STL 기초

Priority Queue

한컴에듀케이션

목 차

- priority queue
- heap
- STL PQ
- reference PQ
- Lazy update
- Real-time update

Priority Queue

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

- 구현

배열 :	push O(1)	pop O(n)
힙 :	**push O(log n)**	**pop O(log n)**

Heap

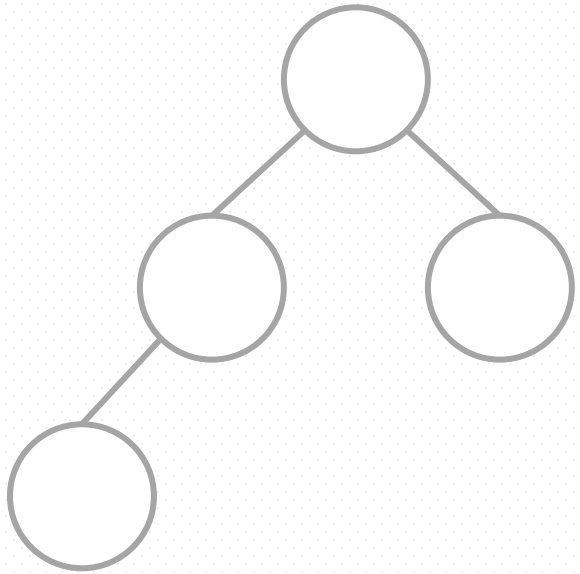
두 가지 특성 만족

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

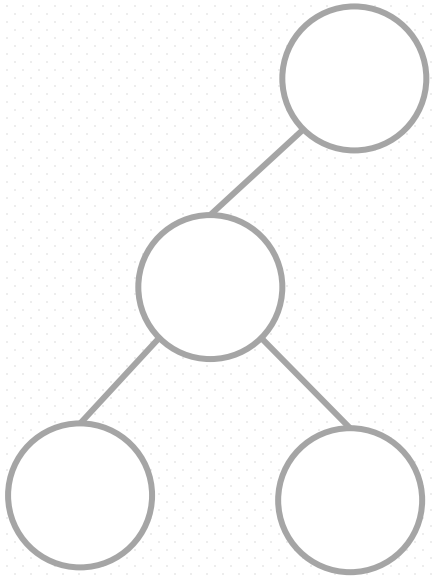
=> 최우선순위 노드는 루트에 존재

Heap

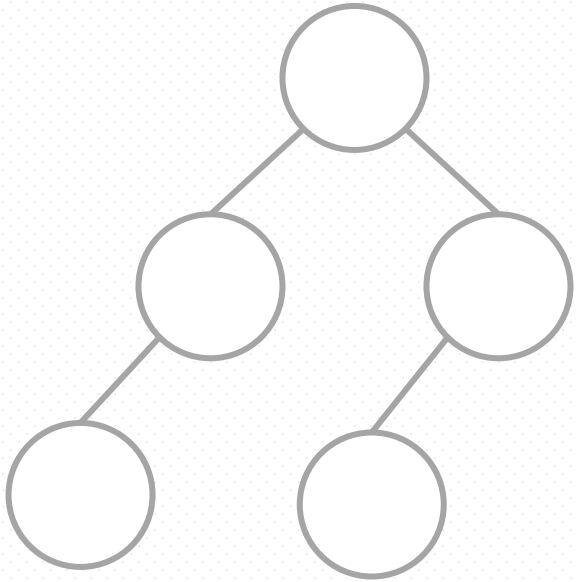
1. **완전 이진 트리**
   - 모든 노드의 자식 노드는 최대 2개
   - 노드는 위에서부터, 왼쪽부터 꽉 채워진다.



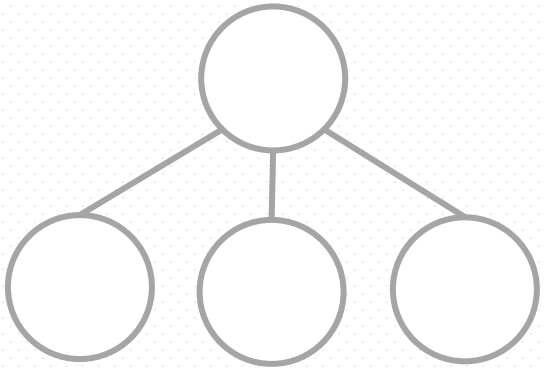
(O)



(X)



(X)

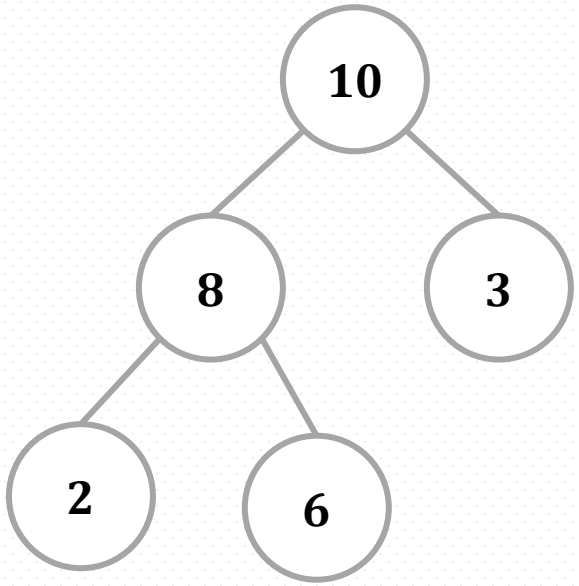


(X)

