정 렬

정렬이란?

- 정렬은 물건을 크기순으로 오름차순이나 내림차순으로 나열하는 것
- 정렬은 컴퓨터공학을 포함한 모든 과학기술 분야에서 가장 기본적이고 중요한 알고리즘 중 하나
- 정렬은 자료 탐색에 있어서 필수적

  (예) 만약 영어사전에서 단어들이 알파벳 순으로 정렬되어 있지 않다면?

정렬의 종류

- 내부정렬(internal sort)
  - 주기억장치 내에서 이루어지는 정렬 방식(RAM)
    - 선택정렬
    - 삽입정렬
    - 버블정렬
    - merge sort
    - quick sort
    - radix sort

- 외부 정렬(external sort)
  - 외부정렬은 보조기억장치를 이용하는 방식(하드디스크)

선택 정렬

- 선택 정렬(selection sort)
  - 정렬 되지 않은 인덱스의 맨 앞에서 부터, 이를 포함한 그 이후의 배열값 중 가장 작은 값을 찾은 후 교환

선택 정렬

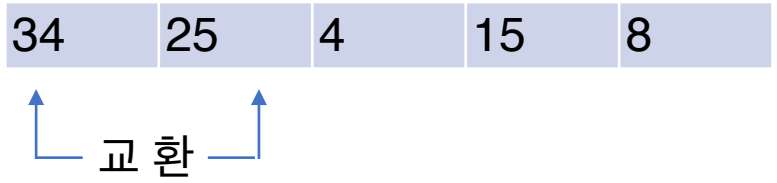
- 코드

```
def selectionSort(v):
    for i in range(0,len(v)):
        temp = i
        for j in range(i+1,len(v)):
            if(v[temp]>v[j]):
                temp = j
        v[i],v[temp] = v[temp],v[i]
```

버블 정렬

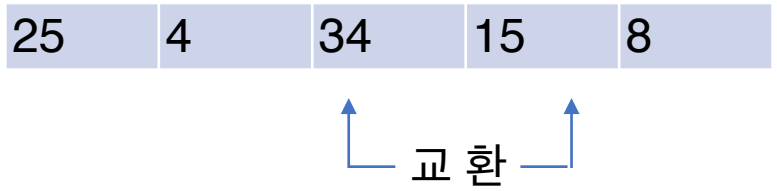
- 버블 정렬(bubble sort)
  - 배열에서 인접한 두 데이터 A[i]와 A[i+1]을 비교하여 두 데이터를 서로 교환하며 정렬

1.

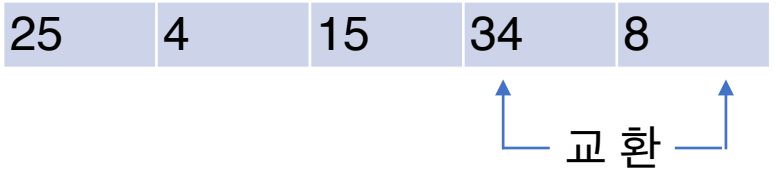


2.

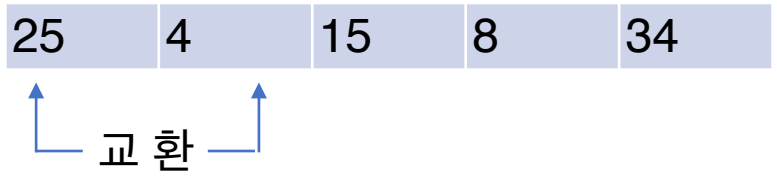
3.



4.



5.



6.

버블 정렬

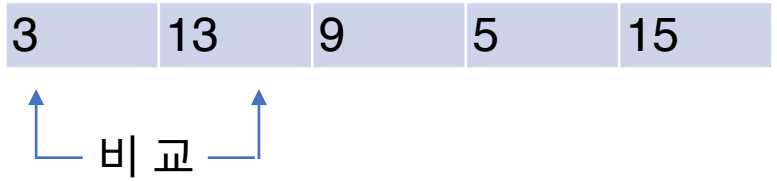
- 코드

```
def bubbleSort(v):
    for i in range(0,len(v)-1):
        for j in range(1, len(v)-i):
            if(v[j-1]>v[j]):
                v[j-1],v[j] = v[j],v[j-1]
```

삽입 정렬

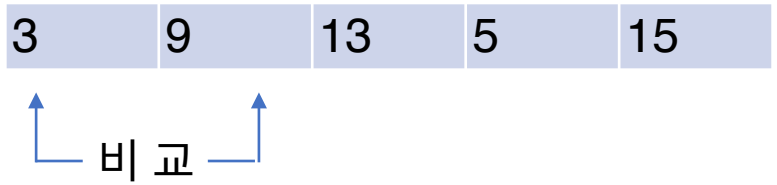
- 삽입 정렬(insertion sort)
  - 새로운 데이터를 정렬된 데이터에 삽입해 나가는 과정을 반복

1.

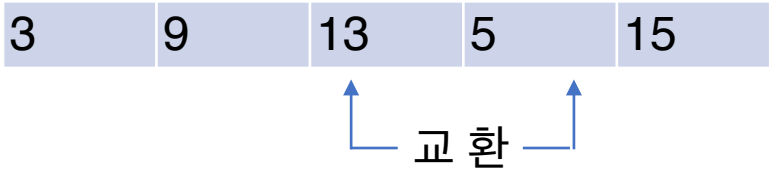


2.

3.

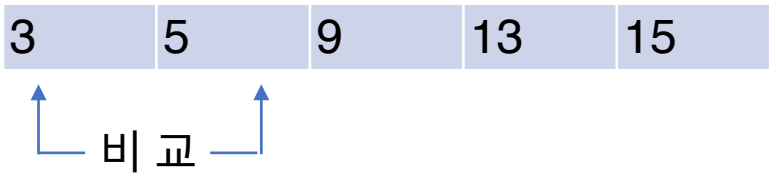


4.



5.

6.



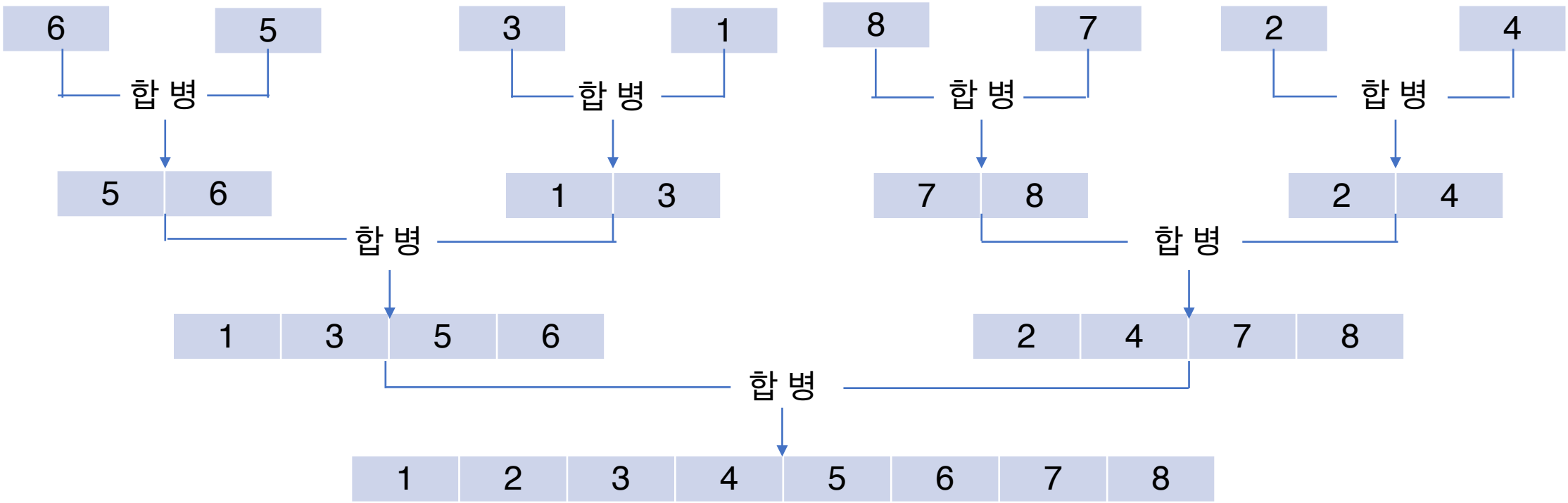
삽입 정렬

- 코드

```
def insertionSort(v):
    for i in range(1, len(v)):
        key = v[i]
        j = i-1
        while(j>=0 and key<v[j]):
            v[j],v[j+1] = v[j+1],v[j]
            j-=1
```

합병 정렬

- 합병 정렬(merge sort)
  - 입력으로 하나의 배열을 받고, 연산 중에 두개의 배열로 쪼게 나간 뒤, 합치면서 정렬해 최후에 하나의 정렬 출력



