# 제 12 강의 . 정렬 알고리즘

<학습 목차>

1. 버블 정렬(bubble sort)
2. 삽입 정렬(insertion sort)
3. 퀵 정렬(quick sort)
4. 힙 정렬(heap sort)
5. 정렬 요약

<학습 가이드>

**자료구조와 알고리즘** 분야는 밀접한 관계에 있다. 자료구조에 따라 알고리즘의 효율이 바뀐다. 트리나 그래프에서 이러한 예를 많이 살펴보았다. 자료구조를 좀 복잡하게 가져가면 알고리즘의 효율성을 높이는 예들이 많이 있다.

정렬은 알고리즘의 시작이다. 정렬은 컴퓨터가 발달하면서 가장 많이 연구되고 개발되어온 알고리즘 분야이다.

이 장에서는 기본적인 정렬 알고리즘인 **버블 정렬**, **삽입 정렬**을 살펴보고 좀 더 효율적인 정렬 방법인 **퀵정렬**과 **힙정렬**을 살펴본다. 버블과 삽입정렬은 평균 수행시간의 복잡도가 **$O(n^2)$**이고 퀵정렬과 힙정렬은 평균 수행시간의 복잡도가 **$O(n \log n)$**이다.

데이터가 100개인 경우 어느 정렬프로그램이나 비슷한 시간이 걸린다. 그러나 데이터가 1,000,000개 있다고 가정해보자 알고리즘의 평균 수행복잡도가 **$O(n^2)$**일 경우 1,000,000,000,000번의 비교를 예상할 수 있다. 복잡도가 **$O(n \log n)$**일 경우 1,000,000 *20번으로 줄어들 수 있다.

## ◐ 정렬(Sorting) ◑

데이터를 컴퓨터에 저장하는 이유는 저장 후 사용을 하기 위해서이다. 즉 필요한 데이터를 검색하기 위해서 이다. 그런데 이 리스트의 데이터를 아무렇게나 입력된 순서대로 둔다면 검색 시간이 어떻게 될까?

컴퓨터에서 다루는 데이터의 수는 100, 1000개일 수도 있지만 1백 만개, 1억 개인 경우가 훨씬 더 많다. 이 데이터를 이름순서나, 주민등록번호순으로 정리해 둔다면 찾는 과정이 훨씬 쉬워진다. 정렬은 데이터 검색을 빠르게 해준다.

정렬이 일어나는 장소에 따라 2가지로 구분한다.

**내부정렬(internal sorting)** :

데이터의 크기가 주 기억장소 용량보다 적을 경우 기억장소를 활용하여 정렬하는 방법

=> 버블정렬(bubble sort), 삽입정렬(insertion sort), 선택정렬(selection sort), 퀵정렬(quick sort), 쉘정렬(shell sort), 힙정렬(heap sort)

**외부정렬(external sorting)** :

데이터의 크기가 주기억장소의 용량보다 클 경우 외부 기억장치(디스크, 테이프 등)를 사용하여 정렬하는 방법

=> 머지 정렬(merge sort)

## ◐ 정렬(Sorting) ◑

우리가 알고있는 선택정렬 알고리즘을 살펴보자.

0. 선택정렬

어떤 정렬 알고리즘이 빠른지는 시간을 측정해보면 알수있지만 컴퓨터 성능, 데이터 구성에 따라 시간이 달라질수있기 때문에 적절하지 않을 수 있다.

다른 방법으로 연산의 개수를 세보는 방법이 있을 수 있다.

선택정렬의 경우 비교, 교환이 주 연산인데 비교, 교환의 연산 수를 계산해보자.

비교 : n(n-1)/2, 교환 : n-1

앞으로 나올 다른 정렬 알고리즘도 비교, 교환의 연산수를 계산해보도록 한다.

## 1. 버블정렬(bubble sort) – 교환정렬

**(버블정렬)** : 버블 정렬은 정렬이 진행되는 모양이 비누거품(bubble)과 같다고 하여 붙여진 이름이다. 나란히 있는 두개의 데이터를 계속하여 바꾸어 나간다.

**(과정)**

**1단계** : list[i]와 list[i+1]를 i = 0,1,2, …, n-2에 대하여 비교하여 만약 뒤 데이터가 값이 더 작으면 바꾼다
(swap list[i]와 list[i+1])
이 과정을 거치면 가장 큰 값이 맨 뒤로 이동한다.

**2단계** : list[i]와 list[i+1]를 i = 0,1,2, …, n-3에 대하여 비교하여 만약 뒤 데이터가 값이 더 작으면 바꾼다
(swap list[i]와 list[i+1])
이 과정을 거치면 두 번째 큰 값이 뒤에서 두 번째에 위치한다.

**3단계** : list[i]와 list[i+1]를 i = 0,1,2, …, n-4에 대하여 비교하여 만약 뒤 데이터가 값이 더 작으면 바꾼다
(swap list[i]와 list[i+1])
이 과정을 거치면 세 번째 큰 값이 뒤에서 세 번째에 위치한다.

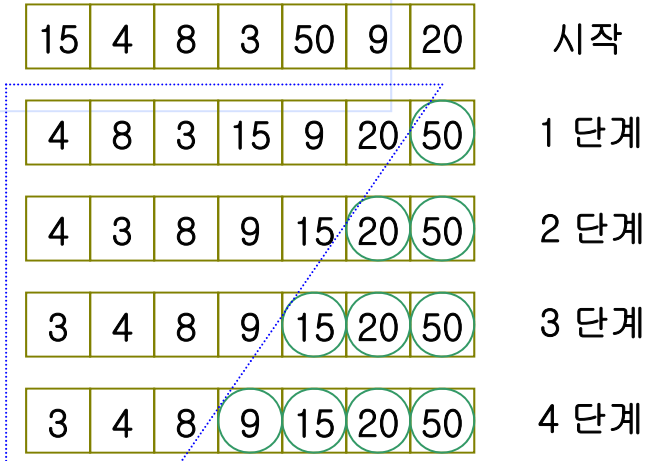
…

**n-1단계** : list[i]와 list[i+1]를 i = 0에 대하여 비교하여 만약 뒤 데이터가 값이 더 작으면 바꾼다
(swap list[i]와 list[i+1])
이 과정을 거치면 n-1번째 큰 값이 뒤에서 n-1번째에 위치한다.

```
/* list에 대한 기본 버블정렬 알고리즘 */
void bubble_sort(element list[], int n) {
        int i, j;
        element next;
        for(i = n-1; i > 0; i--) {          /*1*/
             for(j = 0; j < i; j++) {       /*2*/
                  if(list[j] > list[j + 1] { /*3*/
                       swap(list[j], list[j + 1]);
                  }
             }
        }
}
```

버블정렬 알고리즘

◐ 버블 정렬의 예와 각 단계 진행 과정



## (버블정렬 프로그램의 분석)

버블정렬 프로그램의 각 단계별 첨자의 변화는 다음과 같다.
또 프로그램의 if 문에서 비교를 하게 되며 비교횟수(필요에 따라 교환)는 아래와 같다.

| (단계) | (첨자 변화) | (비교 횟수) |
|---|---|---|
| 1단계 : | i=n-1 : j=0,1,2,...,n-4,n-3,n-2 | (n-1)번 |
| 2단계 : | i=n-2 : j=0,1,2,...,n-4,n-3 | (n-2)번 |
| 3단계 : | i=n-3 : j=0,1,2,...,n-4 | (n-3)번 |
| ... | ... | |
| n-2단계 : | i=2 : j=1,0 | 2번 |
| n-1단계 : | i=1 : j=0 | 1번 |

따라서 정렬을 하기위한 전체 비교횟수 **T(n) = n(n-1)/2** 이다.
비교 횟수로 따지면 수행시간 복잡도는 O( n(n-1)/2 ) = **$O(n^2)$**이다.
n에 관한 2차 함수로 프로그램 수행 시간이 걸린다.

