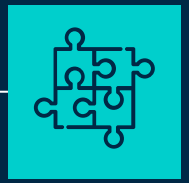
# Data structure: Heap

자람 38기 정성우

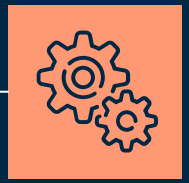
# 목차



01

## Heap이란?

Heap의 ADT
Heap의 특징
min/max heap



02

## Heap의 삽입/삭제

Upheap
Downheap

03

## Heap의 구현과 활용

구현 방법
활용 예

# Heap이란?

Heap의 ADT
Heap의 특징
min/max heap

01

# Heap이란?

## ADT(Abstract Data Type)의 정의

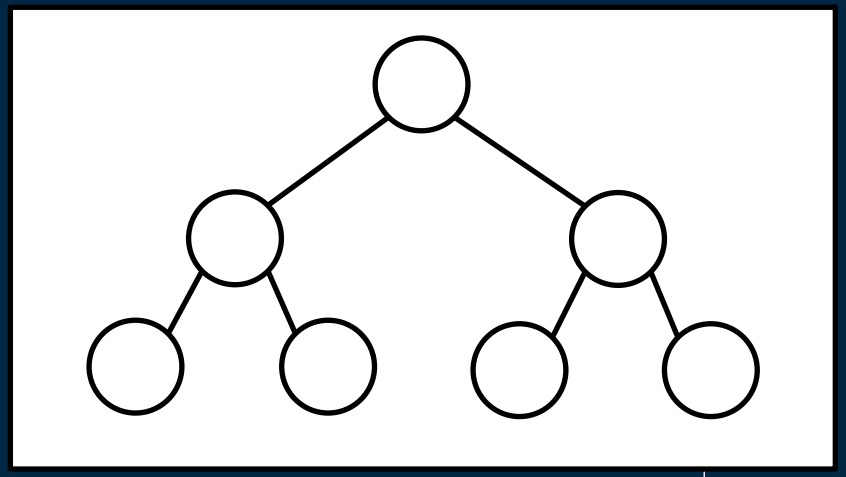
컴퓨터 과학에서 자료들과
그 자료들에 대한 연산들을 명기한 것.

구현 방법을 명시하지 않음.
이는 프로그래머의 몫.

# Heap이란?

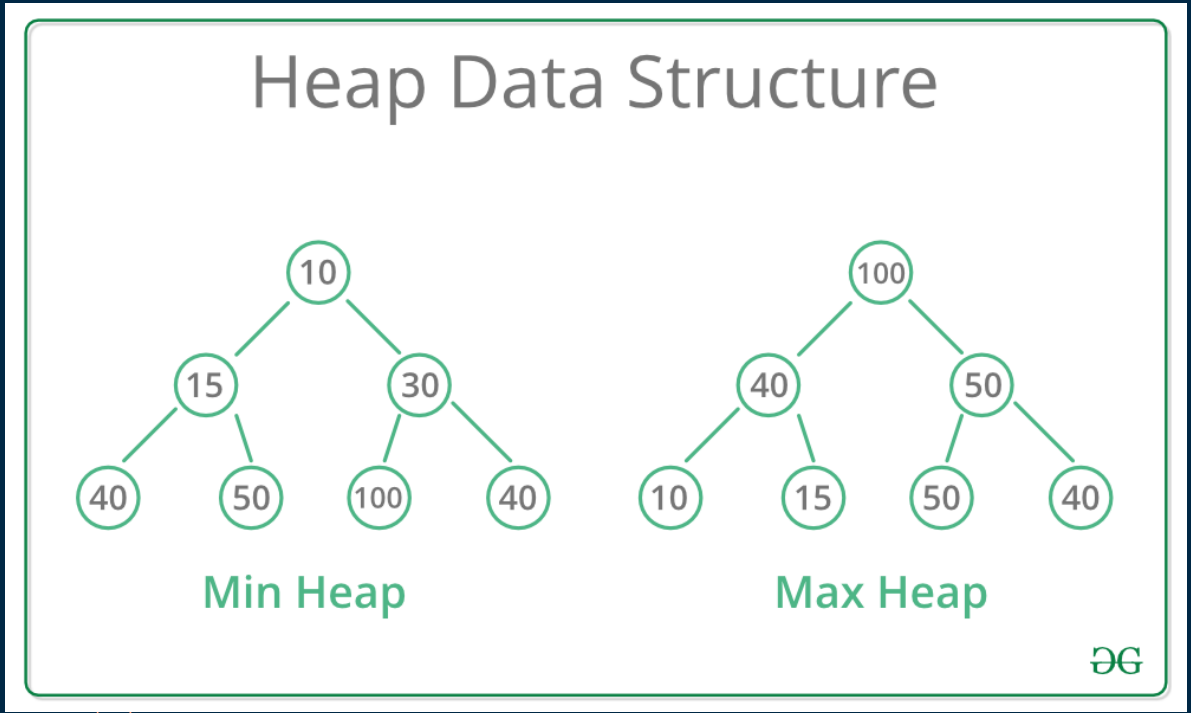
**Structural property:**
완전 이진 트리(Complete Binary Tree)

**Relational property:**
Min heap: 자식 노드의 값이 부모 노드의 값보다 크거나 같아야함
Max heap: 자식 노드의 값이 부모 노드의 값보다 작거나 같아야함



Height <= O(log n)

# Heap이란?


Heap Data Structure
10
15
30
40
50
100
40
Min Heap
100
40
50
10
15
50
40
Max Heap

