 있따.)

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