합병 정렬

- 코드

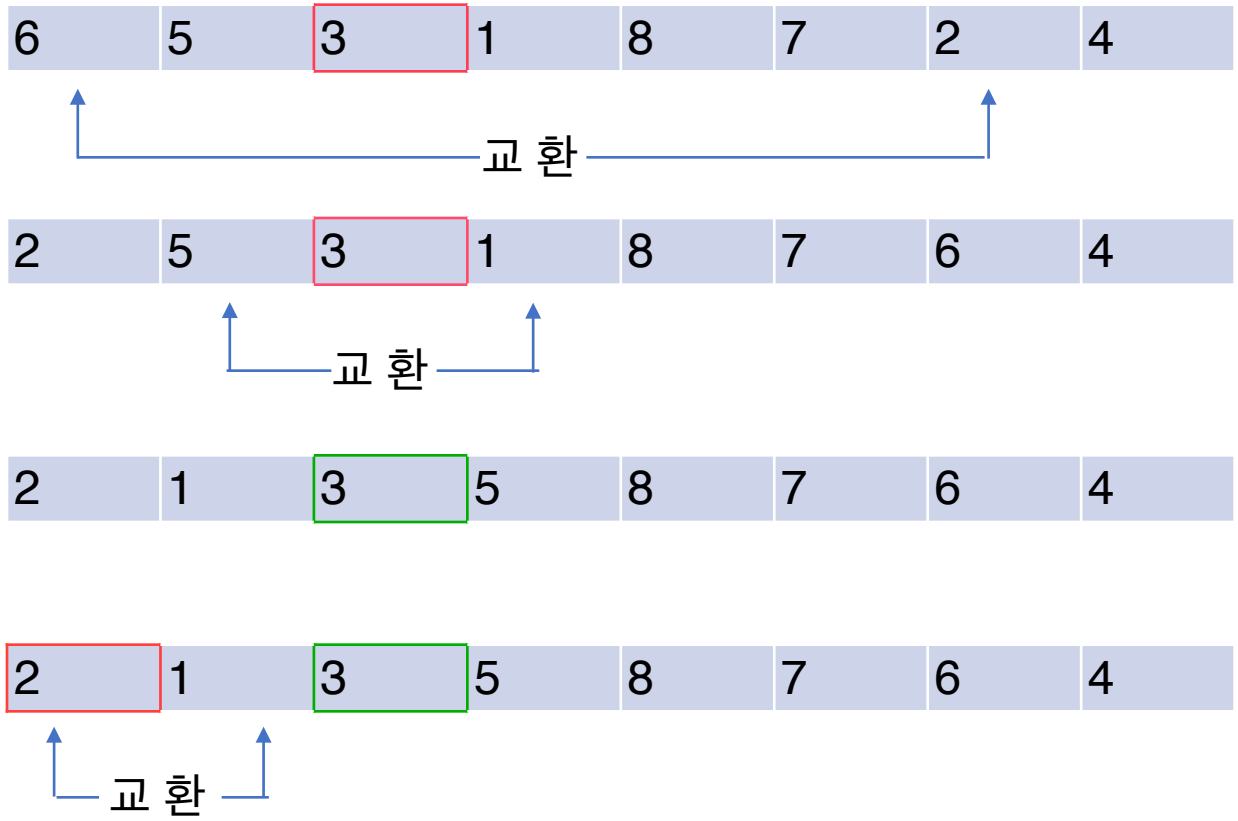
```
def merge(v, start, end, mid):
    res = []
    i = start
    j = mid+1
    copy = 0
    while(i<=mid and j<=end):
        if(v[i]<v[j]):
            res.append(v[i])
            i+=1
        elif(v[i]>v[j]):
            res.append(v[j])
            j+=1
    while(i<=mid):
        res.append(v[i])
        i+=1
    while(j<=end):
        res.append(v[j])
        j+=1
    for k in range(start,end+1):
        v[k] = res[copy]
        copy+=1
```

```
def mergeSort(v, start, end):
    if(start<end):
        mid = (start+end)//2
        mergeSort(v,start,mid)
        mergeSort(v,mid+1,end)
        merge(v,start,end,mid)
```

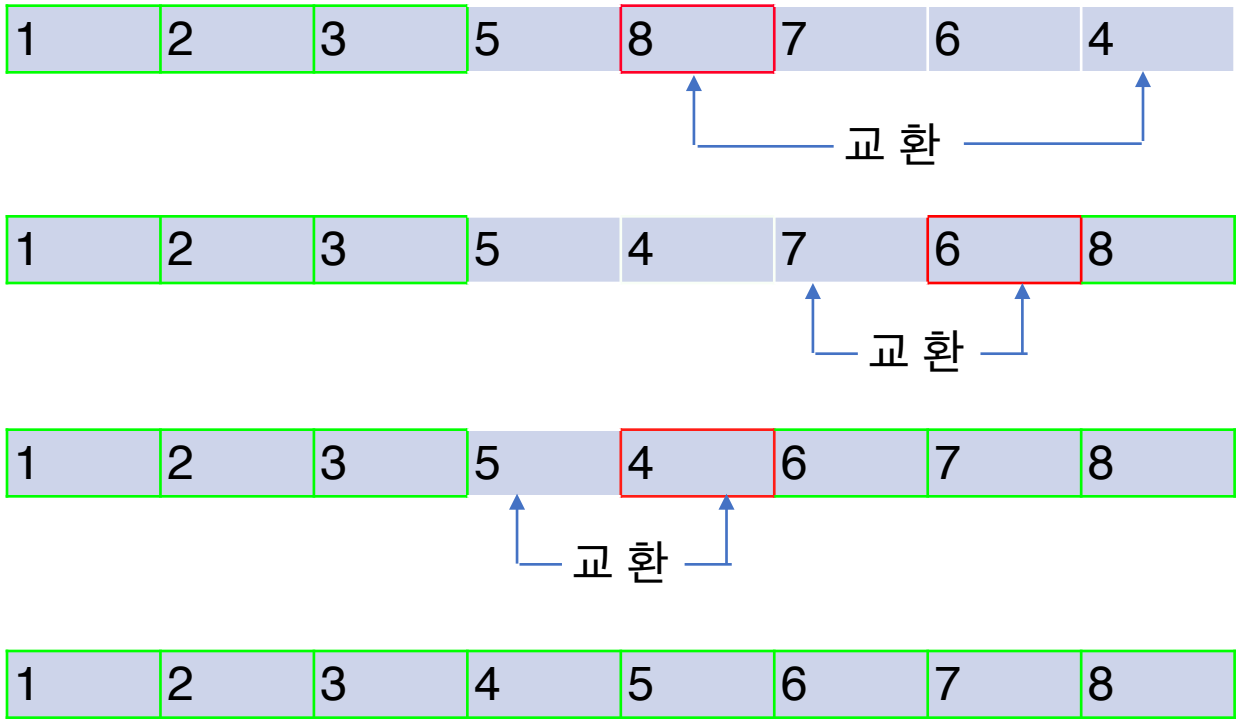
퀵 정렬

- 퀵 정렬(quick sort)
  - pivot point라고 기준이 되는 값을 하나 설정
  - 이 값을 기준으로 작은 값은 왼쪽, 큰 값은 오른쪽으로 옮기는 방식으로 정렬을 진행한다
- 기본 로직
  - pivot point를 잡는다. (보통 맨 앞, 맨 뒤, 중간값, 랜덤값)
  - 가장 왼쪽 배열의 인덱스를 저장하는 left 변수, 가장 오른쪽 배열의 인덱스를 저장한 right 변수 생성
  - right부터 비교를 진행. 비교는 right가 left보다 클 때만 반복. 비교한 배열값이 pivot point보다 크면 right 하나 감소시키고 비교를 반복. pivot point보다 작은 배열 값을 찾으면 반복을 중지
  - left부터 비교를 진행. 비교는 right가 left보다 클 때만 반복하며 비교한 배열값이 pivot point보다 작으면 left를 하나 증가시키고 비교를 반복. pivot point보다 큰 배열 값을 찾으면, 반복을 중지
  - left 인덱스의 값과 right 인덱스의 값을 바꿔준다.
  - left<right가 만족 할 때 까지 반복
  - left의 값과 pivot point를 바꿔준다.
  - 맨 왼쪽부터 left-1까지, left+1부터 맨 오른쪽까지로 나눠 퀵 정렬을 반복

퀵 정렬



퀵 정렬



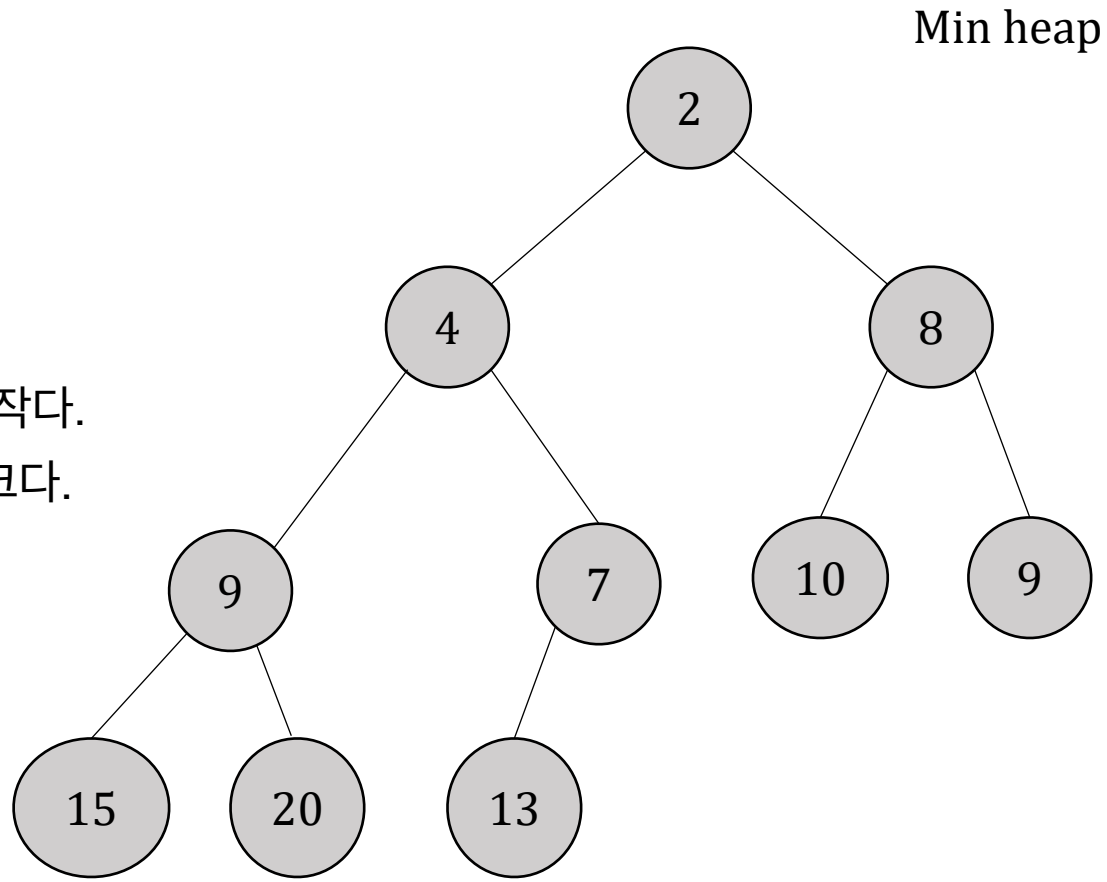
퀵 정렬

- 코드

```
def quickSort(v, start, end):
    pivot = v[start]
    bs = start
    be = end
    while(start < end):
        while(pivot <= v[end] and start<end):
            end-=1
        if(start>end):
            break
        while(pivot >= v[start] and start<end):
            start+=1
        if(start > end):
            break
        v[start],v[end] = v[end],v[start]
    v[bs],v[start] = v[start],v[bs]
    if(bs<start):
        quickSort(v,bs,start-1)
    if(be>end):
        quickSort(v,start+1,be)
```

힙 정렬

- 힙 정렬(heap sort)
  - Data 구조는 완전 이진 트리 형태이다.
  - max heap인 경우 한 노드의 자식은 부모보다 무조건 작다.
  - min heap의 경우 한 노드의 자식은 부모보다 무조건 크다.



