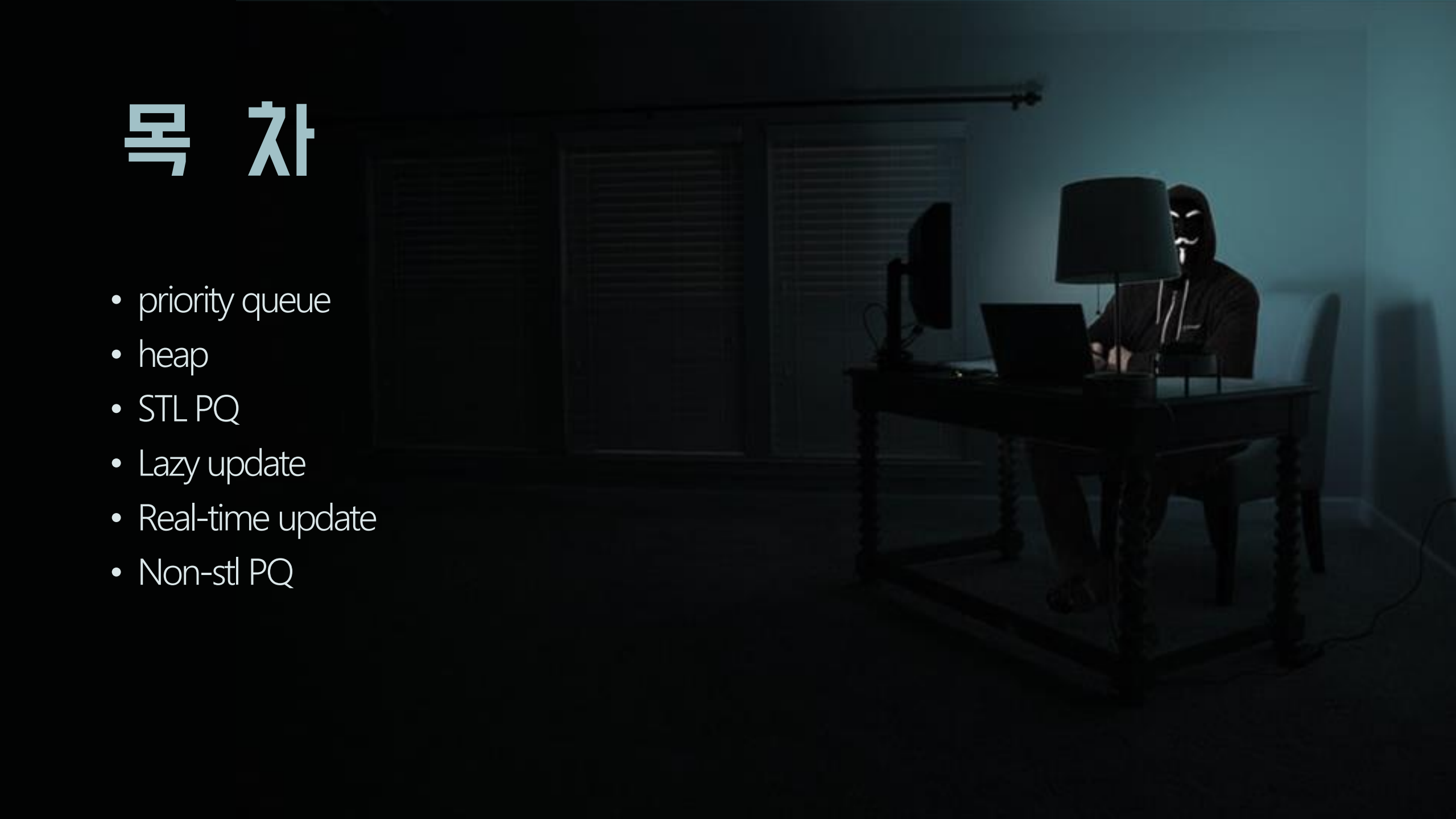
STL 기초

Priority Queue

한컴코드패스

목 차



- priority queue
- heap
- STL PQ
- Lazy update
- Real-time update
- Non-stl PQ

Priority Queue

- stack : 나중에 들어온 데이터가 먼저 나감
- queue : 먼저 들어온 데이터가 먼저 나감
- **priority queue : 우선순위 높은 데이터가 먼저 나감**

- 구현

sequence	push O(1)	pop O(n)
heap	**push O(log n)**	**pop O(log n)**

Heap

두 가지 특성 만족

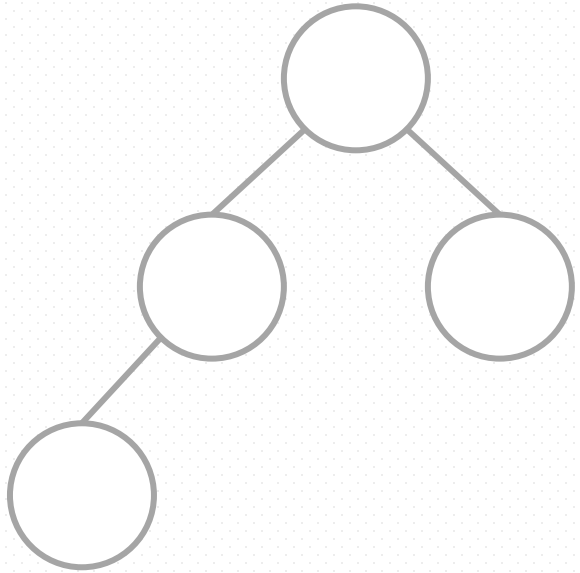
1. 완전 이진 트리
2. 부모 노드의 우선순위가 자식 노드의 우선순위보다 높다.

=> 최우선순위 노드는 루트에 존재

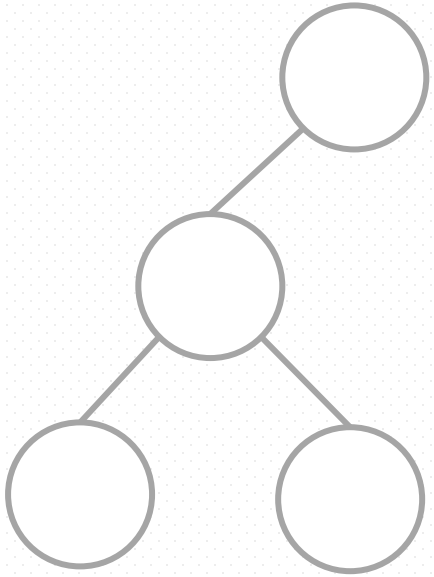
Heap

1. 완전 이진 트리

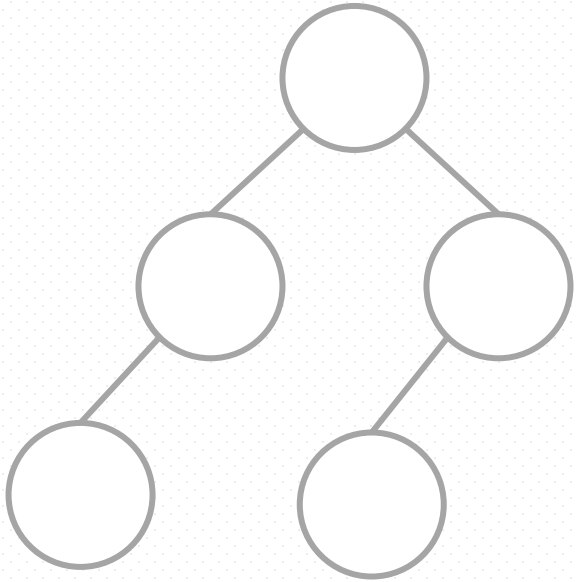
- 모든 노드의 자식 노드는 최대 2개
- 노드는 위에서부터, 왼쪽부터 꽉 채워진다.



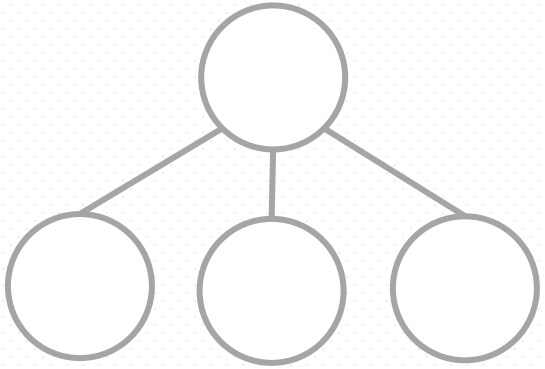
(O)



(X)



(X)

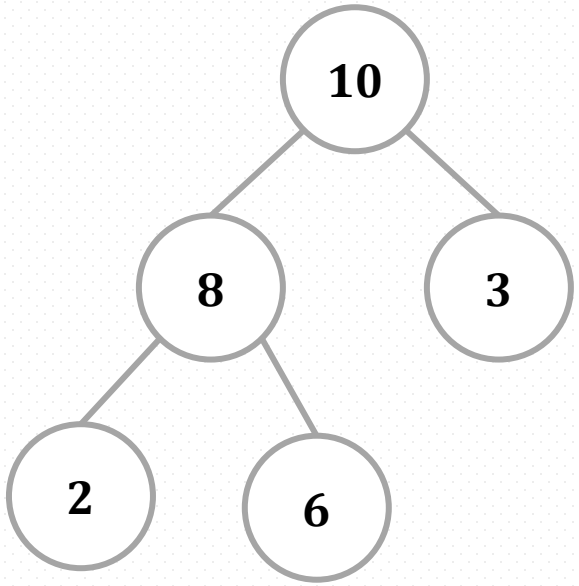


(X)

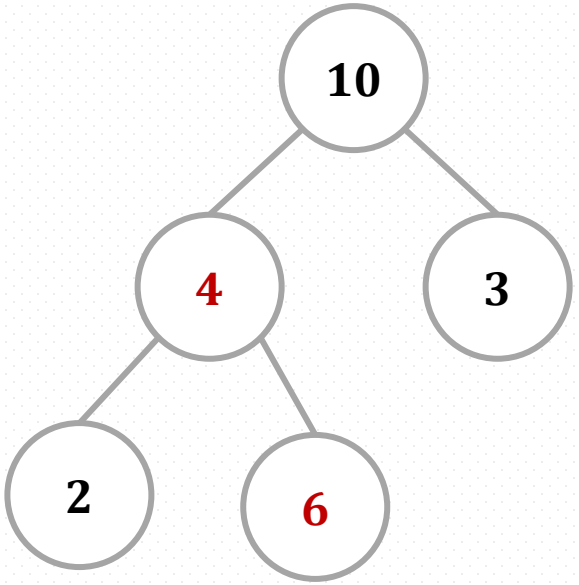
Heap

2. 부모 노드의 우선순위가 자식 노드의 우선순위보다 높다.

ex) 값이 클수록 우선순위가 높다 : MAX_HEAP



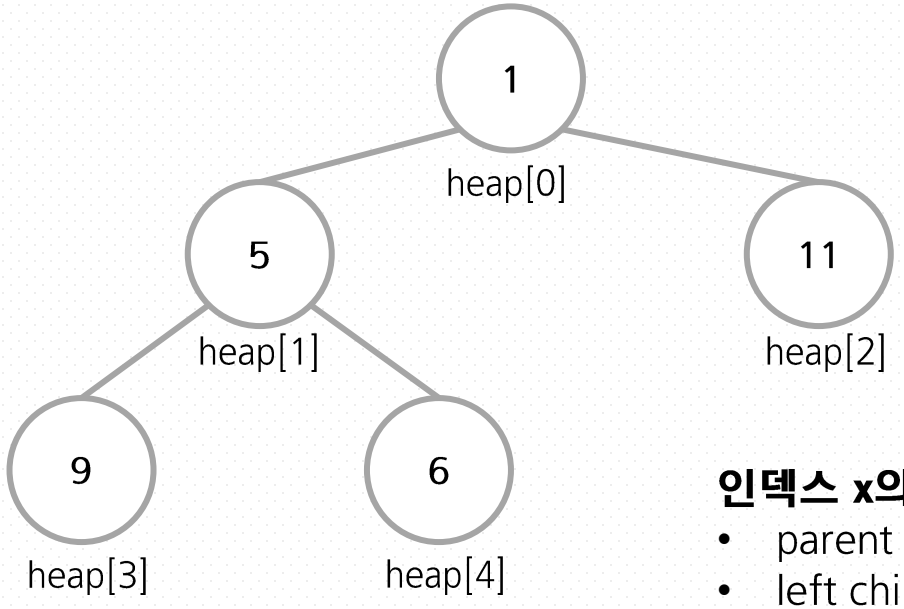
(O)



(X)

Heap : 구성

- 표현은 트리 형태로 하지만 실제 저장은 단순 배열만으로 가능하다.
- 완전 이진 트리이기 때문에 각 노드의 부모, 자식의 인덱스를 수식으로 구할 수 있다.