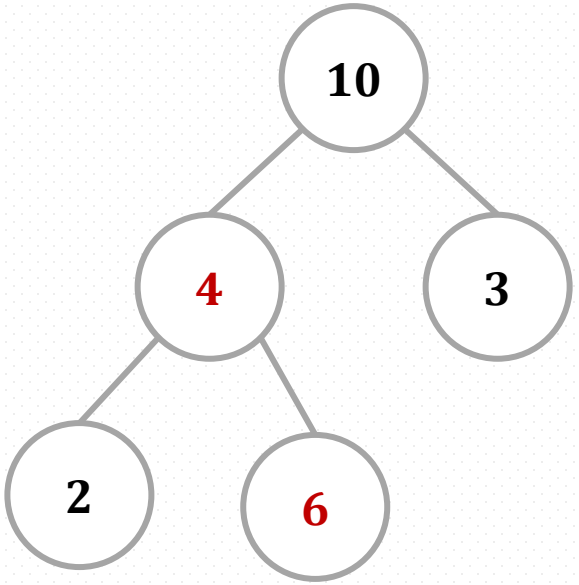
Heap

2. 부모 노드의 우선순위가 자식 노드의 우선순위보다 높다.

ex) 값이 클수록 우선순위가 높다 : MAX_HEAP



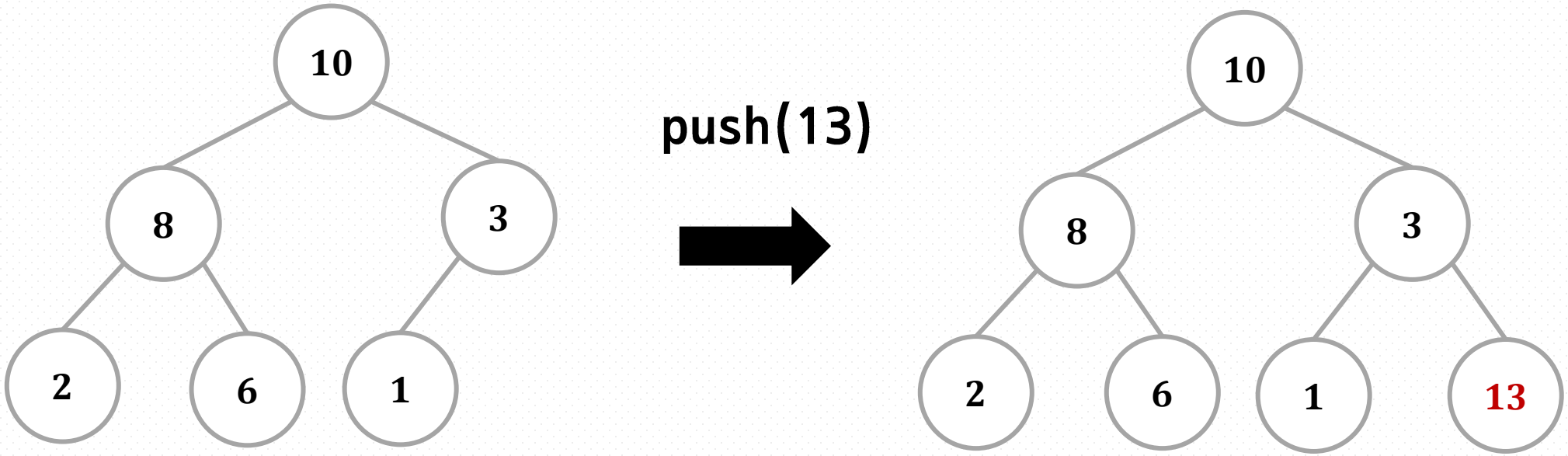
(O)



(X)

Heap : push

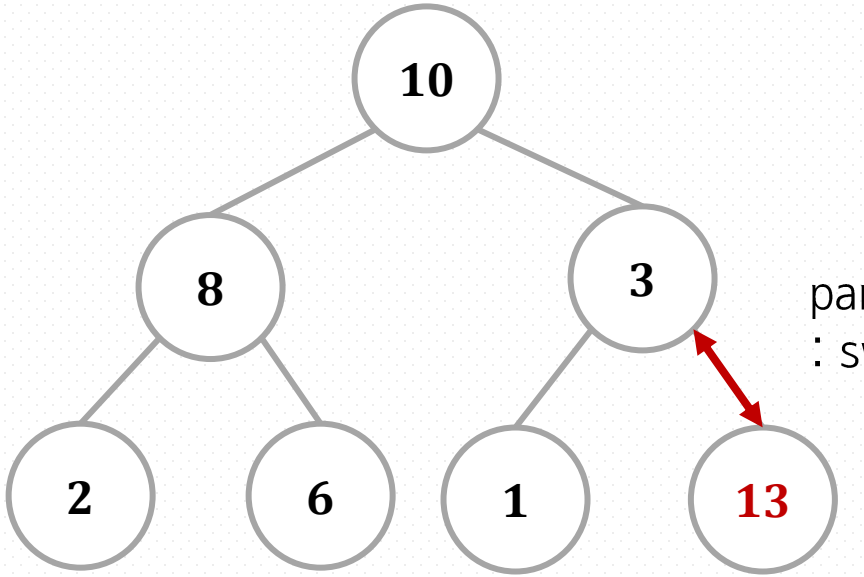
1. 마지막 위치에 추가
2. 추가한 노드를 부모 노드랑 비교하며 우선순위 높으면 바꿔 올라간다.