# Heap의 삽입/삭제

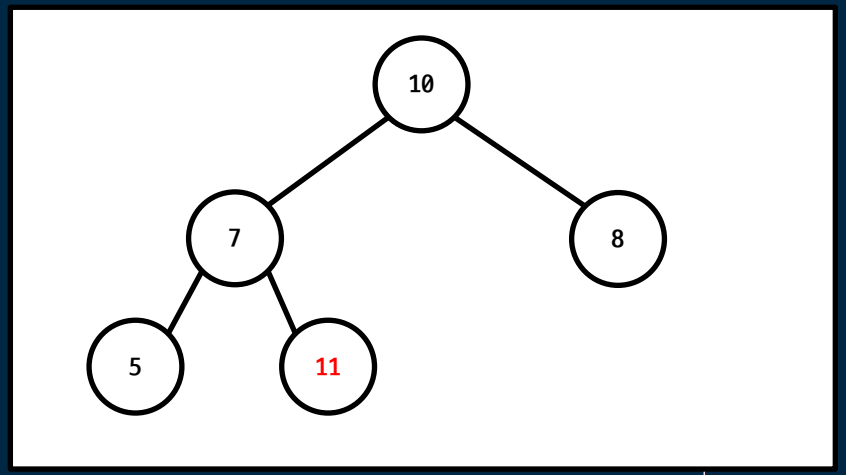
02

Upheap
Downheap

# Heap의 삽입

**Structural property:**
삽입시 반드시 Complete binary tree를 유지하며 삽입해야함

**Relational property:**
Structural property를 유지하기 위한 삽입에 따라 Relational property가 깨짐

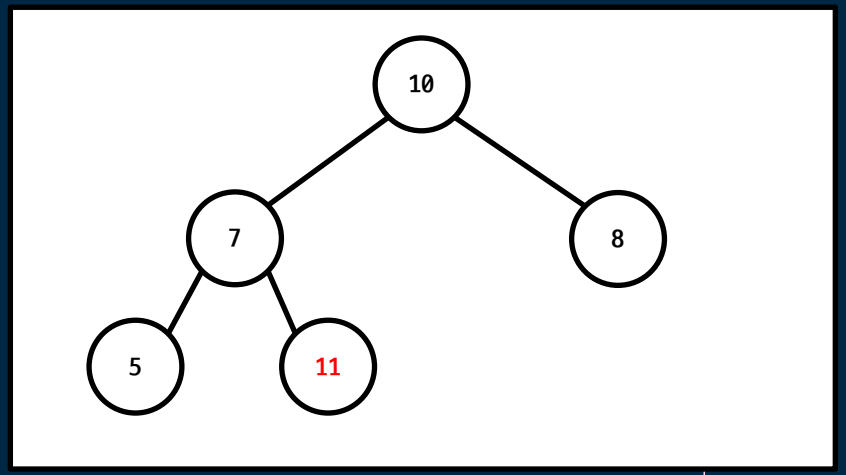


# Heap의 삽입

## Upheap 연산

삽입 후 깨진 Relational property를 회복하기 위한 연산

삽입된 노드로부터 출발하여 부모 노드의 값과 재귀적으로 비교하여
min heap: 부모가 더 크면 바꾸기
max heap: 부모가 더 작으면 바꾸기
Relational property를 회복시켜주어야함

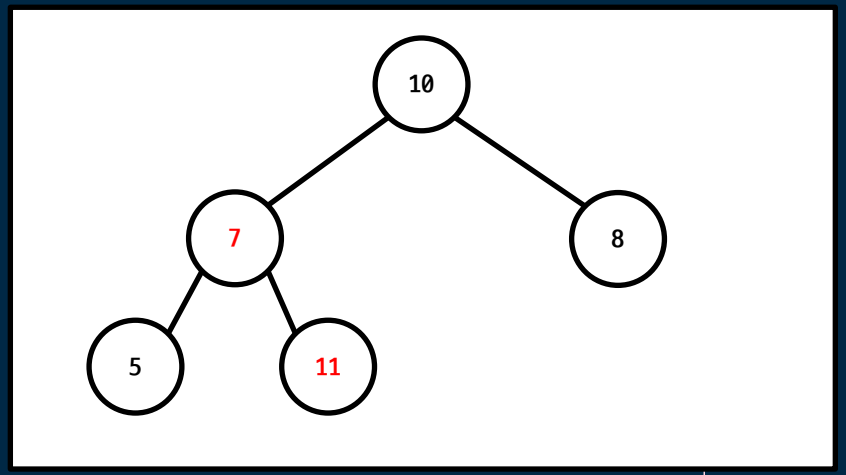


# Heap의 삽입

## Upheap 연산

삽입 후 깨진 Relational property를 회복하기 위한 연산

삽입된 노드로부터 출발하여 부모 노드의 값과 재귀적으로 비교하여
min heap: 부모가 더 크면 바꾸기
max heap: 부모가 더 작으면 바꾸기
Relational property를 회복시켜주어야함

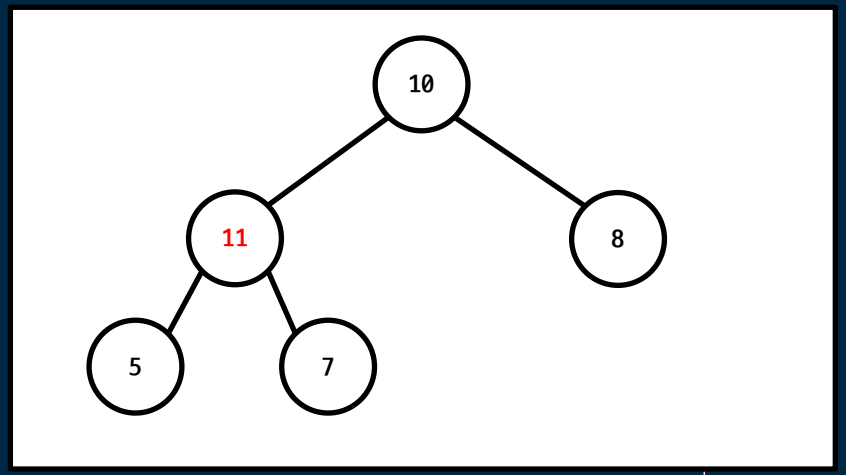


# Heap의 삽입

## Upheap 연산

삽입 후 깨진 Relational property를 회복하기 위한 연산

삽입된 노드로부터 출발하여 부모 노드의 값과 재귀적으로 비교하여
min heap: 부모가 더 크면 바꾸기
max heap: 부모가 더 작으면 바꾸기
Relational property를 회복시켜주어야함

