Chap 7. The Sorting Problem

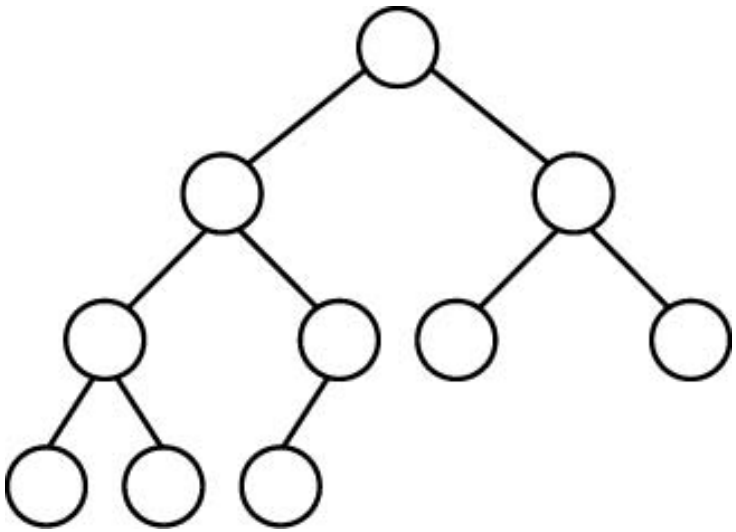
6. Heapsort
9. Sorting by Distribution (Radix Sort)

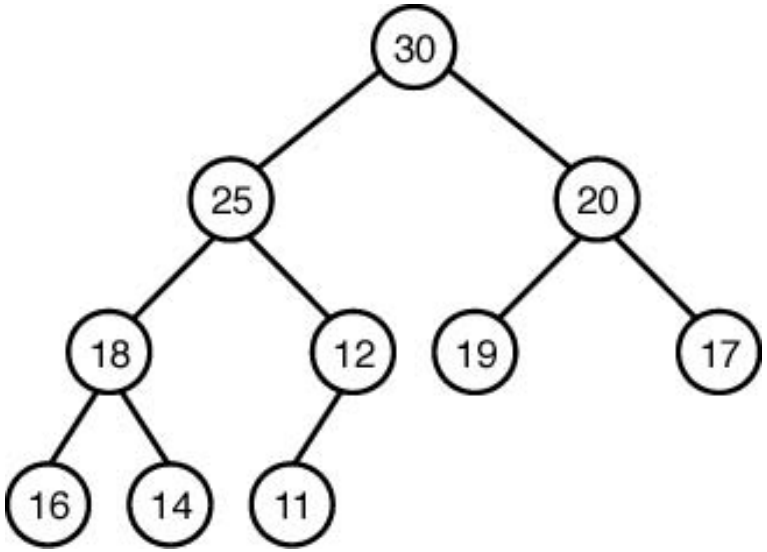
Heapsort

- 완전이진트리 (complete binary tree)
  - 트리의 내부에 있는 모든 마디에 두 개씩 자식마디가 있는 이진트리
  - 따라서 모든 잎의 깊이(depth) d는 동일하다.
- 실질적인 완전이진트리 (essentially complete binary tree)
  - 깊이 d - 1까지는 완전이진트리이고,
  - 깊이 d의 마디는 왼쪽 끝에서부터 채워진 이진트리.
- 힙의 성질 (heap property)
  - 어떤 마디에 저장된 값은 그 마디의 자식마디에 저장된 값보다 크거나 같다.
- 힙 (heap)
  - 힙의 성질을 만족하는 실질적인 완전이진트리