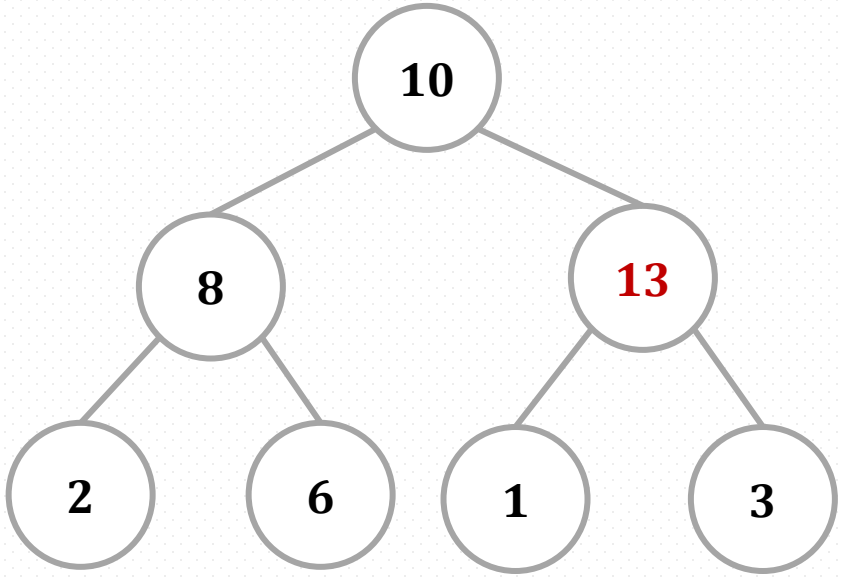
Max_Heap example

Heap : push

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



parent(3) < child(13)
: swap



Max_Heap example

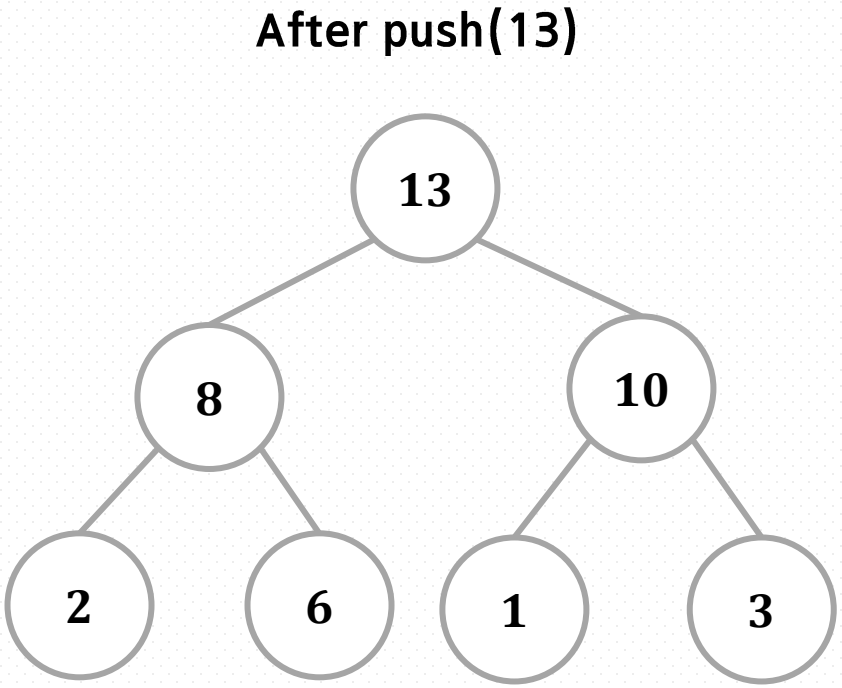
Heap : push

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

Max_Heap example

Heap : push

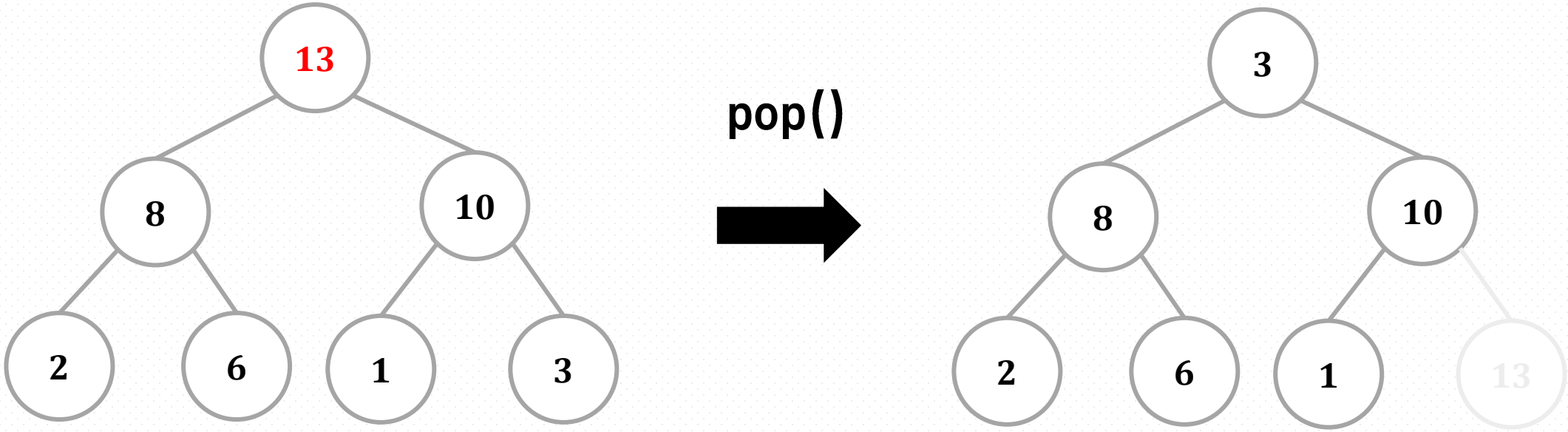
1. 마지막 위치에 추가
2. 추가한 노드를 부모 노드랑 비교하며 우선순위 높으면 바꿔 올라간다.



