

# 알튜비튜

## 우선순위 큐

오늘은 STL에서 제공하는 container adaptor인 priority queue에 대해 알아봅니다.  
가장 최근의 데이터를 뽑는 스택, 제일 먼저 들어간 데이터를 뽑는 큐와 달리 우선순위가 가장 높은 데이터를 뽑는 자료구조입니다.

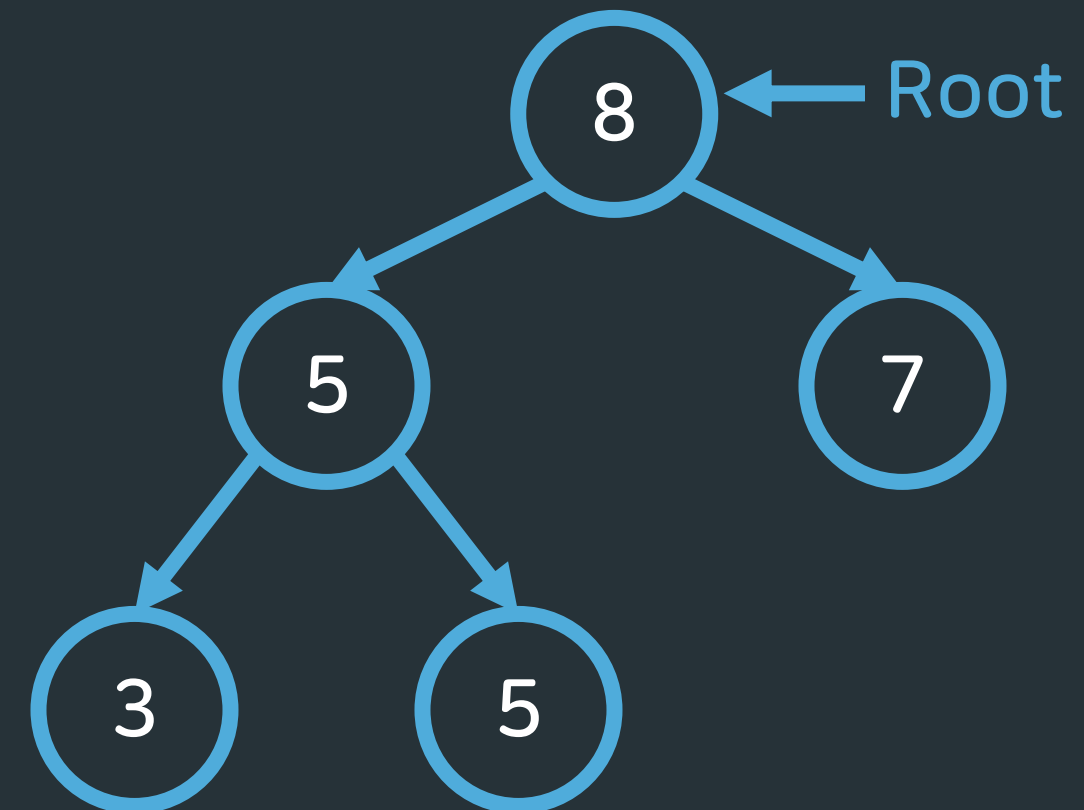


FAMILY  
EMERGENCY ROOM



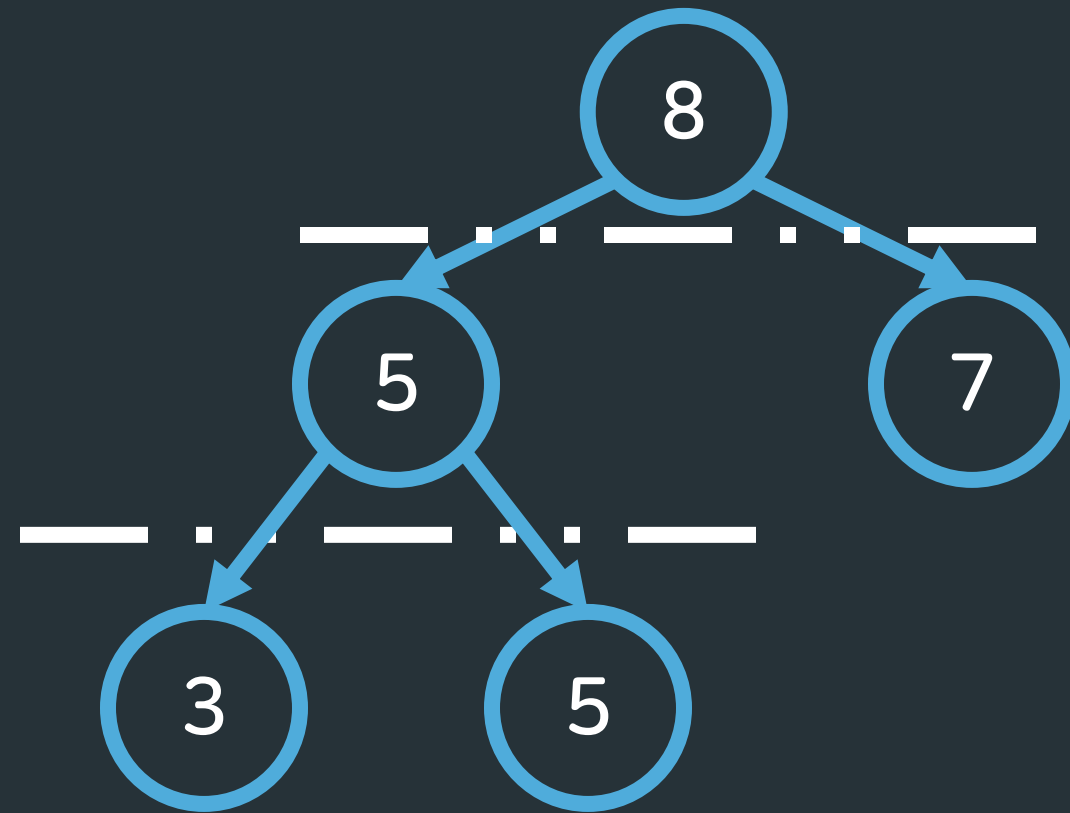
## Priority Queue

- 우선순위가 높은 데이터가 먼저 나옴
- 자료의 Root 노드에서만 모든 연산이 이루어짐
- 모든 연산에 대한 시간 복잡도는  $O(\log n)$
- Heap으로 구현
- Heap의 조건
  1. 완전 이진 트리
  2. 상위 노드의 값은 모든 하위 노드의 값보다 우선순위가 크거나 같다