**(참고)** - 개선된 버블정렬

버블정렬 각 단계에서 데이터의 이동이 일어나지 않으면 다음 단계로 진행할 필요가 없다. 즉 중간과정에서 정렬이 끝나면 더 이상 비교와 교환이 필요 없다. (변화가 있는지 "**flag**" 변수를 사용하여 점검한다.)

개선된 버블정렬 알고리즘

```
/* list에 대한 개선된 버블정렬 알고리즘 */
void bubble_sort(element list[], int n) {
        int i, j;
        int flag = 1;
        element next;
        for(i = n - 1; flag > 0; i--) {    /*1*/
                flag = 0;
                for(j = 0; j < i; j++) {                    /*2*/
                        if(list[j] > list[j + 1] {
                                swap(list[j], list[j + 1]);
                                flag = 1;                   /*3*/
                        }
                }
        }
}
```

## 2. 삽입정렬(Insertion Sort)

**(삽입정렬)** : j = 1,...,n-1에 대하여 각 단계에서 list[j]를 앞 방향으로 비교해가면서 교환해 나간다. 더 작은 값이 나오면 멈춘다. 처음 j=1,2,3...순서대로 진행을 하기 때문에 j번째 진행될 때는 j-1개의 앞쪽 데이터는 정렬이 끝난 상태이다. list[I]에 대하여 앞으로 진행하며 값을 찾아가는 과정이 된다.
- list[j]가 앞으로 이동하면서 list[j]보다 큰 데이터를 한 칸씩 뒤로 이동

**(과정)**

**1단계** : list[1]을 list[0]과 비교하여 만약 뒤(list[1]) 데이터가 값이 더 작으면 바꾼다
(swap list[2]와 list[1])
이 과정을 거치면 list[1], list[2]는 정렬된 상태가 된다.

**2단계** : list[2]을 list[1], list[0]과 순서대로 비교하여 만약 뒤 데이터가 값이 더 작으면 바꾼다
(swap list[i]와 list[i+1])
이 과정을 거치면 list[i], i = 0,1,2는 정렬된 상태가 된다.

**3단계** : list[3]을 list[i], i = 2,1,0 순서대로 비교하여 만약 뒤 데이터가 값이 더 작으면 바꾼다
(swap list[i]와 list[i+1])
이 과정을 거치면 list[i], i = 0,1,2,3은 정렬된 상태가 된다.

...

**n-1단계** : list[n-1]을 list[i], i = n-2,n-3,...,2,1,0 순서대로 비교하여 만약 뒤 데이터가 값이 더 작으면 바꾼다(swap list[i]와 list[i+1])
이 과정을 거치면 list[i], i = 0,1,2,3,...,n-1은 정렬된 상태가 된다.

삽입정렬

```
/* 삽입정렬 – 데이터를 앞으로 이동하면서 끼워넣는다 */

void insertion_sort(element list[], int n) {
        int i, j;
        element next;
        for(i = 1; i < n; i++) {                        /*1*/
                next = list[i];
                for(j = i - 1; j >= 0 &&
                        next.key < list[j].key; j--)    /*2*/
                        { list[j + 1] = list[j];};
                list[j + 1] = next;
        }
}
```

예 1) 삽입정렬의 진행 – n = 5, input sequence: (5, 4, 3, 2, 1)

| i | [0] | [1] | [2] | [3] | [4] | |
|---|---|---|---|---|---|---|
| – | 5 | 4 | 3 | 2 | 1 | 0단계 : 초기데이터 |
| 1 | 4 | 5 | 3 | 2 | 1 | 1단계 : 데이터 4를 앞으로 이동 |
| 2 | 3 | 4 | 5 | 2 | 1 | 2단계 : 데이터 3을 앞으로 이동 |
| 3 | 2 | 3 | 4 | 5 | 1 | 3단계 : 데이터 2를 앞으로 이동 |
| 4 | 1 | 2 | 3 | 4 | 5 | 4단계 : 데이터 1을 앞으로 이동 |

예 2) 삽입정렬의 진행 – n = 5, input sequence: (3, 2, 5, 1, 4)

| i | [0] | [1] | [2] | [3] | [4] | |
|---|---|---|---|---|---|---|
| – | 3 | 2 | 5 | 1 | 4 | 0단계 : 초기데이터 |
| 1 | 2 | 3 | 5 | 1 | 4 | 1단계 : 데이터 3를 2와 비교 앞으로 이동 |
| 2 | 2 | 3 | 5 | 1 | 4 | 2단계 : 데이터 5을 3과 비교 이동없음 |
| 3 | 1 | 2 | 3 | 5 | 4 | 3단계 : 데이터 1를 5,3,2순으로 비교이동 |
| 4 | 1 | 2 | 3 | 4 | 5 | 4단계 : 데이터 4을 5와 비교 이동 |

## (삽입정렬 프로그램의 분석)

삽입정렬 프로그램의 각 단계별 첨자의 변화는 다음과 같다.
또 프로그램의 if 문에서 비교를 하게 되며 비교횟수(필요에 따라 교환)는 아래와 같다.

| (단계) | (첨자 변화) | (비교횟수) – 최대의 경우 |
|---|---|---|
| 1단계 : | i=1 : j=0 | 1번 |
| 2단계 : | i=2 : j=1,0, | 2번 |
| 3단계 : | i=3 : j=2,1.0 | 3번 |
| … | … | |
| n-2단계 : | i=n-2 : j=n-3,,…,2,1,0 | n-2번 |
| n-1단계 : | i=n-1 : j=n-2,n-3,…,2,1,0 | n-1번 |

각 단계에서 비교횟수는 비교 값보다 더 작은 값이 나오면 멈추므로 실제 더 작으나 최대를 가정한다.
따라서 정렬을 하기위한 전체 비교횟수 **T(n) = n(n-1)/2** 이다.
비교 횟수로 따지면 수행시간 복잡도는 O(n(n-1)/2 ) = **$O(n^2)$**이다.

## 3. 머지정렬(Merge Sort)

**(머지정렬)**

# Q/A

1. (정렬 과정)

다음 데이터를 정렬할 때 단계별로 데이터의 모습을 보인 것이다.
각 알고리즘에 대하여 2단계에 맞는 데이터 값, 비교횟수, 교환횟수는?
(초기데이터) 5 6 1 4 2 8 3 7

| | | |
|---|---|---|
| (1) 버블정렬 | (초기) | 5 6 1 4 2 8 7 3 |
| | (1단계) | 5 1 4 2 6 7 3 8 |
| | (2단계) | |
| (2) 선택정렬 | (초기) | 5 6 1 4 2 8 7 3 |
| | (1단계) | 5 6 1 4 2 3 7 8 |
| | (2단계) | |
| (3) 삽입정렬 | (초기) | 5 6 1 4 2 8 7 3 |
| | (1단계) | 5 6 1 4 2 8 7 3 |
| | (2단계) | |
| (4) 머지정렬 | (초기) | 5 6 1 4 2 8 7 3 |
| | (1단계) | 5 6 1 4 2 8 3 7 |
| | (2단계) | |

## 3. 퀵정렬(Quick Sort)

퀵 정렬은 정렬 속도가 빠르다 해서 붙여진 이름이다.
그러나 앞의 버블정렬과 선택정렬보다는 속도가 빠르지만 더 빠른 방법들도 있다. (예 힙정렬)

**(방법)**

**과정 1**. 리스트에서 기준데이터(pivot value) 1개를 지정한 다음 리스트의 데이터들을 앞과 뒤 양쪽에서 가운데쪽으로 1개씩 비교하여 기준데이터보다 큰 값을 리스트 앞에서 찾아서 리스트의 뒤쪽으로, 기준데이터보다 작은 값은 리스트의 뒤쪽에서 찾아서 리스트의 앞쪽으로 둔다. 양쪽에서 비교하여 오기 때문에 이동할 데이터를 찾으면 서로 맞 바꾼다. 이 과정이 끝나면 리스트는 자연스럽게 두개로 분리된다