- 완전 이진 트리
  - 위에서 아래로 왼쪽에서 오른쪽으로 완전히 채워져 있는 트리
  - 마지막 레벨을 제외하고 모든 레벨이 완전히 채워져 있으며 마지막 레벨은 왼쪽부터 채워져 있다.

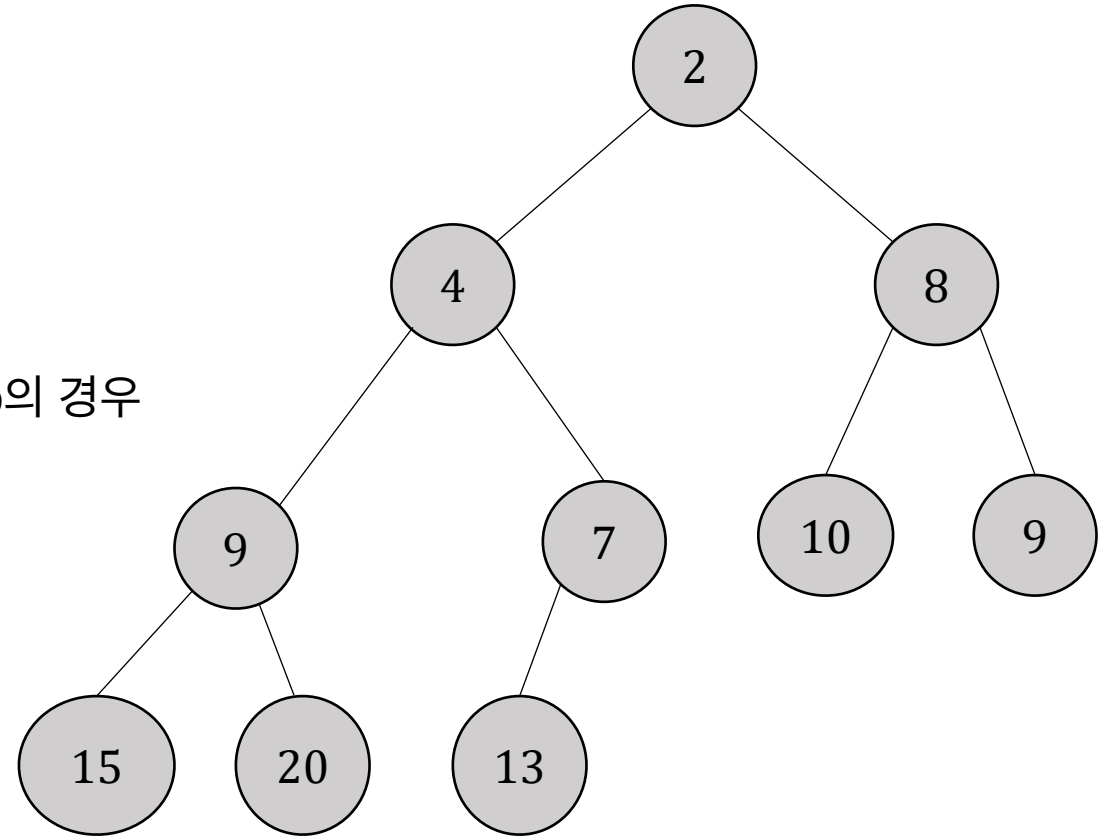
힙 정렬

- priority_queue
  - 우선순위 큐는 힙 구조로 되어 있다.
  - pop(), top(), push() 연산이 수행된다.

- 이와 같은 트리가 있을 때, root node가 top()이고 min heap의 경우 가장 작은 값을 갖고 max heap의 경우 가장 큰 값을 갖는다.

- top()연산의 경우 O(1)에 수행할 수 있다.

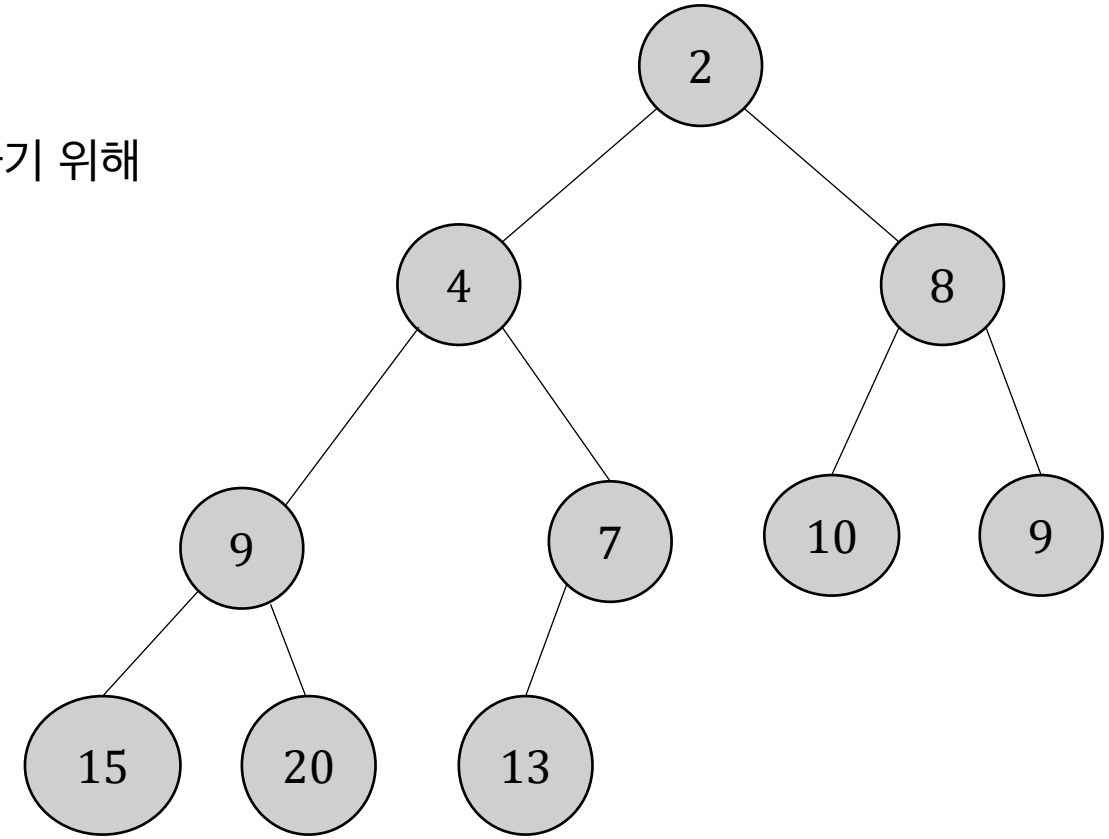
Min heap



힙 정렬

- pop()을 구현하기 위해서는 루트 노드를 제거 합니다.
- 루트 노드를 제거 후 heap property를 만족하도록 처리를 하기 위해 Downheap()을 수행 해야합니다.

