힙 정렬

작성일시	@2023년 6월 18일 오후 12:24
종류	Algorithm
유형	
자료	
복습	☐
Spring Framwork	

💡 1. 장점
최악의 경우에도 O(NlogN)으로 유지가 된다.
힙의 특성상 부분 정렬을 할 때 좋다.

2. 단점
일반적인 O(NlogN)정렬 알고리즘에 비해 성능이 약간 떨어진다.
한 번 최대힙을 만들면서 불안정 정렬상태에서 최댓값만 갖고 정렬을 하기 때문에 안정정렬이 아니다.

▼ 힙 자료구조 기반 구현된 우선순위 큐 자료구조

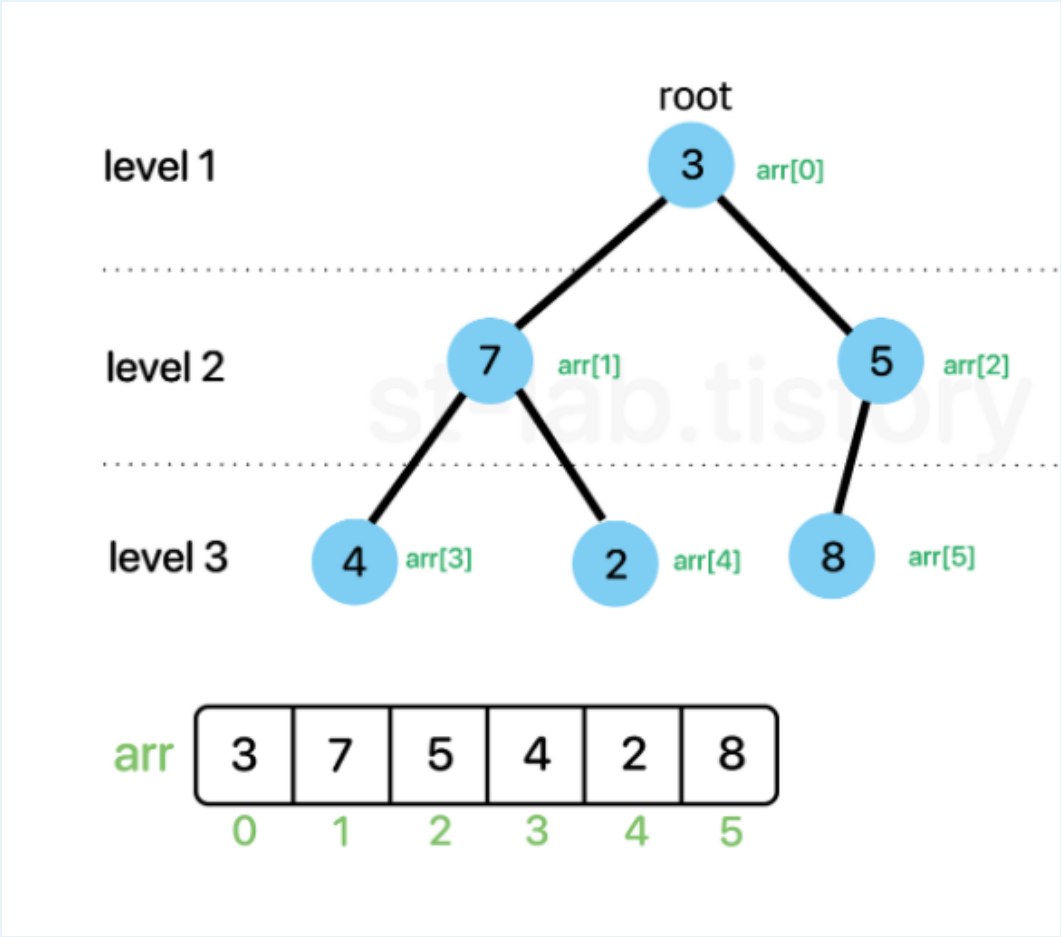
```
import java.util.PriorityQueue;

public class test {
  public static void main(String[] args) {

    int[] arr = {3, 7, 5, 4, 2, 8};
    System.out.print(" 정렬 전 original 배열 : ");
    for(int val : arr) {
      System.out.print(val + " ");
    }

    PriorityQueue<Integer> heap = new PriorityQueue<Integer>();

    // 배열을 힙으로 만든다.
    for(int i = 0; i < arr.length; i++) {
      heap.add(arr[i]);
    }

    // 힙에서 우선순위가 가장 높은 원소(root노드)를 하나씩 뽑는다.
```

```
    for(int i = 0; i < arr.length; i++) {
      arr[i] = heap.poll();
    }


    System.out.print("\n 정렬 후 배열 : ");
    for(int val : arr) {
      System.out.print(val + " ");
    }

  }
}
```