인덱스 x의 부모, 자식

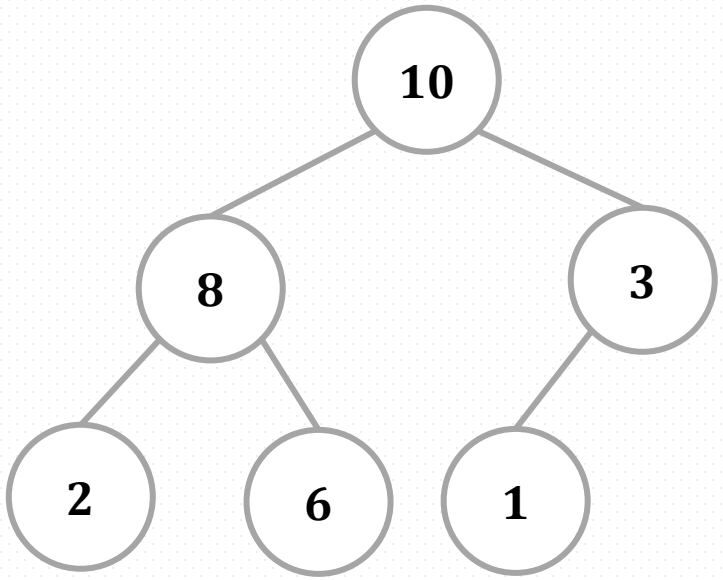
- parent = (x-1)/2
- left child = x * 2 + 1
- right child = x * 2 + 2

index	0	1	2	3	4
value	1	5	11	9	6

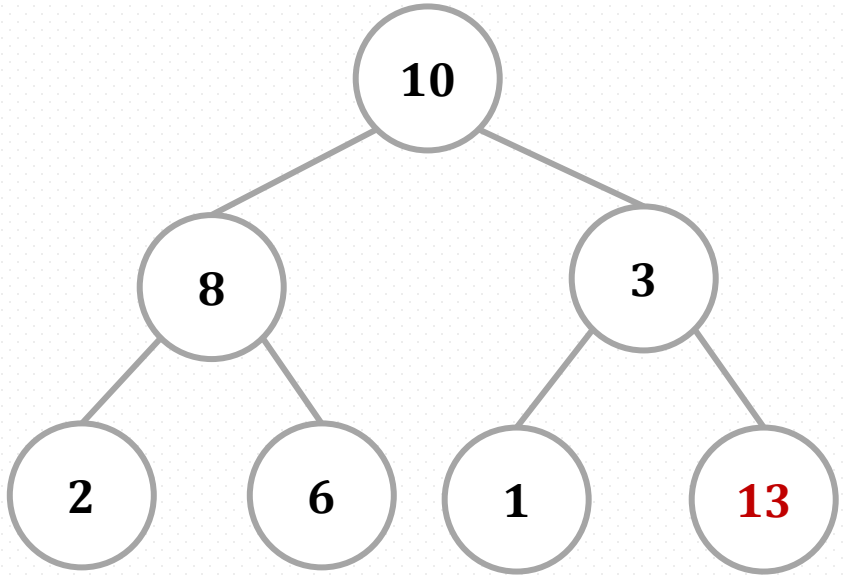
Heap : push

1. 마지막 위치에 추가
2. 추가한 노드를 부모 노드랑 비교하며 우선순위 높으면 바꿔 올라간다.

max heap example



index	0	1	2	3	4	5
value	10	8	3	2	6	1

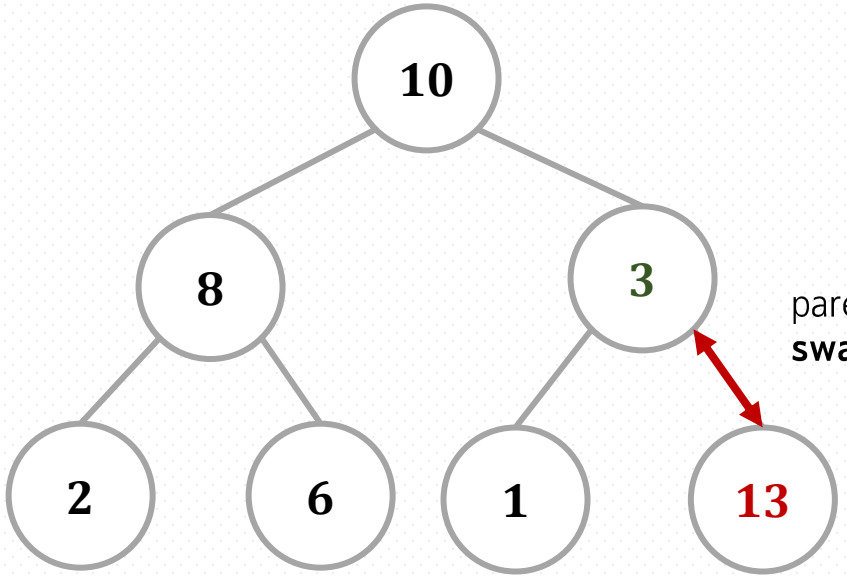


index	0	1	2	3	4	5	6
value	10	8	3	2	6	1	13

Heap : push

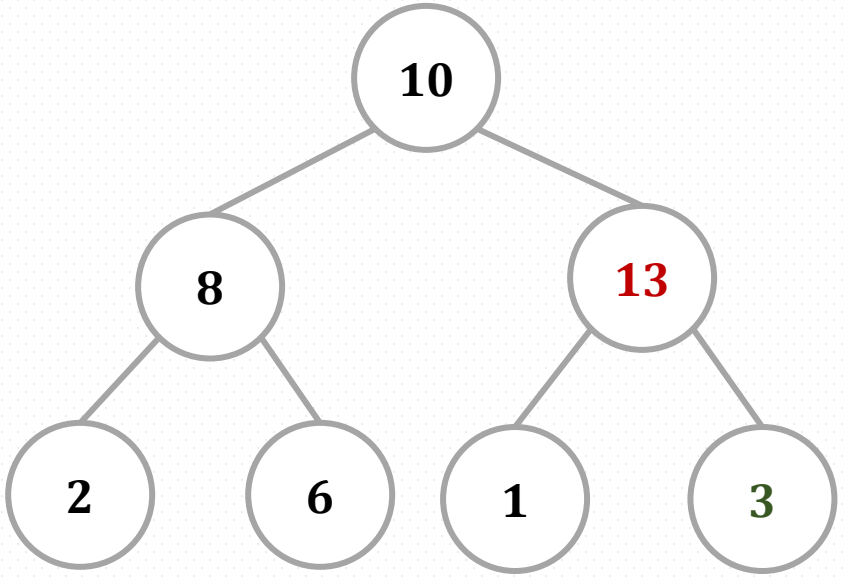
1. 마지막 위치에 추가
2. 추가한 노드를 부모 노드랑 비교하며 우선순위 높으면 바꿔 올라간다.

max heap example



parent(3) < child(13)
swap

index	0	1	2	3	4	5	6
value	10	8	3	2	6	1	13

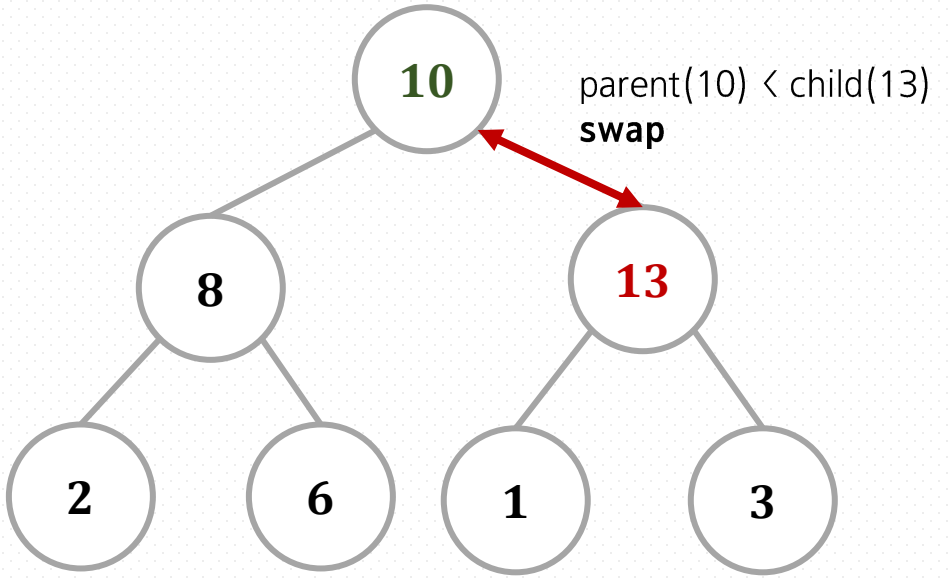


index	0	1	2	3	4	5	6
value	10	8	13	2	6	1	3

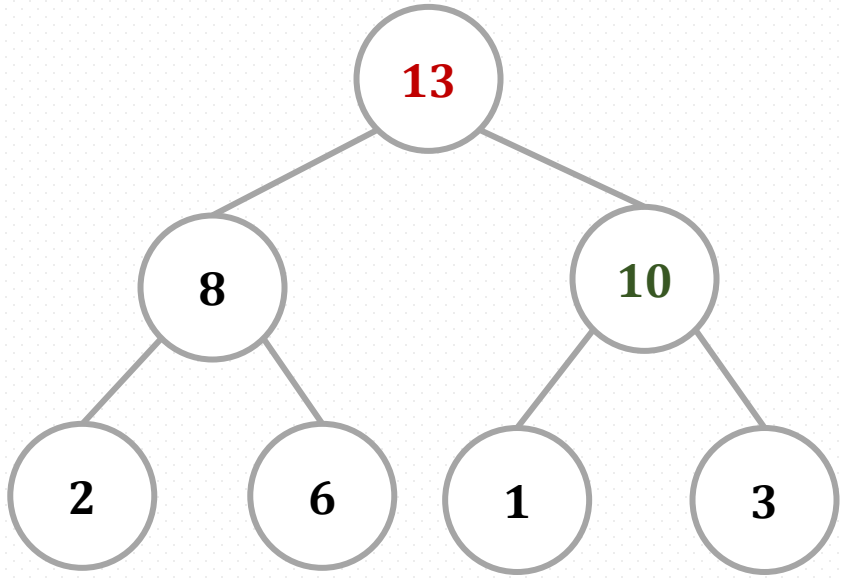
Heap : push

1. 마지막 위치에 추가
2. 추가한 노드를 부모 노드랑 비교하며 우선순위 높으면 바꿔 올라간다.