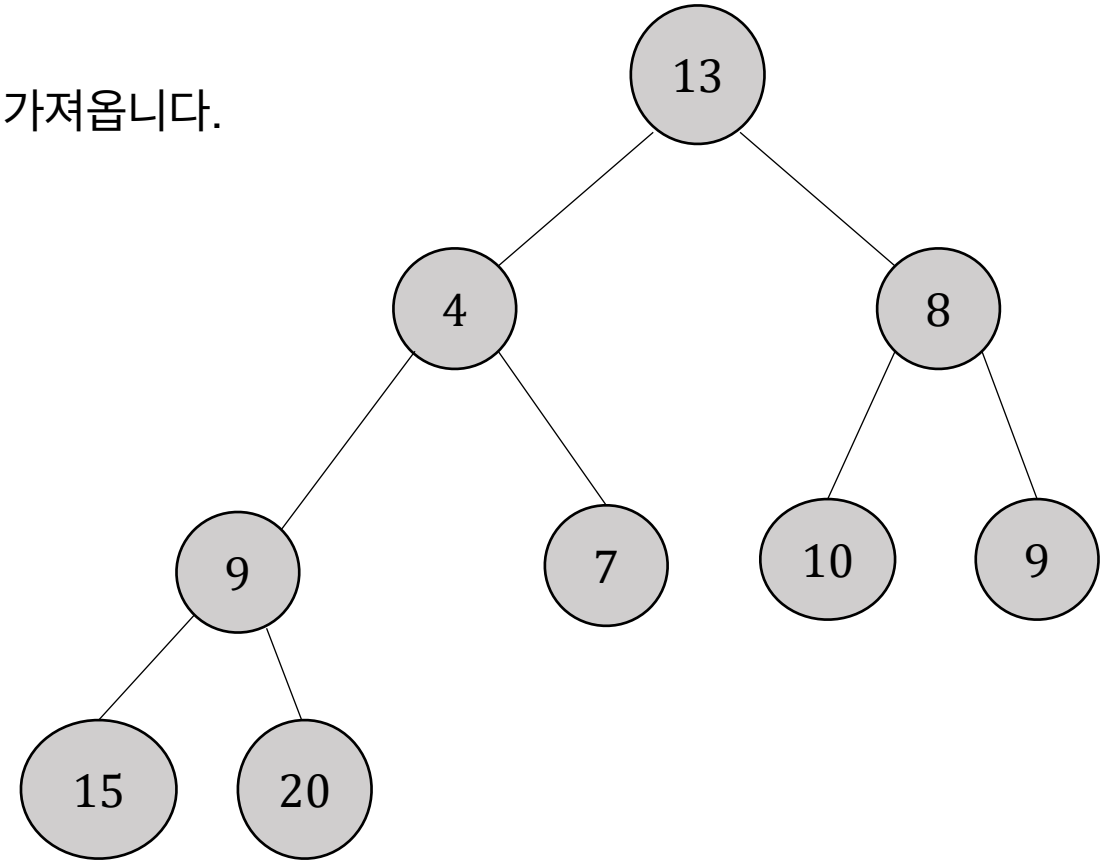
Min heap



힙 정렬

Min heap

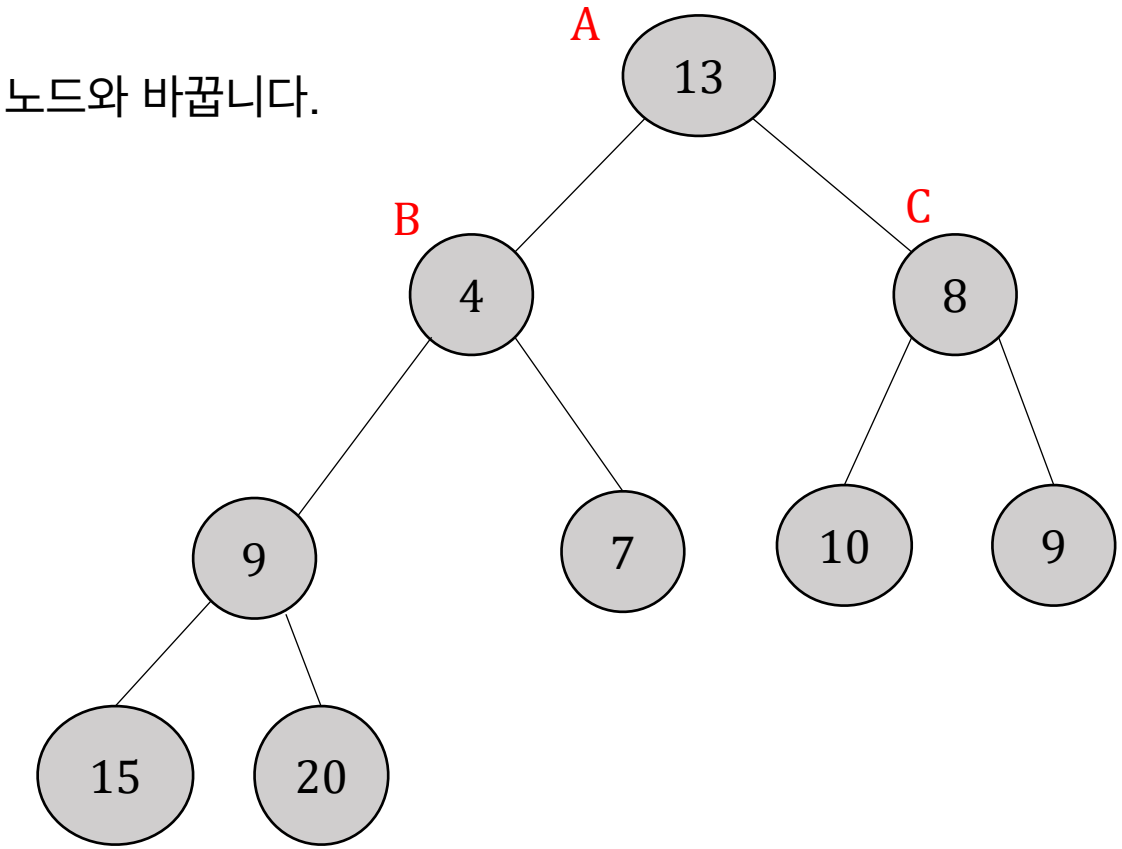
- 루트 노드 2를 제거 후 가장 마지막에 있는 13을 루트 노드로 가져옵니다.



힙 정렬

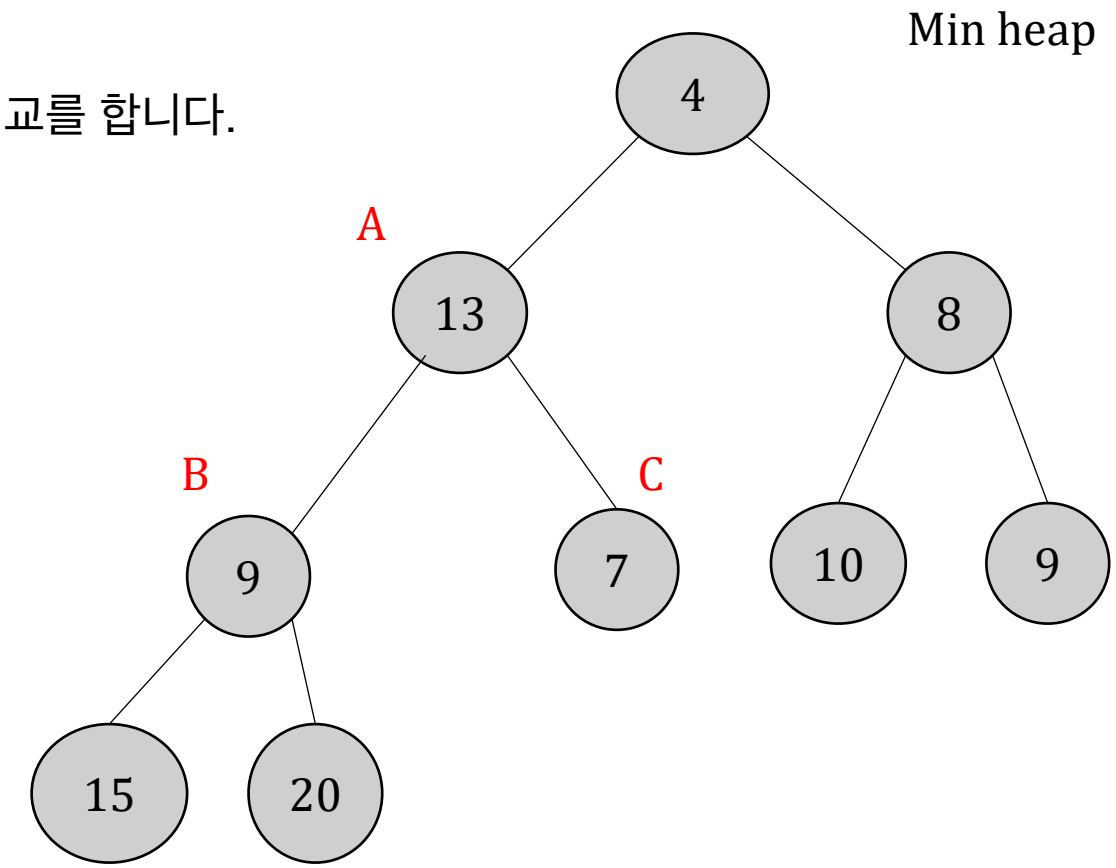
Min heap

- 왼쪽(B)과 오른쪽(C) 자식을 비교하여 자기보다 작은 노드와 바꿉니다.
- 둘 다 작을 경우 둘 중 더 작은 노드와 바꿉니다.
- Heap property를 만족하면 종료합니다.



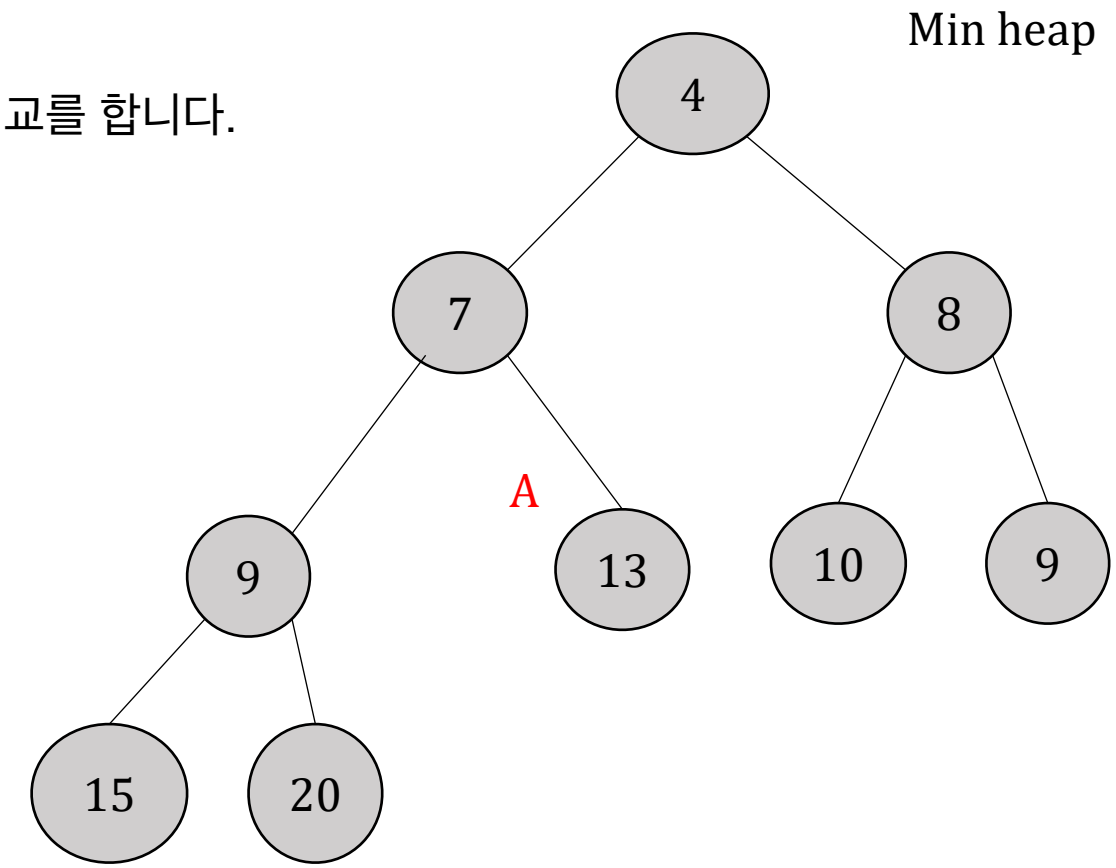
힙 정렬

- 바꾼 자식 노드를 A로 놓고 다시 자식 B그리고 C와 비교를 합니다.
- B와 C중 더 작은 노드와 바꿉니다.
- Heap property를 만족하면 종료합니다.



