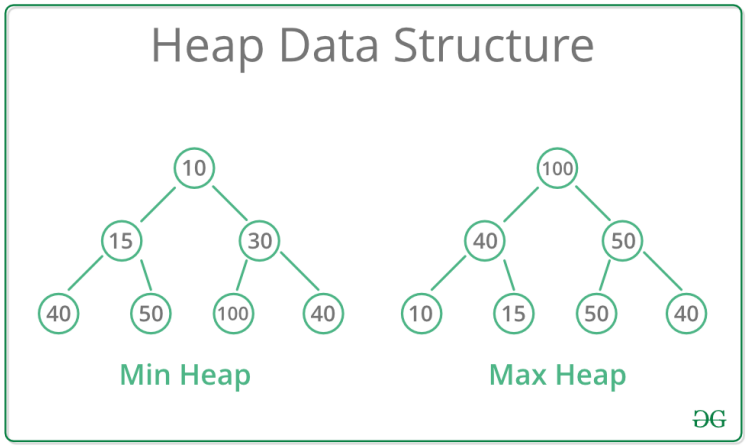
# Heap

- **완전 이진 트리**에 있는 노드 중에서 키 값이 가장 큰 노드나 키 값이 가장 작은 노드를 찾기 위해 만든 자료구조
- 시간 복잡도 : log2N




## 최대 힙

- 키 값이 가장 큰 노드를 찾기 위한 완전 이진 트리
- 부모 노드의 키 값 > 자식 노드의 키 값
- 루트 노드 : 키 값이 가장 큰 노드
- 특징
  - 힙 정렬 : 데이터를 계속 빼내면 내림차순으로 숫자가 나오게 된다.

## 최소 힙

- 키 값이 가장 작은 노드를 찾기 위한 완전 이진 트리
- 부모 노드의 키 값 < 자식 노드의 키 값
- 루트 노드 : 키 값이 가장 작은 노드
- 특징
  - 힙 정렬 : 데이터를 계속 빼내면 오름차순으로 숫자가 나오게 된다.

## 우선순위 큐(Priority Queue)

- 우선순위 큐

    - 일반 큐처럼 FIFO 순서가 아니라, 우선순위가 높은(최대, 최소) 순서대로 먼저 나간다.
- java.util.PriorityQueue
    - Heap 자료구조로 되어 있다.(힙을 사용하는 게 우선순위 큐를 구현하는 가장 효율적인 방법)
        - 노드 하나의 추가/삭제 시간 복잡도가 O(logN)
        - 최대값/최소값을 O(1)에 구할 수 있다.
    - Comparator 혹은 Comparable 인터페이스를 구현해야 최대힙, 최소힙에 따른 원소들의 순서를 유지할 수 있다.
- 배열을 이용해 트리 형태를 쉽게 구현 가능
    - `n` 노드의 자식은 `2n`, `2n+1`

## Comparable vs Comparator

- Comparable
    - 원소 스스로가 타원소와 비교해서 결과를 준다.
- Comparator
    - 두 원소를 비교해서 판단을 내려줌

## 힙 삽입

- 원소 삽입
    - 새로운 노드가 추가되면 완전 이진 트리의 마지막 노드로 추가
    - 크기 비교를 위해 부모 노드와 비교, 부모보다 자식이 더 크면 스왑
    - 크기 비교 할 때마다 비교 할 게 절반씩 줄어듦으로 시간 복잡도가 log 2의 N이다.

```
public void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
  }

  public void offer(int data) {
    if (isFull()) {
        increaseCapacity(); // 처음 생성한 배열의 크기가 꽉 찼을 때
    }
    elements[++pos] = data;  // 힙크기 하나 증가하고 마지막 노드에 data 삽입
    int idx = pos; // 현재 위치 변수 설정
    while (idx > 1 && elements[idx] > elements[idx / 2]) { // 자식것이 크면 교환
```

```
      swap(elements, idx, idx / 2);
      idx /= 2;
    }
  }
```

## 힙 삭제

- 원소 삭제
  - 힙에서는 루트 노드만을 (꺼내면서) 삭제할 수 있음
  - 맨 마지막의 노드를 루트 자리에 옮긴 후 맨 마지막 노드를 삭제
  - 작은 값이 루트에 있는 상태이므로 루트와 자식 노드 비교 후 스왑하는 작업 필요

```
public int poll() {
    if (pos == 0) {
      return -1;  // 데이터 없을때
    }
    int result = elements[1]; // 루트반환 - 현재 가장 큰 값
    elements[1] = elements[pos]; // 마지막 데이터를 루트 위치값으로 설정
    elements[pos] = 0; // 마지막 데이터를 지움
    pos--; // 사이즈 하나 줄임

    // 힙을 재정렬한다.
    heapify();
    return result;
  }

  public void heapify() { // 내부의 데이터를 heap 구조로 만드는 작업 (log N)
    // 첫번째 있던 정보와 그 자식노드와 비교해서 자식노드가 크면 그 노드와 변경하면서 가장 아래 level까지한다.
    int idx = 1;
    while (idx * 2 <= pos) {
      if (elements[idx] >= elements[idx * 2] && elements[idx] >= elements[idx * 2 + 1]) {
        break;
      }

      // 그렇지 않으면 두 자식 중에 큰 값을 가진것과 교환한다.
      if (elements[idx * 2] > elements[idx * 2 + 1]) {
        swap(elements, idx, idx * 2);
        idx = idx * 2;
      } else {
        swap(elements, idx, idx * 2 + 1);
        idx = idx * 2 + 1;
      }
    }
  }
```