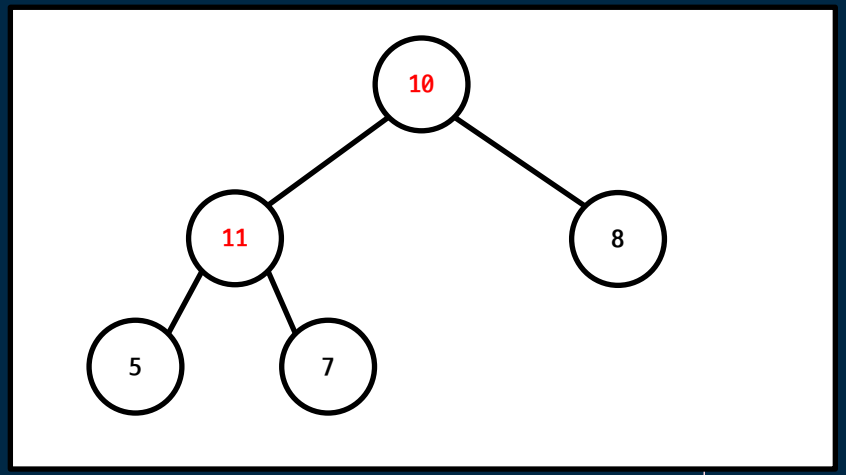


# Heap의 삽입

## Upheap 연산

삽입 후 깨진 Relational property를 회복하기 위한 연산

삽입된 노드로부터 출발하여 부모 노드의 값과 재귀적으로 비교하여
min heap: 부모가 더 크면 바꾸기
max heap: 부모가 더 작으면 바꾸기
Relational property를 회복시켜주어야함

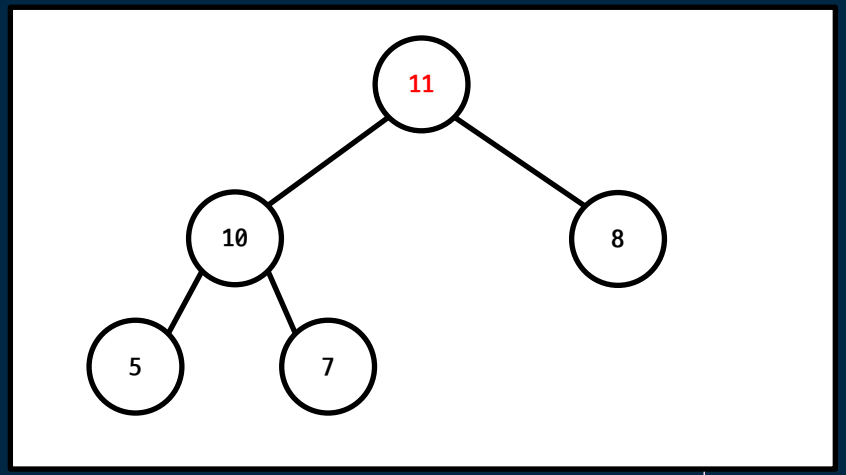


# Heap의 삽입

## Upheap 연산

삽입 후 깨진 Relational property를 회복하기 위한 연산

삽입된 노드로부터 출발하여 부모 노드의 값과 재귀적으로 비교하여
min heap: 부모가 더 크면 바꾸기
max heap: 부모가 더 작으면 바꾸기
Relational property를 회복시켜주어야함

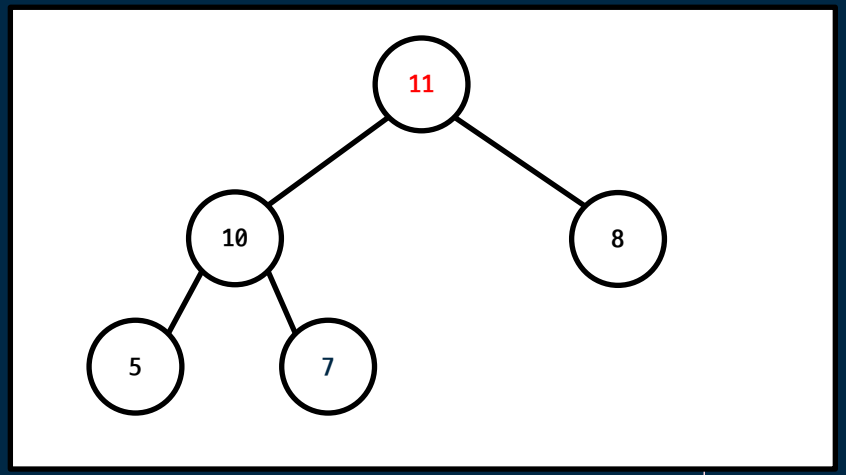



Running time: O(log n)