max heap example



index	0	1	2	3	4	5	6
value	10	8	13	2	6	1	3

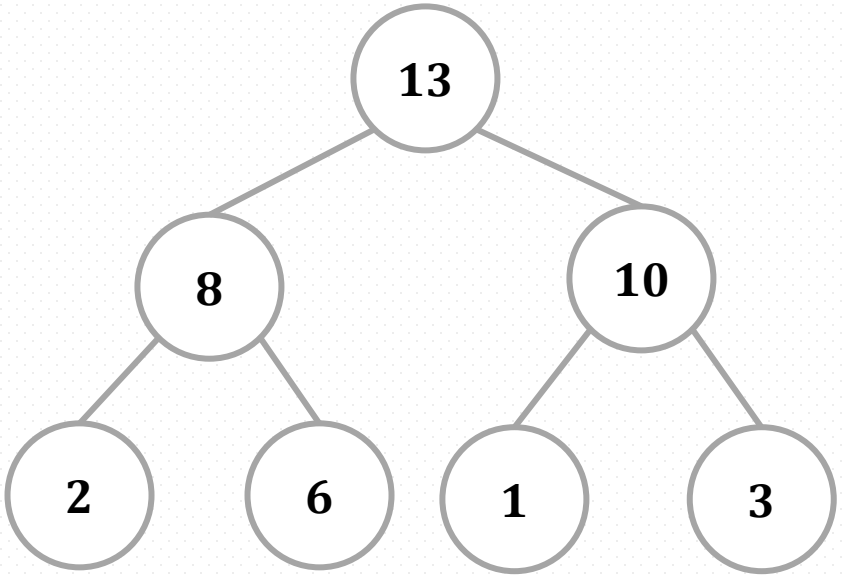


index	0	1	2	3	4	5	6
value	13	8	10	2	6	1	3

Heap : push

1. 마지막 위치에 추가
2. 추가한 노드를 부모 노드랑 비교하며 우선순위 높으면 바꿔 올라간다.

max heap example

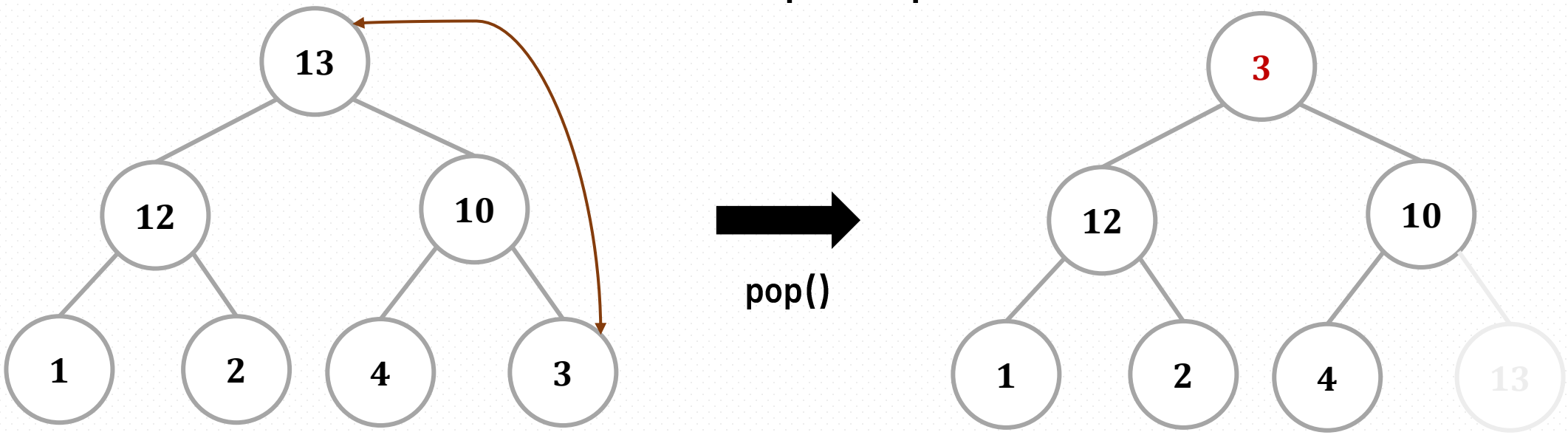


더 이상 올라갈 곳이 없으므로 push 종료

Heap : pop

1. 루트와 마지막 노드를 바꾸고 heap size를 감소시킨다.
2. 루트 노드에서부터 우선순위 높은 자식 노드와 비교하여 우선순위가 낮으면 바꿔 내려간다.

max heap example



index	0	1	2	3	4	5	6
value	13	12	10	1	2	4	3

index	0	1	2	3	4	5	6
value	3	12	10	1	2	4	~~13~~

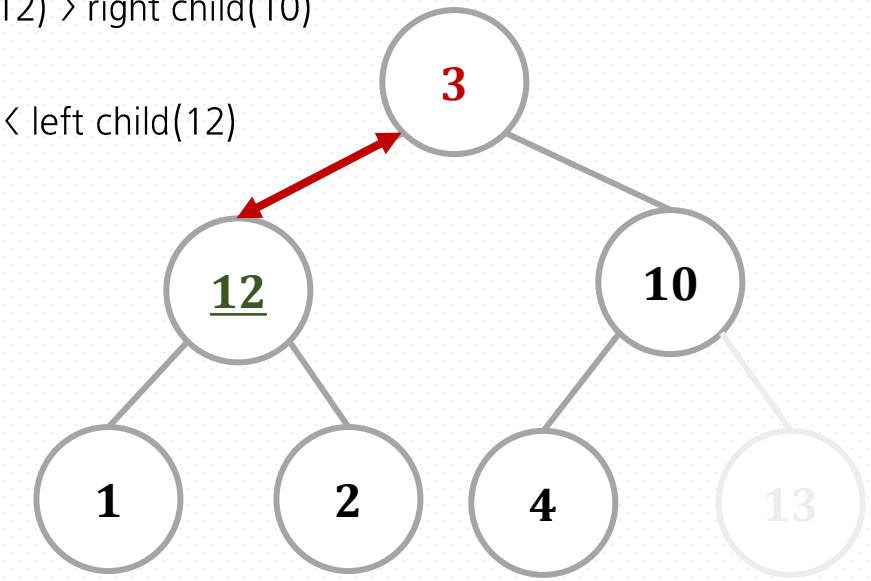
Heap : pop

1. 루트와 마지막 노드를 바꾸고 heap size를 감소시킨다.
2. 루트 노드에서부터 우선순위 높은 자식 노드와 비교하여 우선순위가 낮으면 바꿔 내려간다.

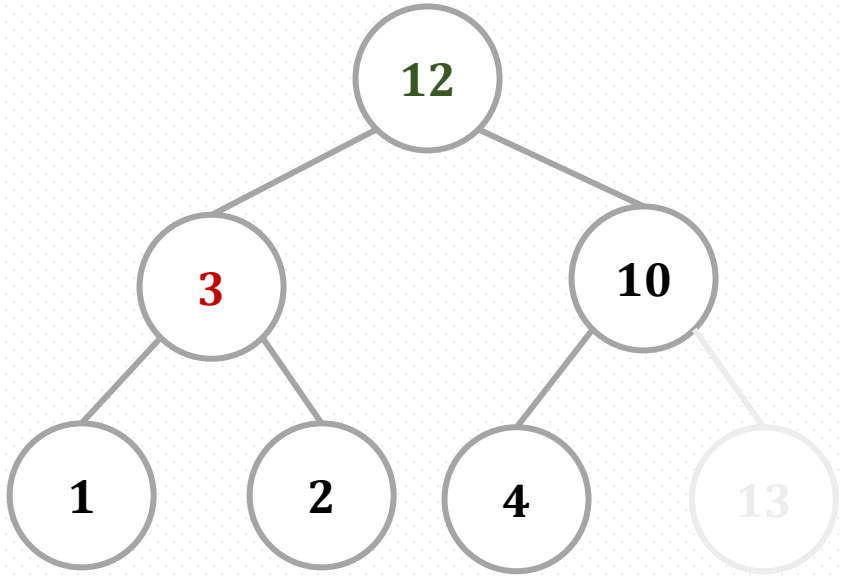
max heap example

1. select left
left child(12) > right child(10)

2. swap
parent(3) < left child(12)



index	0	1	2	3	4	5	6
value	3	12	10	1	2	4	~~13~~



index	0	1	2	3	4	5	6
value	12	3	10	1	2	4	~~13~~

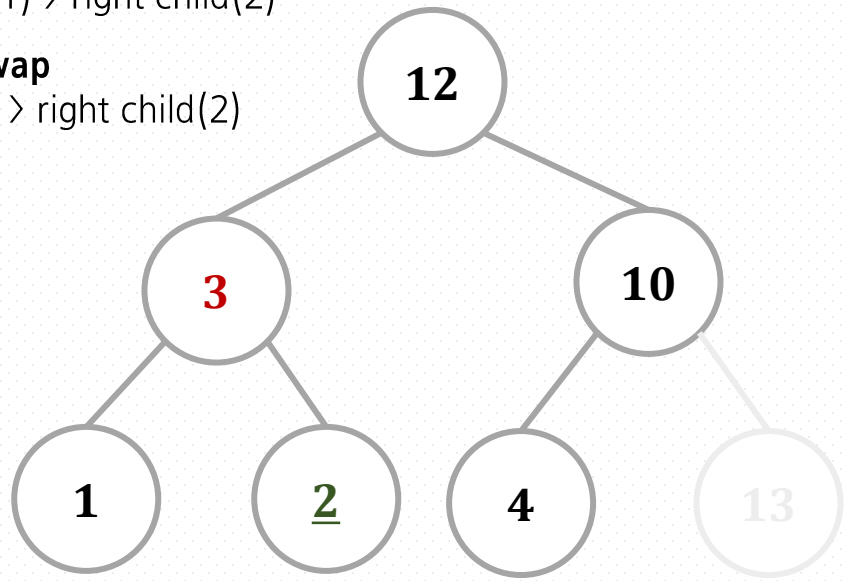
Heap : pop

1. 루트와 마지막 노드를 바꾸고 heap size를 감소시킨다.
2. 루트 노드에서부터 우선순위 높은 자식 노드와 비교하여 우선순위가 낮으면 바꿔 내려간다.