! 문제점 : 별도의 공간을 마련하는 것은 메모리를 그만큼 따로 잡아줘야 하기 때문에 비효율적이다.

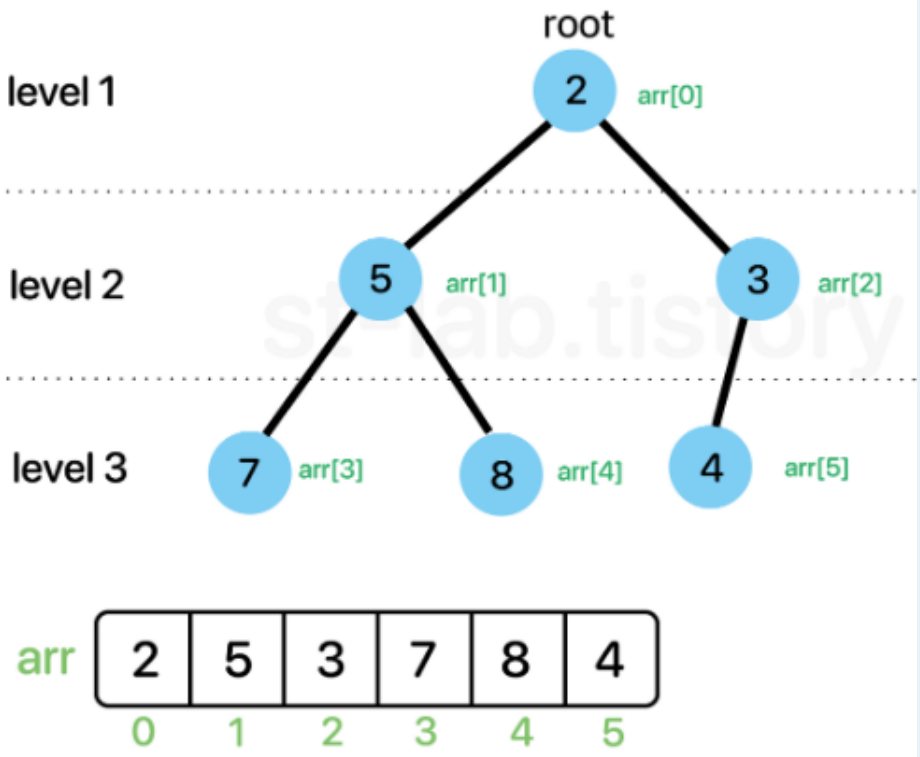
예시 코드를 힙 구조로 만들기

int[] arr = {3,7,5,4,2,8};
위 배열을 완전이진트리형태로 본다면 다음과 같다.