(기준데이터보다 작은 값들, 큰 값들)

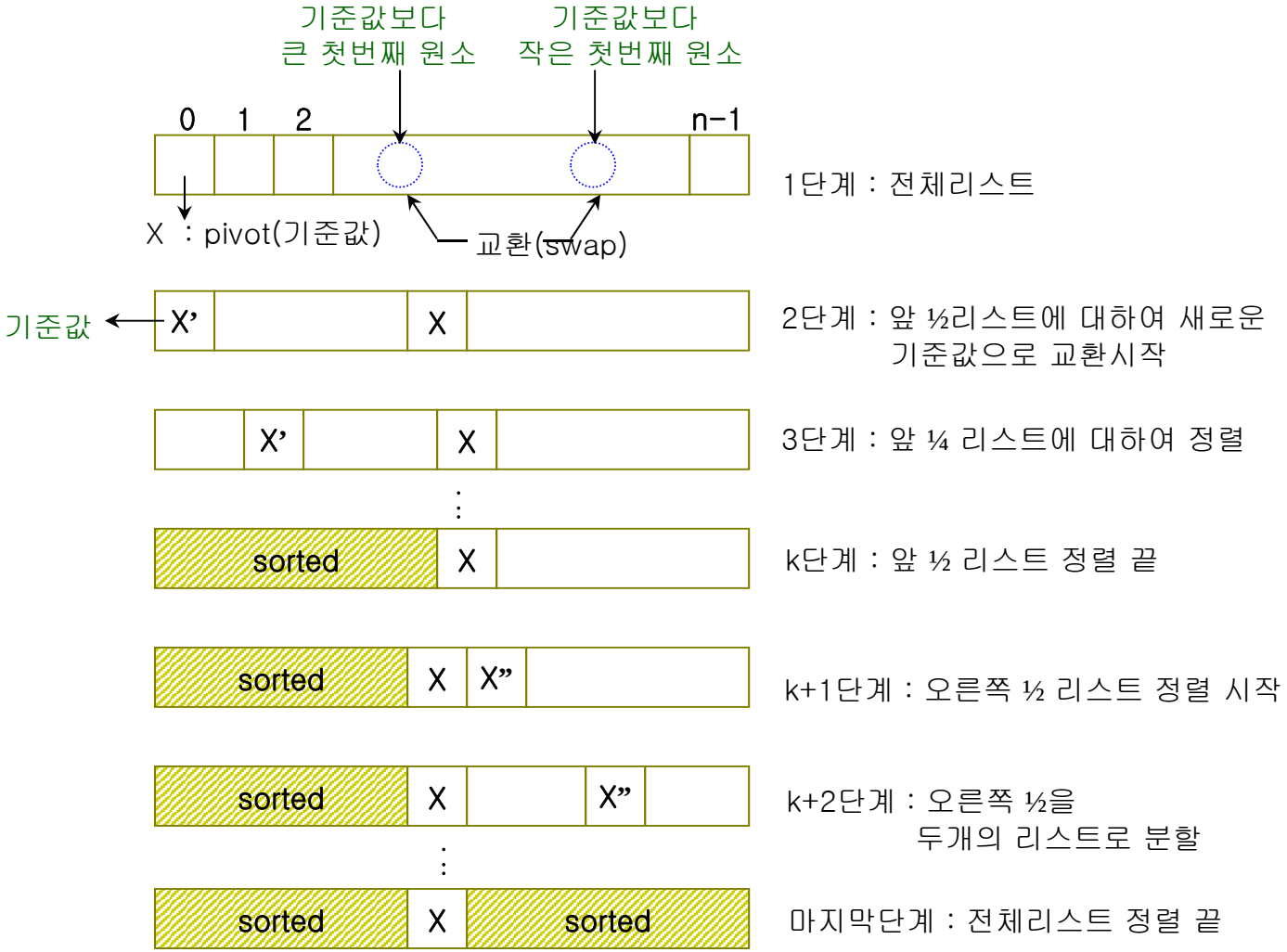
기준데이터는 리스트의 가운데 비교하여 오면서 만나는 위치에 둔다.

**과정 2**. 첫번째 과정이 끝나면 두 개의 리스트(기준값보다 작은 리스트와 큰 리스트)에 대하여 각각 같은 방법으로 첫번째 과정과 같은 방법으로 분리한다. 기준값은 리스트에서 새로 정한다. 과정 2를 반복하면 나중에 데이터개수가 1개 있는 리스트가 남게 되며 이 때는 자동으로 정렬이 끝나게 된다.

**(정리)**

1번 과정이 끝나면 리스트의 평균적으로 크기가 ½로 줄어든다. 매번 데이터는 ½로 줄어든다. 이러한 방법을 분할-정복(divide and conquer) 기법이라고 한다. 즉 문제를 작은 문제로 바꾸어 해결해 나간다.

## 퀵 정렬의 진행 과정


기준값보다
큰 첫번째 원소
기준값보다
작은 첫번째 원소
0
1
2
n-1
X : pivot(기준값)
교환(swap)
기준값
X'
X
X'
X
sorted
X
sorted
X
X"
sorted
X
X"
sorted
X
sorted
1단계 : 전체리스트
2단계 : 앞 ½리스트에 대하여 새로운
기준값으로 교환시작
3단계 : 앞 ¼ 리스트에 대하여 정렬
k단계 : 앞 ½ 리스트 정렬 끝
k+1단계 : 오른쪽 ½ 리스트 정렬 시작
k+2단계 : 오른쪽 ½을
두개의 리스트로 분할
마지막단계 : 전체리스트 정렬 끝

## 예) 퀵 정렬의 예

- 입력리스트 : 10 개 (26,5,37,1,61,11,59,15,48,19)

| $R_0$ | $R_1$ | $R_2$ | $R_3$ | $R_4$ | $R_5$ | $R_6$ | $R_7$ | $R_8$ | $R_9$ | left | right | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 26 | 5 | 37 | 1 | 61 | 11 | 59 | 15 | 48 | 19 | 0 | 9 | 1단계 |
| 11 | 5 | 19 | 1 | 15 | 26 | 59 | 61 | 48 | 37 | 0 | 4 | 2단계 |
| 1 | 5 | 11 | 19 | 15 | 26 | 59 | 61 | 48 | 37 | 0 | 1 | 3단계 |
| 1 | 5 | 11 | 19 | 15 | 26 | 59 | 61 | 48 | 37 | 3 | 4 | 4단계 |
| 1 | 5 | 11 | 15 | 19 | 26 | 59 | 61 | 48 | 37 | 6 | 9 | 5단계 |
| 1 | 5 | 11 | 15 | 19 | 26 | 48 | 37 | 59 | 61 | 6 | 7 | 6단계 |
| 1 | 5 | 11 | 15 | 19 | 26 | 37 | 48 | 59 | 61 | 9 | 9 | 7단계 |
| 1 | 5 | 11 | 15 | 19 | 26 | 37 | 48 | 59 | 61 | | | 8단계 |

- 각 단계마다 기준값은 첫번째 원소로 한다.
- 1단계 - 기준값은 26이고 19와 37, 61과 15가 바뀐다.
  중간 도착점은 R5이므로 R5와 기준값 R1을 교환한다.
  최종 결과는 2단계 시작처럼 된다.
- 5단계 - 오른쪽 ½ 리스트의 정렬을 시작한다.
- 8단계 – 최종 결과이다.

```
/* 퀵 정렬 프로그램 */
void quicksort(element list[], int left, int right)
{
    int pivot, i, j; element temp;
    if(left < right) {                          /*1*/
        i = left; j = right + 1;
        pivot = list[left].key;
        do {                                    /*2*/
            do      i++;
            while(list[i].key < pivot && i<=right); /*3*/
            do      j--;
            while(list[j].key > pivot);     /*4*/
            if(i < j)
                    SWAP(list[i], list[j], temp);
        } while(i < j);
        SWAP(list[left], list[j], temp);
        quicksort(list, left, j - 1);
        quicksort(list, j + 1, right);
    }
}
```

퀵 정렬 프로그램
quicksort.c

## 퀵 정렬의 수행시간 분석(time complexity)

n개의 데이터에 대한 수행 시간을 T(n)이라고 하면 첫번째 단계를 거치면 기준 값(pivot value)을 중심으로 데이터는 n/2개의 리스트가 2개가 생긴다. 이렇게 두 개의 분리된 리스트로 만들기 위해서 데이터 n개에 대하여 n번의 비교를 해야 한다.