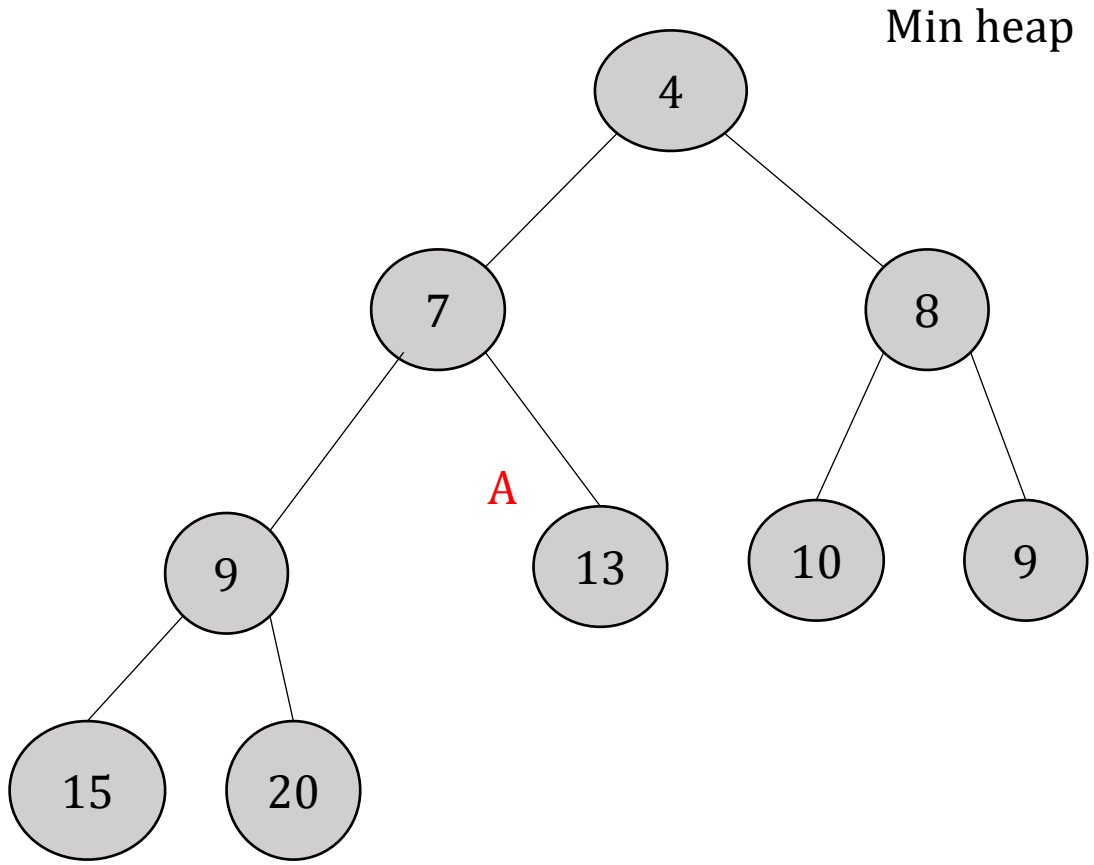
힙 정렬

- 바꾼 자식 노드를 A로 놓고 다시 자식 B그리고 C와 비교를 합니다.
- B와 C중 더 작은 노드와 바꿉니다.
- Heap property를 만족하면 종료합니다.



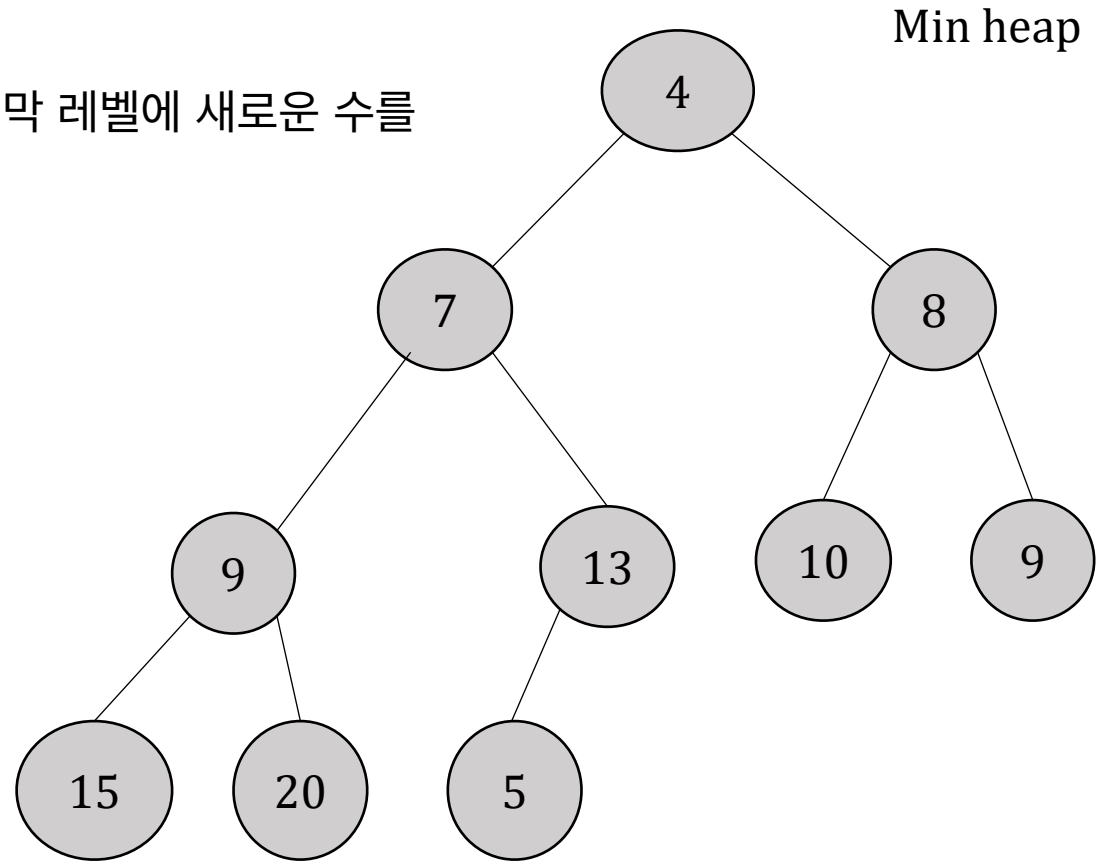
힙 정렬

- pop()의 시간복잡도
  - tree의 level(높이)만큼이 걸린다.
  - O(logN)



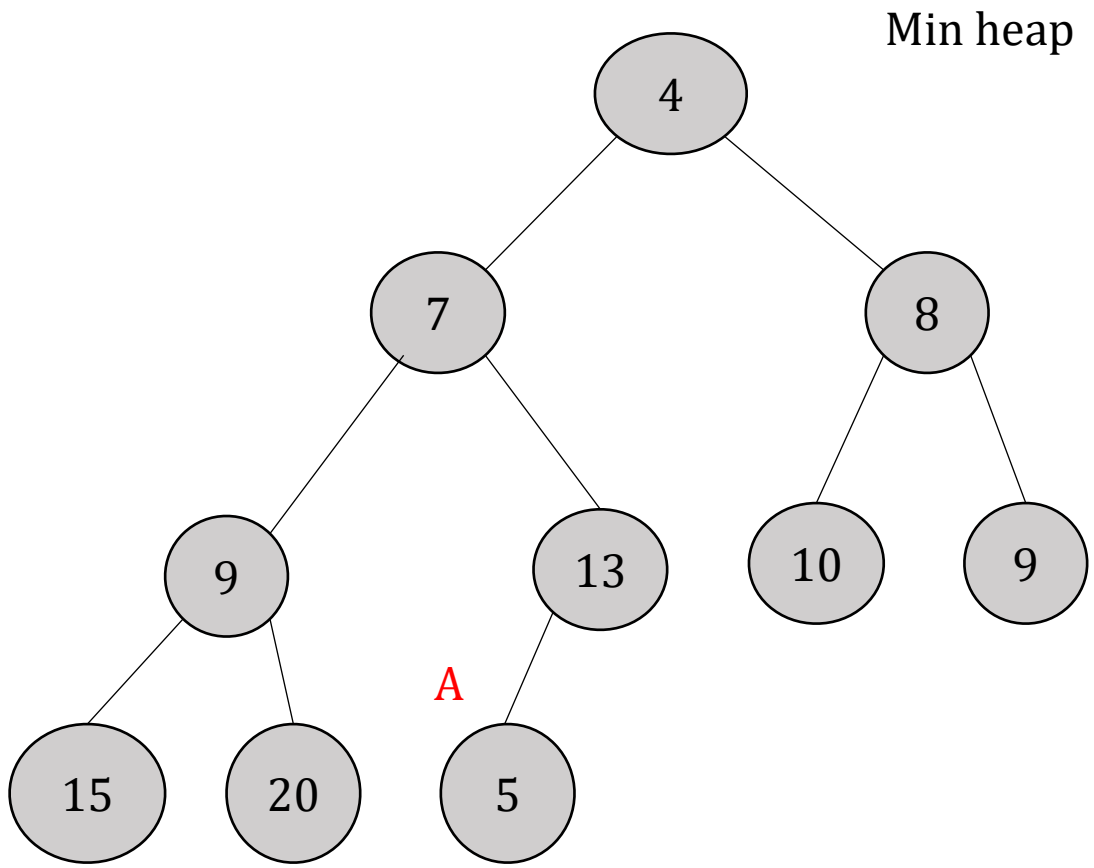
힙 정렬

- push()를 구현하기 위해서 완전 이진 트리에 맞게 마지막 레벨에 새로운 수를 넣고 Upheap()을 수행합니다.