# Heap의 삭제

**Structural property:**
삭제되는 노드 자체는 반드시 Complete binary tree가 유지될 수 있도록 마지막 노드를 삭제해야 함

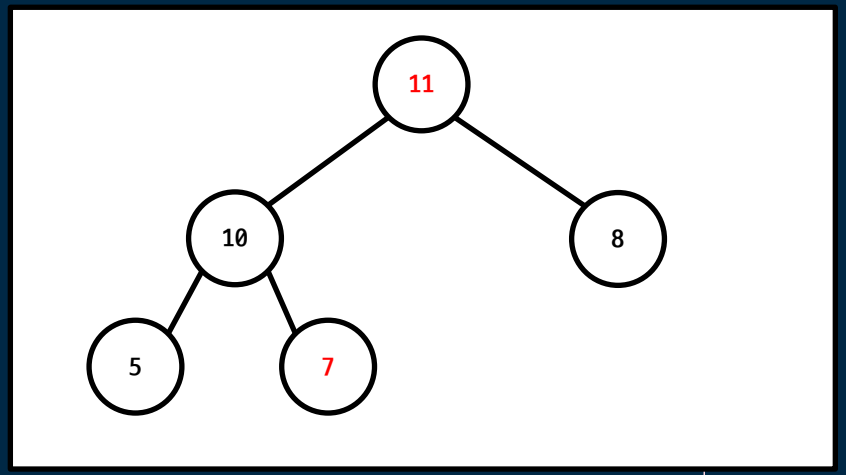
**Relational property:**
삭제되는 값 자체는 반드시 루트의 값으로, 그렇기에 실제 삭제되는 노드와 루트의 값을 바꾸고 삭제하여 Relational property가 깨짐



# Heap의 삭제

**Structural property:**
삭제되는 노드 자체는 반드시 Complete binary tree가 유지될 수 있도록 마지막 노드를 삭제해야 함

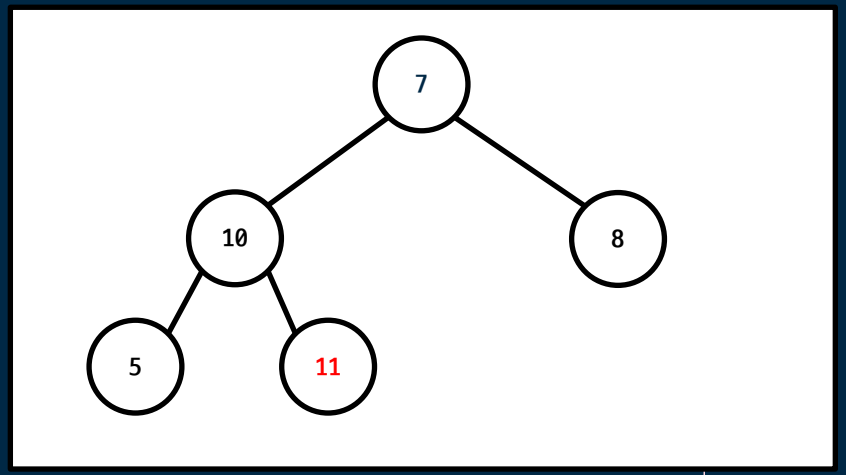
**Relational property:**
삭제되는 값 자체는 반드시 루트의 값으로, 그렇기에 실제 삭제되는 노드와 루트의 값을 바꾸고 삭제하여 Relational property가 깨짐



# Heap의 삭제

**Structural property:**
삭제되는 노드 자체는 반드시 Complete binary tree가 유지될 수 있도록 마지막 노드를 삭제해야 함

**Relational property:**
삭제되는 값 자체는 반드시 루트의 값으로, 그렇기에 실제 삭제되는 노드와 루트의 값을 바꾸고 삭제하여 Relational property가 깨짐

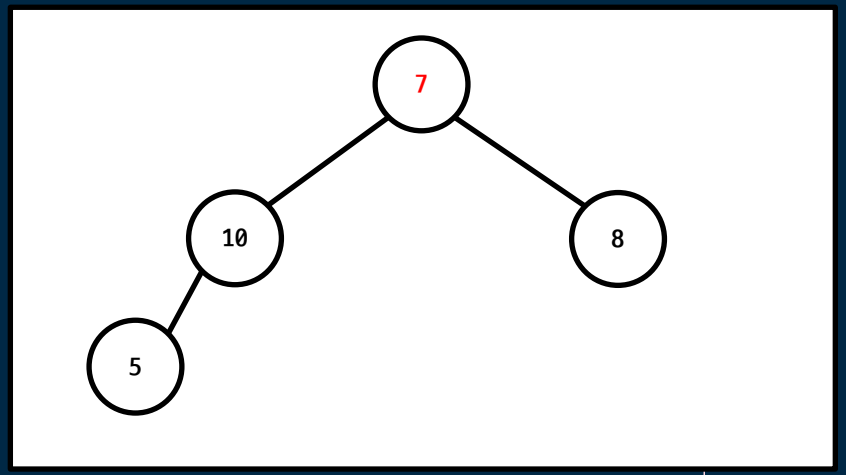


# Heap의 삭제

## Downheap 연산

삭제 후 깨진 Relational property를 회복하기 위한 연산

루트 노드로부터 출발하여 두 자식 노드의 값들과 재귀적으로 비교하여
min heap: 가장 작은 자식과 바꾸기
max heap: 가장 큰 자식과 바꾸기
Relational property를 회복시켜주어야함