Heapsort

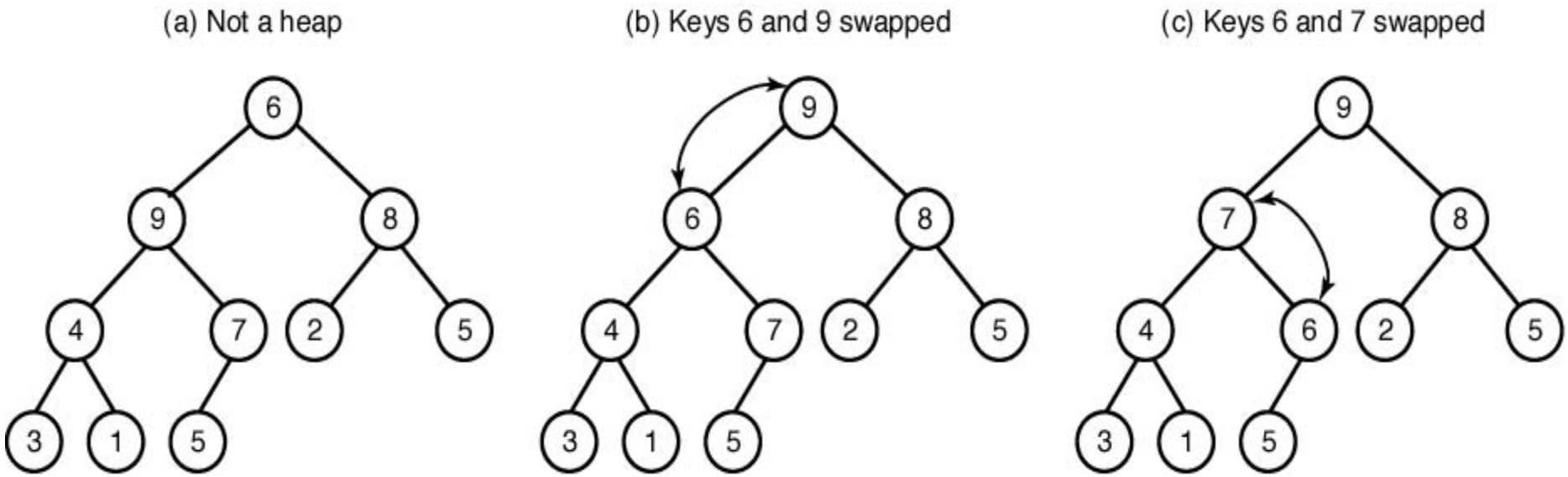
- 보기





Heapsort

- 보기: siftdown



Heapsort

- 알고리즘: **siftdown**

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

Heapsort

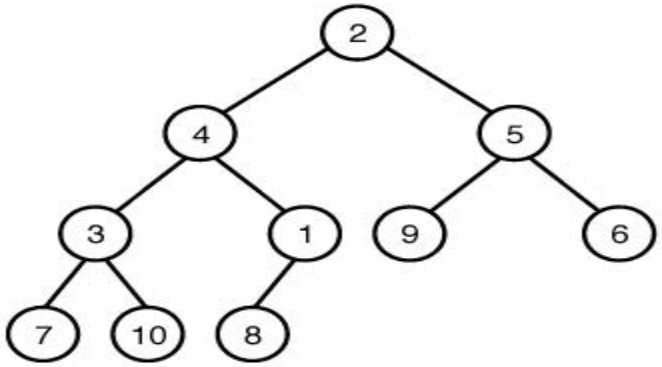
```
void heapsort(int n, heap H, keytype S[]){
        makeheap(n,H);
        removekeys(n,H,S);
}
```

```
void makeheap(int n, heap& H){
        index i;
        heap Hsub;
        for(i=d-1; i>=0; i--)
                for(all subtree Hsub whose roots have depth i)
                        siftdown(Hsub);
}
```

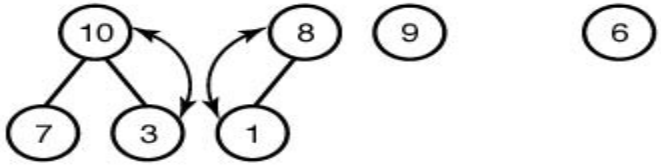
Heapsort

- 보기: makeheap

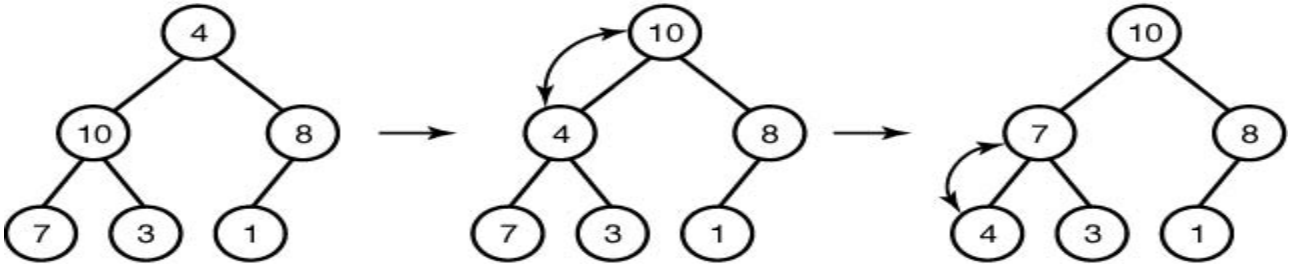
(a) The initial structure



(b) The subtrees, whose roots have depth $d-1$, are made into heaps.