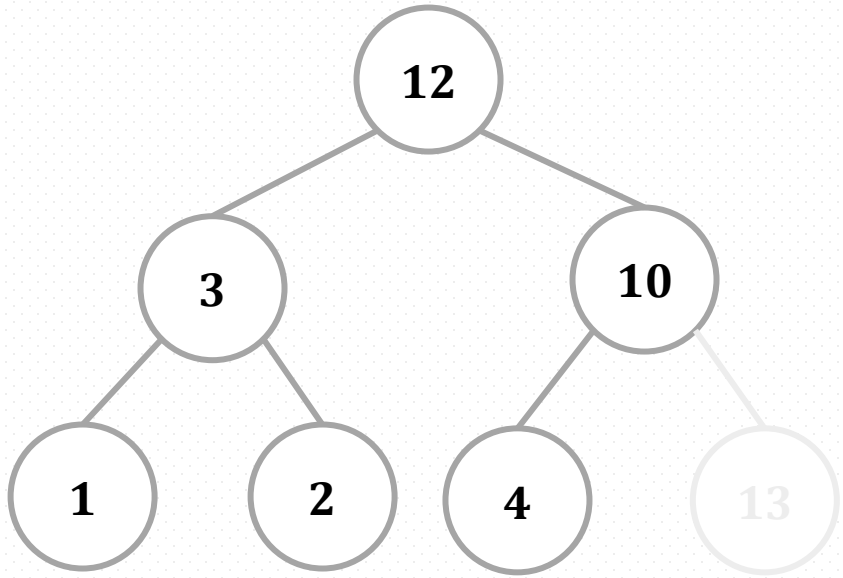
max heap example

1. select right
left child(1) > right child(2)

2. do not swap
parent(3) > right child(2)



index	0	1	2	3	4	5	6
value	12	3	10	1	2	4	~~13~~

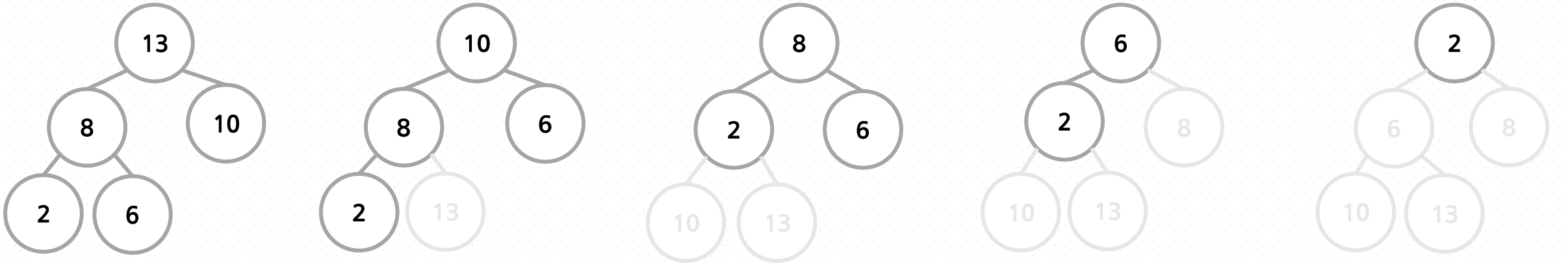


pop 종료

Heap Sort

1. n개의 data를 heap에 push
2. n번 pop 진행 (pop 할 때, 루트와 마지막 값 swap)
   <u>최우선순위 값이 순차적으로 뒤쪽부터 쌓이다.</u>

max heap example



0	1	2	3	4
13	8	10	2	6

0	1	2	3	4
10	8	6	2	13

0	1	2	3	4
8	2	6	10	13

0	1	2	3	4
6	2	8	10	13

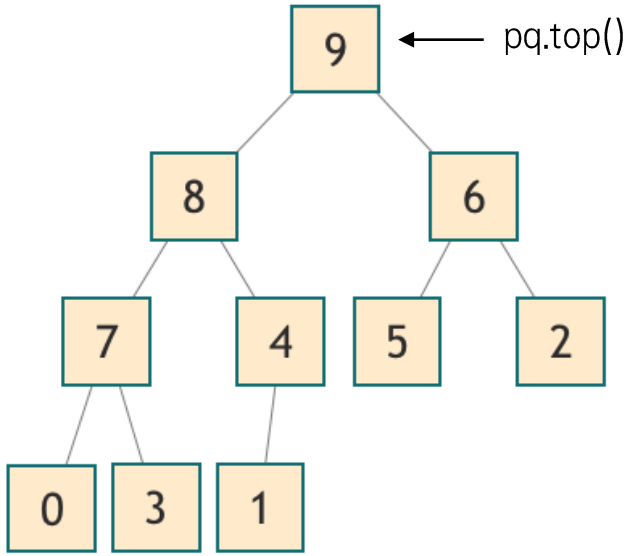
0	1	2	3	4
2	6	8	10	13

- max heap : 큰 값이 뒤에 쌓인다 => 오름차순 정렬
- min heap : 작은 값이 뒤에 쌓인다 => 내림차순 정렬

STL Priority_Queue

- #include<queue>
- heap으로 구성
- default : vector<T> , less<T>
- top만 접근 가능
- **compare 기준으로 정렬했을 때 가장 오른쪽 element를 top으로 반환**

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



<max_pq example>

```
priority_queue<T> pq
priority_queue<T, vector<T>, Compare> pq

pq = {}// clear

void pq.push(x)
void pq.pop()
T    pq.top()
int  pq.size()
bool pq.empty()
```

Compare 설정 방법

※ Function Object으로 설정

1. pre-defined function object

- **less<T>** : 큰 값이 top
  key type에 operator< 정의 필수

```
template<class T>
struct less {
  bool operator()(const T& lhs, const T& rhs) const {
    return lhs < rhs;
  }
}
```

- **greater<T>** : 작은 값이 top
  key type에 operator> 정의 필수

```
template<class T>
struct greater {
  bool operator()(const T& lhs, const T& rhs) const {
    return lhs > rhs;
  }
}
```

2. user-defined function object

- **int 절대값 오름차순**
  : 절대값 큰 값이 top

```
struct AbsComp {
    bool operator()(const int &l, const int &r) const {
        return abs(l) < abs(r);
    }
};
```

참조자(&) : 권장
const : 필수

Compare 설정 예 - int

- **less<int> : 값이 클수록 우선순위 높음**

```
priority_queue<int> maxpq;
priority_queue<int, vector<int>, less<int>> maxpq;
```

- **greater<int> : 값이 작을수록 우선순위 높음**

```
priority_queue<int, vector<int>, greater<int>> minpq;
```

- **절대값 오름차순 : 절대값이 클수록 우선순위 높음**

```
struct AbsComp {
    bool operator()(const int &l, const int &r) const {
        return abs(l) < abs(r);
    }
};

priority_queue<int, vector<int>, AbsComp> abspq;
```

Compare 설정 예 – custom data type

1. **default - less<T> 활용** operator< 정의 필수

```
struct Data {
    int a, b, c;
    bool operator<(const Data& r) const {
        return a < r.a;
    }
};

priority_queue<Data> pq;
priority_queue<Data, vector<Data>, less<Data>> pq;
```

2. **user-defined function object**

```
struct Data { int a, b, c; };

struct Comp {
    bool operator()(const Data&l, const Data&r) const {
        return l.a < r.a;
    }
};

priority_queue<Data, vector<Data>, Comp> pq;
```

a 값이 클수록 오른쪽

=> a 값이 가장 큰게 top

Compare Requirements

- **Function object**

Predicate return type : bool

find_if
all_of
…

BinaryPredicate return type : bool , argument : two
bool f(T a, T b)

lower_bound
upper_buond
unordered_set
unordered_map
…

Compare

set
map
sort
max
priority_queue
…

strict weak ordering

1. !comp(a,a)
2. comp(a,b) then !comp(b,a)
3. comp(a,b) && comp(b,c) then comp(a,c)
4. !comp(a,b) && !comp(b,a) then (a equal to b)

※ comp(a,b) example : a<b or a>b

Summary

- >, < 연산자만 사용
- 좌우항이 같은 형태여야 함
  ex) l.x < r.x , l.x+l.y < r.x+r.y , ~~l.x < r.y~~

임의 원소의 변경, 삭제

PQ는 주어진 push, pop외에 다른 방식의 데이터 조작을 허용하지 않는다.
이를 위해서 추가적인 로직이 필요하며 대표적으로 두가지가 있다.

1. Lazy Update

실시간으로 힙 내부를 최신화 하지 않고 top을 구하는 과정에서 유효하지 않을 값을 버린다.

- 최신 정보를 저장하는 별도의 배열을 만들어 관리한다.
- PQ가 실시간 업데이트 되지 않으므로 유효하지 않은 값들도 들어가있게 된다.
- top을 확인할 때, 유효하지 않는 값들은 버린다.
  이때 판단은 우선순위 1개를 구하는지 여러 개를 구하는지에 따라 달라질 수 있다.

2. Real-time Update (indexed heap)

실시간으로 PQ 내부를 최신화 한다.
이를 위해서 내부의 원소 접근이 필요하며 STL PQ는 사용할 수 없다.

- 각 원소들의 heap index를 기록하는 별도의 배열(pos[])이 필요하다.
- heap의 위치가 swap 될때마다 pos[] 배열도 같이 swap된다.
- 특정 id의 value값이 변경되거나 삭제 될 때, pos[id]로 바뀐 id의 heap index에 접근한다.
- PQ에 push(), pop() 외에 erase(), update() 기능을 추가해줘야 하며 push(), pop()을 응용하여 구현 가능하다.

Lazy Update

원소가 삭제 될 때

- 최신 정보를 저장하는 배열에 삭제됐음을 표시
- heap에는 아무 변화 없음

원소가 변경 될 때

- 최신 정보를 저장하는 배열에 변경된 값을 기록
- heap에 변경된 정보로 push

우선순위 값 한 개 확인할 때

- top에 있는 값이 유효한 정보인지 확인한다.
  => 최신 값이랑 일치하면 된다.
- 맞다면 처리해주고 종료, 아니라면 pop 한 뒤 다시 반복한다.

우선순위 값 여러 개 확인할 때

- 유효한 우선순위 값을 순차적으로 담을 배열을 생성하여 관리한다.
- top에 있는 값이 유효한 정보인지 확인한다.
  => 최신 값이랑 일치해야 하고, 직전에 뽑은 우선순위 값이랑은 달라야 한다.
- 맞다면 배열에 담아준다.
- 일괄적으로 pop한 뒤, 원하는 개수를 찾을 때까지 반복한다.
- 다 구한 뒤에는 배열에 담은 값들을 pq에 다시 push 한다.

Lazy Update Example

※ 질의1 문제 참조

초기 상태

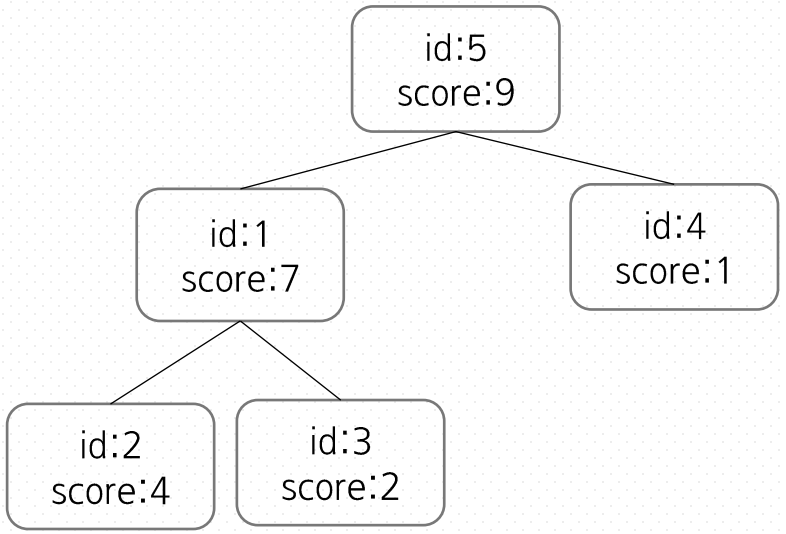
- S[id] = 최신 score 기록, 0이면 존재하지 않음
  (문제에서 score 범위가 1~10억이므로)
- pq는 max heap으로 구성
  우선순위 기준은 **(1) score 큰 순, (2) id 큰 순**

실제 최신 정보를 저장하는 배열

S

id	1	2	3	4	5
score	7	4	2	1	9

pq: max heap



Lazy Update Example

※ 질의1 문제 참조

1번이 삭제된 경우

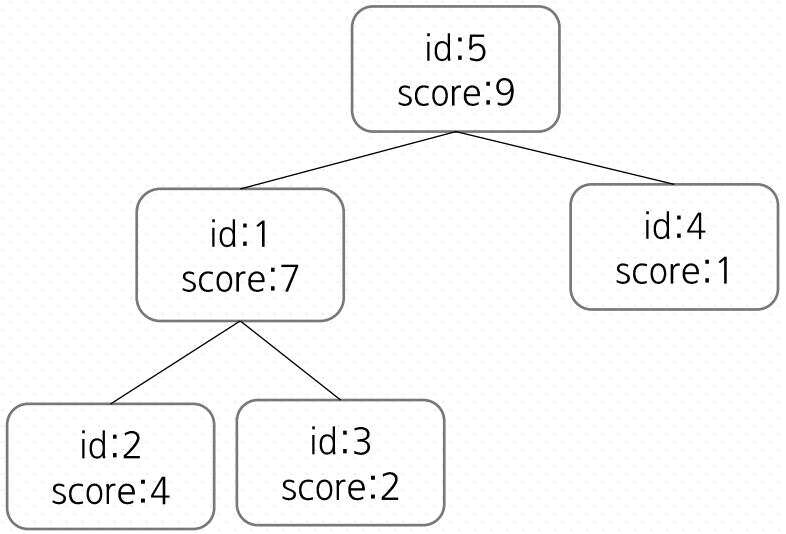
- S[1]을 0으로 하여 삭제됐음을 표시한다.
- pq는 변화없다.