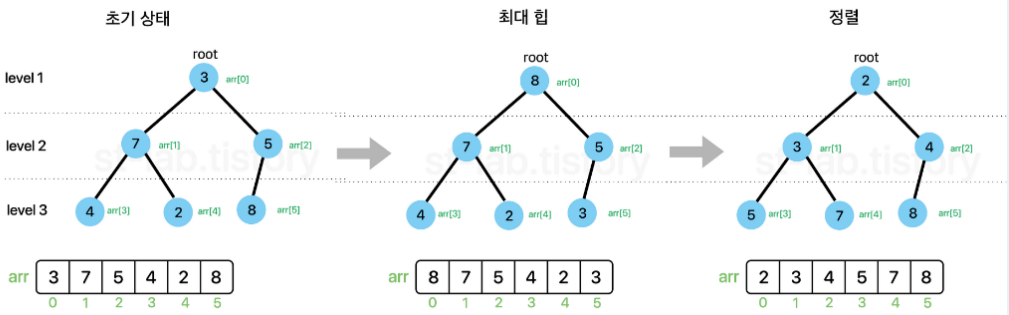
해당 트리를 최대 힙 조건을 만족하도록 재구성해야한다.
왜 최소 힙이 아닌 최대 힙 조건을 만족해야 할까?

힙은 부모노드와 자식 노드의 우선순위에 있어 항상 부모노드가 자식노드보다 높다.
이때 형제노드 사이에서의 우선순위는 고려되지 않는다.
즉, 힙 구조는 반 정렬상태이지 완전 정렬상태가 아니다.
바로 이 점 때문에 오름차순으로 구현할 시, 최소힙으로 정렬하여 쓰게 되면 형제간 우선순위는 고려되지 않아 정렬이 되지 않는다.