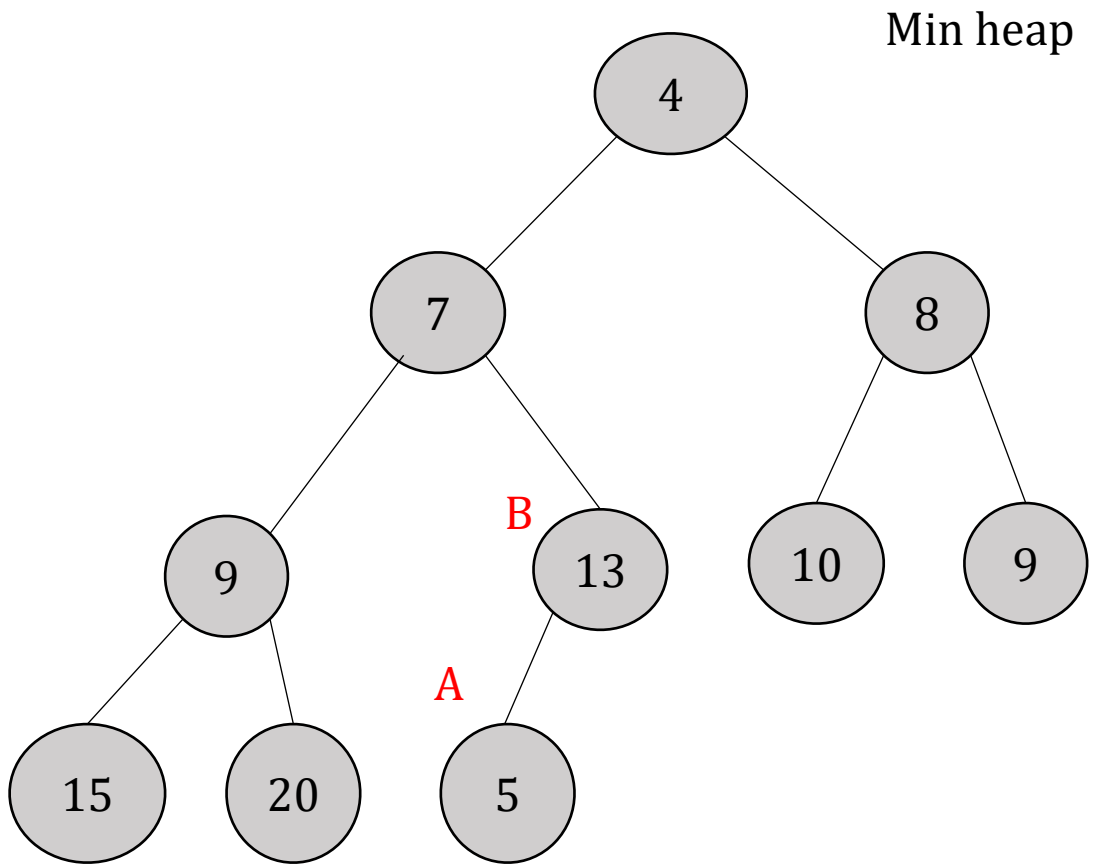
힙 정렬

- 새로 입력한 노드를 A로 놓습니다.



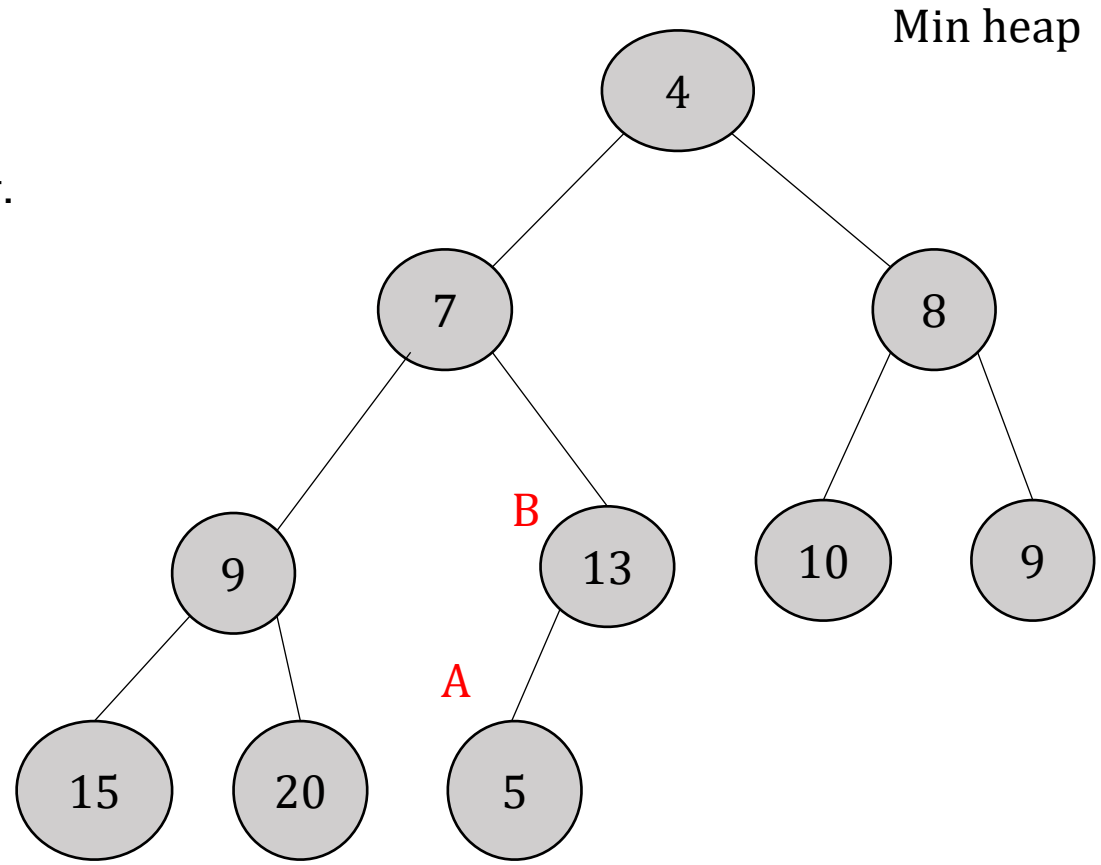
힙 정렬

- A의 부모 노드를 B로 놓습니다.



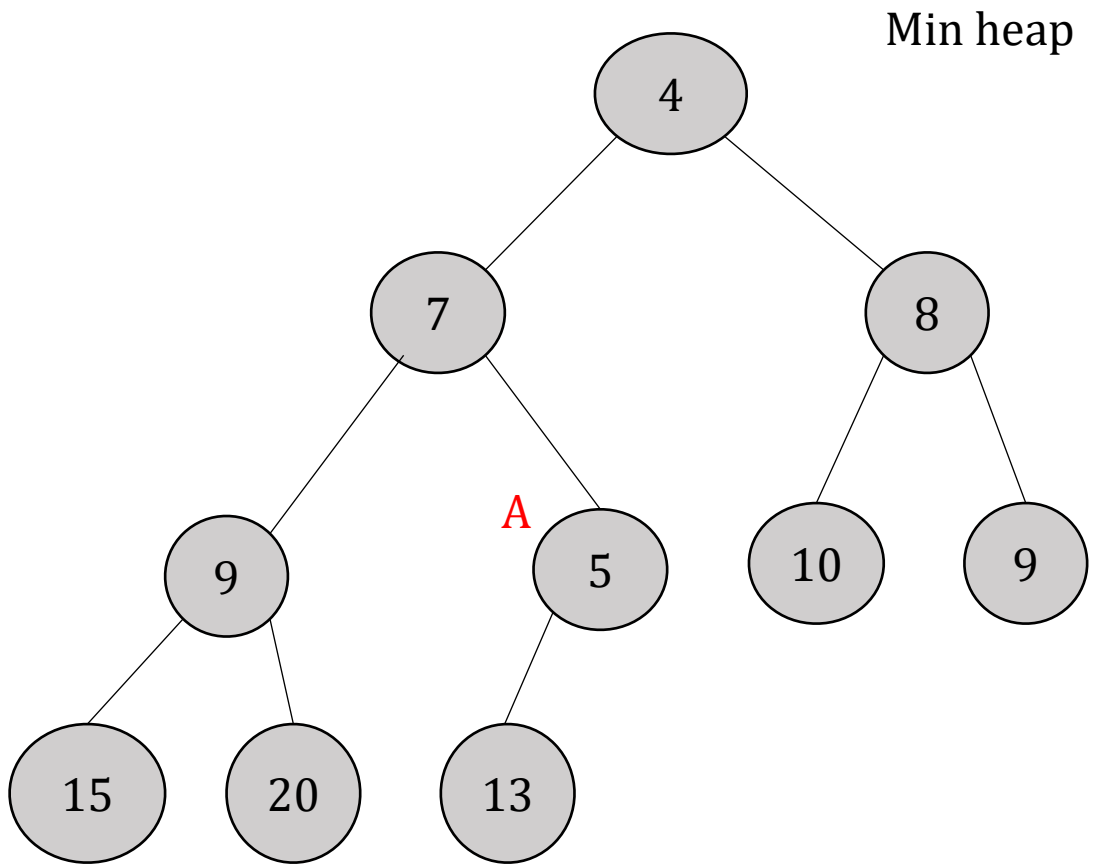
힙 정렬

- Heap property를 만족하면 종료합니다.
- 만족하지 않을 경우 A와 B를 바꾸고 B를 A로 놓습니다.



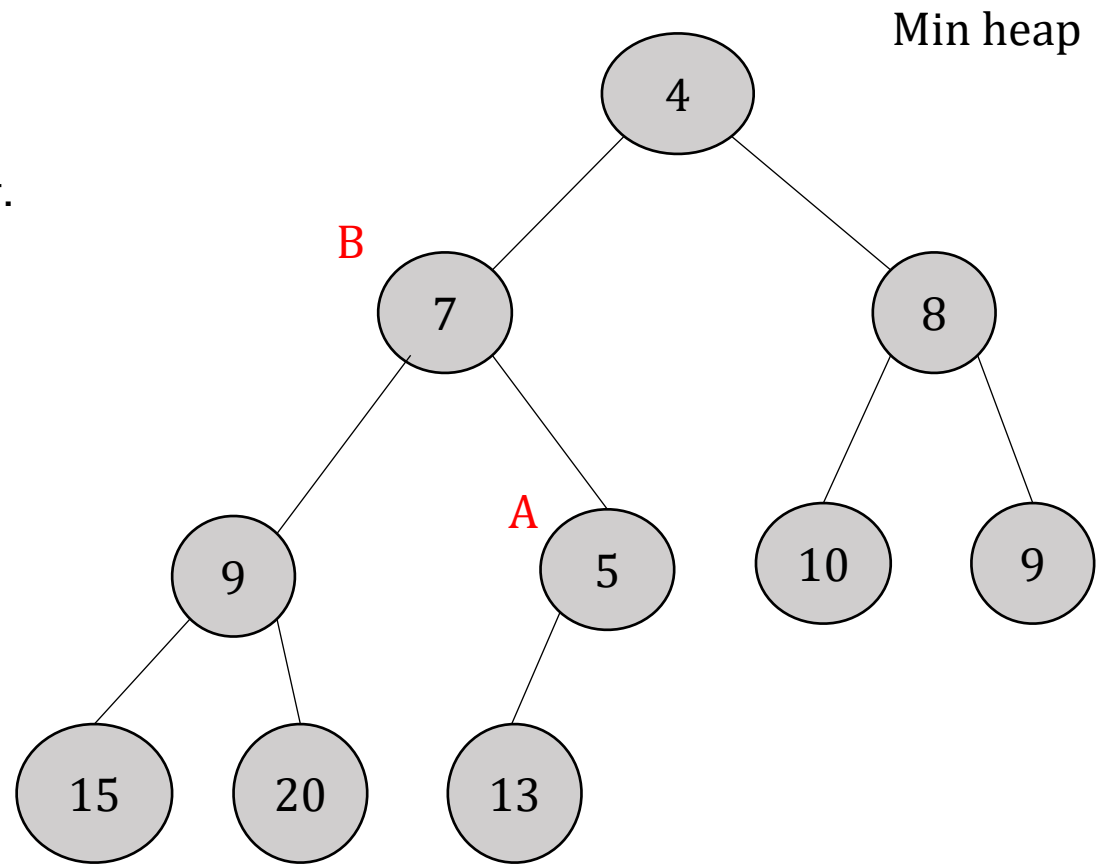
힙 정렬

- A의 부모 노드를 B로 놓습니다.