실제 최신 정보를 저장하는 배열

S

id	1	2	3	4	5
score	0	4	2	1	9

pq: max heap



Lazy Update Example

※ 질의1 문제 참조

실제 최신 정보를 저장하는 배열

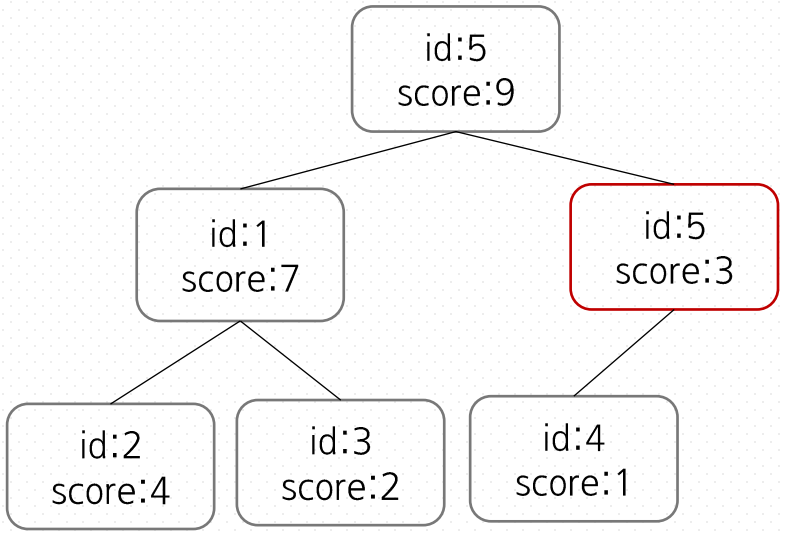
S

id	1	2	3	4	5
score	0	4	2	1	3

5번의 score가 3으로 변경되는 경우

- S[5]를 3으로 하여 변경됐음을 표시한다.
- pq에 {id=5, score=3}을 push하여 최신 정보를 등록한다.
- 기존 노드인 {id=5, score=9}는 유효하지 않지만 pq에 남아있게 된다.

pq: max heap



Lazy Update Example

※ 질의1 문제 참조

실제 최신 정보를 저장하는 배열

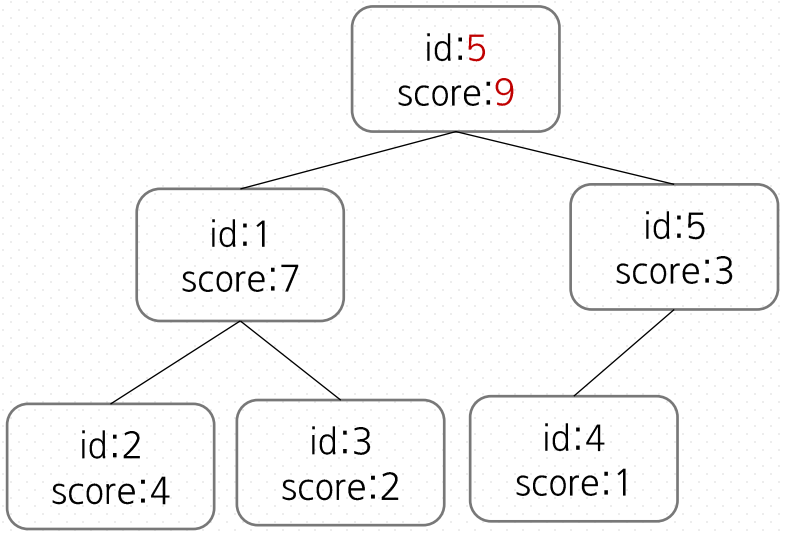
S

id	1	2	3	4	5
score	0	4	2	1	3

우선순위 값 1개를 구하는 경우

- top 노드인 {id=5, score=9} 가 유효한지 확인한다.
  최신 score인 S[5]의 값과 top의 score가 다르므로 유효하지 않다.
  => pop

pq: max heap



Lazy Update Example

※ 질의1 문제 참조

우선순위 값 1개를 구하는 경우(cont.)

- top 노드인 {id=5, score=9} 가 유효한지 확인한다.
  최신 score인 S[5]와 top의 score가 다르므로 유효하지 않다.
  => pop

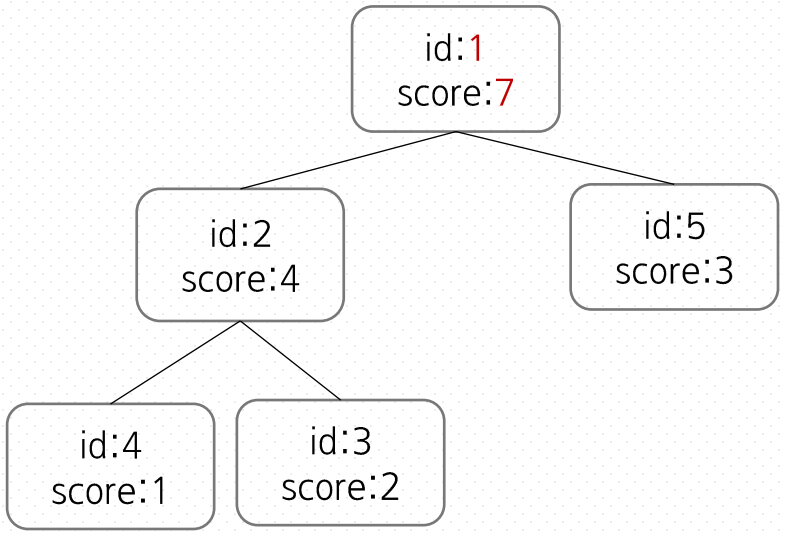
- top 노드인 {id=1, score=7} 이 유효한지 확인한다.
  최신 score인 S[1]와 top의 score가 다르므로 유효하지 않다.
  => pop

실제 최신 정보를 저장하는 배열

S

id	1	2	3	4	5
score	0	4	2	1	3

pq: max heap



Lazy Update Example

※ 질의1 문제 참조

실제 최신 정보를 저장하는 배열

S

id	1	2	3	4	5
score	0	4	2	1	3

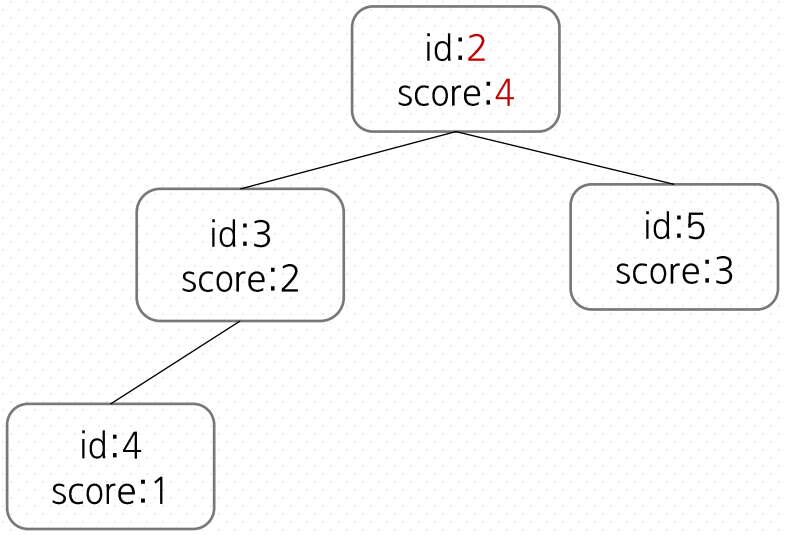
우선순위 값 1개를 구하는 경우(cont.)

- top 노드인 {id=5, score=9} 가 유효한지 확인한다.
  최신 score인 S[5]와 top의 score가 다르므로 유효하지 않다.
  => pop

- top 노드인 {id=1, score=7} 이 유효한지 확인한다.
  최신 score인 S[1]와 top의 score가 다르므로 유효하지 않다.
  => pop

- top 노드인 {id=2, score=4} 가 유효한지 확인한다.
  최신 score인 S[2]와 top의 score가 같으므로 유효하다.
  => **우선순위 값 확정**

pq: max heap



Lazy Update Example

※ 질의+ 문제 참조

우선순위 값 3개를 구하는 경우

1. **우선순위 3개 구하는 방법**
   - pq는 최우선순위 값을 빠르게 구할 수 있지만 3번째 값을 바로 구하지는 못한다.
   - 이를 위해 pop을 해나가며 별도 배열에 기록해야 한다.
   - 그렇게 3개의 값을 배열에 기록했으면 요구사항을 수행하고 다시 pq에 push로 돌려 넣어준다.

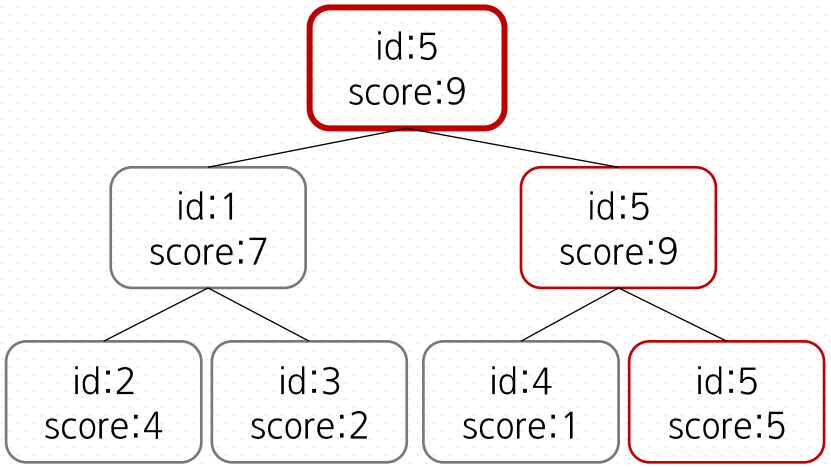
2. **유효성 검증**
   - 초기 상태에서 update(id=5, score=5), update(id=5, score=9)가 수행된 경우를 생각해보자.
   - S[5] = 9,
     pq에는 id=5인 노드가 총 3개 등록된다. 이중, 1개만이 최신 정보가 된다.
   - 유효성 검증을 할때 단순 S[id] 하고만 비교하게 되면 2개가 유효하다고 판별된다.
     우선순위 1개를 구할 때는 상관 없지만 3개를 구하는 경우라면 {5, 9}는 한 개만 기록되어야 한다.
   - 이를 위해, **중복 검사**를 추가적으로 해줘야 한다.
     직전에 유효하다고 기록한 정보가 top과 같으면 유효하지 않다고 판별한다.
     따라서, {5, 9}는 한번만 기록된다.