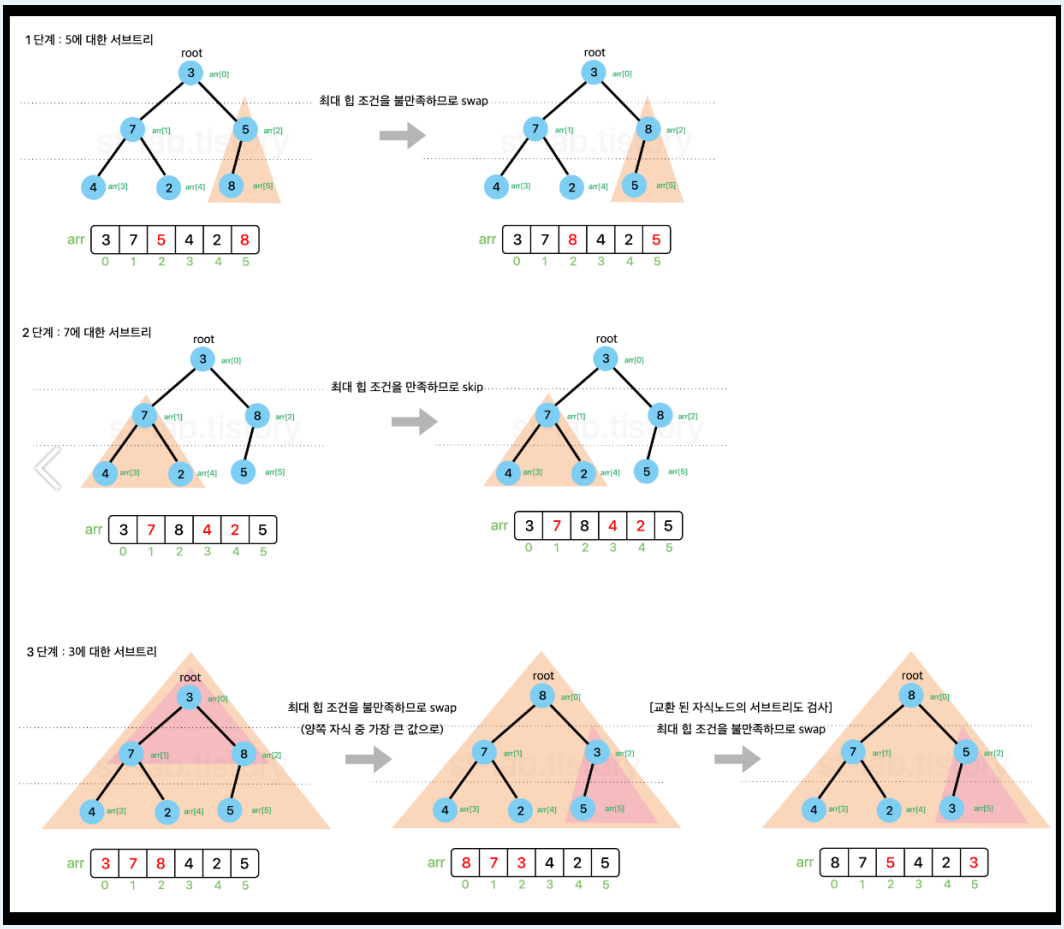
잘못된 정렬

결과적으로 최종 결과물이 정렬된 형태를 원하기 때문에 한 과정을 추가하는 것이다. 최대힙을 구성하여 root노드가 항상 큰 값을 갖게 만들고 그렇게 만들어진 최대힙에서 root노드 값을(큰 값)을 하나씩 삭제하며 뒤에서부터 배치하는 것이다.



과정 1: 최대 힙 만들기

초기 상태의 배열 자체를 별도의 배열 선언없이 최대 힙으로 만들어야 한다.
가장 작은 서브트리부터 최대 힙을 만족하도록 순차적으로 진행한다.



위와 같이 힙을 만드는 과정을 heapify라고 하며 부모노드부터 자식노드로 진행되기 때문에 sift-down과정이라고도 한다.

가장 마지막 노드의 부모노드 서브트리를 검사하는 코드를 만들어 보자.

```
//부모 인덱스를 얻는 함수
private static int getParent(int child) {
return (child-1)/2;
}
//두 인덱스의 원소를 교환하는 함수
private static void swap(int[] a, int i, int j) {
int tmp= a[i];
a[i] = a[j];
a[j] = tmp;
}

//힙을 만드는 함수
private static void heapify(int[] a, int parentIdx, int lastIdx){
```

```
/*
*현재 트리에서 부모 노드의 자식노드 인덱스를 각각 구해준다.
*현재 부모 인덱스를 가장 큰 값을 갖고 있다고 가정한다.
*/
int leftChildIdx = 2 * parentIdx +1;
int rightChildIdx = 2 * parentIdx + 2;
int largestIdx = parentIdx;

/*
left child node와 비교

자식노드 인덱스가 끝의 원소 인덱스를 넘어가지 않으면서
현재 가장 큰 인덱스보다 왼쪽 자식노드의 값이 더 클경우
가장 큰 인덱스를 가리키는 largestIdx를 왼쪽 자식 노드인덱스로 바꿔준다.
*/
if(leftChildIdx <= lastIdx && a[largestIdx] < a[leftChildIdx]) {
largestIdx = leftChildIdx;
}

/*
     *  left right node와 비교
     *
     *  자식노드 인덱스가 끝의 원소 인덱스를 넘어가지 않으면서
     *  현재 가장 큰 인덱스보다 오른쪽 자식노드의 값이 더 클경우
     *  가장 큰 인덱스를 가리키는 largestIdx를 오른쪽 자식노드인덱스로 바꿔준다.
     *
     */
    if(rightChildIdx < lastIdx && a[largestIdx] < a[rightChildIdx]) {
      largestIdx = rightChildIdx;
    }

/*
largestIdx와 부모노드가같지 않다는 것은
위 자식 노드 비교 과정에서 현재 부모노드보다 큰 노드가 존재한다는 뜻이다.
그럴 경우 해당 자식 노드와 부모노드를 교환해주고
교환 된 자식노드를 부모노드로 삼은 서브트리를 검사하도록 재귀 호출 한다.
*/
if(parentIdx != largestIdx) {
swap(a, largestIdx, parentIdx);
heapify(a, largestIdx, lastIdx);
}
}
}
```