(c) The left subtree, whose root has depth $d-2$, are made into a heap.



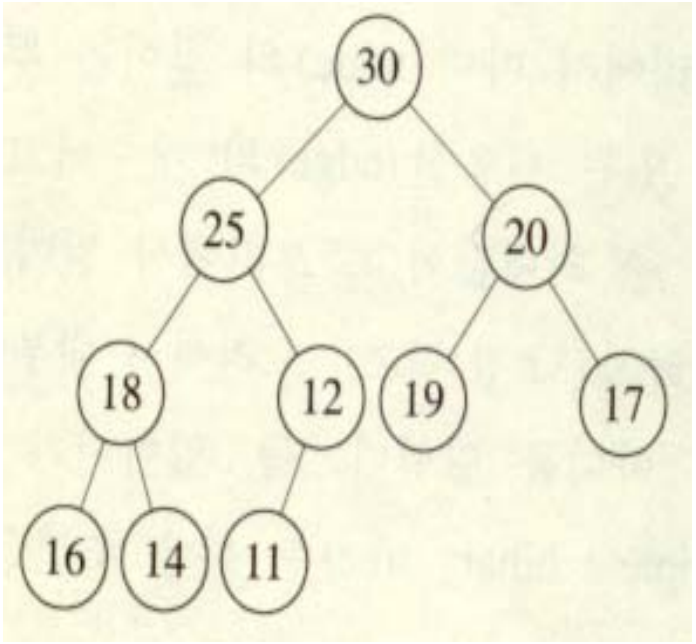
Heapsort

```
void removekeys(int n, heap H, keytype S[]){
        index i;
        for(i=n; i>=1; i--)
                S[i] = root(H);
}
```

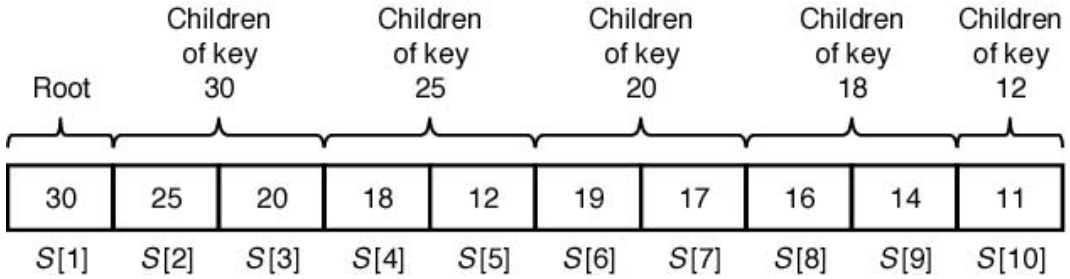
```
keytype root(heap& H) {
        keytype keyout;
        keyout = key at the root;
        move the key at the bottom node to the root;
        delete the bottom node;
        siftdown(H);
        return keyout;
}
```

Heap Data Structure

- 실질적인 완전이진트리 ⇒ 배열
  - 어떤 마디의 왼쪽 자식마디의 인덱스는 그 마디의 인덱스의 두 배
  - 어떤 마디의 오른쪽 자식마디의 인덱스는 그 마디의 인덱스의 두 배보다 1만큼 크다.
  - 어떤 마디의 부모마디의 인덱스는 그 마디의 인덱스의 ½의 floor