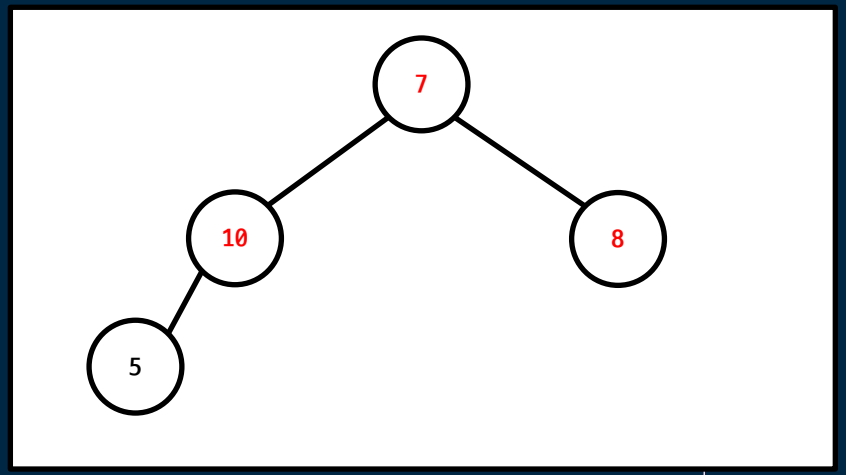


# Heap의 삭제

## Downheap 연산

삭제 후 깨진 Relational property를 회복하기 위한 연산

루트 노드로부터 출발하여 두 자식 노드의 값들과 재귀적으로 비교하여
min heap: 가장 작은 자식과 바꾸기
max heap: 가장 큰 자식과 바꾸기
Relational property를 회복시켜주어야함

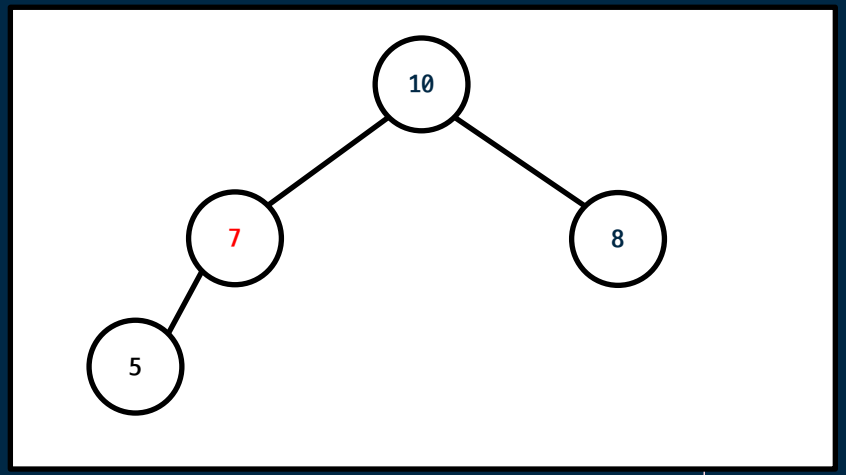


# Heap의 삭제

## Downheap 연산

삭제 후 깨진 Relational property를 회복하기 위한 연산

루트 노드로부터 출발하여 두 자식 노드의 값들과 재귀적으로 비교하여
min heap: 가장 작은 자식과 바꾸기
max heap: 가장 큰 자식과 바꾸기
Relational property를 회복시켜주어야함

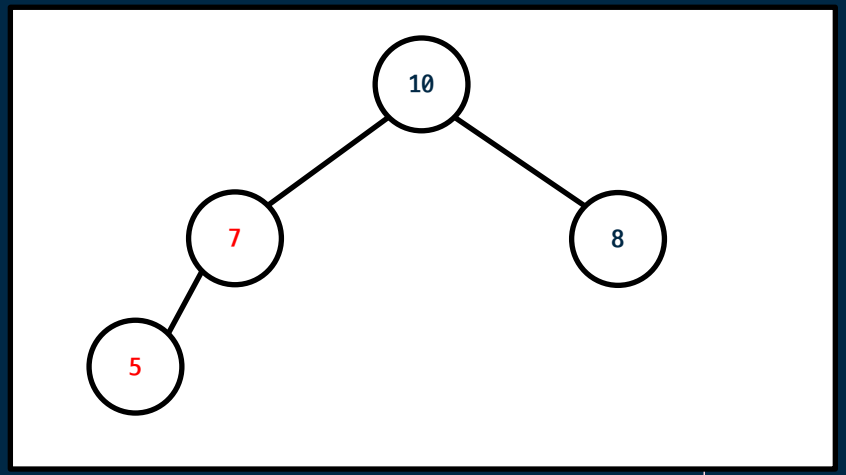


# Heap의 삭제

## Downheap 연산

삭제 후 깨진 Relational property를 회복하기 위한 연산

루트 노드로부터 출발하여 두 자식 노드의 값들과 재귀적으로 비교하여
min heap: 가장 작은 자식과 바꾸기
max heap: 가장 큰 자식과 바꾸기
Relational property를 회복시켜주어야함

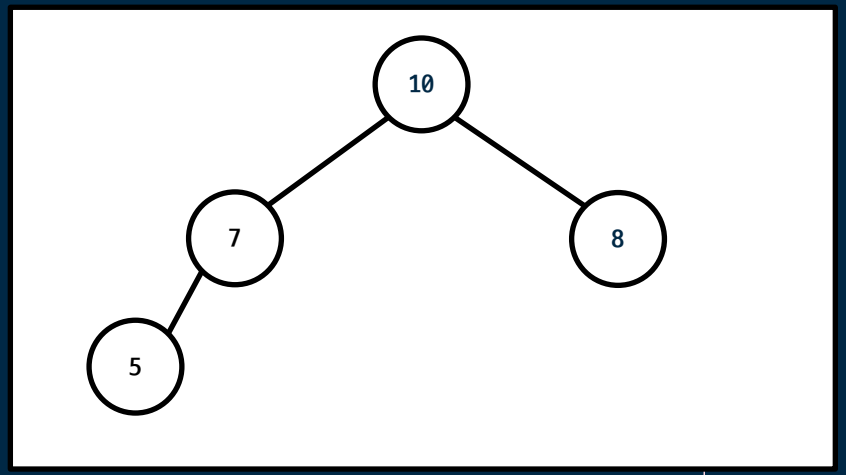


# Heap의 삭제

## Downheap 연산

삭제 후 깨진 Relational property를 회복하기 위한 연산

루트 노드로부터 출발하여 두 자식 노드의 값들과 재귀적으로 비교하여
min heap: 가장 작은 자식과 바꾸기
max heap: 가장 큰 자식과 바꾸기
Relational property를 회복시켜주어야함