즉, n번 비교 + 2개의 ½ * n개의 필요한 정렬시간

$$T(n) \le c \cdot n + 2 \cdot T(n/2)$$

마찬가지로 나머지 데이터에 대하여 계속 반복을 한다면

$$\begin{aligned} T(n) &\le c \cdot n + 2 \cdot T(n/2) \\ &\le c \cdot n + 2(c \cdot n/2 + 2 \cdot T(n/4)) \\ &\le 2 \cdot c \cdot n + 4 \cdot T(n/4) \\ &\cdots \\ &\le c \cdot n \cdot \log_2 n + n \cdot T(1) = O(n \cdot \log_2 n) \end{aligned}$$

데이터의 크기 분포가 고르다면 평균 수행시간은 다음과 같다.

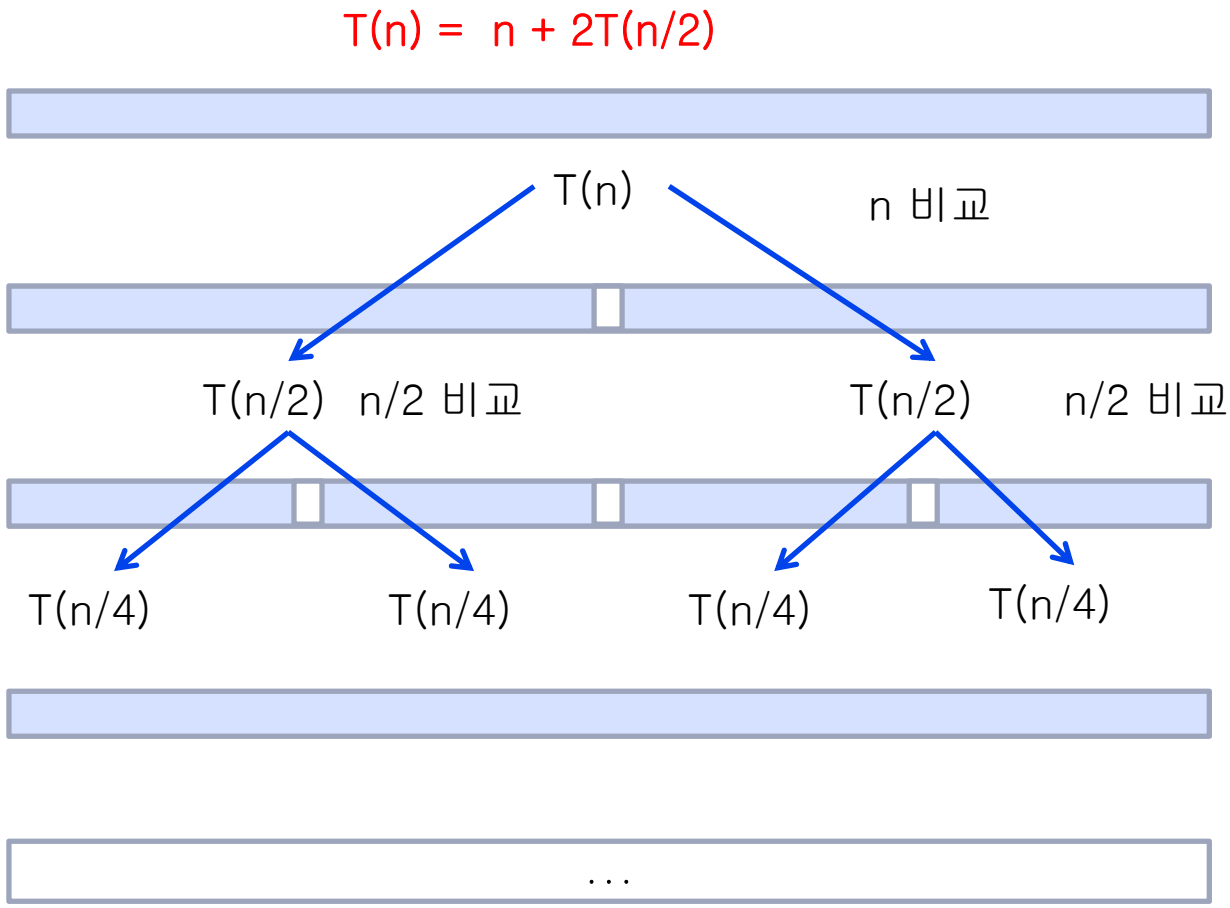
평균 수행시간 : **$O(n \cdot \log_2 n)$**

그러나 데이터분포가 고르지 않다면?

(예를 들어 이미 정렬되어 있다면 기준 값이 한쪽 끝 값을 갖는다.)

- 최악시간은 **$O(n^2)$**

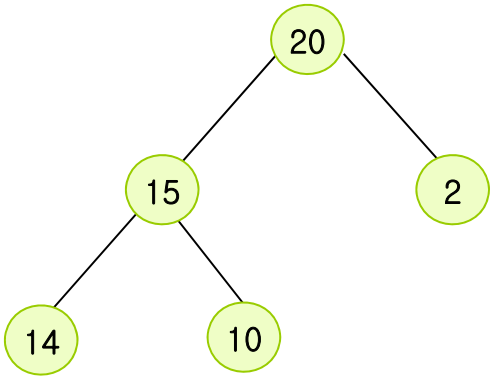
## 퀵 정렬의 수행시간 분석(time complexity) - 그림



## 4. 힙 정렬(Heap Sort)

### (1) 힙(heap)

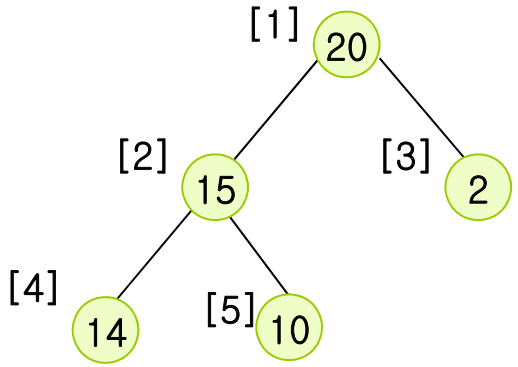
힙(heap)의 정의 : 힙은 트리 중에서 부모노드의 원소 값이 자식노드의 원소 값보다 큰 완전 이진 트리이다.(정확히 말하면 부모노드의 값이 큰 경우 Max Heap 이라고 하고 작은 경우를 Min Heap이라고 한다), 힙 구조에서 가장 큰 값의 위치는 루트에 있으며 가장 작은 값은 리프 노드 중에 있게 된다. 힙은 완전 이진 트리 이므로 트리 구조이지만 배열에 저장하는 것이 더 효율적이다.