Max_Heap example

Heap : pop

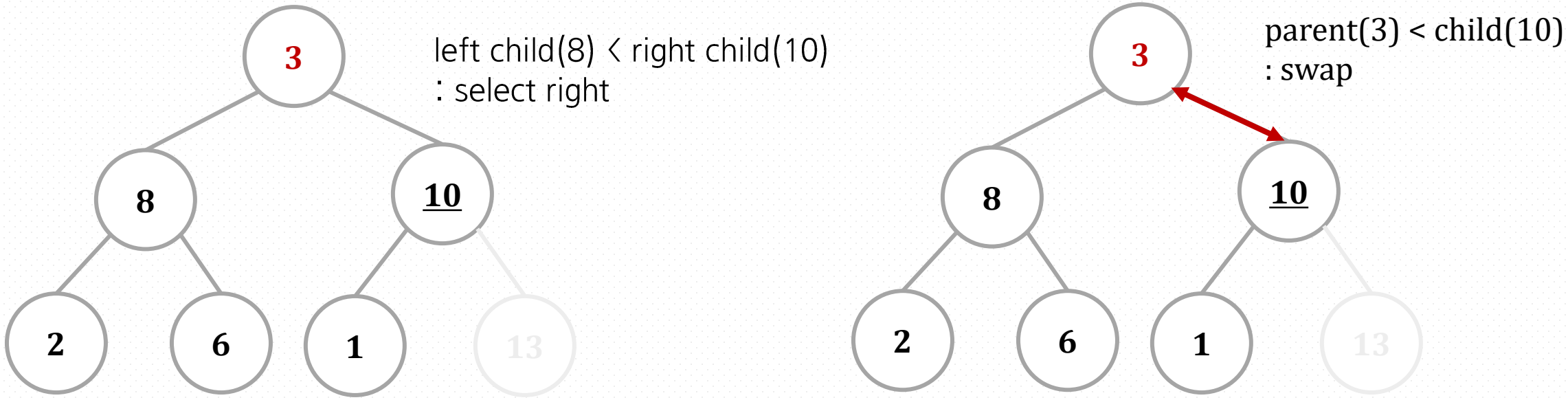
1. 루트와 마지막 노드를 바꾸고 heap size를 감소시킨다.
2. 루트 노드에서 우선순위 높은 자식 노드와 비교하여 우선순위가 낮으면 바꿔 내려간다.



Max_Heap example

Heap : pop

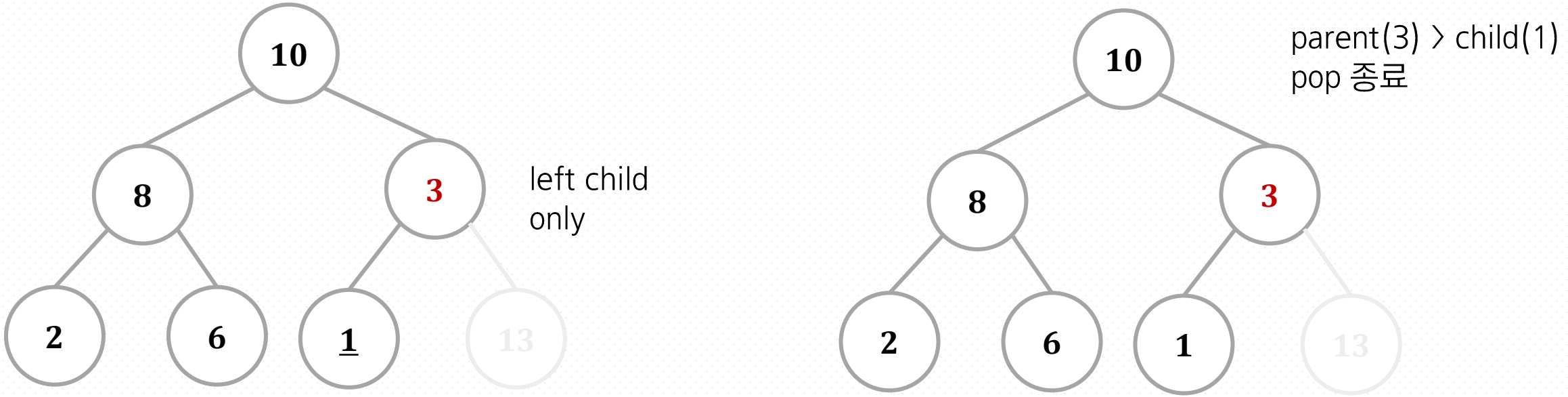
1. 루트와 마지막 노드를 바꾸고 heap size를 감소시킨다.
2. 루트 노드에서 우선순위 높은 자식 노드와 비교하여 우선순위가 낮으면 바꿔 내려간다.