⇒



Heap Data Structure

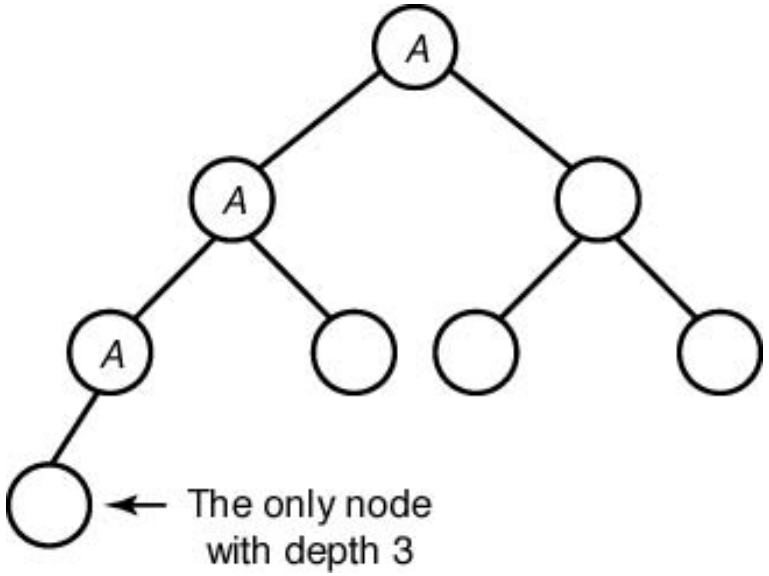
```
struct heap{
    keytype  S[1..n];
    int heapsize;
}
void siftdown(heap& H, index i){
    index parent, largerchild; keytype siftkey; bool spotfound;
    siftkey = H.S[i];      parent = i;      spotfound = false;
    while( (2*parent <= H.heapsize) && !spotfound){
          if(2*parent < H.heapsize && H.S[2*parent]<H.S[2*parent+1])
                    largerchild = 2*parent + 1;
          else  largerchild = 2*parent;
          if(siftkey < H.S[largerchild]) {
                    H.S[parent] = H.S[largerchild];
                    parent = largerchild;
          }
          else  spotfound = true;
    }
    H.S[parent] = siftkey;
}
```

Heap Data Structure

```
void makeheap (int n, heap& H){   index i;

      H.heapsize = n;
      for (i=floor(n/2) ; i>= 1; i--)  siftdown(H,i);
}
keytype root (heap& H){     keytype keyout;

      keyout = H.S[1];   H.S[1] = H.S[heapsize];
      H.heapsize --;
      siftdown(H,1);
      return keyout;
}
void removekeys (int n, heap& H, keytype S[]){   index  i;
      for (i=n; i>=1; i--)    S[i] = root(H)
}

void heapsort(int n, heap& H){
        makeheap(n,H);
        removekeys(n,H,H.S);
}
```

Heapsort

- 힙정렬 알고리즘의 공간복잡도
  - 힙을 배열로 구현한 경우에는 제자리정렬 알고리즘이다.
  - 공간복잡도: $\Theta(1)$
- 최악의 경우 시간복잡도 분석
  - 단위연산: siftdown 프로시저에서의 키의 비교
  - 입력크기: n, 총 키의 개수
  - makeheap의 분석: $n = 2^k$라 가정
    - d를 실질적인 완전이진트리의 깊이라고 하면, $d = \lg n$.
    - 이때 d의 깊이를 가진 마디는 하나이고 그 마디는 d개의 조상(ancestor)을 가진다.
    - 일단 깊이가 d인 마디가 없다고 가정하고 키가 sift되는 상한값(upper bound)을 구해 보자.



Heapsort

depth	node수	키가sift되는 최대횟수
0	2^0	$d-1$
1	2^1	$d-2$
2	s^2	$d-3$
:	:	:
j	2^j	$d-j-1$
:	:	:
$d-2$	2^{d-2}	1
$d-1$	2^{d-1}	0

Heapsort

- 따라서 키가 sift되는 총 횟수는 $\sum_{j=0}^{d-1} 2^j (d-j-1)$를 넘지 않는다.
- 이를 아래 식들을 이용하여 계산하면,

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$\sum_{i=1}^{n} i2^i = (n-1)2^{n+1} + 2$$

$$\sum_{j=0}^{d-1} 2^j (d-j-1) = (d-1)\sum_{j=0}^{d-1} 2^j - \sum_{j=0}^{d-1} j2^j = 2^d - d - 1$$

- 깊이가 d인 마디의 sift될 횟수의 상한값인 d를 더하면,

$$2^d - d - 1 + d = 2^d - 1 = n - 1$$

- 그런데 한번 sift될 때마다 2번씩 비교하므로
  실제 비교횟수는 $2(n\text{-}1)$이 된다.