힙 구조의 예

```
/* 힙을 위한 자료구조 선언 */

#define MAX_ELEMENTS 200
#define HEAP_FULL(n) (n == MAX_ELEMENTS - 1)
#define HEAP_EMPTY(n) (!n)
typedef struct {
        int key;
        /* other field */
} element;
element heap[MAX_ELEMENTS];
int n = 0;
```




| | |
|---|---|
| [1] | 20 |
| [2] | 15 |
| [3] | 2 |
| [4] | 14 |
| [5] | 10 |
| [6] | – |
| [7] | – |
| [8] | – |
| [9] | – |
| [10] | – |

힙의 예와 저장

(첨자가 1부터 저장 가정)

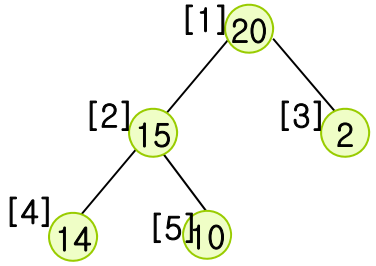
## - 힙에 새로운 노드의 삽입 과정(adjust 과정)

(1) 새로운 노드의 위치를 정한다 ? 마지막 레벨의 마지막 노드 - (b) 참조(2) 삽입할 데이터를 새로운 노드에 놓는다. (3) 새로운 노드와 부모를 비교하여 부모가 더 작으면 바꾸는 과정을 루트에 도달할 때까지 계속한다.

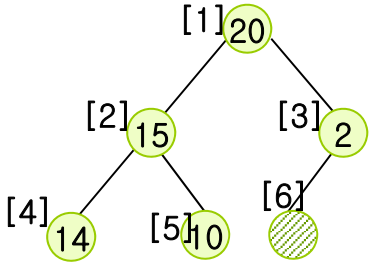
부모 노드의 위치는 i/2에 있다. 수행 시간은 트리의 높이와 같다.

- **새로운 노드 삽입의 수행시간** O(h) : h는 트리의 깊이 = $\log_2 n + 1$

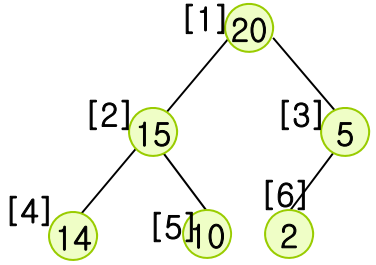
– 힙에 새로운 노드의 삽입 예 (adjust 과정)



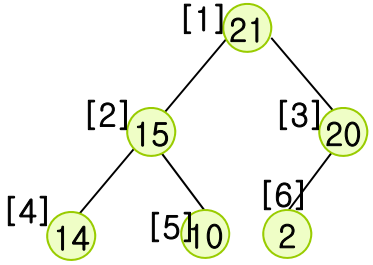

(a) 힙 구조를 가진 트리




(b) 힙에 새로운 노드의 삽입 위치




(c) (a)에 5 삽입,재구성 후의 트리




(d) (a)에 21 삽입,재구성한 후의 트리

## 힙에 데이터를 삽입하는 함수

```
/* 힙에 데이터를 삽입하는 함수 */
void insert_max_heap(element item, int *n) {
        int i;
        if(HEAP_FULL(*n)) {
                fprintf(stderr,"The heap is full. ₩n");
                exit(1);
        }
        i = ++(*n);
        while((i!=1)&&(item.key>heap[i/2].key)) { /*1*/
                heap[i] = heap[i/2];
                i /= 2;
        }
        heap[i] = item;
}
```

## 힙에서 노드의 삭제(deletion from a max heap)

- 힙에서의 삭제는 항상 루트 노드를 삭제한다.(가장 큰 데이터)
- 삭제 후 완전 이진 트리가 되도록 재구성한다.

- 삭제후 힙의 재구성 방법 :

1. 마지막 레벨의 마지막 노드를 루트에 올려 놓는다.
2. 루트노드를 왼쪽 혹은 오른쪽 자식노드와 교환
   (왼쪽과 오른쪽 중 큰 값 - 두 값 중 하나는 현재 트리에서 가장 큰 값이다)한다.
3. 2번 과정을 트리의 밑으로 내려가면서 계속 반복한다.

- 자식 노드의 위치

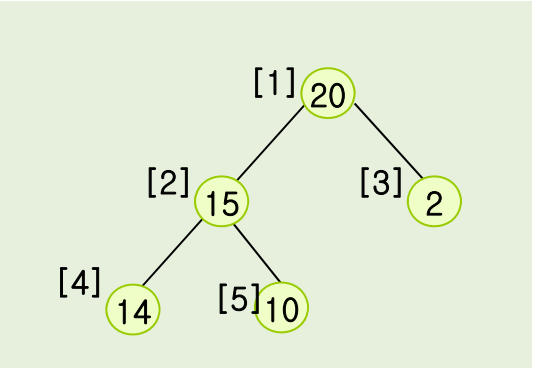
  왼쪽자식의 위치(left_child position) : 2·i

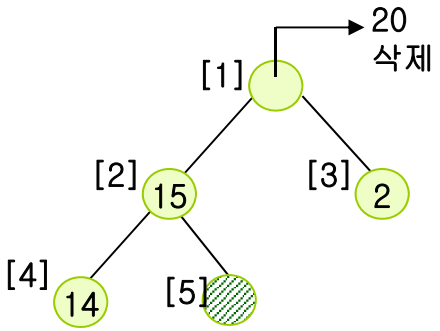
  오른쪽자식의 위치(right_child position) : 2·i + 1

- 삭제 수행시간 :

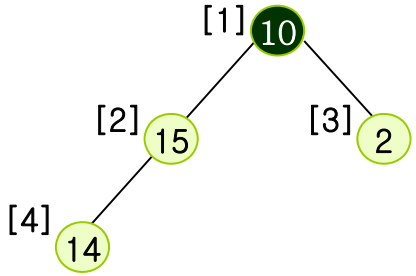
  O(h) : h는 트리의 깊이 = $\log_2 n$ + 1

– 힙에서 노드의 삭제 예(20의 삭제)

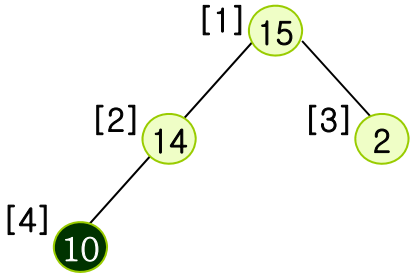



(a) 힙 구조 – 20 삭제




(b) 10을 루트 노드로 이동




(c) 재구성(adjust)한 최종 결과

힙에서 데이터의 삭제 알고리즘

```
/* 힙에서 데이터의 삭제 알고리즘 */
element delete_max_heap(int *n) {
        element item, temp;
        if(HEAP_EMPTY(*n)) {
                fprintf(stderr,"The heap is empty₩n");
                exit(1);
        }
        item = heap[1];
        temp = heap[(*n)--];
        parent = 1; child = 2;
        while(child <= *n) {
                if((child < *n) && (heap[child].key < heap[child+1].key))
                        child++;
                if(temp.key>=heap[child].key) break;
                heap[parent] = heap[child];
                parent = child;
                child *= 2;
        }
        heap[parent] = temp;
        return item;
      }
```

## (2) 힙 정렬 (heapsort)

-힙 구조을 이용하여 정렬을 한다.(Max Heap)

(* 힙정렬은 완전이진트리를 여러 번 다듬어(Adjust) 이쁘게 만든다음(힙구조), 다시 다듬기(Adjust)를 반복하여 정렬하는 방법이다)

1. 리스트를 배열에 저장하여 힙을 만든다
   – Adjust 알고리즘 n/2번 적용하여 완전이진트리 힙 구성
2. 힙에서 n개의 데이터를 삭제하고 힙을 재구성하는 과정을 반복한다.
   (삭제는 항상 루트에서 실행하며 가장 큰 값이 삭제된다)
   - Adjust 알고리즘 n-1번 수행
3. 2번에서 삭제된 노드를 배열에 차례로 저장하면 정렬된 리스트가 된다.
   (실제로는 힙이 저장된 배열에 다시 저장)

- 시간복잡도(time complexity)