Max_Heap example

Heap : pop

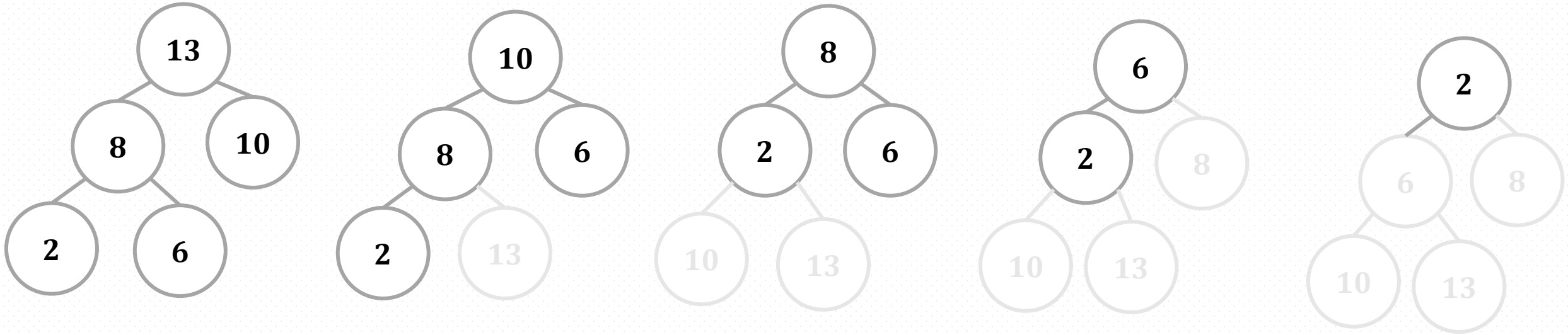
1. 루트와 마지막 노드를 바꾸고 heap size를 감소시킨다.
2. 루트 노드에서 우선순위 높은 자식 노드와 비교하여 우선순위가 낮으면 바꿔 내려간다.



Max_Heap example

Heap Sort

1. n개의 data를 heap에 push
2. n번 pop 진행 (pop 할 때, 루트와 마지막 값 swap)



Max Heap example

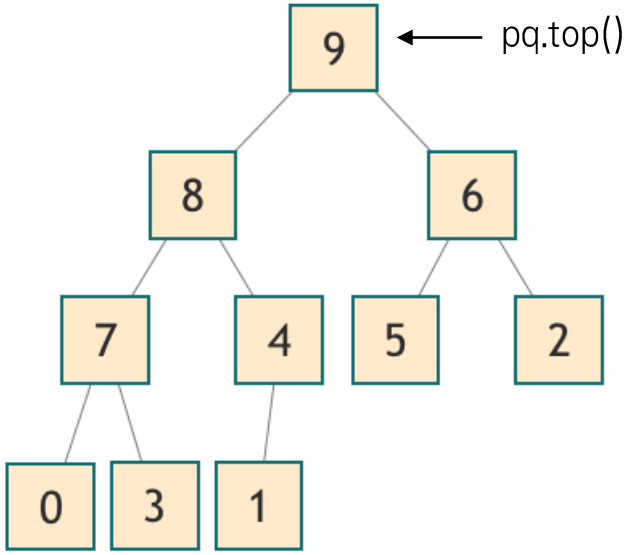
0	1	2	3	4
2	6	8	10	13

Max Heap => 오름차순 Min Heap => 내림차순

STL Priority_Queue

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

```
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 *(다른 방법도 있지만 이 방법만 다룬다)*

1. pre-defined function object

- **less<T>** : 큰 값이 top

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

- **greater<T>** : 작은 값이 top

```
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);
    }
};
```

Compare 설정 예

- compare 기준 : less<int> => 값이 클수록 우선순위 높음

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

- compare 기준 : greater<int> => 값이 작을수록 우선순위 높음

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

- compare 기준 : 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< overloading 필요

```
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) => !comp(b,a)
3. comp(a,b) && comp(b,c) => comp(a,c)
4. !comp(a,b) && !comp(b,a) => (a==b)

※comp(a,b) : a<b or a>b

요약

- <, > operator만 사용
- 좌우항이 같은 형태
  ex) l.x < r.x , l.x+l.y < r.x+r.y , ~~l.x < r.y~~

Update / Erase

- **Lazy Update**
  - 실제 업데이트 되는 정보를 별도의 객체(배열)에 저장해 놓는다.
  - heap에는 push 되는 시점의 정보가 들어간다.
  - top을 확인할 때, 유효하지 않는 값들은 버린다.
  - 유효하지 않은 값이란 최신 데이터랑 push된 시점의 데이터가 같지 않은 값이다.
  - heap에는 유효하지 않은 값들도 들어가 있다.

- **Real-time Update**
  - 각 원소들의 heap index를 기록하는 별도의 배열(pos[])이 필요하다.
  - heap의 위치가 swap 될때마다 pos[] 배열도 같이 swap된다.
  - 특정 id가 update되거나 erase 될 때,
    pos[id]로 바뀐 id의 heap index에 접근한다.
  - up, down을 통해 본인 자리를 찾아간다.

Lazy Update

- **원소가 erase될 때**
  실제 데이터를 저장하는 객체(배열)의 값을 삭제됐다고 표시한다.
  top을 확인할 때 실제 정보와 같은지 판별하여 그렇지 않다면 pop하고
  유효한 정보가 나올 때까지 반복한다.

- **원소가 update 될 때**
  update된 정보를 heap에 push 한다.
  그리고 top을 확인할 때 실제 정보와 같은지 판별하여 그렇지 않다면
  pop하고 유효한 정보가 나올 때까지 반복한다.

Lazy Update Example

heap에는 유효하지 않은 정보들도 들어가 있을것이므로
실제 유효한 정보들을 저장하는 별도의 객체(배열)가 필요하다.