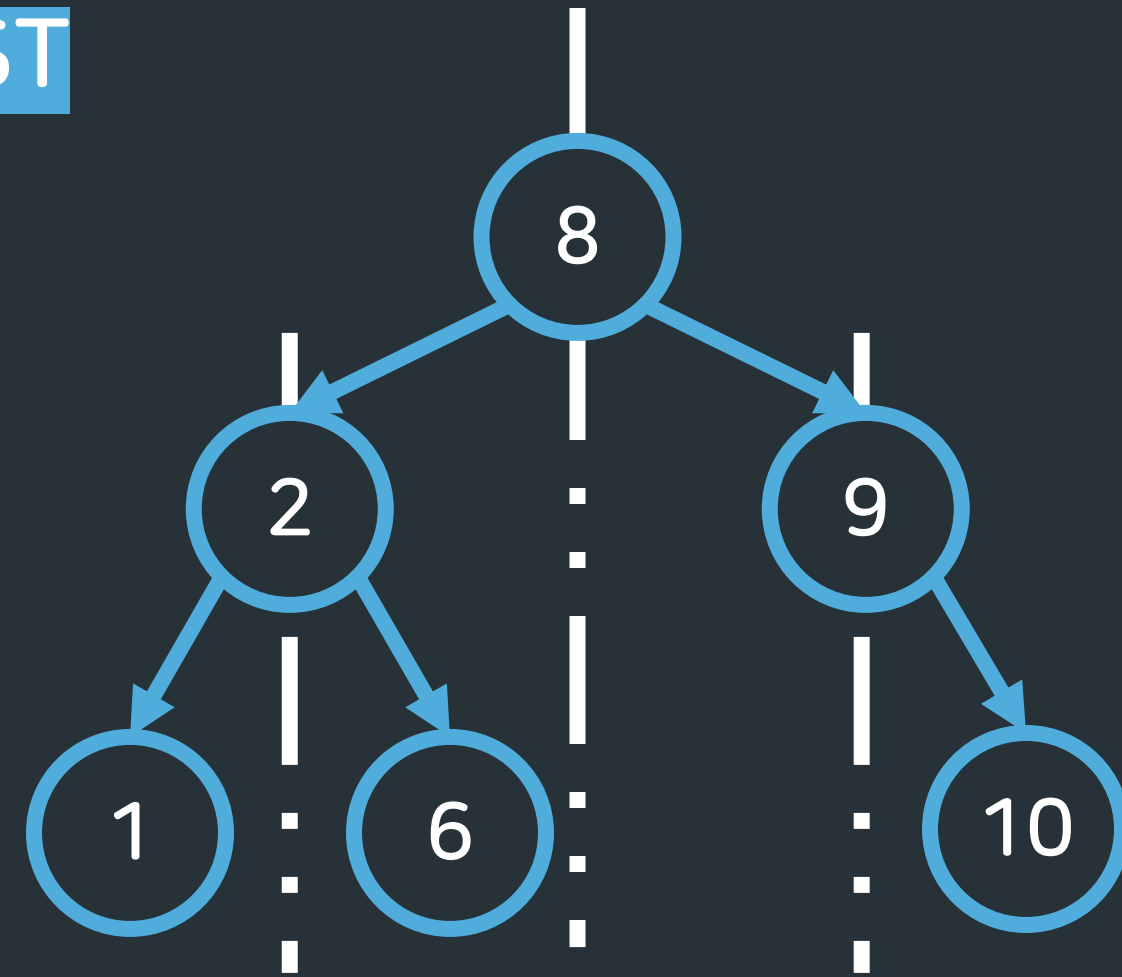
# Heap과 BST의 차이

Heap



(상위)  $\geq$  (하위)  
상하관계

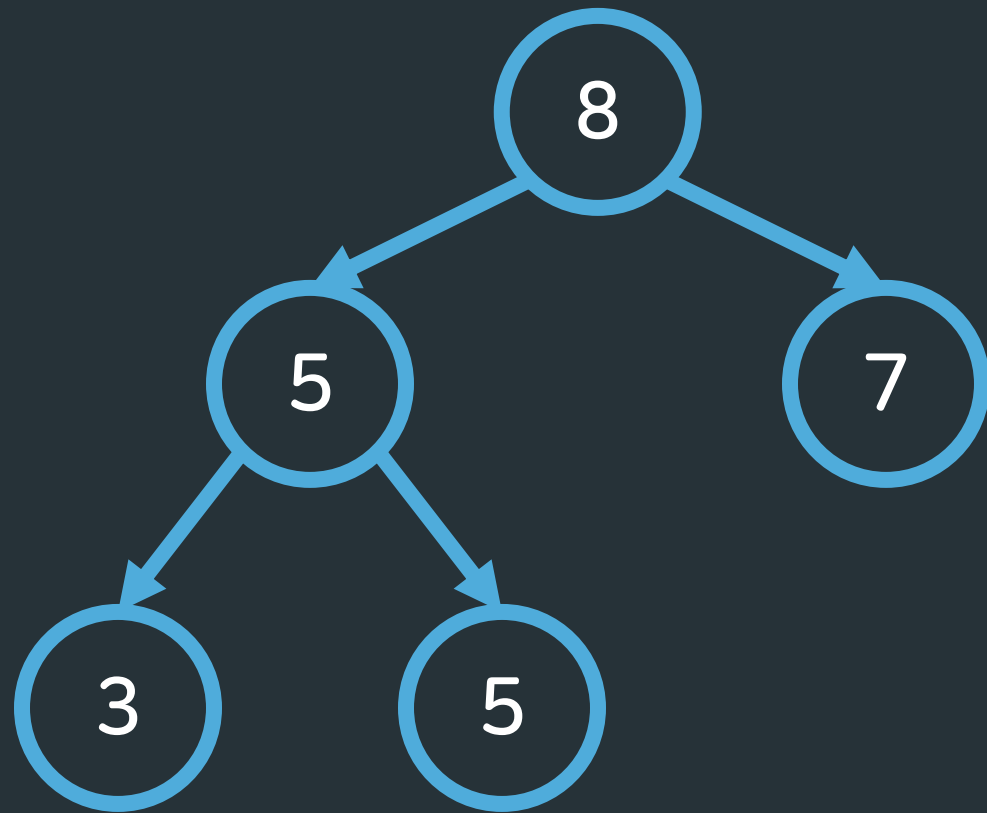
BST



(왼쪽)  $<$  (루트)  $<$  (오른쪽)  
좌우관계

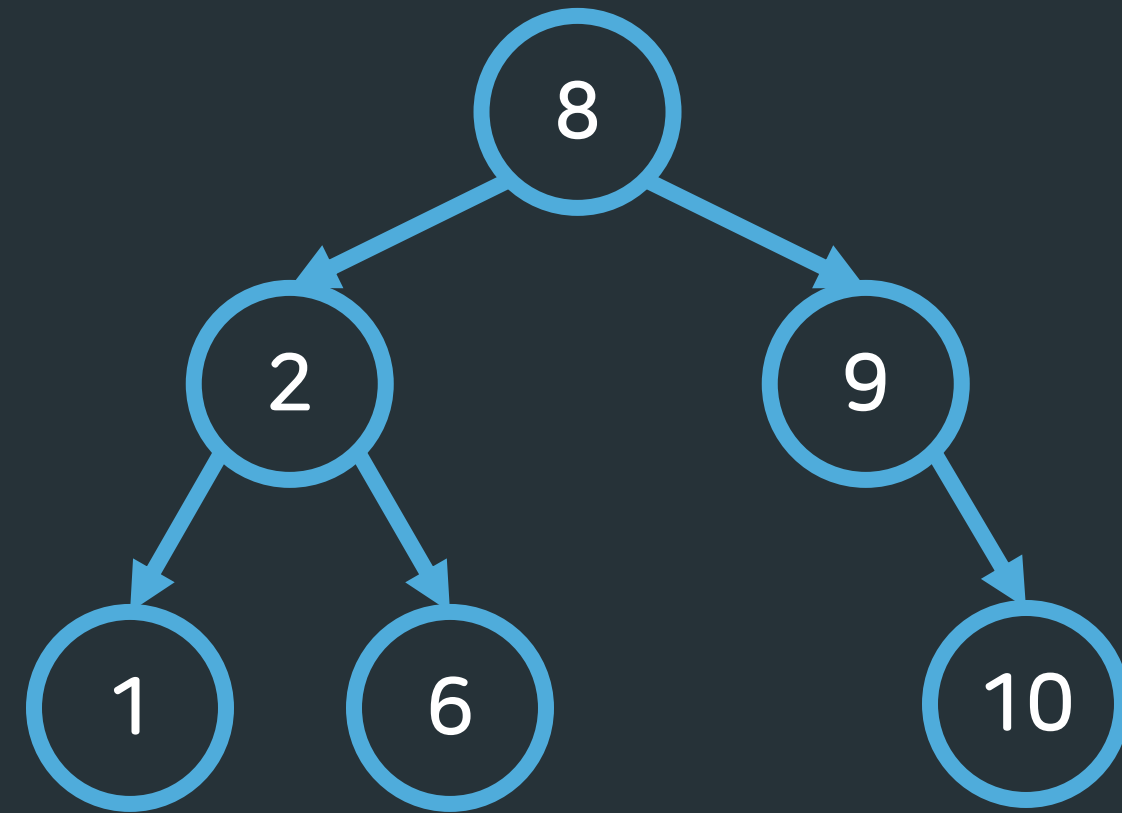
# Heap과 BST의 차이

Heap