Running time: O(log n)

# Heap의 삭제: Heap 정렬

## Heap의 삭제는 반드시 루트에서만 일어난다.

즉 min heap은 오름차순으로 정렬
max heap은 내림차순으로 정렬됨

즉 Heap안의 모든 값을 정렬하기 위한 Running time은
일단 Heap에 값을 넣기 위한 삽입 연산(O(log n)) n번
수행 후 삭제 연산(O(log n))을 n번 수행하므로

## Heap 정렬 Running time: O(2n log n) = O(n log n)

# Heap의 구현과 활용

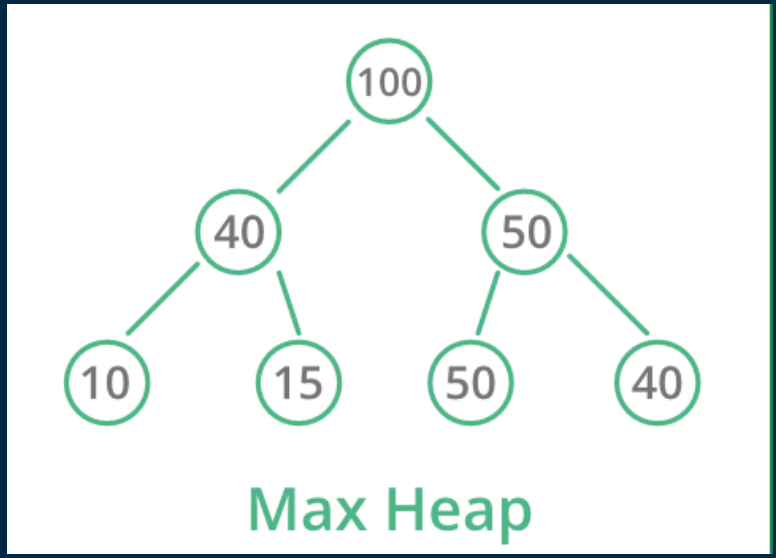
03

구현 방법
활용 예

# Heap의 구현 방법

1. 구조체를 이용한 구현
2. 배열을 이용한 구현

# Heap의 구현 방법: 구조체




Complete binary tree를 유지하기위해
어떤 노드의 자식으로 삽입/삭제 되어야하는지
그 노드의 주소를 어떻게 아는가?

# Heap의 구현 방법: 구조체

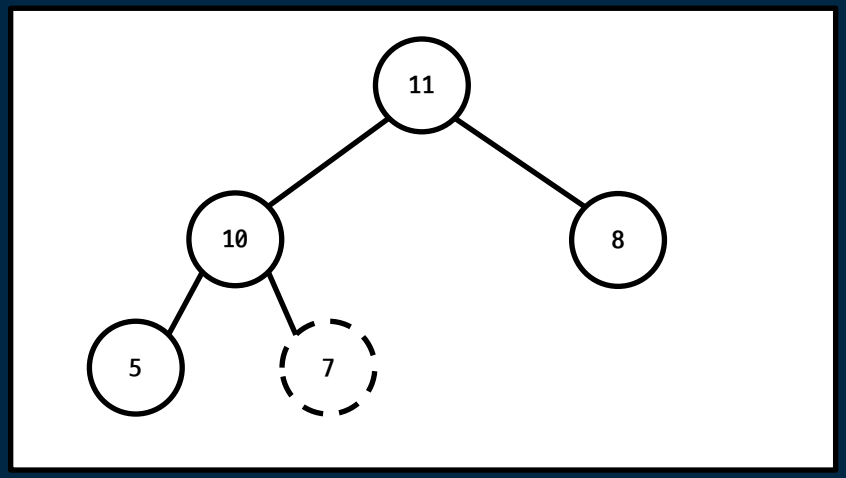
추가로 알아야할 변수

need_leaf: 포화 이진트리가 되기 위해 필요한 노드 수
current_leaf: 현재 leaf에 존재하는 노드 수

# Heap의 구현 방법: 구조체: 삽입

```
need_leaf: 4
current_leaf: 1
```

예시

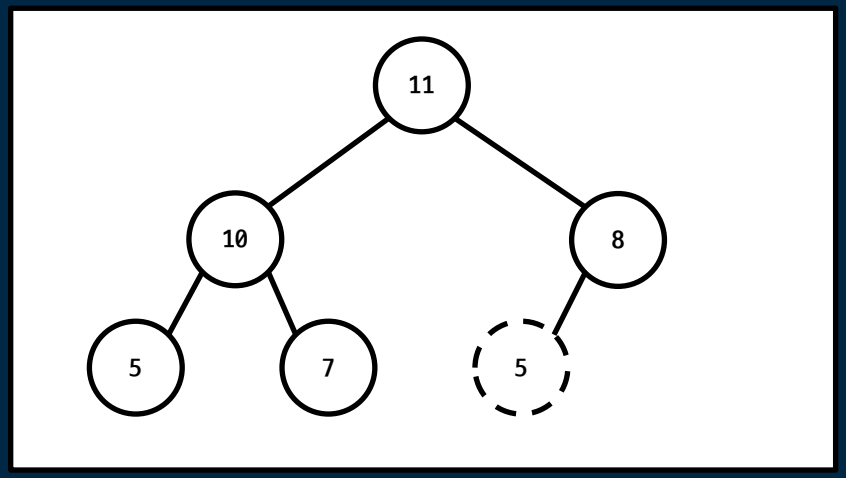



1. 4 / 2 = 2 > 1 (왼쪽)
2. 2 / 2 = 1 <= 1 (오른쪽)