실제 최신 정보를 저장하는 배열

S

id	1	2	3	4	5
score	7	4	2	1	9

pq: max heap



Lazy Update Example

※ 질의+ 문제 참조

우선순위 값 3개를 구하는 경우

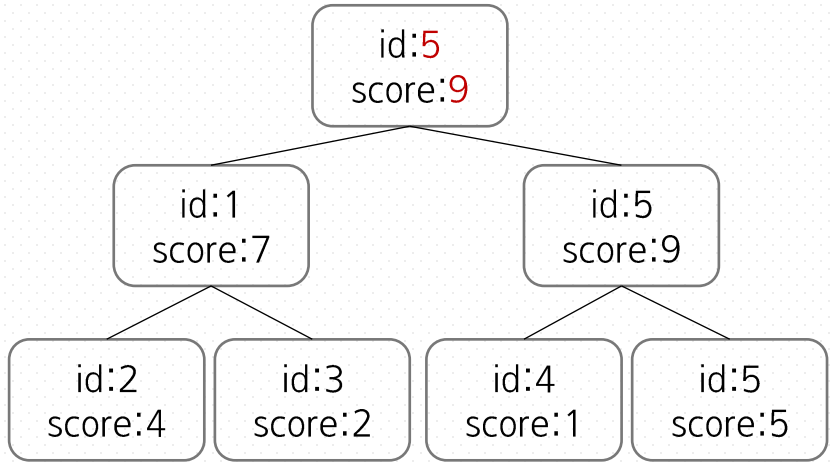
- top 노드인 {5, 9} 가 최신 값이랑 동일하다.
  아직 기록된 유효한 정보가 없으므로 중복도 없다. => 기록, pop

실제 최신 정보를 저장하는 배열

S

id	1	2	3	4	5
score	7	4	2	1	9

pq: max heap



	0	1	2
id	5		
score	9		

Lazy Update Example

※ 질의+ 문제 참조

우선순위 값 3개를 구하는 경우(cont.)

- top 노드인 {5, 9} 가 최신 값이랑 동일하다.
  아직 기록된 유효한 정보가 없으므로 중복도 없다. => 기록, pop

- top 노드인 {5, 9} 가 최신 값이랑 동일하다.
  직전에 기록된 유효한 정보와 동일하므로 유효하지 않다. => pop
  (직전만 확인하면 되는 이유는 중복이라면 연속돼서 나오기 때문)

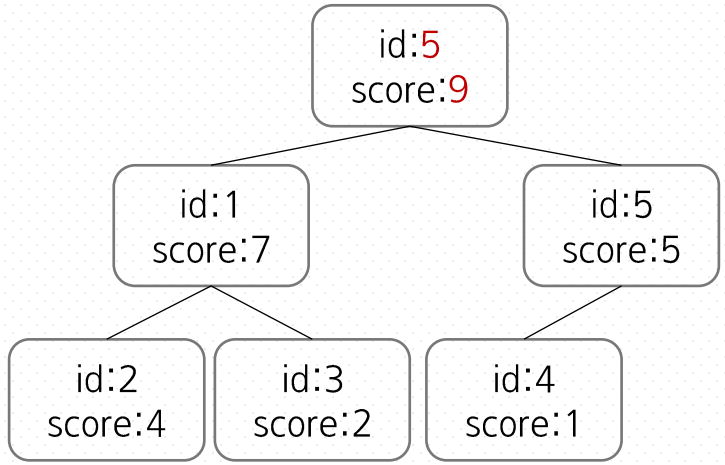
	0	1	2
id	5		
score	9		

실제 최신 정보를 저장하는 배열

S

id	1	2	3	4	5
score	7	4	2	1	9

pq: max heap



Lazy Update Example

※ 질의+ 문제 참조

우선순위 값 3개를 구하는 경우(cont.)

- top 노드인 {5, 9} 가 최신 값이랑 동일하다.
  아직 기록된 유효한 정보가 없으므로 중복도 없다. => 기록, pop

- top 노드인 {5, 9} 가 최신 값이랑 동일하다.
  직전에 기록된 유효한 정보와 동일하므로 유효하지 않다. => pop
  (직전만 확인하면 되는 이유는 중복이라면 연속돼서 나오기 때문)

- top 노드인 {1, 7} 이 최신 값이랑 동일하다.
  직전에 기록된 유효한 정보와 동일하지 않으므로 유효하다. => 기록, pop

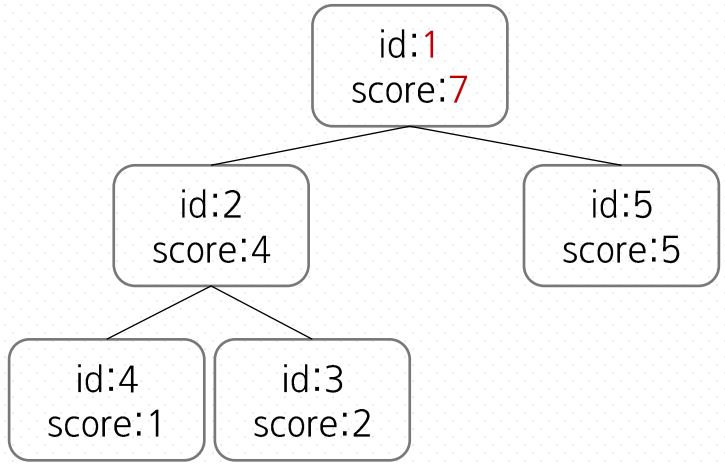
	0	1	2
id	5	1	
score	9	7	

실제 최신 정보를 저장하는 배열

S

id	1	2	3	4	5
score	7	4	2	1	9

pq: max heap



Lazy Update Example

※ 질의+ 문제 참조

우선순위 값 3개를 구하는 경우(cont.)

- top 노드인 {5, 9} 가 최신 값이랑 동일하다.
  아직 기록된 유효한 정보가 없으므로 중복도 없다. => 기록, pop
- top 노드인 {5, 9} 가 최신 값이랑 동일하다.
  직전에 기록된 유효한 정보와 동일하므로 유효하지 않다. => pop
  (직전만 확인하면 되는 이유는 중복이라면 연속돼서 나오기 때문)
- top 노드인 {1, 7} 이 최신 값이랑 동일하다.
  직전에 기록된 유효한 정보와 동일하지 않으므로 유효하다. => 기록, pop
- top 노드인 {5, 5} 가 최신 값과 다르므로 유효하지 않다. => pop

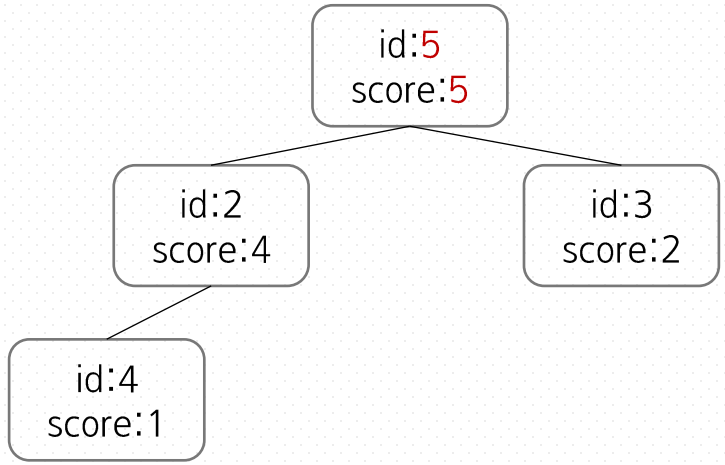
	0	1	2
id	5	1	
score	9	7	

실제 최신 정보를 저장하는 배열

S

id	1	2	3	4	5
score	7	4	2	1	9

pq: max heap



Lazy Update Example

※ 질의+ 문제 참조

우선순위 값 3개를 구하는 경우(cont.)

- top 노드인 {5, 9} 가 최신 값이랑 동일하다.
  아직 기록된 유효한 정보가 없으므로 중복도 없다. => 기록, pop
- top 노드인 {5, 9} 가 최신 값이랑 동일하다.
  직전에 기록된 유효한 정보와 동일하므로 유효하지 않다. => pop
  (직전만 확인하면 되는 이유는 중복이라면 연속돼서 나오기 때문)
- top 노드인 {1, 7} 이 최신 값이랑 동일하다.
  직전에 기록된 유효한 정보와 동일하지 않으므로 유효하다. => 기록, pop
- top 노드인 {5, 5} 가 최신 값과 다르므로 유효하지 않다. => pop
- top 노드인 {2, 5} 가 최신 값이랑 동일하다.
  직전에 기록된 유효한 정보와 다르므로 유효하다. => 기록, pop

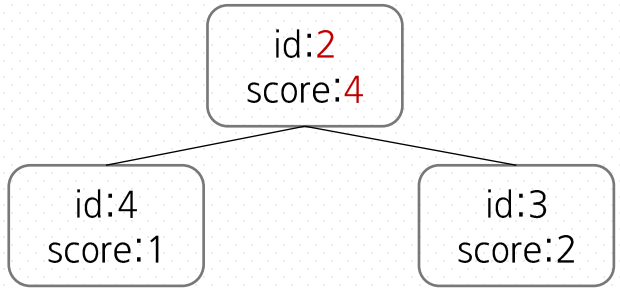
	0	1	2
id	5	1	2
score	9	7	4

실제 최신 정보를 저장하는 배열

S

id	1	2	3	4	5
score	7	4	2	1	9

pq: max heap



Lazy Update Example

※ 질의+ 문제 참조

우선순위 값 3개를 구하는 경우(cont.)

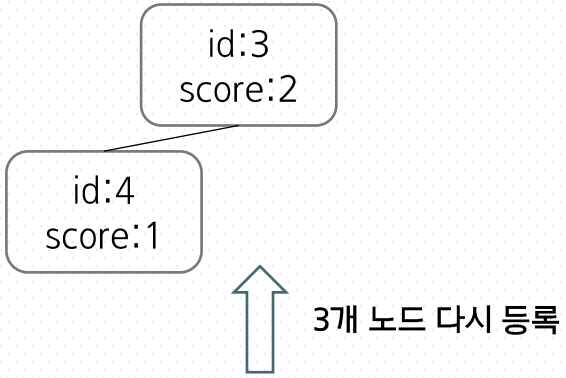
- 우선순위 값을 전부 구했으므로 필요한 동작 수행 후, 전부 재등록 한다.

실제 최신 정보를 저장하는 배열

S

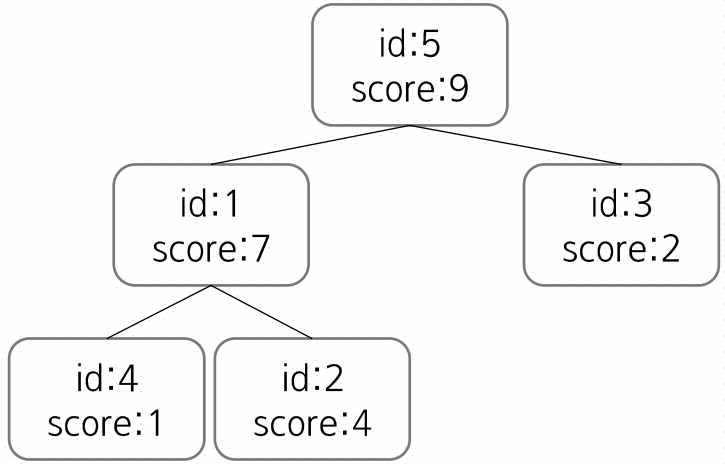
id	1	2	3	4	5
score	7	4	2	1	9

우선순위 값구한 직후의 pq



	0	1	2
id	5	1	2
score	9	7	4

재등록 후 pq



Real-Time Update

PQ 기능 추가

- STL PQ는 사용하지 못하고 직접 구현한 뒤, update(), erase() 기능을 추가해줘야 한다.

원소가 삭제 될 때 : erase()

- 마지막 원소를 해당 위치(pos[id])로 갖고 와서 up(), down()으로 본인 자리를 찾아간다.

원소가 변경 될 때 : update()

- 해당 위치(pos[id]) 값을 update 해주고 up(), down()으로 본인 자리를 찾아간다.

우선순위 값을 확인할 때

- heap이 실시간 업데이트가 완료된 상태이므로 루트 노드가 top임이 보장된다.