▼ 전체 트리 검사

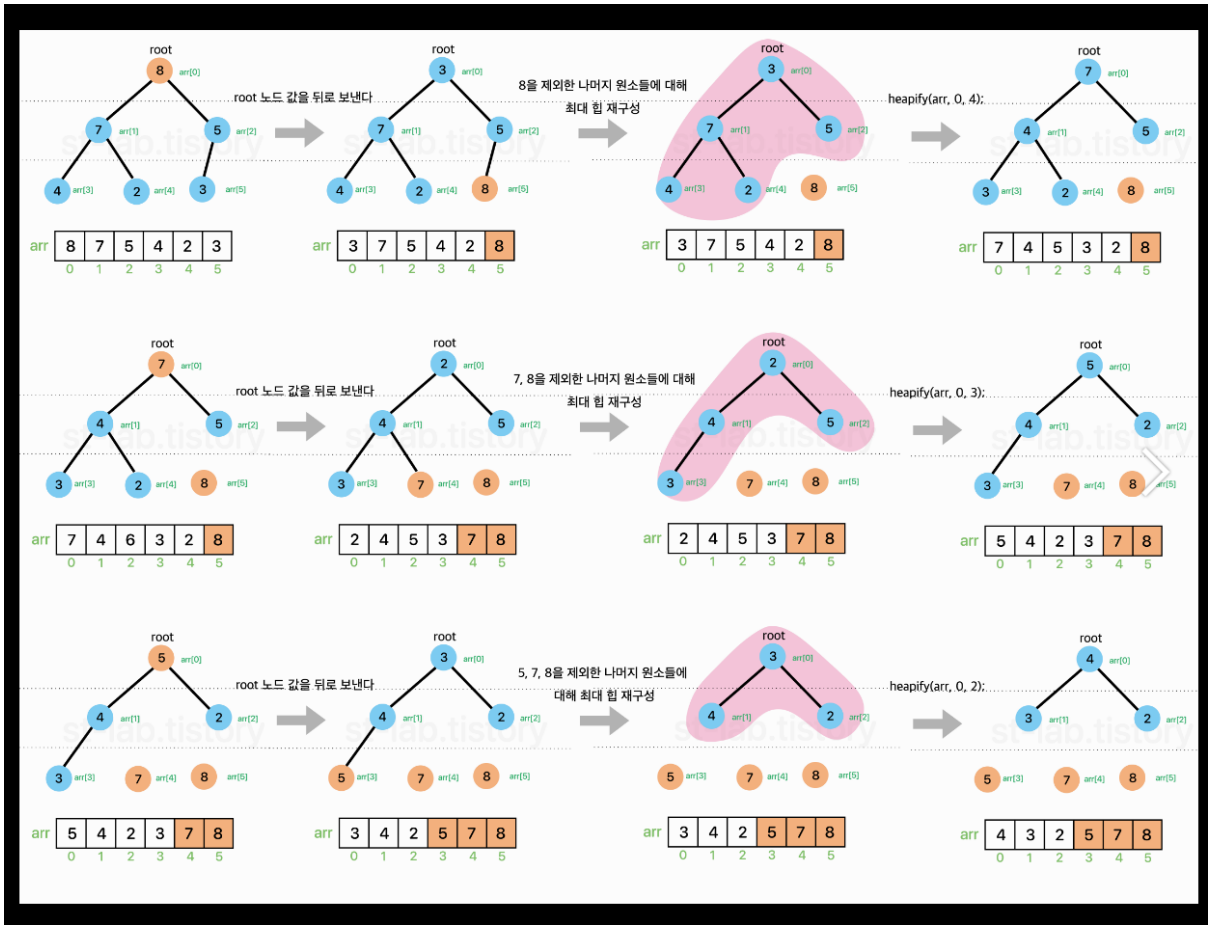
```
public static void heapsort(int[] arr) {
int size = arr.length;

/*
  부모노드와 heapify과정에서 음수가 발생하면 잘못 된 참조가 발생하기 때문에
```