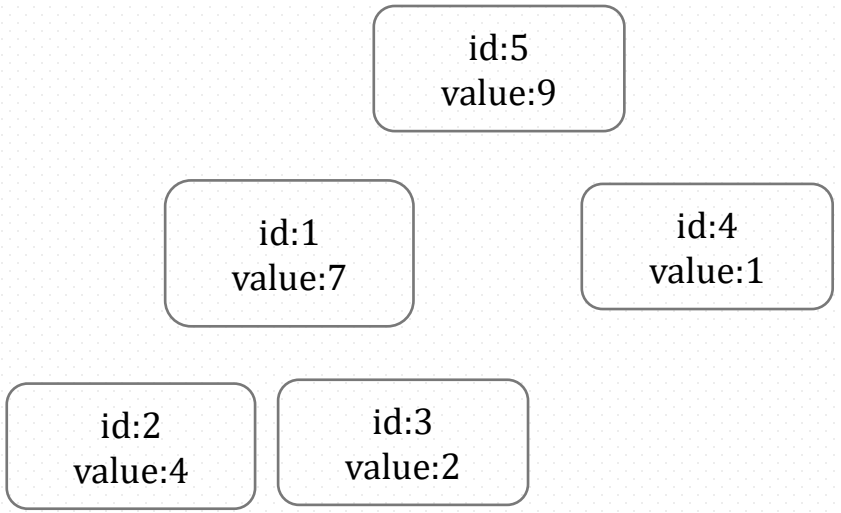
유효한지 판단해주기 위한 식별 값들이 heap data에 포함되어야 한다.
(실제 객체의 id, 변하는 value)

실제 유효한 정보를 저장하는 배열

S

1	2	3	4	5
7	4	2	1	9

Max Heap



Lazy Update Example

1번이 삭제된 경우

value 범위를 벗어나는 값을 설정하여 지워졌음을 표시한다.

예제에서는 value가 양수라고 가정하고 -1로 표현해 볼 수 있다.

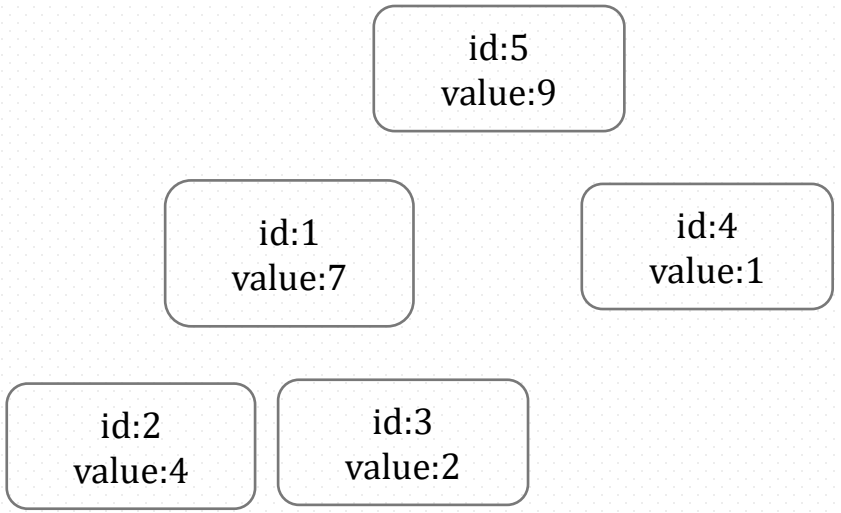
힙은 따로 처리해줄 필요가 없다.

실제 유효한 정보를 저장하는 배열

S

1	2	3	4	5
-1	4	2	1	9

Max Heap



Lazy Update Example

우선순위 값을 구하는 경우

Lazy update의 경우, top에 왔을 때 유효성 검증을 하여 사용해야 한다.

heap[1]의 value 와 S[5]의 값이 같은지 비교한다.

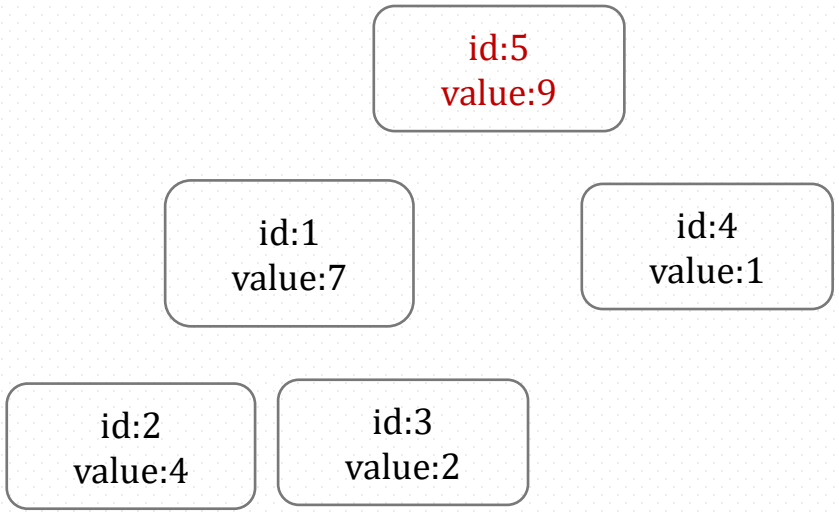
같으므로 유효한 값임을 확인 하고 처리해준 뒤 pop 한다.

실제 유효한 정보를 저장하는 배열

S

1	2	3	4	5
-1	4	2	1	9

Max Heap



Lazy Update Example

또 우선순위 값을 구하는 경우

heap[1]의 value 와 S[1]의 값이 같은지 비교한다.

다르므로 해당 heap[1]의 정보는 유효하지 않다고 판별하여 그냥 pop()해준다.