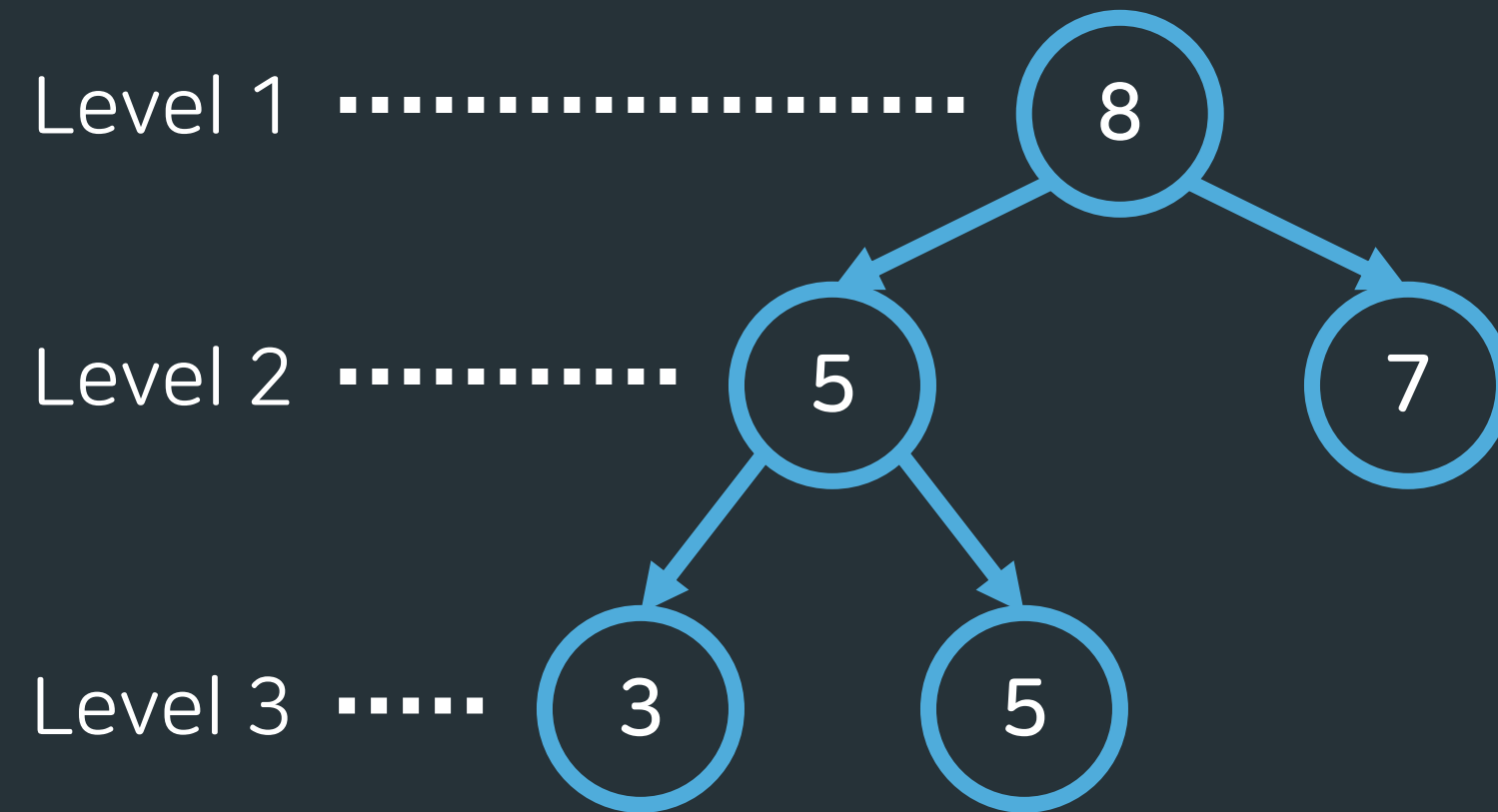


중복 0  
완전 이진 트리

BST



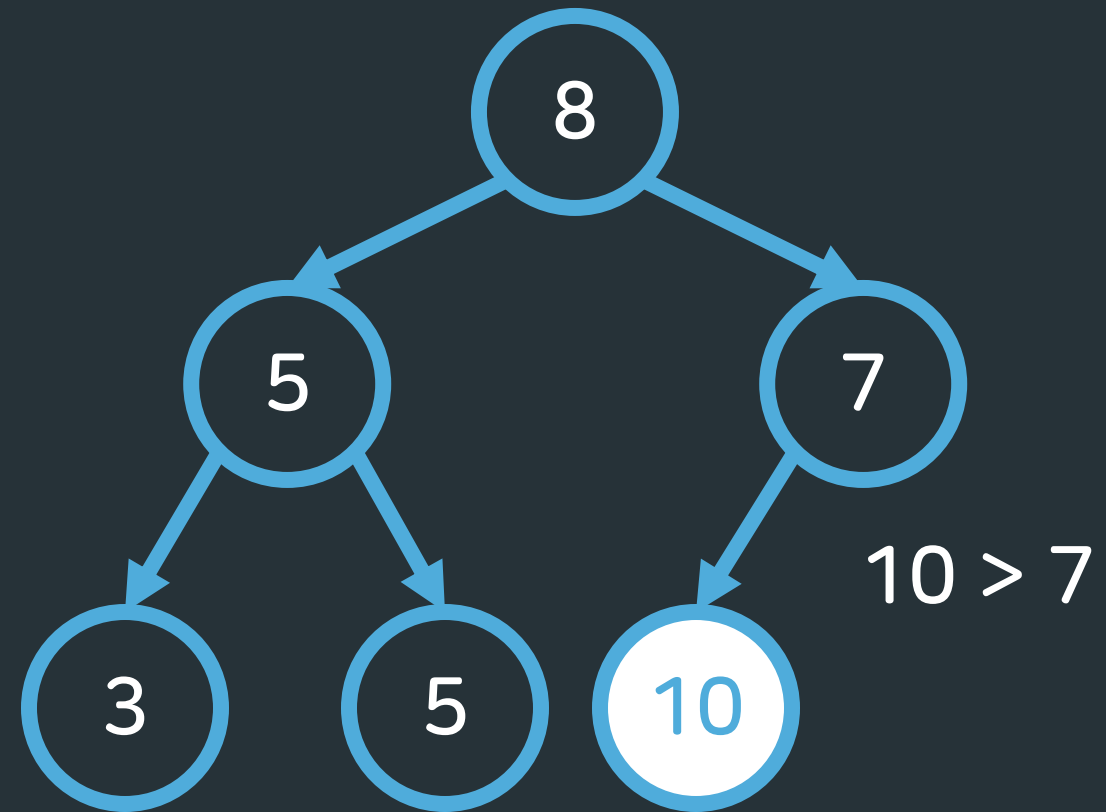
중복 X  
완전 이진 트리일 필요 없음



## Complete Binary Tree

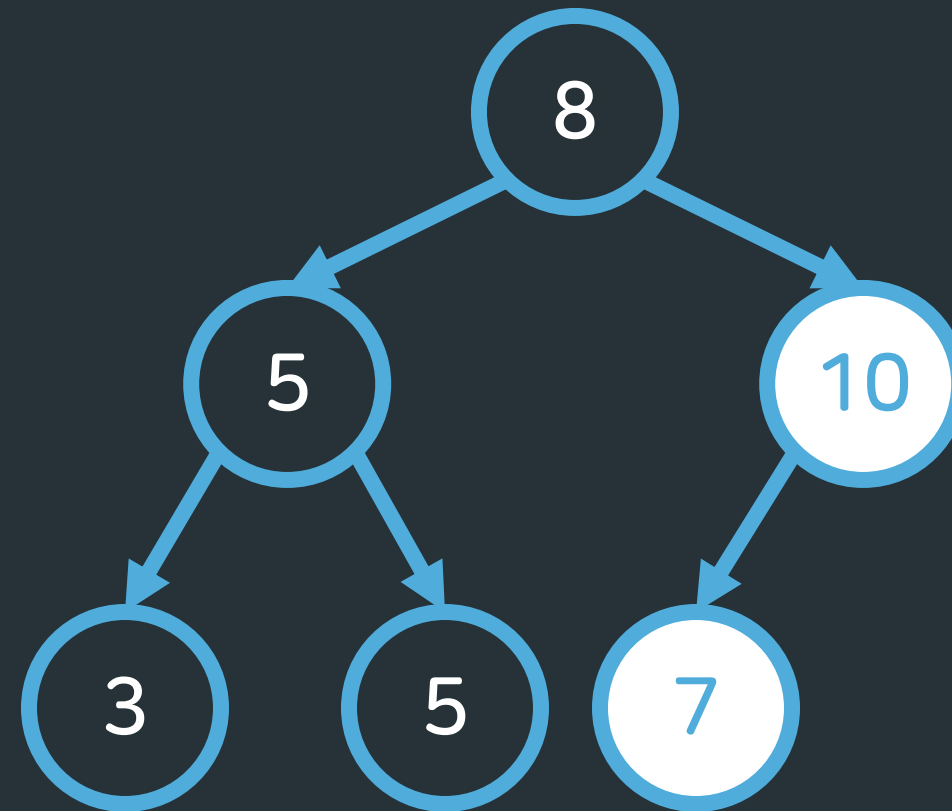
- 마지막 레벨을 제외하고 모든 레벨을 다 채움
- 마지막 레벨의 모든 노드는 왼쪽부터 빈 공간 없이 채움