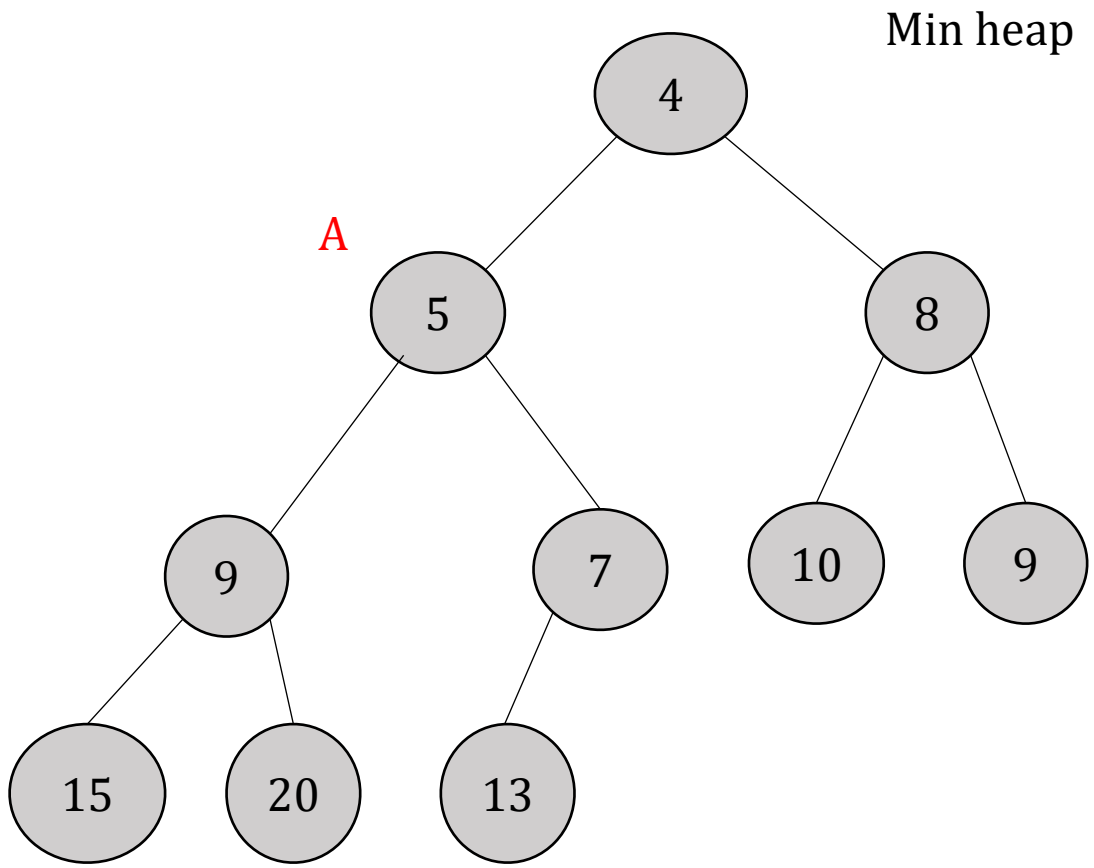
힙 정렬

- Heap property를 만족하면 종료를 합니다.
- 만족하지 않을 경우 A와 B를 바꾸고 B를 A로 놓습니다.



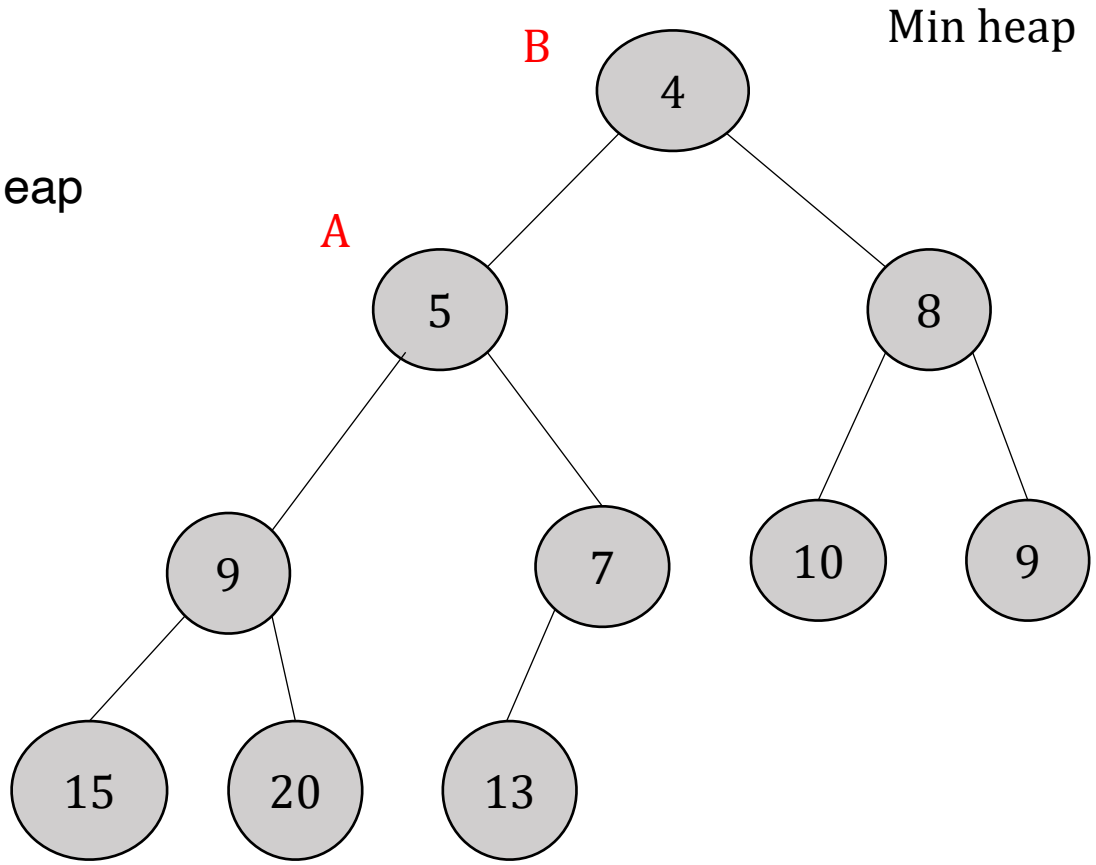
힙 정렬

- A의 부모 노드를 B로 놓습니다.



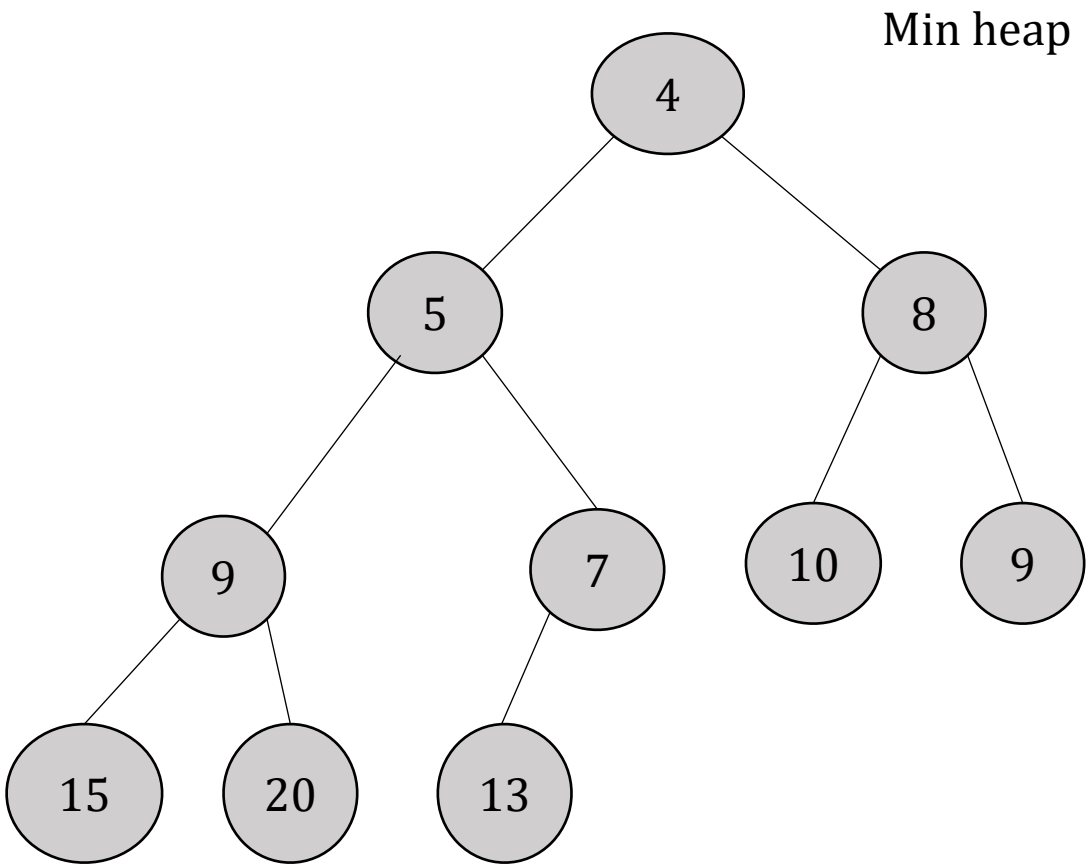
힙 정렬

- Heap property를 만족하면 종료합니다.
- 만족하지 않을 경우 A와 B를 바꾸고 B를 A로 놓지만 Heap property를 만족하므로 종료합니다.