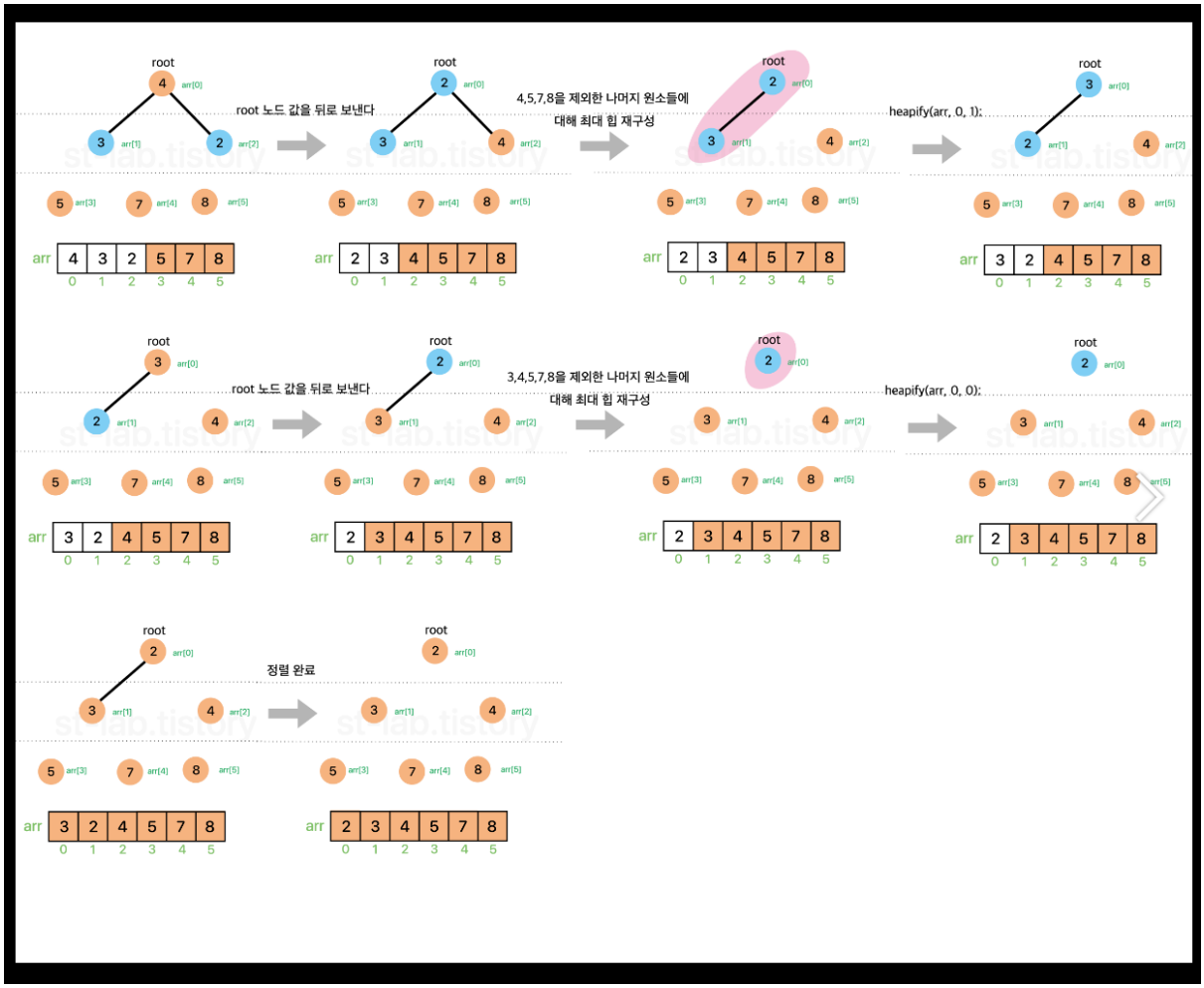
```
  원소가 1이거나 0일때에는 바로 종료한다.
*/
if(size<2) {
return;
}

//가장 마지막 노드의 부모 노드 인덱스
int parentIdx = getParent(size-1);

//max heap만들기
for(int i= parentIdx; i>=0; i--) {

//부모 노드(i값)을 1씩 줄이면서 heap조건을 만족시키도록 재구성한다.
heapify(a,i,size-1);
  }
}

private static void heapify(int[] a, int parentIdx, int lastIdx){
위의 내용과 동일
}
```