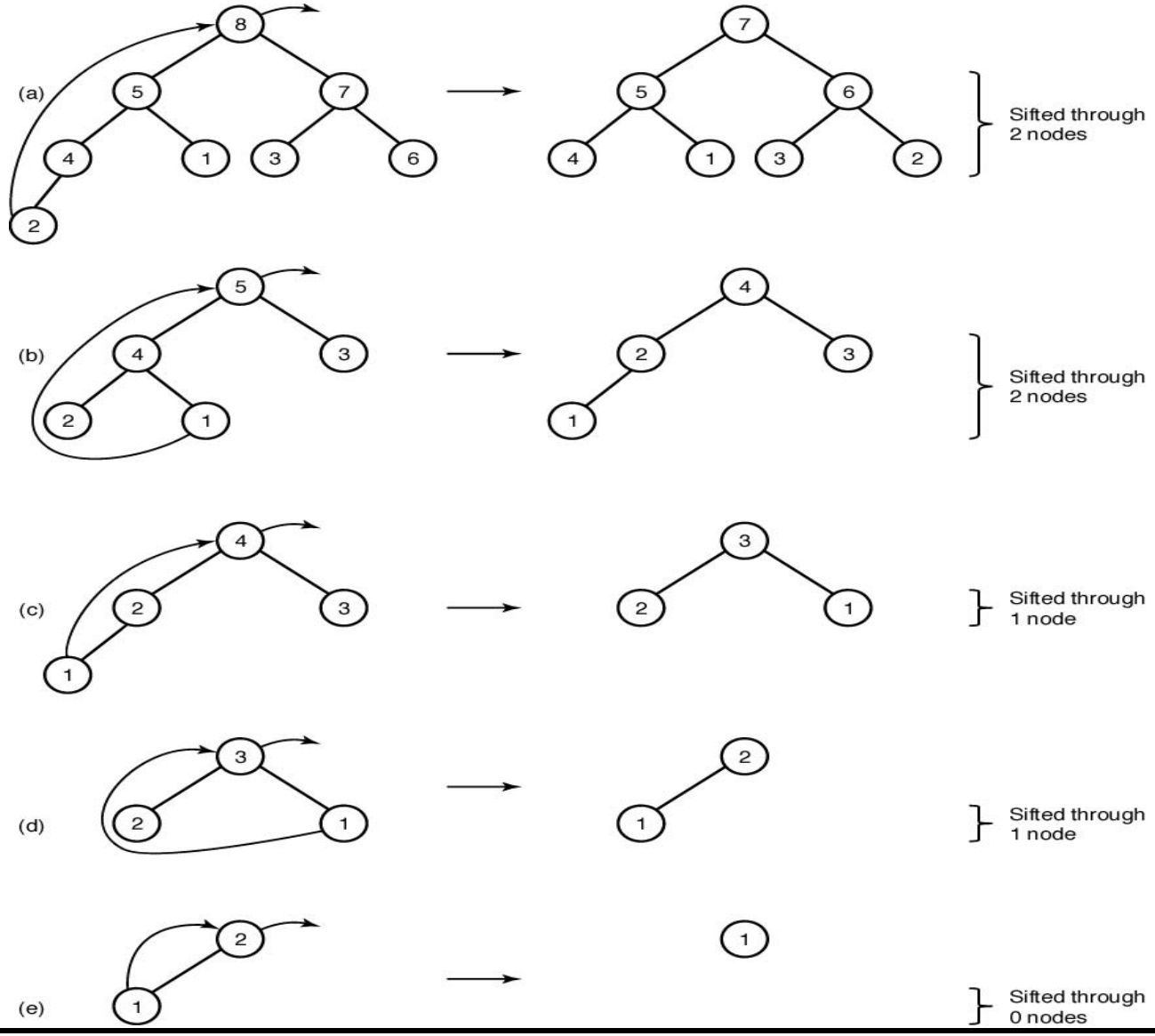
Heapsort

- removekeys의 분석: $n = 2^k$라 가정
  - 먼저 $n = 8$이고 $d = \lg 8 = 3$인 경우를 생각해 보자.
  - Depth 2인 4개의 키를 제거하는데 sift되는 횟수가 2회, Depth 1인 2개의 키를 제거하는데 sift되는 횟수가 1회, 그리고 마지막 키를 제거 하는데 sift횟수는 0회이다.
  - 따라서 총 sift횟수는 $1(2) + 2(4) = \sum_{j=1}^{3-1} j2^j$가 된다.
  - 일반적인 경우는

$$\sum_{j=0}^{d-1} j2^j = (d-2)2^d + 2 = d \cdot 2^d - 2 \cdot 2^d + 2 = n\lg n - 2n + 2$$

  - 그런데 한번 sift될 때마다 2번씩 비교하므로 실제 비교횟수는 $2n\lg n - 4n + 4$ 가 된다.