힙 정렬

- push() 함수의 시간 복잡도
  - Down Heap과 마찬가지로 O(logN)



정렬 알고리즘의 비교

알고리즘	최선	평균	최악
삽입 정렬	O(n)	O(n^2)	O(n^2)
선택 정렬	O(n^2)	O(n^2)	O(n^2)
버블 정렬	O(n^2)	O(n^2)	O(n^2)
합병 정렬	O(nlog(n))	O(nlog(n))	O(nlog(n))
퀵 정렬	O(nlog(n))	O(nlog(n))	O(n^2)
힙 정렬	O(nlog(n))	O(nlog(n))	O(nlog(n))

실 습

- Sorting

```
li = [6,5,3,1,8,100,7,2,4]
li.sort()
print(li)
```

```
li = [(6,5),(1,2),(3,5),(7,3),(2,2)]
li.sort()
print(li)
```

```
li = [(6,5),(1,2),(3,5),(7,3),(2,2),(1,1),(2,3),(2,1)]
temp = sorted(li)
li.sort()
print(temp)
print(li)
```

```
li = [(6,5),(1,2),(3,5),(7,3),(2,2),(1,1),(2,3),(2,1)]
li.sort(key = lambda x : (x[0]))
print(li)
```

```
li = [(6,5),(1,2),(3,5),(7,3),(2,2),(1,1),(2,3),(2,1)]
li.sort(key = lambda x : (x[0],-x[1]))
print(li)
```

- 결과

```
[1, 2, 3, 4, 5, 6, 7, 8, 100]
[(1, 2), (2, 2), (3, 5), (6, 5), (7, 3)]
[(1, 1), (1, 2), (2, 1), (2, 2), (2, 3), (3, 5), (6, 5), (7, 3)]
[(1, 1), (1, 2), (2, 1), (2, 2), (2, 3), (3, 5), (6, 5), (7, 3)]
[(1, 2), (1, 1), (2, 2), (2, 3), (2, 1), (3, 5), (6, 5), (7, 3)]
[(1, 2), (1, 1), (2, 3), (2, 2), (2, 1), (3, 5), (6, 5), (7, 3)]
```

실 습

- priority_queue (max_heap)

```
import heapq

li = [5,1,4,8,7,9,3,6,2]
heapq.heapify(li)
print(li)

heap = []
heapq.heappush(heap,10)
heapq.heappush(heap,20)
heapq.heappush(heap,15)
print(heap)

top = heapq.heappop(heap)
print(top)
print(heap)

print(heap[0])
```

- 결과

[1, 2, 3, 5, 7, 9, 4, 6, 8]
[10, 20, 15]
10
[15, 20]
15

실 습

- priority_queue (min heap)

```
import heapq

heap = []
for i in li:
    heapq.heappush(heap, -i)
while(heap):
    print(-heapq.heappop(heap))
```

- 결과

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

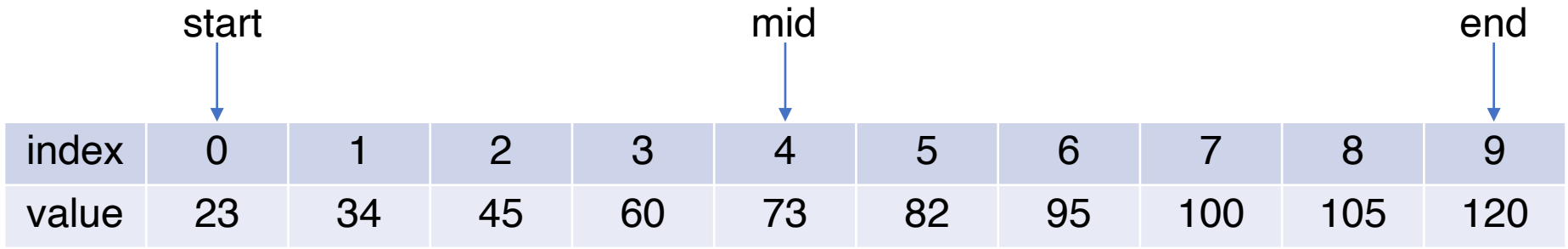
이분 탐색

이분 탐색이란?

- 탐색이란?
  - 탐색(search)는 기본적으로 여러 개의 자료 중에서 원하는 자료를 찾는 작업이다.

- 이분 탐색이란?
  - n개의 원소를 갖는 정렬된 배열의 경우 log(n)에 탐색을 할 수 있는 효율적인 탐색 방법

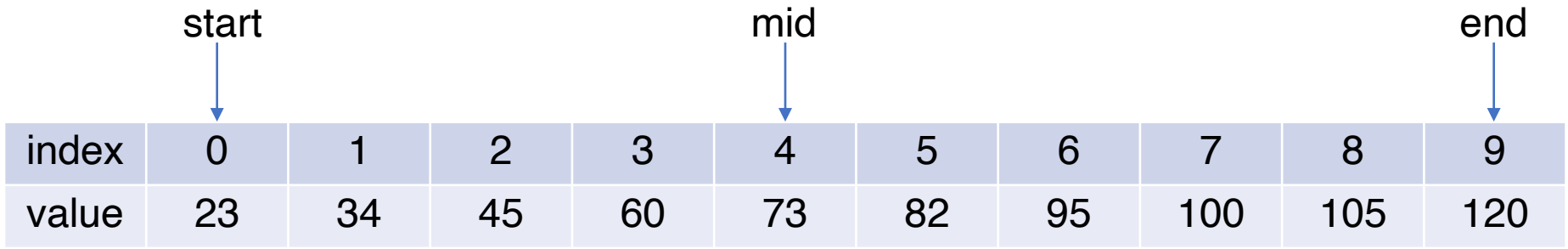
이분 탐색



index	0	1	2	3	4	5	6	7	8	9
value	23	34	45	60	73	82	95	100	105	120

- 위의 그림과 정렬된 배열이 있고, target이 95라 가정.
- start와 end 변수를 생성하고 각각 0과 9로 설정.
- mid 변수를 선언하여 (start+end)/2 로 초기화 한다.
- mid = (0+9)/2 = 4

이분 탐색



index	0	1	2	3	4	5	6	7	8	9
value	23	34	45	60	73	82	95	100	105	120

- start > end 이면 종료
- mid index에 있는 값과 target(95) 와 비교를 한다.
- target이 더 작은 경우 end를 mid-1로 설정 후 mid를 다시 계산
- target이 더 큰 경우 start를 mid+1로 설정 후 mid를 다시 계산
- target이 실제로 더 크므로 start=mid+1로 설정
- mid = (start+end)/2 = (5+9)/2 = 7

이분 탐색

start mid end

index	0	1	2	3	4	5	6	7	8	9
value	23	34	45	60	73	82	95	100	105	120

- start > end 이면 종료
- mid index에 있는 값과 target(95) 와 비교를 한다.
- target이 더 작은 경우 end를 mid-1로 설정 후 mid를 다시 계산
- target이 더 큰 경우 start를 mid+1로 설정 후 mid를 다시 계산
- target이 mid index에 있는 value(100)보다 작으므로 end=mid-1로 설정
- mid = (start+end)/2 = (5+6)/2 = 5

이분 탐색

start, mid end

index	0	1	2	3	4	5	6	7	8	9
value	23	34	45	60	73	82	95	100	105	120

- start > end 이면 종료
- mid index에 있는 값과 target(95) 와 비교를 한다.
- target이 더 작은 경우 end를 mid-1로 설정 후 mid를 다시 계산
- target이 더 큰 경우 start를 mid+1로 설정 후 mid를 다시 계산
- target이 mid index에 있는 value(82)보다 크므로 start=mid+1로 설정
- mid = (start+end)/2 = (6+6)/2 = 6

이분 탐색

start, mid, end

index	0	1	2	3	4	5	6	7	8	9
value	23	34	45	60	73	82	95	100	105	120

- start > end 이면 종료
- mid index에 있는 값과 target(95) 와 비교를 한다.
- target이 더 작은 경우 end를 mid-1로 설정 후 mid를 다시 계산
- target이 더 큰 경우 start를 mid+1로 설정 후 mid를 다시 계산
- target이 mid index에 있는 value(95)와 같으므로 찾았으니 종료

이분 탐색

- 코드

```
def binary_search(arr):
    start = 0
    end = len(arr)-1
    target = 95
    while(start<=end):
        mid = (start+end)//2
        if(arr[mid]==target):
            print("find")
            quit()
        elif(arr[mid]<target):
            start = mid+1
        elif(arr[mid]>target):
            end = mid-1
    print('can\'t find')

arr = [23,45,120,105,95,82,73,100,60,34]
arr.sort()
binary_search(arr)
```

- 결과

find

이분 탐색

- lower bound
  - 정렬된 배열에서 찾고자 하는 key 값이 있을 때 key보다 크거나 같은 첫번째 위치(이상)을 반환

- upper bound
  - 정렬된 배열에서 찾고자 하는 key 값이 있을 때 key보다 큰 첫번째 위치(초과)를 반환

- 예)
  - 배열 {1, 3, 3, 5, 7}이 있을 경우 찾고자 하는 key가 3이면, Upper Bound = 3(index), Lower Bound = 1(index)
  - 배열 {1, 3, 3, 5, 7}이 있을 경우 찾고자 하는 key가 2이면, Upper Bound = 1(index), Lower Bound = 1(index)

이분 탐색

- lower_bound

```
def lower_bound(start, end, target):
    while(start<end):
        mid = (start+end)//2
        if(arr[mid] < target):
            start = mid+1
        else:
            end = mid
    return end+1
```

```
from bisect import bisect_left

a = [1,3,5,7,9]
print(bisect_left(a, x=3))
```

1

- upper_bound

```
def upper_bound(start, end, target):
    while(start<end):
        mid = (start+end)//2
        if(arr[mid]<=target):
            start = mid+1
        else:
            end = mid
    return end+1
```

```
from bisect import bisect_right

a = [1,3,5,7,9]
print(bisect_right(a, x=3))
```

3

Set / Map

- Set / Map
  - Map
    - key와 value의 쌍으로 이루어진 트리.
    - key를 기준으로 정렬된 상태
  - Set
    - key 값만 있고 value가 없는 정렬된 집합

- 구조
  - 균형 이진 트리인 AVL 트리와 Red-Black 트리로 구성