# Heap의 구현 방법: 구조체: 삽입

```
need_leaf: 4
current_leaf: 2
```

예시

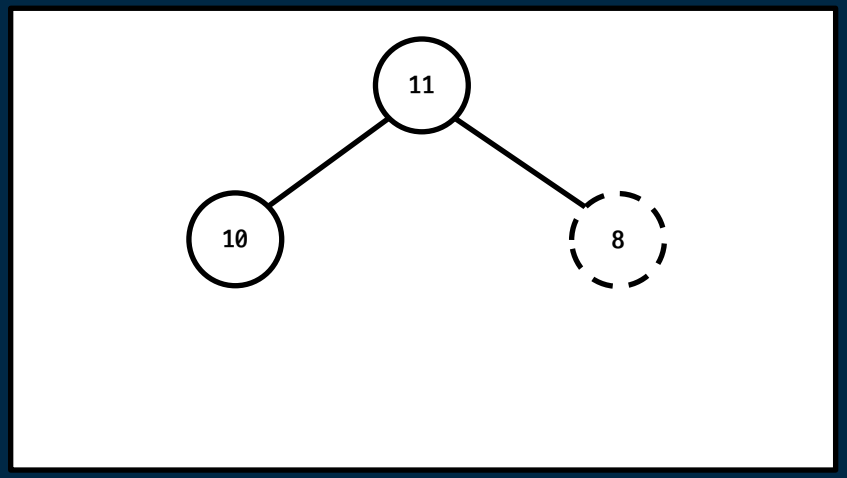



1. 4 / 2 = 2 <= 2 (오른쪽)
2. 2 / 2 = 1 > 2 - 2 = 0 (왼쪽)

# Heap의 구현 방법: 구조체: 삽입

need_leaf: 2
current_leaf: 1

예시

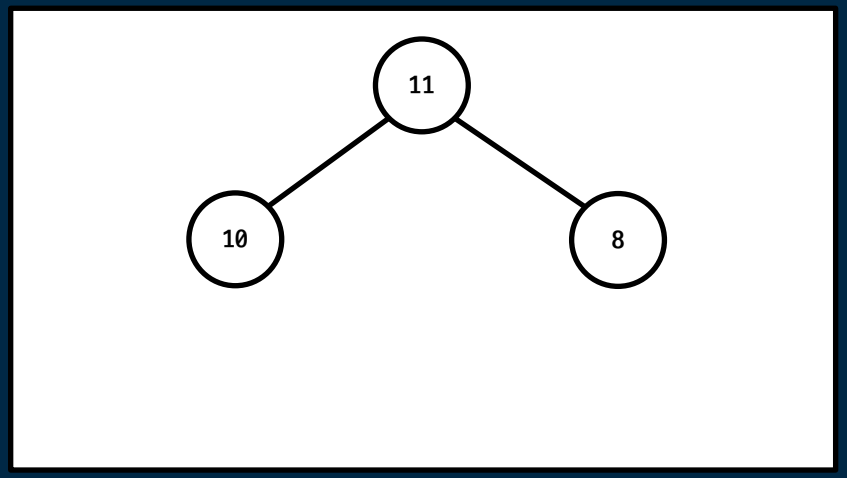



1. 2 / 2 = 1 <= 1 (오른쪽)