과정2 : 정렬하기

위 과정1을 마치면 거의 끝난 상태이다.
최대 힙을 만들었으면 이제 오름차순으로 정렬해야 한다.
힙은 최댓값 혹은 최솟값을 빠르게 찾기 위한 자료구조이다. 따라서 root노드는 항상 최댓값을 갖고 있다. 그 점을 이용하여 최댓값을 하나씩 삭제하면서 배열의 맨뒤부터 채워 나간다.

1. root원소를 배열의 뒤로 보낸다.
2. 뒤에 채운 원소를 제외한 나머지 배열 원소들을 다시 최대힙을 만족하도록 재구성한다.


root
root 노드 값을 뒤로 보낸다
8을 제외한 나머지 원소들에 대해
최대 힙 재구성
heapify(arr, 0, 4);
arr 8 7 5 4 2 3
arr 3 7 5 4 2 8
arr 3 7 5 4 2 8
arr 7 4 5 3 2 8
root
root 노드 값을 뒤로 보낸다
7, 8을 제외한 나머지 원소들에 대해
최대 힙 재구성
heapify(arr, 0, 3);
arr 7 4 6 3 2 8
arr 2 4 5 3 7 8
arr 2 4 5 3 7 8
arr 5 4 2 3 7 8
root
root 노드 값을 뒤로 보낸다
5, 7, 8을 제외한 나머지 원소들에
대해 최대 힙 재구성
heapify(arr, 0, 2);
arr 5 4 2 3 7 8
arr 3 4 2 5 7 8
arr 3 4 2 5 7 8
arr 4 3 2 5 7 8



▼ HeapSort 코드 구현

```
public static void heapSort(int[] a) {
  int size = a.length;

if(size <2) return;

int parentIdx = getParent(size-1);

for(int i=parentIdx ; i>=0 ; i--) {
heapify(a, i , size-1);
}

for(int i= size; i>0;i--) {
/*
root인 0번째 인덱스와 i번쨰 인덱스의 값을 교환해준 뒤
0~(i-1)까지의 부분트리에 대해 max heap을 만족하도록 재구성한다.
swap(a, 0 , i);
heapigy(a, 0 , i-1);
}
}
```

```
private static void heapify(int[] a, int parentIdx, int lastIdx){

/*
heapify내용
*/
}
}
```

💡 힙을 구성하는 단계에서 시간이 많이 소요될 것 같지만 O(nlogn)의 시간 복잡도를 갖는다.

▼ 반복문을 사용한 heapify

```
private static void heapify(int[] a, int parentIdx, int lastIdx) {
int leftChildIdx;
int rightChildIdx;
int largestIdx;

/*
현재 부모 인덱스의 자식 노드 인덱스가
마지막 인덱스를 넘지 않을 때 까지 반복한다.

이때 왼쪽 자식 노드를 기준으로 해야 한다.
오른쪽 자식 노드를 기준으로 범위를 갖고 검사하면
마지막 부모 인덱스가 왼쪽 자식만 갖고 있을 경우
비교 교환을 할 수 없다.
*/

while(parentIdx *2 ) +1 <= lastIdx) {
leftChildIdx = (parentIdx *2 +1;
rightChildIdx = (parentIdx *2 ) +2;
largestIdx = parentIdx;

if(a[leftChildIdx] > a[largestIdx]) largestIdx = leftChildIdx;

if(a[rightChildIdx] > a[largestIdx]) largestIdx = rightChildIdx;

if(rightChildIdx <= lastIdx && a[rightChildIdx] > a[largestIdx]{
largestIdx = rightChildIdx;
}

if(largestIdx != parentIdx) {
swap(a, parentIdx, largestIdx);
parentIdx = largestIdx;
}
else{
return;
}
}
```

```
}
}
```