Heapsort: Removing the keys from a heap



Heapsort

- 두 분석의 통합:
  - 키를 비교하는 총 횟수는 n이 2^k일 때
    $2(n\text{-}1) + 2n \lg n - 4n + 4 = 2(n \lg n - 2n + 1) \approx 2n \lg n$ 을 넘지 않는다.
  - 따라서 최악의 경우 $W(n) \in \Theta(2n \lg n)$.
  - 일반적으로 모든 n에 대해서도 $W(n) \in \Theta(2n \lg n)$이라고 할 수 있다.
    - 왜냐하면 $W(n)$은 결국은 증가함수이기 때문이다.

Comparison of Mergesort, Quicksort, Heapsort

알고리즘	비교횟수	지정횟수	추가저장장소사용량
합병정렬	$W(n) = A(n) = n \lg n$	$T(n) = 2n \lg n$	$\Theta(n)$
빠른정렬	$W(n) = n^2/2$ $A(n) = 1.38n \lg n$	$A(n) = 0.69n \lg n$	제자리정렬
힙정렬	$W(n) = A(n) = 2n \lg n$	$W(n) = A(n) = n \lg n$	제자리정렬

정렬알고리즘의 분류

- **선택하여 정렬하는 부류의 알고리즘**
  - 순서대로 항목을 선택하여 적절한 곳에 위치시키는 알고리즘.
  - 교환정렬, 삽입정렬, 선택정렬, 힙정렬 알고리즘이 이 부류에 속한다.
  - 공간복잡도는 모두 $\Theta(1)$이다. 모두 제자리정렬 알고리즘

- **자리를 바꾸어서 정렬하는 부류의 알고리즘**
  - 인접해 있는 항목을 교환하여서(자리를 바꾸어서) 정렬이 수행되는 알고리즘.
  - 거품정렬(Bubble Sort($\Theta(n^2)$)), 합병정렬, 빠른정렬 알고리즘이 이 부류에 속한다.
  - 거품정렬을 제외하고는 모두 제자리정렬 알고리즘이 아니다.