  평균수행시간 : **$O(n \cdot \log_2 n)$**
  최악의 경우 : **$O(n \cdot \log_2 n)$**

-재구성(adjust): 이진 트리를 힙으로 만들기 위하여 재구성(adjust) 과정을 거친다.
재구성 시간 : O(h) h : 트리의 깊이

/* 재구성 알고리즘 */

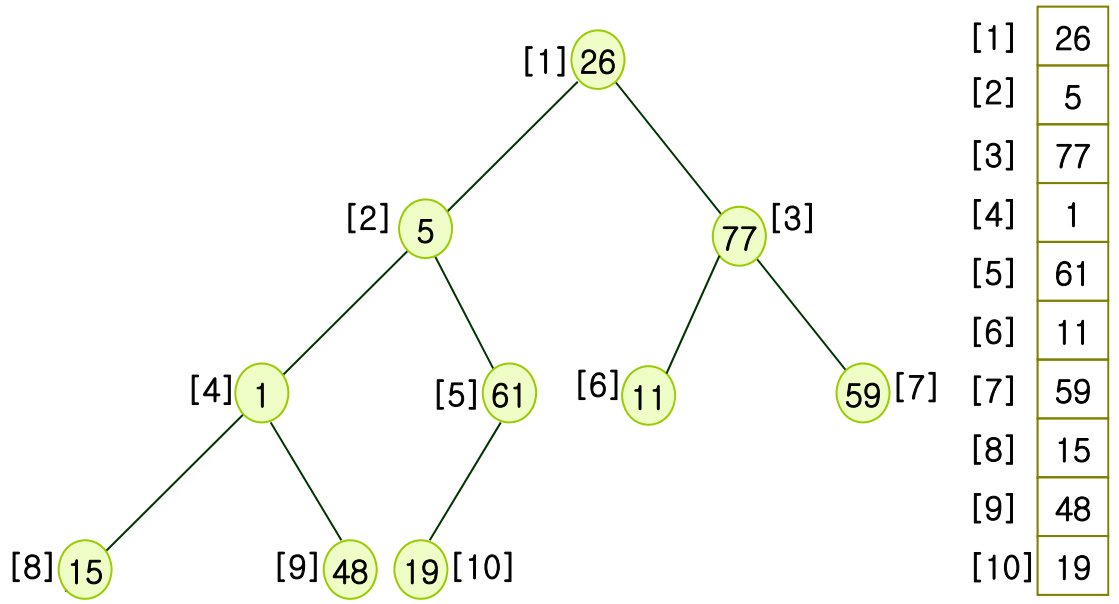
```
void adjust(element list[], int root, int n) {
        int child, rootkey; element temp;
        temp = list[root];
        rootkey = list[root].key;
        child = 2 * root; /* left child */
        while(child <= n) {
                if((child < n) &&
                        (list[child].key < list[child+1].key))
                        child++;
                if(rootkey > list[child].key) break;
                else {
                        list[child/2] = list[child];
                        child *= 2;
                }
        }
        list[child/2] = temp;
}
```

/* 힙 정렬 */

```
/* 힙 정렬 */
void heapsort(element list[], int n) {
        int i, j;
        element temp;
        for(i = n / 2; i > 0; i--) /*1*/
                adjust (list, i, n);
        for(i = n - 1; i > 0; i--) { /*2*/
                SWAP(list[1], list[i + 1], temp);
                adjust (list, 1, i);
        }
}
```

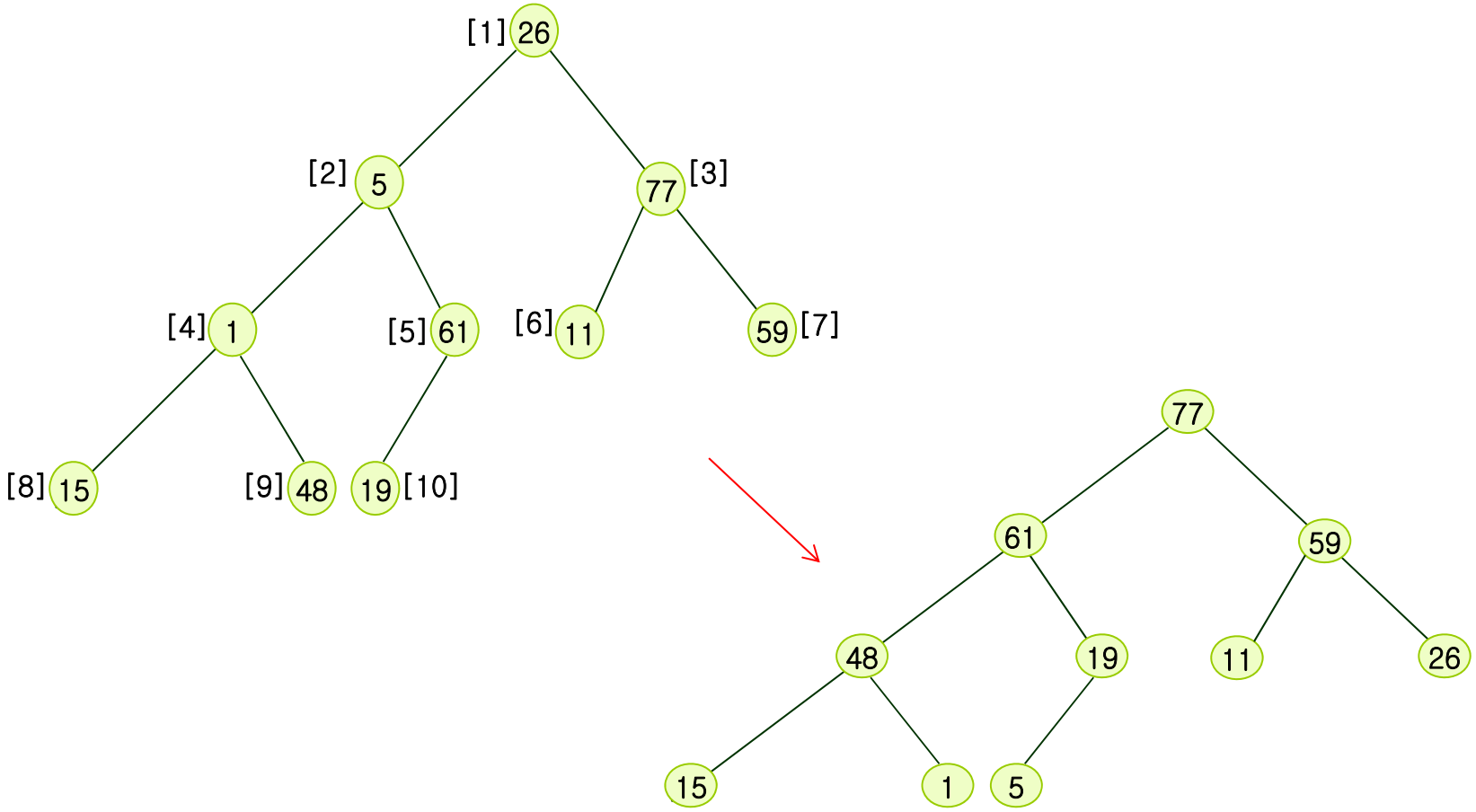
힙 정렬의 예)

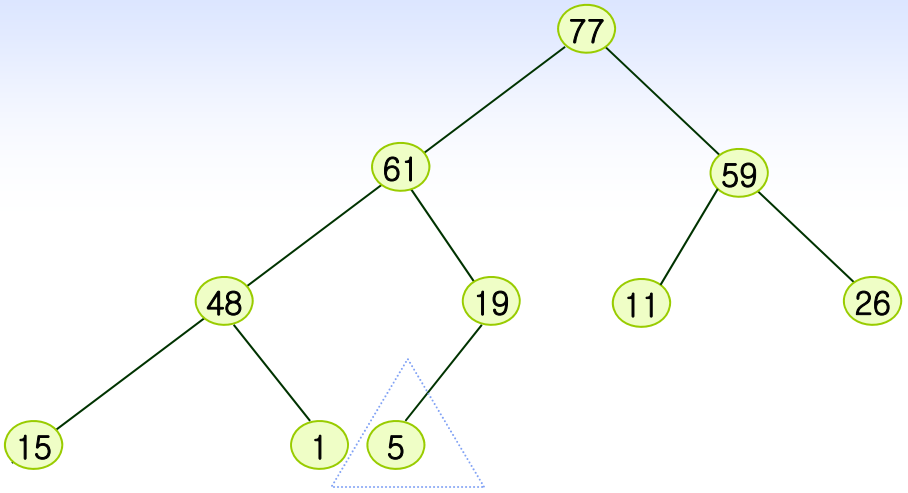
- 입력데이터 (26,5,77,1,61,11,59,15,48,19)