# 최대 힙에 데이터 삽입



key = 10

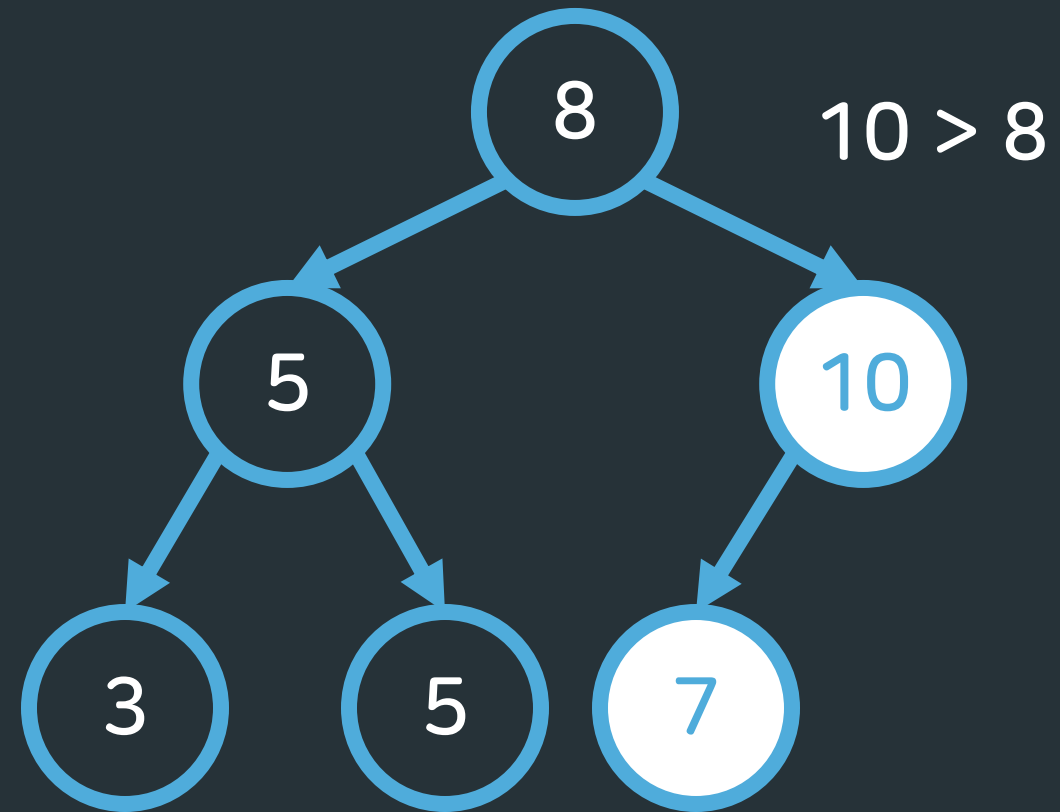
# 최대 힙에 데이터 삽입



key = 10

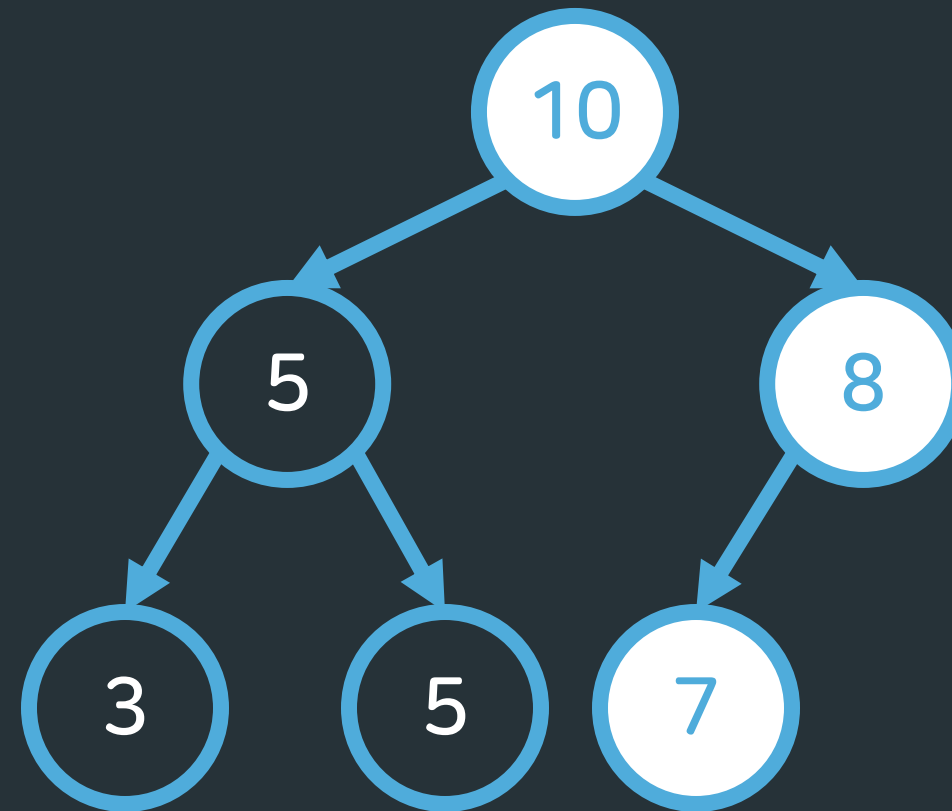


# 최대 힙에 데이터 삽입



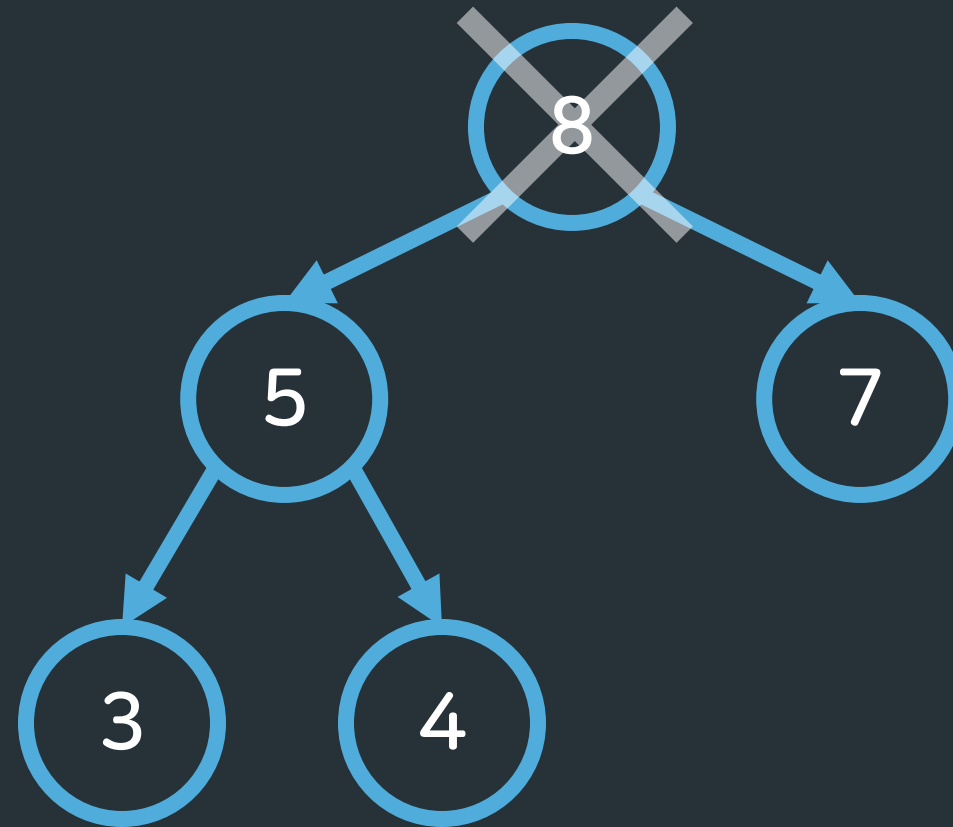
key = 10