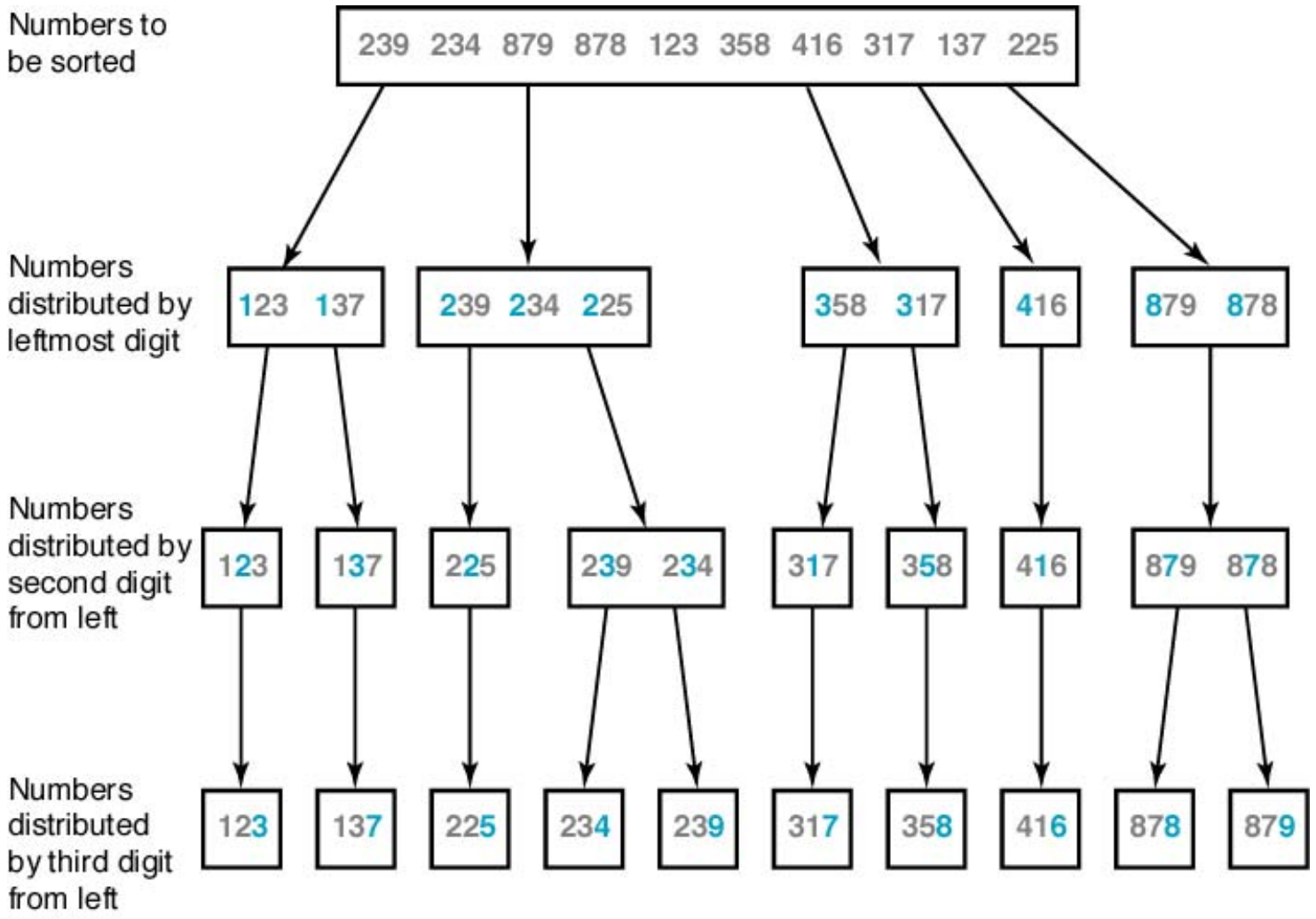
Radix Sort: 분배에 의한 정렬

- 키에 대해서 아무런 정보가 없는 경우에는 키들을 비교하는 것 이외에는 다른 방법이 없으므로 $\Theta(n \lg n)$보다 더 좋은 알고리즘을 만드는 것은 불가능하다.

- 그러나 키에 대한 어느 정도의 정보를 알고 있는 경우에는 달라질 수 있다.
  - 가령 키들이 음이 아닌 수이고 디지트(digit)의 개수가 모두 같다면,
    첫번째 디지트가 같은 수끼리 따로 모으고,
    그 중에서 두 번째 디지트가 같은 수끼리 따로 모으고,
    마지막 디지트가지 이런 식으로 계속 모은다면,
    각 디지트를 한번씩 만 조사를 하면 정렬을 완료할 수 있다.
  - 다음에 예가 제시되어 있는데, 이런 방법으로 정렬하는 것을
    "분배에 의한 정렬(sorting by distribution)" - 기수정렬(radix sort) -
    이라고 한다.