# Heap의 구현 방법: 구조체: 삽입

need_leaf: 4
current_leaf: 0

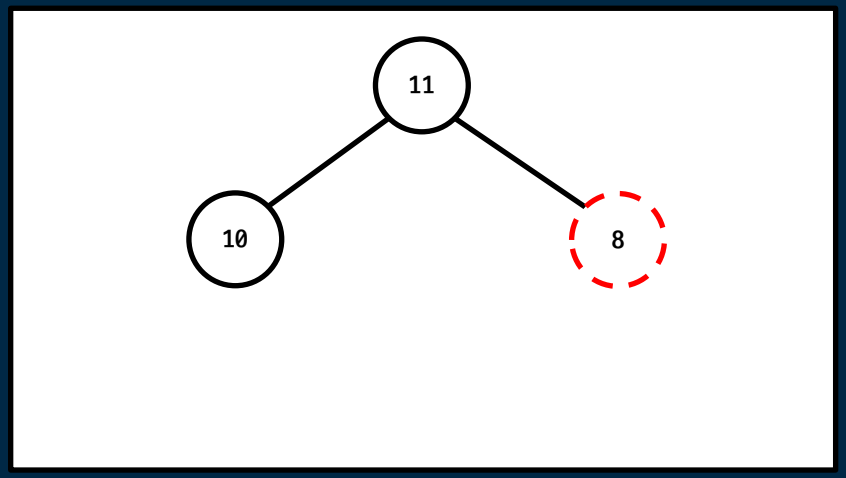
예시



# Heap의 구현 방법: 구조체: 삭제

```
need_leaf: 4
current_leaf: 0
```

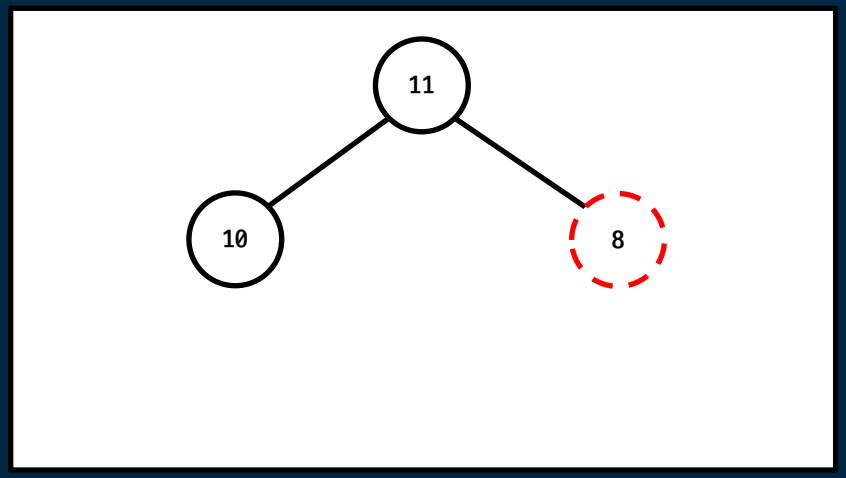
예시



# Heap의 구현 방법: 구조체: 삭제

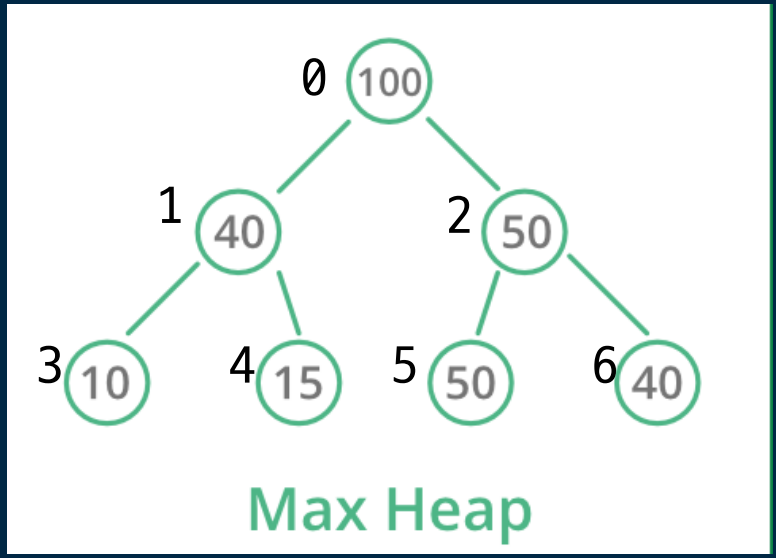
need_leaf: 2
current_leaf: 1

예시




1. 2 / 2 = 1 <= 1 (오른쪽)

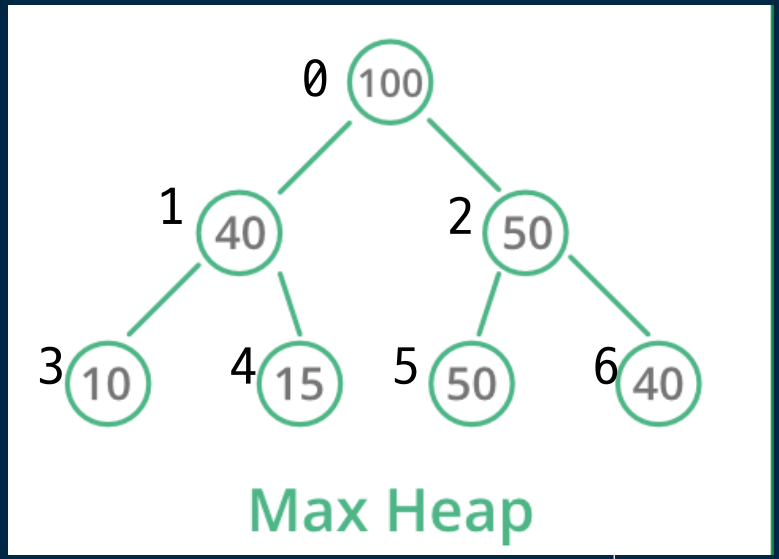
# Heap의 구현 방법: 배열




| 100 | 40 | 50 | 10 | 15 | 50 | 40 |
|---|---|---|---|---|---|---|

# Heap의 구현 방법: 배열: upheap

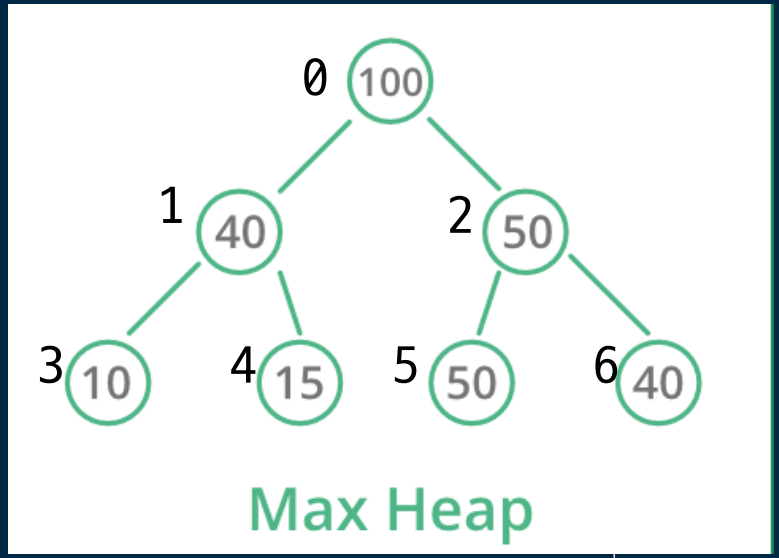
부모로 가기:
현재 index가 홀수일 경우: index / 2
현재 index가 짝수일 경우: index / 2 – 1




| 100 | 40 | 50 | 10 | 15 | 50 | 40 |
|---|---|---|---|---|---|---|

# Heap의 구현 방법: 배열: downheap

왼쪽 자식으로 가기: index * 2 + 1
오른쪽 자식으로 가기: index * 2 + 2




| 100 | 40 | 50 | 10 | 15 | 50 | 40 |
|---|---|---|---|---|---|---|

# Heap의 활용 및 효율

**활용 예:**

네트워크 트래픽 제어, 운영체제 작업 스케줄링 등
**우선순위**가 있는 작업들에 대한 처리시 유용

끝