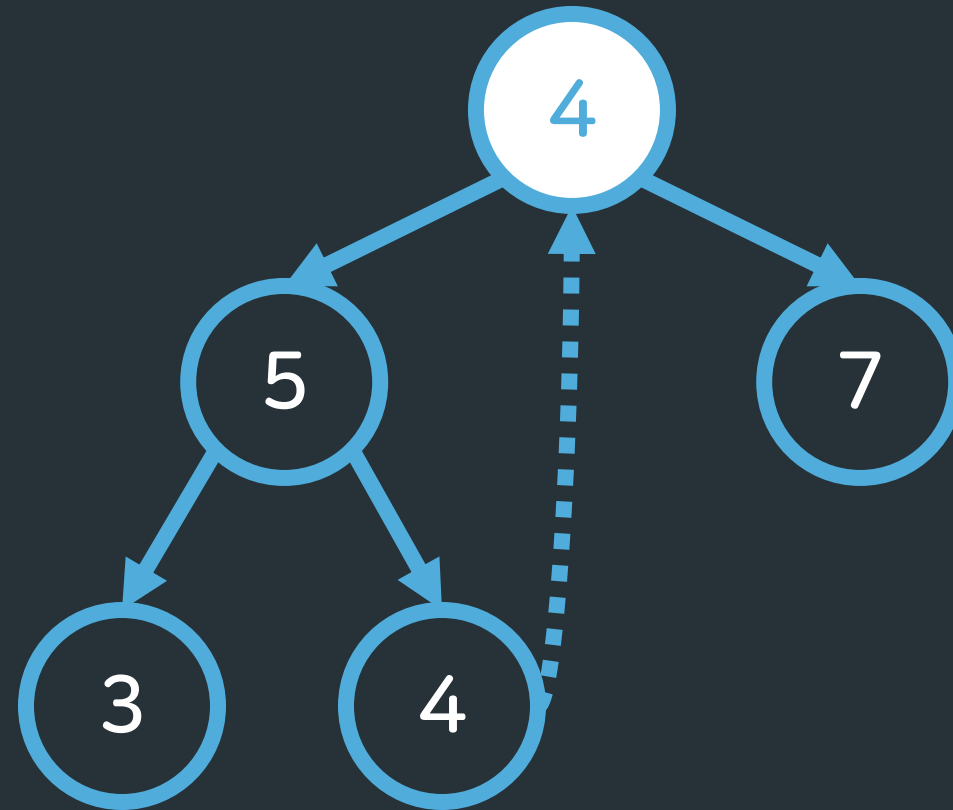
# 최대 힙에 데이터 삽입



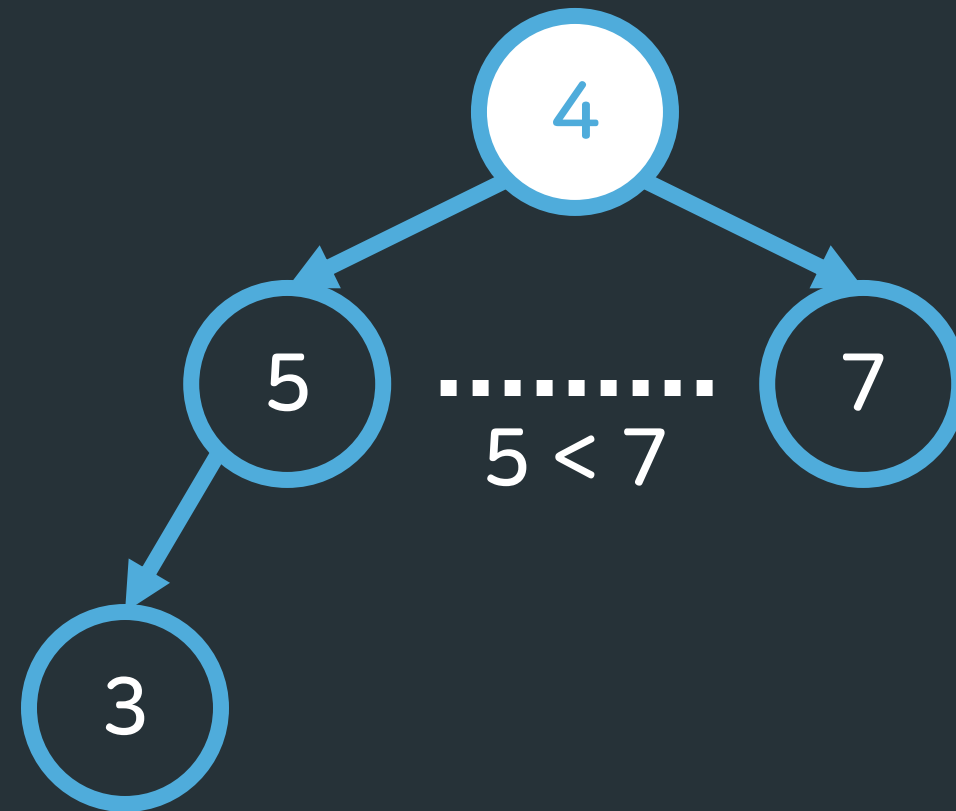
key = 10

# 최대 힙에서 데이터 삭제

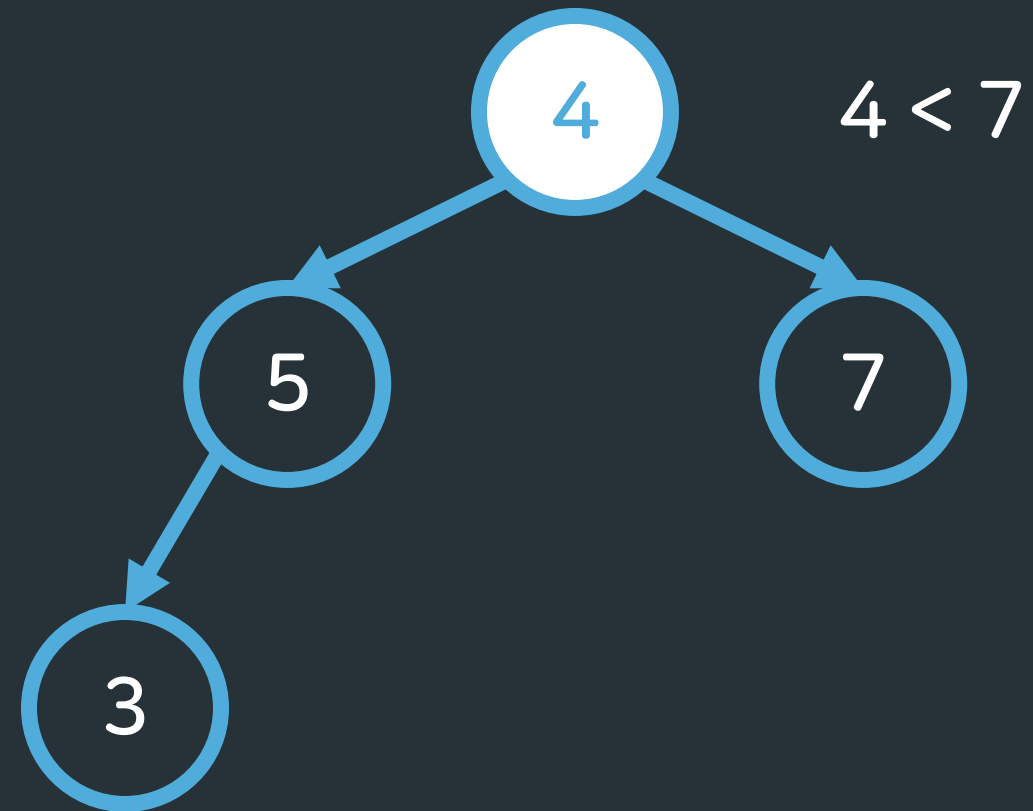




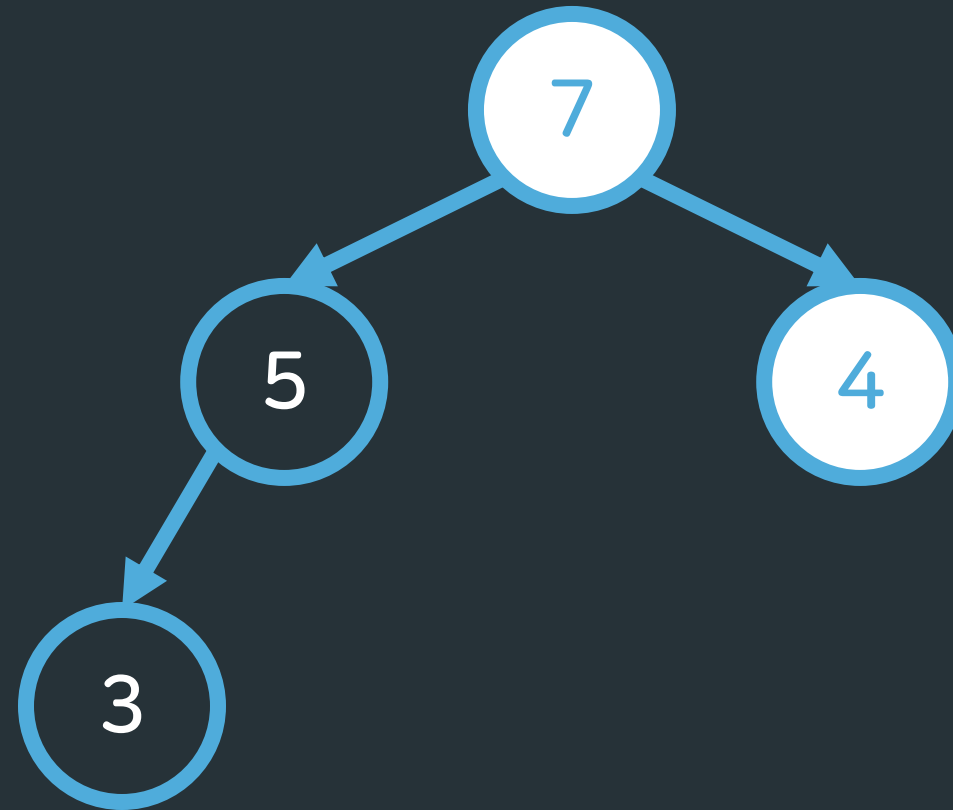