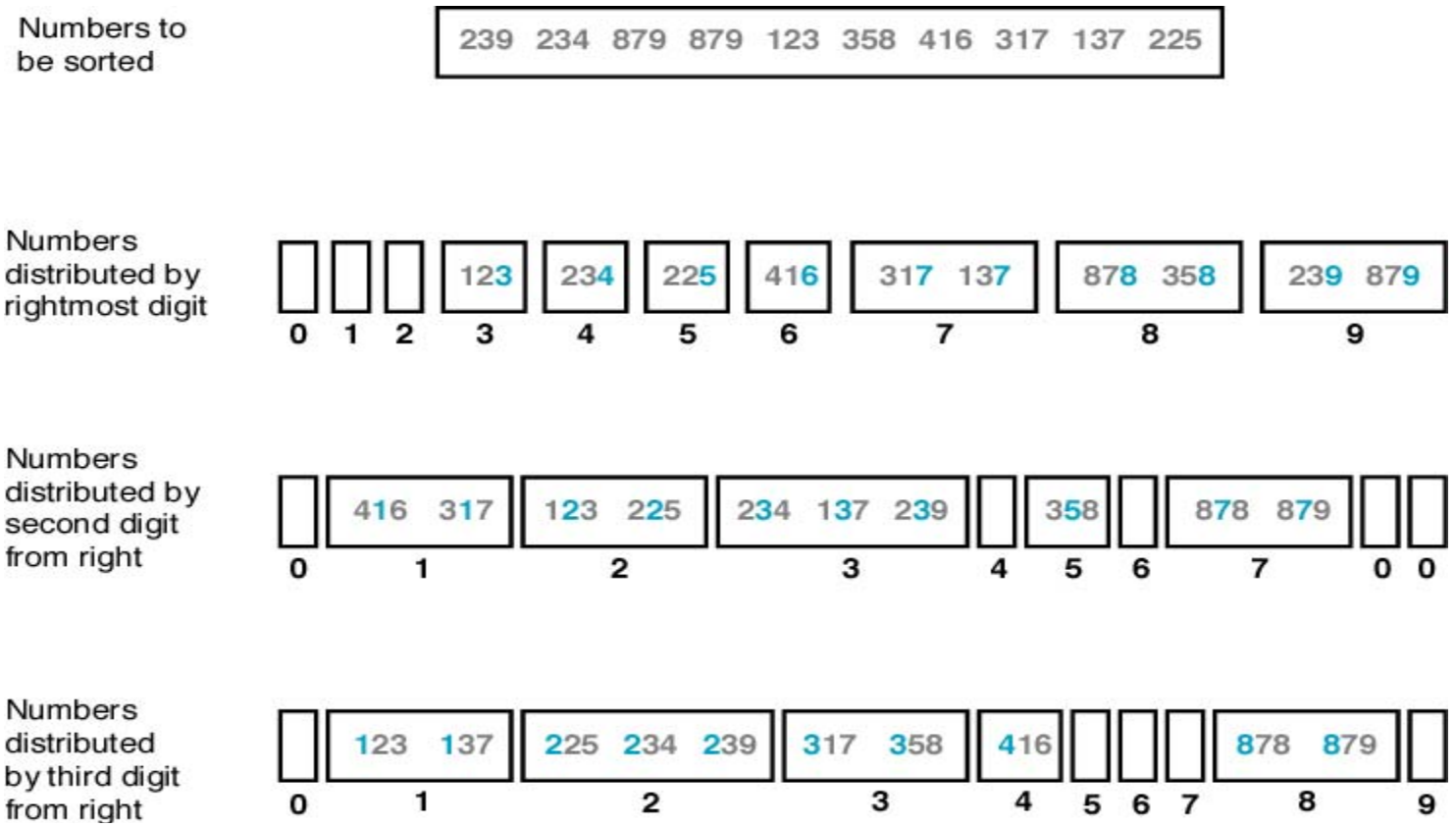
Radix Sort: 분배에 의한 정렬

- 보기 - 기수정렬



Radix Sort: 분배에 의한 정렬

- 기수정렬 알고리즘
  - 이와 같이 정렬하는 경우 항상 뭉치(pile)를 구성하는 개수가 일정하지 않으므로 관리하기가 쉽지 않다. 이를 해결하기 위해서는 다음 예와 같이 끝에 있는 디지트부터 먼저 조사를 시작하면 된다.



Radix Sort: 분배에 의한 정렬

- 기수정렬 알고리즘의 분석
  - 단위연산: 뭉치에 수를 추가하는 연산
  - 입력크기: 정렬하는 정수의 개수 = n,
    각 정수를 이루는 디지트의 최대 갯수 = $numdigits$
  - 모든 경우 시간복잡도 분석:

$$T(n) = numdigits(n+10) \in \Theta(numdigits \times n)$$

    - 따라서 $numdigits$가 n과 같으면, 시간복잡도는 $\Theta(n^2)$가 된다.
    - 그러나 일반적으로 서로 다른 n개의 수가 있을 때
      그것을 표현하는데 필요한 디지트의 수는 $\lg n$으로 볼 수 있다.
    - Ex) 주민등록번호는 13개의 디지트로 되어 있는데,
      표현할 수 있는 개수는 10,000,000,000,000개 이다.
      - 이 10조개의 번호를 기수정렬하는데 걸리는 시간은
        $10{,}000{,}000{,}000{,}000 \times \log_{10} 10{,}000{,}000{,}000{,}000 = 130$조
  - 공간복잡도 분석
    - 추가적으로 필요한 공간은 키를 연결된 리스트로 만드는데 필요한 공간,
      즉, $\Theta(n)$