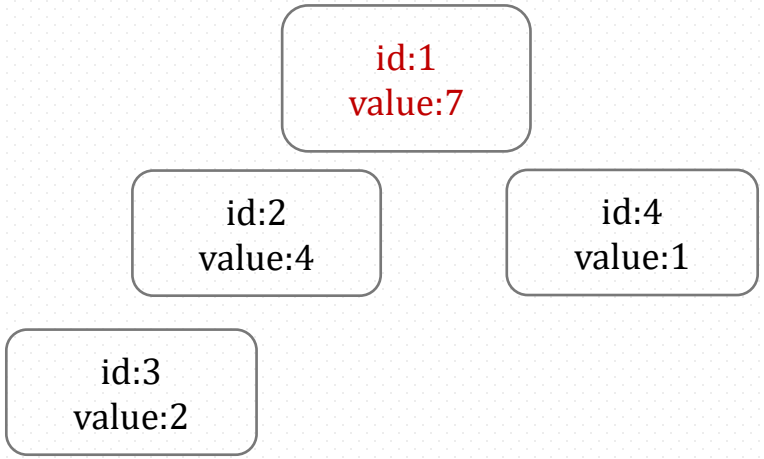
새로운 heap[1]에 대해서도 똑같이 확인하며 유효한 값이 나올 때까지 반복한다.

실제 유효한 정보를 저장하는 배열

S

1	2	3	4	5
-1	4	2	1	-1

Max Heap



Lazy Update Example

1번 지우기 전 상태에서
2번을 6으로 update 하는 경우

S에는 실시간으로 값을 바꿔준다.

heap에는 업데이트된 값으로 push 해 준다.

heap에 똑같은 index 2에 대한 정보가 중복되어 들어간다.

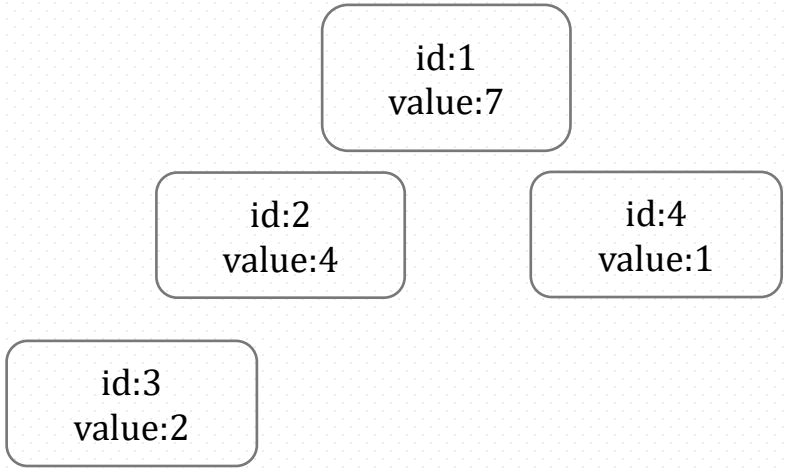
해당 정보들은 heap[1]에 왔을 때, erase와 마찬가지로 실제 정보와 비교하여 유효성 판별이 가능하다.

실제 유효한 정보를 저장하는 배열

S

1	2	3	4	5
7	4	2	1	-1

Max Heap



Lazy Update Example

2번을 6으로 update

S에는 실시간으로 값을 바꿔준다.

heap에는 업데이트된 값으로 push 해 준다.

heap에 똑같은 index 2에 대한 정보가 중복되어 들어간다.

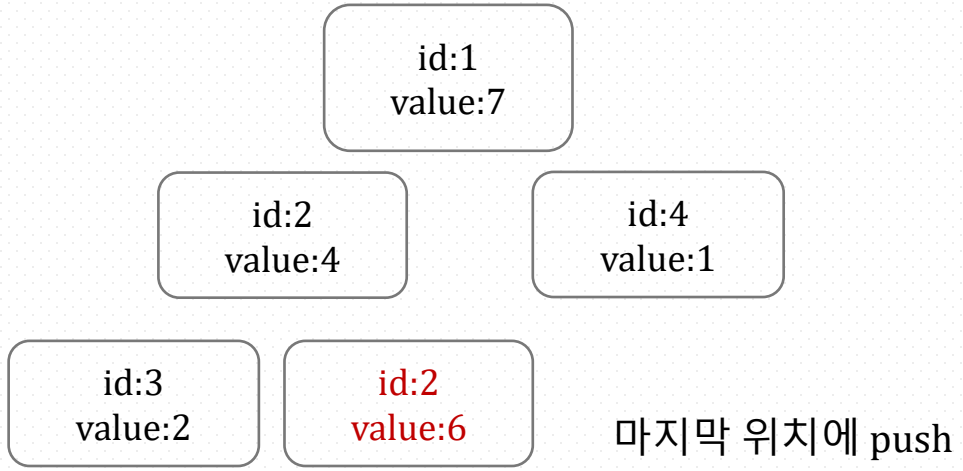
해당 정보들은 heap[1]에 왔을 때, erase와 마찬가지로 실제 정보와 비교하여 유효성 판별이 가능하다.

실제 유효한 정보를 저장하는 배열

S

1	2	3	4	5
7	6	2	1	-1

Max Heap



마지막 위치에 push

Lazy Update Example

2번을 6으로 update

S에는 실시간으로 값을 바꿔준다.

heap에는 업데이트된 값으로 push 해 준다.

heap에 똑같은 index 2에 대한 정보가 중복되어 들어간다.