Real-Time Update Example

실제 최신 정보 저장

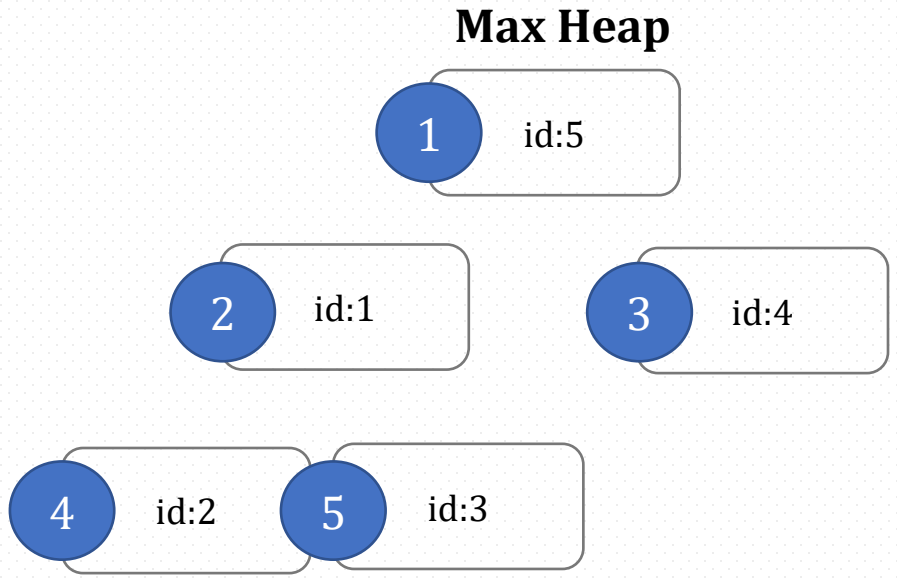
S	1	2	3	4	5
	7	4	2	1	9

id의 heap index 번호

Pos	1	2	3	4	5
	2	4	5	3	1

초기 상태

- S : 최신 정보 저장
  Pos : id별 heap 내의 index 저장
  heap : 1-base 예시
- lazy update와는 다르게 heap을 구성하는 요소는 id값 하나면 충분하다.
- 물론, 별도의 S배열 없이 heap 자체에서 data값까지 같이 기록해도 된다.



Real-Time Update Example

실제 최신 정보 저장

S	1	2	3	4	5
	~~7~~	4	2	1	9

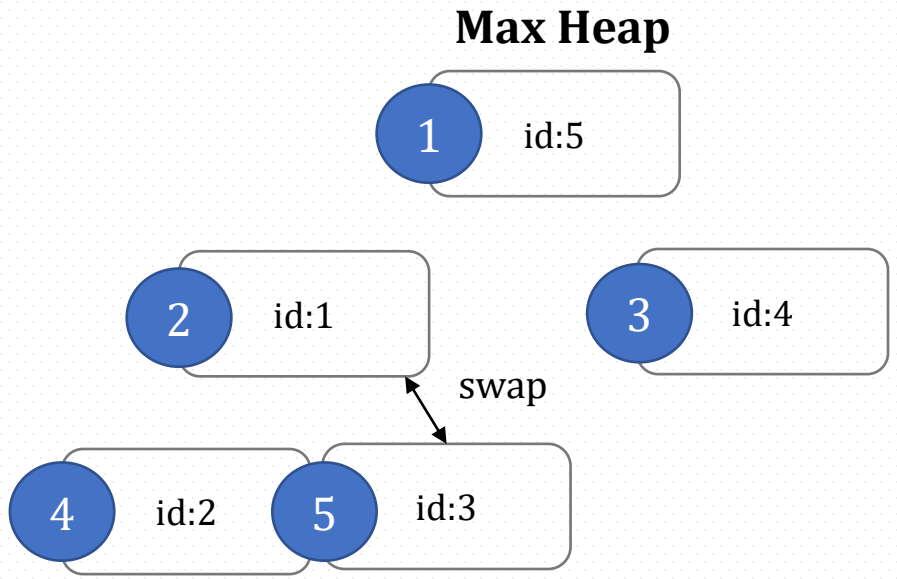
id의 heap index 번호

Pos	1	2	3	4	5
	2	4	5	3	1

swap

S[1]을 지우는 경우

- pop과 비슷하게 heap의 마지막 노드를 지우고자 하는 위치와 바꿔준다.
- 지우고자 하는 위치는 pos[1]을 참조하면 된다.
- heap의 노드가 바뀔 때는 pos도 항상 같이 바뀌어야 한다.
- 바꾼 후, heap의 성질을 만족하기 위한 자리를 찾아가야 한다.



Real-Time Update Example

실제 최신 정보 저장

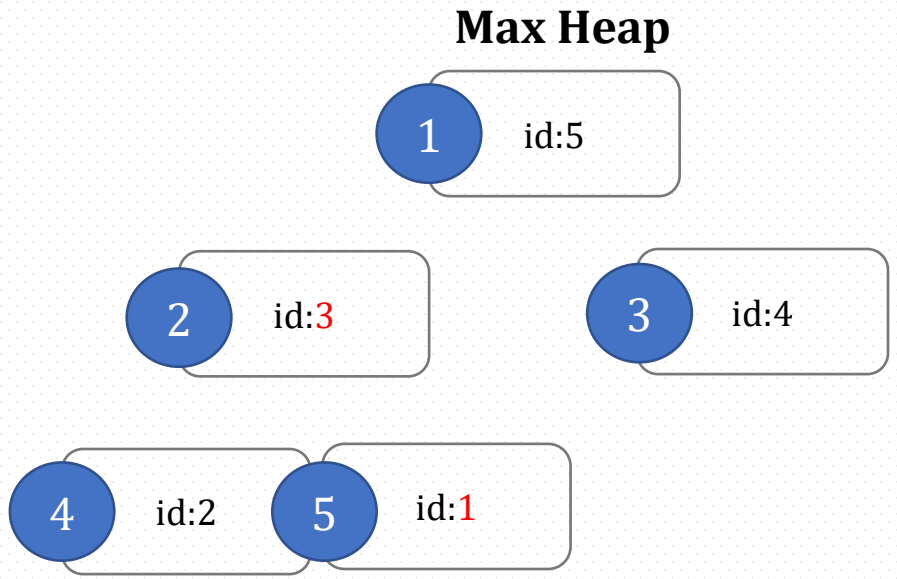
S	1	2	3	4	5
	~~7~~	4	2	1	9

id의 heap index 번호

Pos	1	2	3	4	5
	5	4	2	3	1

S[1]을 지우는 경우

- pop과 비슷하게 heap의 마지막 노드를 지우고자 하는 위치와 바꿔준다.
- 지우고자 하는 위치는 pos[1]을 참조하면 된다.
- heap의 노드가 바뀔 때는 pos도 항상 같이 바뀌어야 한다.
- 바꾼 후, heap의 성질을 만족하기 위한 자리를 찾아가야 한다.



Real-Time Update Example

실제 최신 정보 저장

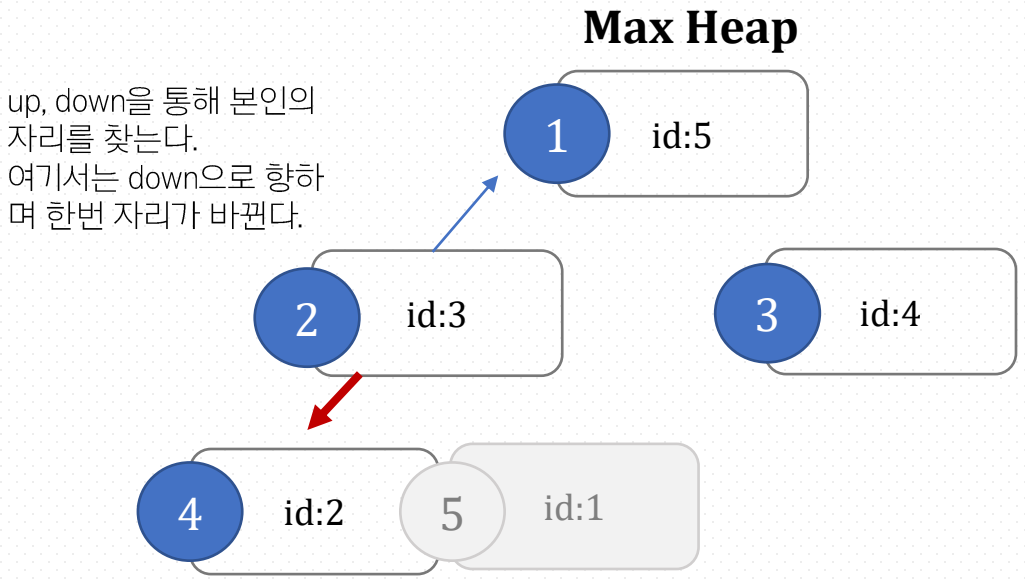
S	1	2	3	4	5
	~~7~~	4	2	1	9

id의 heap index 번호

Pos	1	2	3	4	5
	5	4	2	3	1

S[1]을 지우는 경우

- pop과 비슷하게 heap의 마지막 노드를 지우고자 하는 위치와 바꿔준다.
- 지우고자 하는 위치는 pos[1]을 참조하면 된다.
- heap의 노드가 바뀔 때는 pos도 항상 같이 바뀌어야 한다.
- 바꾼 후, heap의 성질을 만족하기 위한 자리를 찾아가야 한다.



Real-Time Update Example

실제 최신 정보 저장

S	1	2	3	4	5
	~~7~~	4	2	1	9

id의 heap index 번호

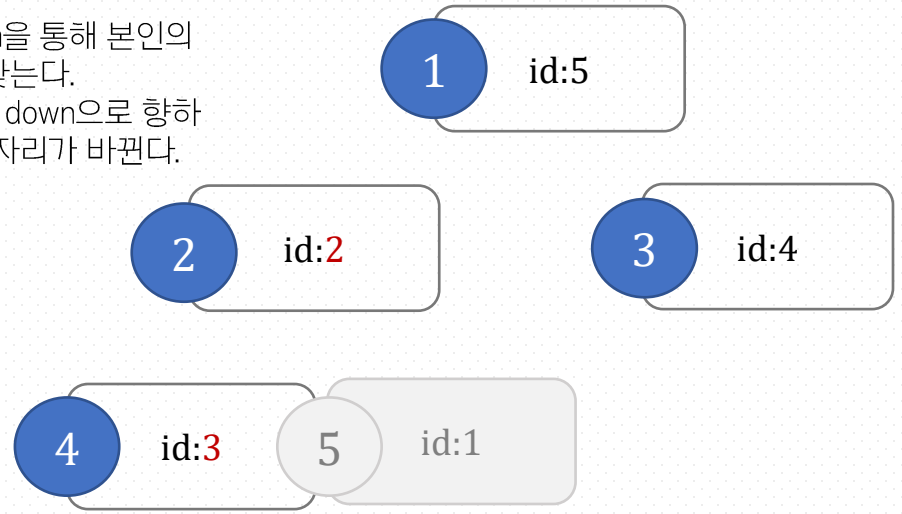
Pos	1	2	3	4	5
	5	2	4	3	1

S[1]을 지우는 경우

- pop과 비슷하게 heap의 마지막 노드를 지우고자 하는 위치와 바꿔준다.
- 지우고자 하는 위치는 pos[1]을 참조하면 된다.
- heap의 노드가 바뀔 때는 pos도 항상 같이 바뀌어야 한다.
- 바꾼 후, heap의 성질을 만족하기 위한 자리를 찾아가야 한다.

Max Heap

up, down을 통해 본인의 자리를 찾는다.
여기서는 down으로 향하며 한번 자리가 바뀐다.