# 최대 힙에서 데이터 삭제



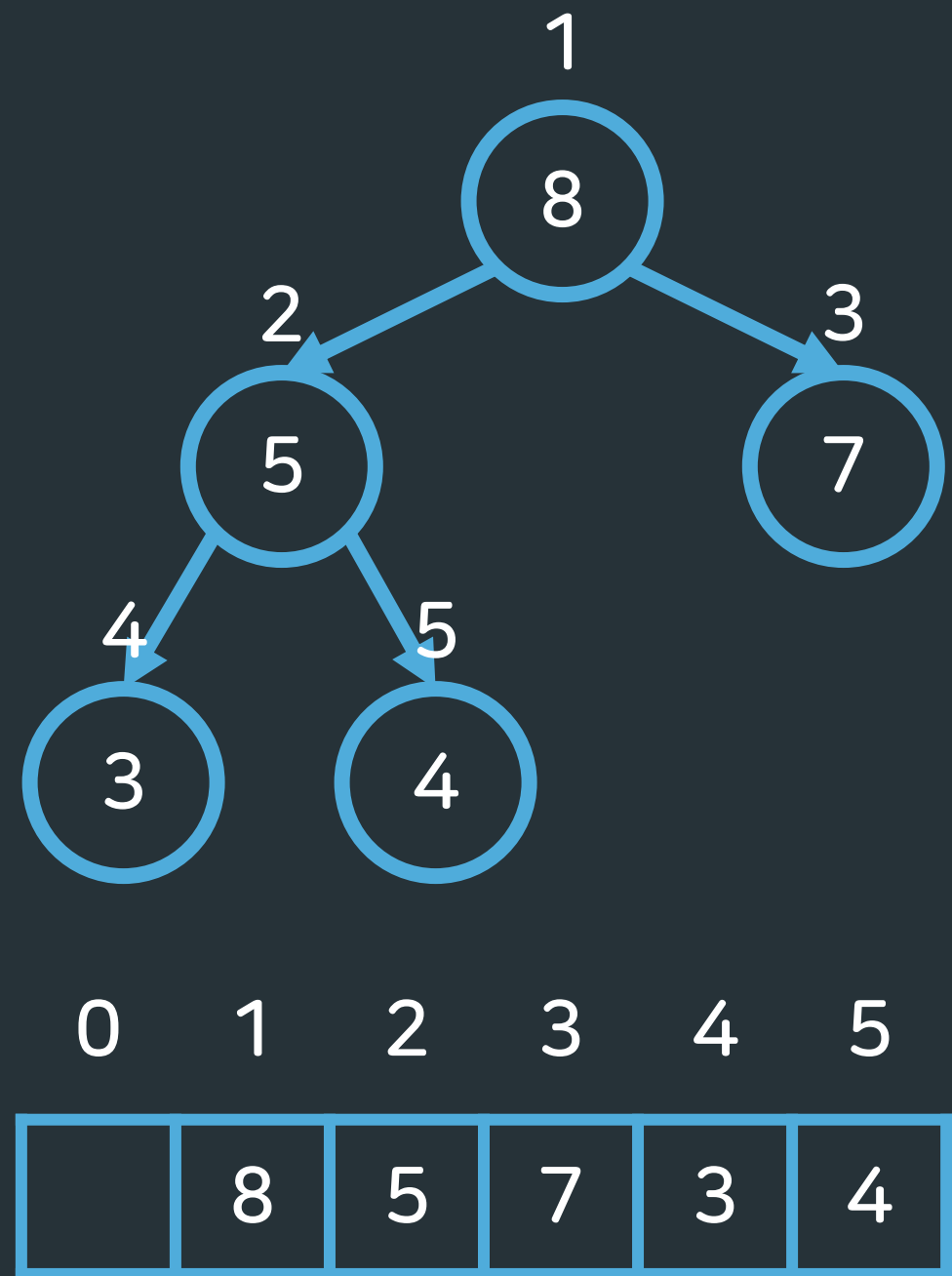
# 최대 힙에서 데이터 삭제



# 최대 힙에서 데이터 삭제

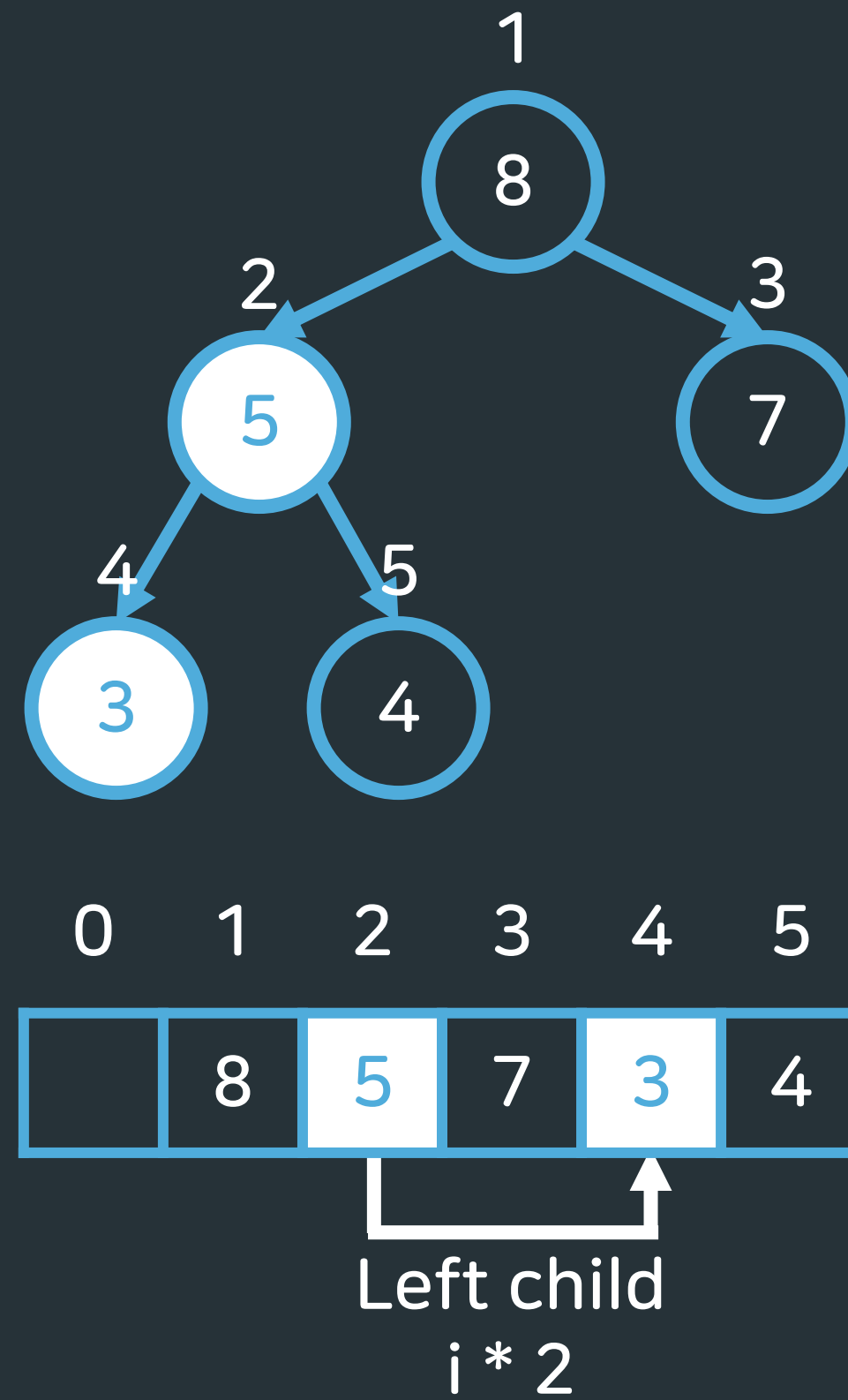


# 배열로 힙 구현하기

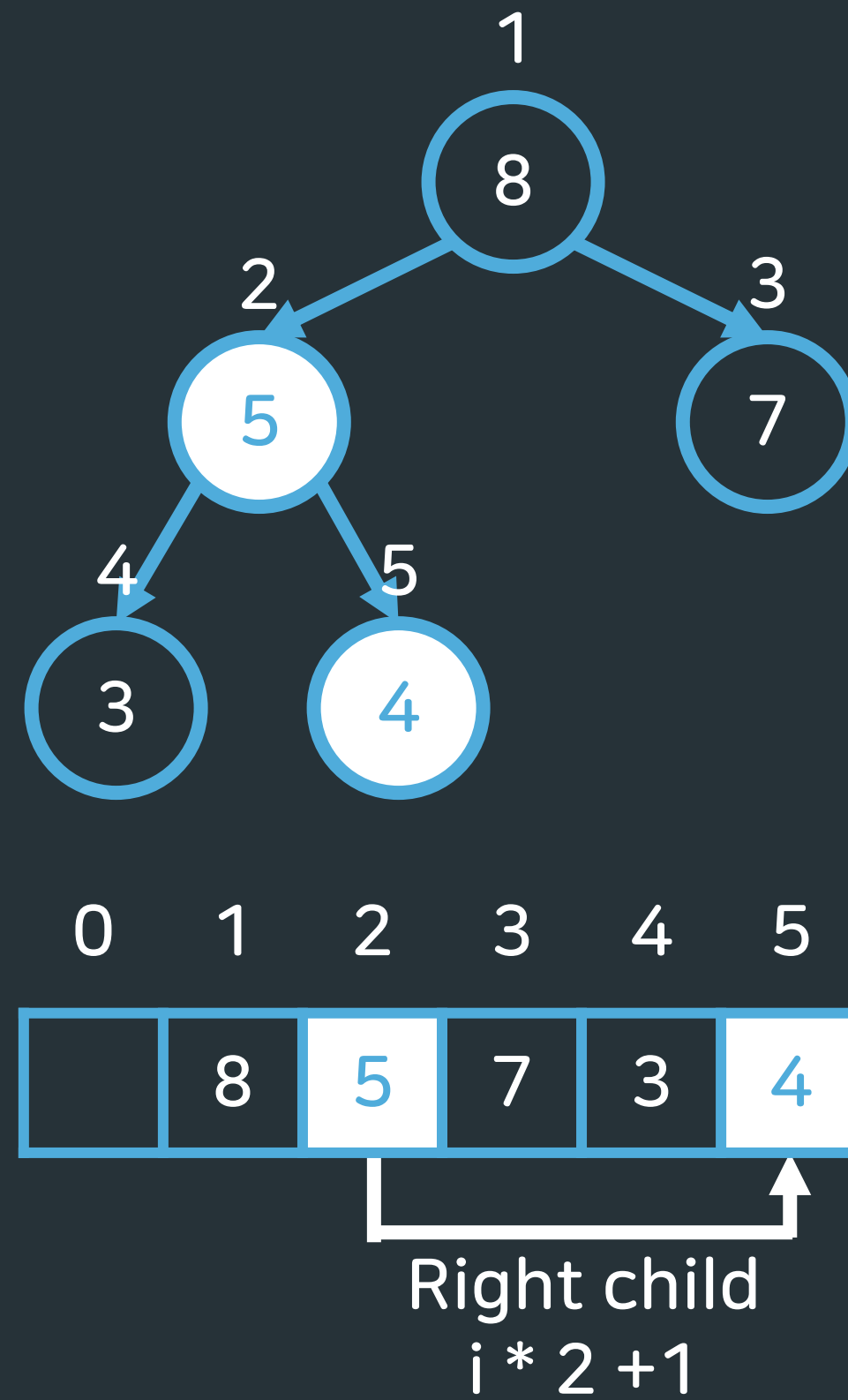




# 배열로 힙 구현하기



# 배열로 힙 구현하기



## /<> 11279번 : 최대 힙 - Silver 2

### 문제

- 다음의 명령을 처리하는 최대 힙 프로그램 만들기
  1. 정수  $x$ 가 주어진다.
  2.  $x$ 가 자연수라면 최대 힙에  $x$  추가
  3.  $x$ 가 0이라면 최대 힙에서 가장 큰 값을 출력하고 제거. 최대 힙이 비었다면 0 출력