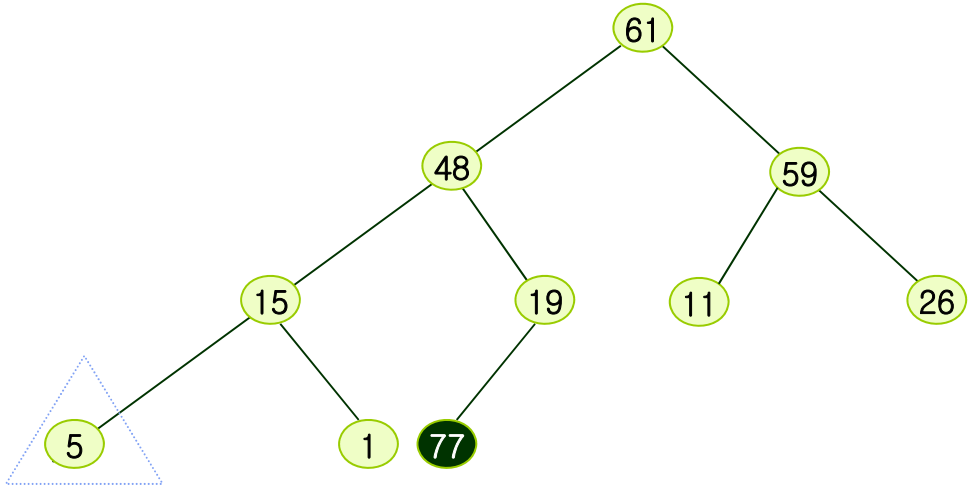

힙 구조(왼쪽)과 배열에 저장된 힙 (배열 첨자 1 부터 데이터 저장을 가정)

## 힙 구성 과정



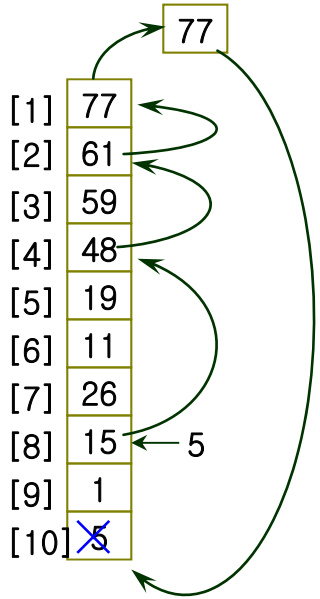


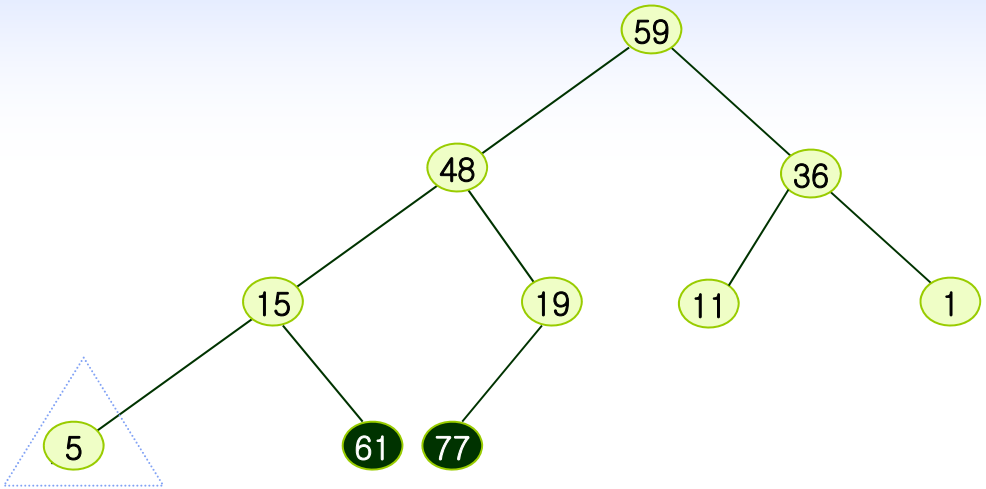

(a) 77의 삭제 (5를 루트로 이동)



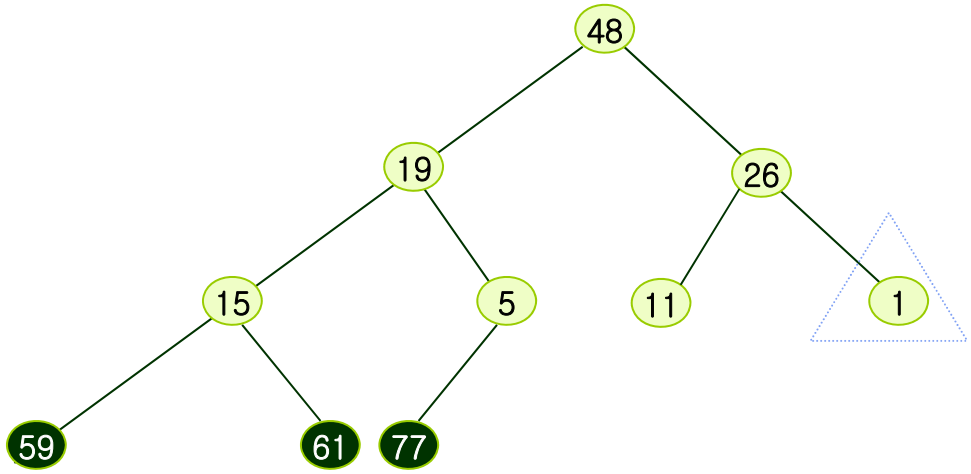

(b) 5의 재구성 – (a) 그림의 루트에 있던 5에 대하여 재구성

(삭제된 77을 마지막에 저장)

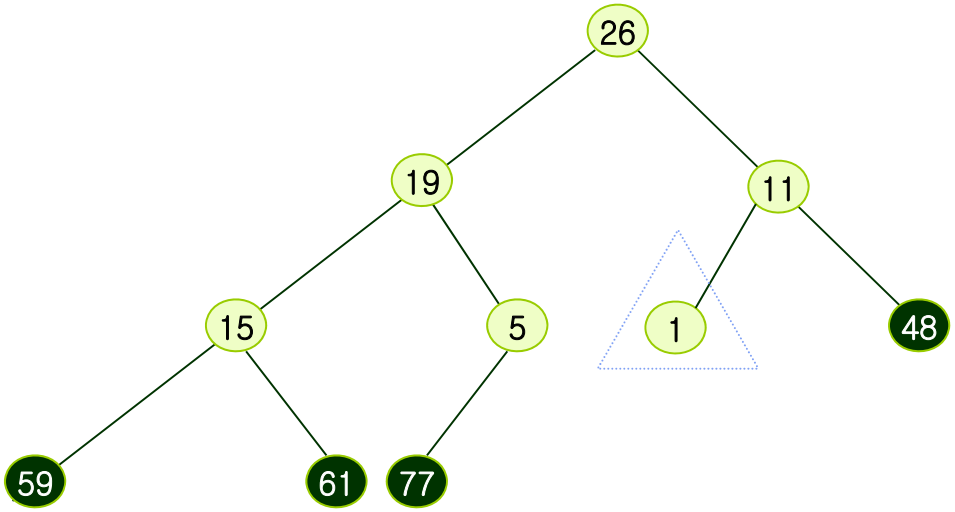





(c) 루트에 저장된 1의 재구성
(61을 마지막에 저장)