Real-Time Update Example

실제 최신 정보 저장

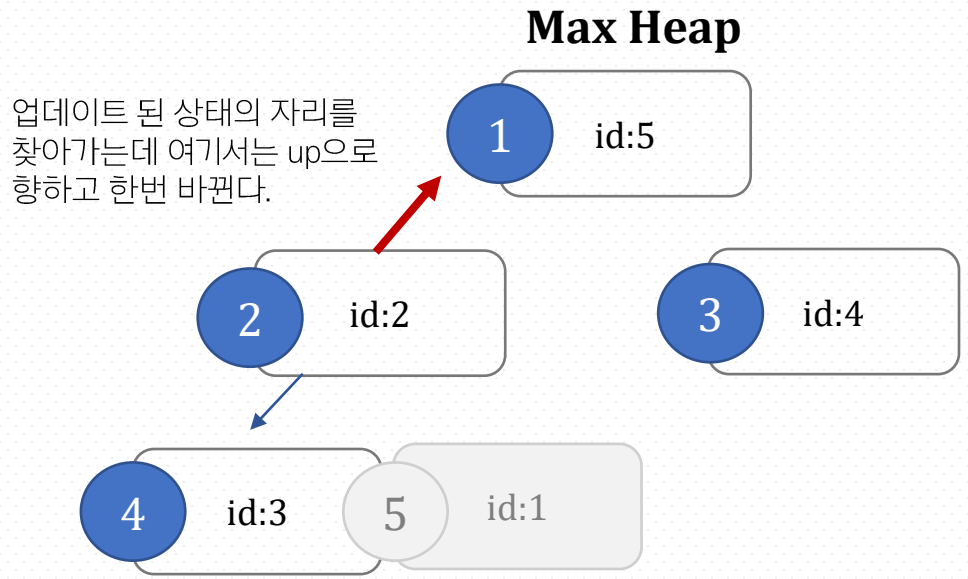
S	~~1~~	2	3	4	5
	~~7~~	10	2	1	9

id의 heap index 번호

Pos	~~1~~	2	3	4	5
	~~5~~	2	4	3	1

S[2]을 10으로 업데이트 하는 경우

- pos[2]를 통해 heap 내의 index를 접근한다.
- 값이 update 되었으므로 up, down을 통해 본인 자리를 찾아간다.



Real-Time Update Example

실제 최신 정보 저장

S

~~1~~	2	3	4	5
~~7~~	10	2	1	9

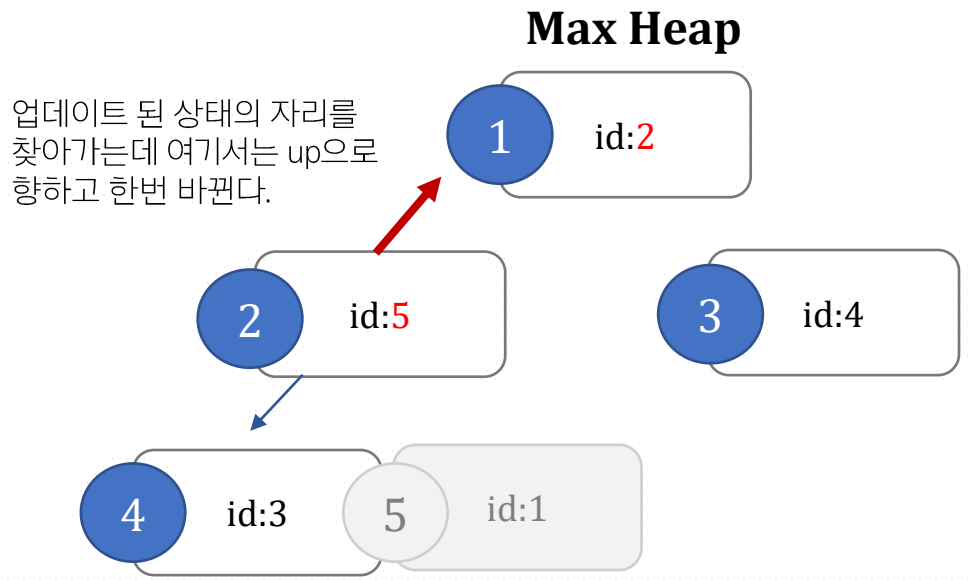
id의 heap index 번호

Pos

~~1~~	2	3	4	5
~~5~~	1	4	3	2

S[2]을 10으로 업데이트 하는 경우

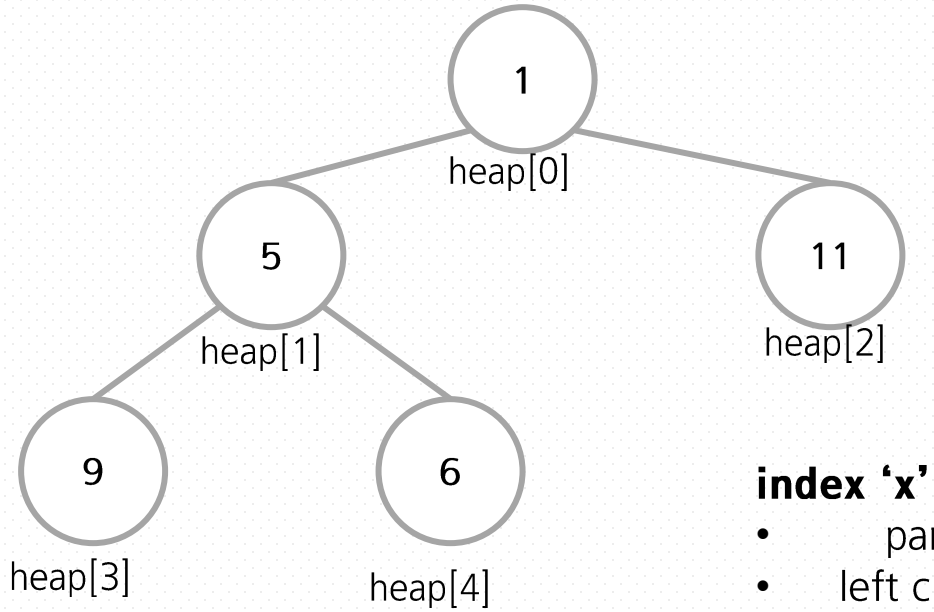
- pos[2]를 통해 heap 내의 index를 접근한다.
- 값이 update 되었으므로 up, down을 통해 본인 자리를 찾아간다.



Non-stl Priority_Queue

SWEA reference code version

1. min_heap
2. 배열 사용
3. 0-base : (0~heapSize-1) index
4. 최우선순위 노드 heap[0]



index ‘x’ 기준

- parent = (x-1)/2
- left child = x * 2 + 1
- right child = x * 2 + 2

index	0	1	2	3	4	5
score	1	5	11	9	6	

Non-stl Priority_Queue

```
#define MAX_SIZE 100

int heap[MAX_SIZE];
int heapSize = 0;

void heapInit(void)
{
    heapSize = 0;
}

int heapPush(int score)
{
    if (heapSize + 1 > MAX_SIZE)
    {
        printf("queue is full!");
        return 0;
    }

    heap[heapSize] = score;

    int current = heapSize;
    while (current > 0 && heap[current] < heap[(current - 1) / 2])
    {
        int temp = heap[(current - 1) / 2];
        heap[(current - 1) / 2] = heap[current];
        heap[current] = temp;
        current = (current - 1) / 2;
    }

    heapSize = heapSize + 1;

    return 1;
}
```

```
int heapPop(int *score)
{
    if (heapSize <= 0)
    {
        return -1;
    }

    *score = heap[0];
    heapSize = heapSize - 1;

    heap[0] = heap[heapSize];

    int current = 0;
    while (current * 2 + 1 < heapSize)
    {
        int child;
        if (current * 2 + 2 == heapSize)
        {
            child = current * 2 + 1;
        }
        else
        {
            child = heap[current * 2 + 1] < heap[current * 2 + 2] ? current * 2 + 1 : current * 2 + 2;
        }

        if (heap[current] < heap[child])
        {
            break;
        }

        int temp = heap[current];
        heap[current] = heap[child];
        heap[child] = temp;

        current = child;
    }
    return 1;
}
```

Non-stl Priority_Queue

```
#define MAX_SIZE 100

struct MinPQ {
    int heap[MAX_SIZE];
    int heapSize = 0;

    void init(void)
    {
        heapSize = 0;
    }

    void push(int score)
    {
        ①
    }

    void pop()
    {
        ②
    }
    int top()
    {
        return heap[0];
    }
};

MinPQ minpq;
```

```
int heapPush(int score)
{
    if (heapSize + 1 > MAX_SIZE)
    {
        printf("queue is full!");
        return 0;
    }

    heap[heapSize] = score;

    int current = heapSize;
    while (current > 0 && heap[current] < heap[(current - 1) / 2])
    {
        int temp = heap[(current - 1) / 2];
        heap[(current - 1) / 2] = heap[current];
        heap[current] = temp;
        current = (current - 1) / 2;
    }

    heapSize = heapSize + 1;

    return 1;
}
```

Non-stl Priority_Queue

```
#define MAX_SIZE 100

struct MinPQ {
    int heap[MAX_SIZE];
    int heapSize = 0;

    void init(void)
    {
        heapSize = 0;
    }

    void push(int score)
    {
        ①
    }

    void pop()
    {
        ②
    }
    int top()
    {
        return heap[0];
    }
};

MinPQ minpq;
```

```
int heapPop(int *score)
{
    if (heapSize <= 0)
    {
        return -1;
    }

    *score = heap[0];
    heapSize = heapSize - 1;

    heap[0] = heap[heapSize];

    int current = 0;
    while (current * 2 + 1 < heapSize)
    {
        int child;
        if (current * 2 + 2 == heapSize)
        {
            child = current * 2 + 1;
        }
        else
        {
            child = heap[current * 2 + 1] < heap[current * 2 + 2] ? current * 2 + 1 : current * 2 + 2;
        }

        if (heap[current] < heap[child])
        {
            break;
        }

        int temp = heap[current];
        heap[current] = heap[child];
        heap[child] = temp;

        current = child;
    }
    return 1;
}
```

Non-stl Priority_Queue

```
void push(int score) {
    heap[heapSize] = score;

    int current = heapSize;
    while (current > 0 && heap[current] < heap[(current - 1) / 2])
    {
        int temp = heap[(current - 1) / 2];
        heap[(current - 1) / 2] = heap[current];
        heap[current] = temp;
        current = (current - 1) / 2;
    }

    heapSize = heapSize + 1;
}
```

- 마지막 위치에 새로운 값 score 추가
- current = 새로 추가한 index로 설정
- current가 root가 아니고 부모 노드보다 우선순위 높은 동안 수행
- current와 부모 노드 swap
- current를 부모 노드 위치로 변경
  swap 하였으므로 기존 current 노드를 가리키게 된다.
- heapSize 증가
  [0 ~ heapSize - 1] 에 노드 존재

Non-stl Priority_Queue

```
void pop() {
    heapSize = heapSize - 1;

    heap[0] = heap[heapSize];

    int current = 0;
    while (current * 2 + 1 < heapSize)
    {
        int child;
        if (current * 2 + 2 == heapSize)
        {
            child = current * 2 + 1;
        }
        else
        {
            child = heap[current * 2 + 1] < heap[current * 2 + 2] ? current * 2 + 1 : current * 2 + 2;
        }

        if (heap[current] < heap[child])
        {
            break;
        }

        int temp = heap[current];
        heap[current] = heap[child];
        heap[child] = temp;

        current = child;
    }
}
```

- heapSize 감소
- 마지막 값을 root로 복사
- current = root 로 설정 후, 아래로 자기 자리 찾아나감
- current가 왼쪽 자식이 있는 동안 반복
- child = current의 우선순위 높은 자식 선택
- 왼쪽 자식밖에 없는 경우
- 왼쪽, 오른쪽 자식 있는 경우, 두 값 비교 후 선택
- current 와 선택된 자식 child 중 current가 우선 순위가 높으면 종료
- child가 우선순위가 높으므로 swap
  current는 child로 이동


감사합니다