### 제한 사항


- 명령의 수  $N$ 의 범위는  $1 \leq N \leq 100,000$
- 명령과 함께 주어지는 정수  $x$ 의 범위는  $0 \leq x \leq 2^{31}$

## 예제 입력

```
13
0
1
2
0
0
3
2
1
0
0
0
0
0
```

## 예제 출력

```
0
2
1
3
2
1
0
0
```



Search:

[Reference](#)
[<queue>](#)
[priority\\_queue](#)

[register](#)
[log in](#)

C++

[Information](#)
[Tutorials](#)
[Reference](#)
[Articles](#)
[Forum](#)

Reference

C library:

Containers:

<array>

<deque>

<forward\_list>

<list>

<map>

<queue>

<set>

<stack>

<unordered\_map>

<unordered\_set>

<vector>

Input/Output:

Multi-threading:

Other:

<queue>

priority\_queue

queue

priority\_queue

priority\_queue::priority\_queue

member functions:

priority\_queue::emplace

priority\_queue::empty

priority\_queue::pop

You were redirected to [cplusplus.com/priority\\_queue](#) || See search results for: "**priority\_queue**"

class template

std::priority\_queue

<queue>

```
template <class T, class Container = vector<T>,
          class Compare = less<typename Container::value_type> > class priority_queue;
```

Priority queue

Priority queues are a type of container adaptors, specifically designed such that its first element is always the greatest of the elements it contains, according to some *strict weak ordering* criterion.

This context is similar to a *heap*, where elements can be inserted at any moment, and only the *max heap* element can be retrieved (the one at the top in the *priority queue*).

Priority queues are implemented as *container adaptors*, which are classes that use an encapsulated object of a specific container class as its *underlying container*, providing a specific set of member functions to access its elements. Elements are *popped* from the "back" of the specific container, which is known as the *top* of the priority queue.

The underlying container may be any of the standard container class templates or some other specifically designed container class. The container shall be accessible through *random access iterators* and support the following operations:

- empty()
- size()
- front()
- push\_back()
- pop\_back()

The standard container classes `vector` and `deque` fulfill these requirements. By default, if no container class is specified for a particular `priority_queue` class instantiation, the standard container `vector` is used.

Support of *random access iterators* is required to keep a heap structure internally at all times. This is done automatically by the container adaptor by automatically calling the algorithm functions `make_heap`, `push_heap` and `pop_heap` when needed.

## 구문

```
template <class Type, class Container= vector <Type>, class Compare= less <typename Container ::value_type>>  
class priority_queue
```

- `priority_queue<T> ex) priority_queue <int> pq;`
- `priority_queue<T , Container, Compare> ex) priority_queue <int , vector<int>, greater<int>> pq;`

## 멤버함수

- `empty()` : 우선순위 큐가 비어 있으면 `true`를 return하고 비어 있지 않으면 `false`를 return
- `size()` : 큐의 요소 수를 return
- `top()` : 우선순위가 가장 큰 요소를 return
- `push(const Type& val)` : val의 우선 순위에 따라 우선순위 큐에 삽입
- `pop()` : 우선순위가 가장 큰 요소를 제거한다

## /<> 11286번 : 절댓값 힙 – Silver 1

### 문제

- 절댓값 힙은 다음 두 가지 연산을 지원
  1. 배열에 정수  $x (x \neq 0)$ 를 삽입
  2. 배열에서 절댓값이 가장 작은 값을 출력하고, 그 값을 배열에서 제거. 절댓값이 가장 작은 값이 여러 개인 경우, 가장 작은 수를 출력하고 그 값을 배열에서 제거.

### 제한 사항

- 연산의 개수  $N$ 의 범위  $1 \leq N \leq 100,000$
- 입력되는 정수  $x$ 의 범위  $-2^{31} < x < 2^{31}$
- 시간 제한 1초 (추가 시간 없음)

# 예제

예제 입력

18  
1  
-1  
0  
0  
0  
1  
1  
-1  
-1  
2  
-2  
0  
0  
0  
0  
0  
0  
0

예제 출력

-1  
1  
0  
-1  
-1  
1  
1  
-2  
2  
0



# 우선 순위를 변경해야 하는데...

## C++

1. 왜 `priority_queue`는 기본이 `Max heap`일까요?
  2. 정렬의 비교함수... 기억하시나요?
- 정렬 → 오름차순
  - 우선순위 큐 → 최대 힙

```
class template  
std::priority_queue  
template <class T, class Container = vector<T>,  
class Compare = less<typename Container::value_type> > class priority_queue;
```

## less<>

```
C++98 C++11 ?
1 template <class T> struct less {
2     bool operator() (const T& x, const T& y) const {return x<y;}
3     typedef T first_argument_type;
4     typedef T second_argument_type;
5     typedef bool result_type;
6 };
```

## greater<>

```
C++98 C++11 ?
1 template <class T> struct greater {
2     bool operator() (const T& x, const T& y) const {return x>y;}
3     typedef T first_argument_type;
4     typedef T second_argument_type;
5     typedef bool result_type;
6 };
```

조건을 만족하도록 정렬  
조건을 만족할 때 Swap

	정렬	우선순위 큐
X	왼쪽	부모 노드
Y	오른쪽	자식 노드
less	오름차순	Max heap
greater	내림차순	Min heap

## 비교 구조체를 직접 작성할 때는,

복잡하게 생각하지 말고, 첫 번째 인자를 *child*, 두 번째 인자를 *parent*로 두고 조건 그대로 작성하기

```
struct cmp {  
    bool operator()(const 자료형 &child, const 자료형 &parent) {  
        // 예: 부모는 자식보다 나이가 많아야 함.  
        return parent.age > child.age;  
    }  
};
```

## 절댓값 힙을 위한 비교 구조체는?

배열에서 절댓값이 가장 작은 값을 출력하고, 그 값을 배열에서 제거.

절댓값이 가장 작은 값이 여러 개일 경우 가장 작은 수를 출력하고 그 값을 배열에서 제거

```
struct cmp {  
    bool operator()(const int &child, const int &parent) {  
        // 1. 절댓값이 다른 경우 절댓값이 작은 수가 부모노드로  
        if (abs(parent) != abs(child)) {  
            return abs(parent) < abs(child);  
        }  
        // 2. 절댓값이 같은 경우 값이 작은 수가 부모노드로  
        return parent < child;  
    }  
};
```


## 정리

- 우선순위 큐는 힙으로 구현하고, 시간 복잡도가  $O(\log n)$ 인 자료구조
- 효율성을 보는 문제에 사용되는 경우가 많음
- 그리디, 최단 경로 알고리즘 풀이에 활용되기도 함
- cmp 정의할 때는 헛갈리지 말기!(sort와 반대)
- 정렬은 cmp 정의 시 첫 번째 인자 입장으로, priority\_queue는 두 번째 인자 입장으로 생각하면 쉬움
- 무한 루프 (pop을 하지 않음), 런타임 에러 (empty 체크 안하고 조회 or 삭제 시도) 조심!!
  
- default는 C++의 경우 Max heap

## 필수

- /<> 2607번 : 비슷한 단어 - Silver 3
- /<> 19640번 : 화장실의 규칙 - Gold 4
- /<> 2075번 : n번째 큰 수 - Silver 2

## 도전

- /<> 1655번 : 가운데를 말해요 - Gold 2
-  디스크 컨트롤러 - Lv.3

과제제출 마감

~ 3월 21일 금요일 18:59

추가제출 마감

~ 3월 23일 일요일 23:59