(d) 루트에 저장된 5의 재구성
(59를 마지막에 저장)




(e) 1의 재구성과 (48을 마지막에 저장)

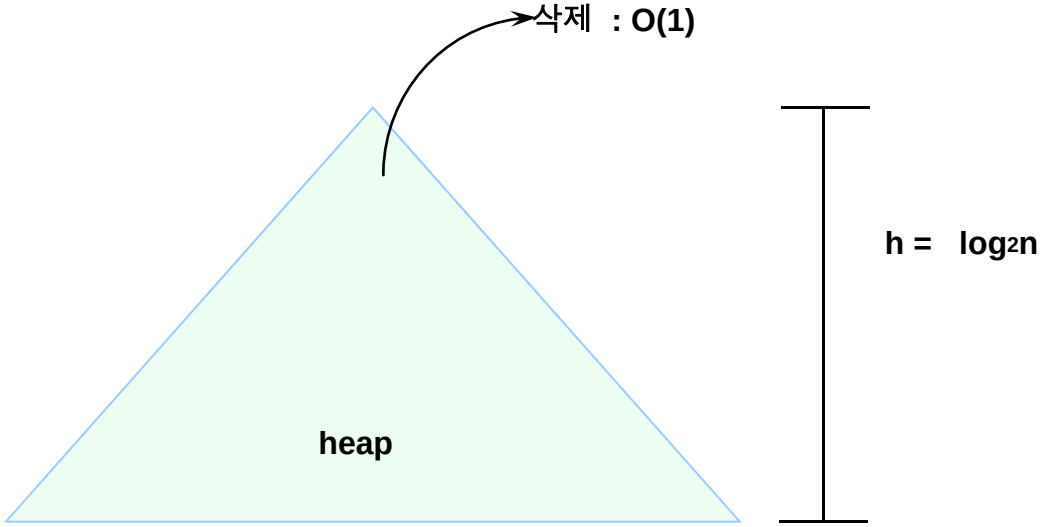
– 힙 정렬의 수행 시간

전체시간 = 힙 구성시간 + n개의 데이터 삭제 및 재구성시간

= 힙 구성시간 + ($\log_2 n$ + $\log_2(n-1)$ + … + $\log_2 2$)

= ($\log_2 n$ + $\log_2(n-1)$ + … + $\log_2 2$) + ($\log_2 n$ + $\log_2(n-1)$ + … + $\log_2 2$)

= $O(n \cdot \log_2 n)$

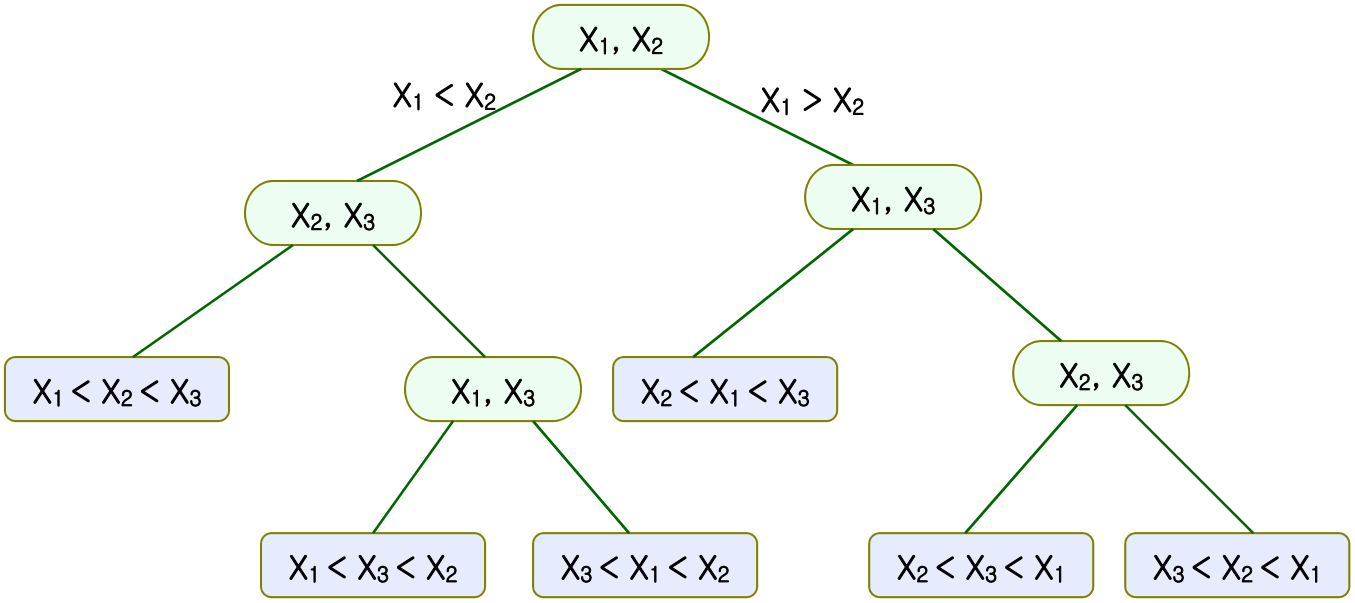


## 5. 정렬 알고리즘 요약

### (1) 최적의 정렬시간

정렬을 할 수 있는 이론적인 최소의 시간은 얼마인가?
- 최적의 시간은 $O(n \cdot \log_2 n)$이다.

-> list ($X_1$, $X_2$, $X_3$)개에 대하여 가능한 가지 수는 다음과 같이 구할 수 있다. 즉 6가지 결과가 가능하고 릾 노드의 수가 6개 이상이 되는 트리가 된다.
-> n개의 데이터는 n!개의 릾 노드가 필요하며 n!개의 릾 노드가 생기려면 트리의 깊이가 n log n 이상이 된다는 것이 증명되어 있다. 따라서 n log n번 이상의 비교를 해야만 정렬이 가능하다.



## (2) 정렬 알고리즘의 정리

| 정렬 알고리즘 | 알고리즘<br>평균 수행시간 | 알고리즘<br>최악 수행시간 | 알고리즘 기법 | 비고 |
|---|---|---|---|---|
| 버블정렬<br>Bubble sort | $O(n^2)$ 비교 | $O(n^2)$ 비교 | 비교와 교환 | flag을 이용하면<br>더 효율적이다. |
| 삽입정렬<br>Insertion sort | $O(n^2)$ 비교 | $O(n^2)$ 비교 | 비교와 교환 | |
| 선택정렬<br>Selection sort | $O(n^2)$ 비교 | $O(n^2)$ 비교 | 비교와 교환 | 교환의 횟수가 버블,<br>삽입정렬보다 작다 |
| 퀵 정렬<br>Quick sort | O(n log n) 비교 | $O(n^2)$ 비교 | Divide and Conquer,<br>순환 알고리즘 | 최악의 경우 $O(n^2)$<br>시간이 걸린다. |
| 힙 정렬<br>Heap sort | O(n log n) 비교 | O(n log n) 비교 | 힙 구조 이용 | 평균과 최악의 경우<br>모두 O(n log n) |

## 학습내용정리

자료구조와 알고리즘의 시작인 여러 가지 정렬 알고리즘을 보고 각 알고리즘을 비교하여 본다.

정렬은 입력데이터(리스트)에 대하여 데이터 값의 크기 순서대로 재구성하는 것을 말한다.

기본적인 정렬 알고리즘은 수행시간이 $O(n^2)$이 걸린다. 여기에는 버블정렬, 선택정렬, 삽입정렬 등이 있다.

효율적인 알고리즘으로는 수행시간이 $O(n \log n)$이 걸린다. 여기에는 퀵정렬, 힙 정렬, 머지 정렬이 있다.

정렬 알고리즘은 이 외에도 쉘 정렬, 기수정렬 등 수 많은 방법들이 있다. 이 방법들은 모두 비교 연산을 위주로 하여 정렬하는 방식이다.