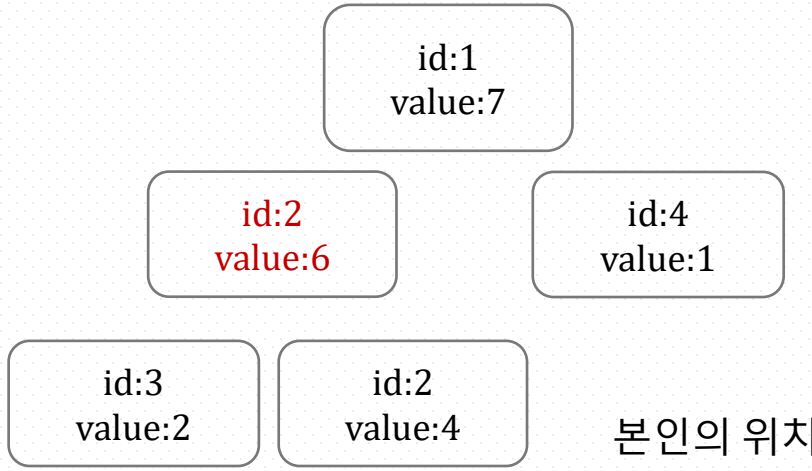
해당 정보들은 heap[1]에 왔을 때, erase와 마찬가지로 실제 정보와 비교하여 유효성 판별이 가능하다.

실제 유효한 정보를 저장하는 배열

S

1	2	3	4	5
7	6	2	1	-1

Max Heap



본인의 위치를 찾아감

Lazy Update 중복 처리

앞 example에서 우선순위 값 3개를 구해야 할 때,

1. (유효한 top 구한 뒤, 기록하고 pop) 과정을 3번 반복
2. 처리 후에 3개 다시 push

but, id:5 인 값의 value가 9->5->9 로 바뀐 경우라면,
{5, 9} 가 2번 들어가 있으므로 유효한 top 3개를 구하는 과정에서 {5, 9} 가 두 번 포함된다.

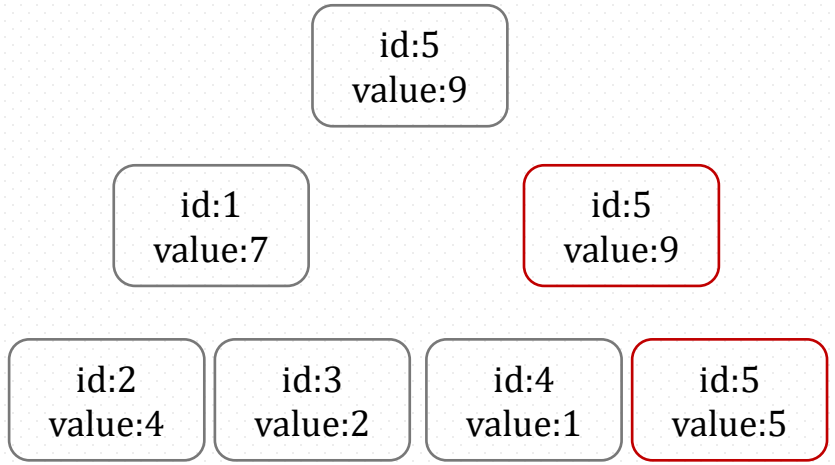
따라서, 유효한 top을 구하더라도 직전 뽑아낸 top이랑 중복되는 경우는 버려줘야 한다.

실제 유효한 정보를 저장하는 배열

S

1	2	3	4	5
7	4	2	1	9

Max Heap



Real-Time Update

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

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

Real-Time Update Example

실제 유효한 정보 저장

S	1	2	3	4	5
	7	4	2	1	9

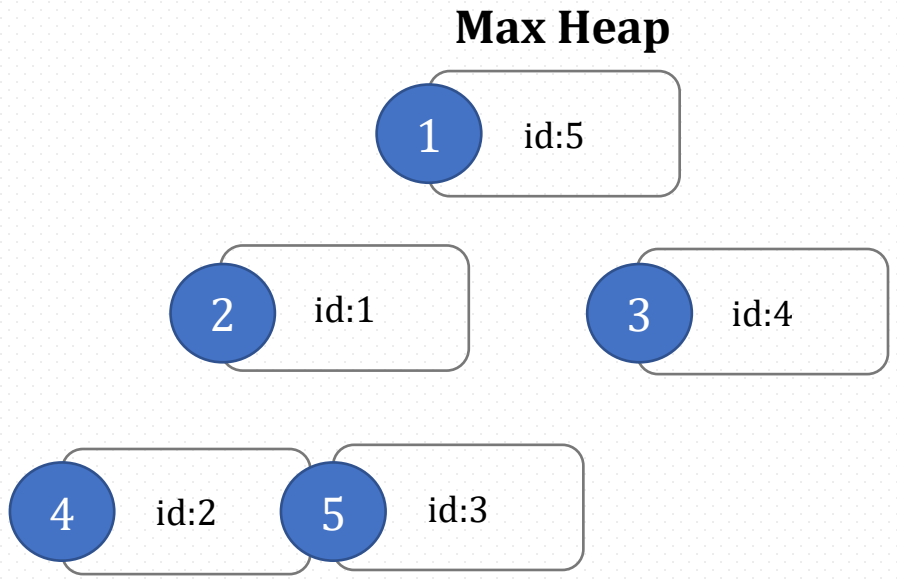
id의 heap index 번호

Pos	1	2	3	4	5
	2	4	5	3	1

- 실제 정보가 저장되는 S
- id별 heap 내의 index를 저장하는 Pos

- lazy update와는 다르게 heap을 구성하는 요소는 id값 하나면 충분하다.

- 물론, 별도의 S배열 없이 heap 자체에서만 data값까지 같이 기록해도 된다.



Real-Time Update Example

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, 힙의 성질을 만족하기 위한 자리를 찾아가야 한다.

실제 유효한 정보 저장

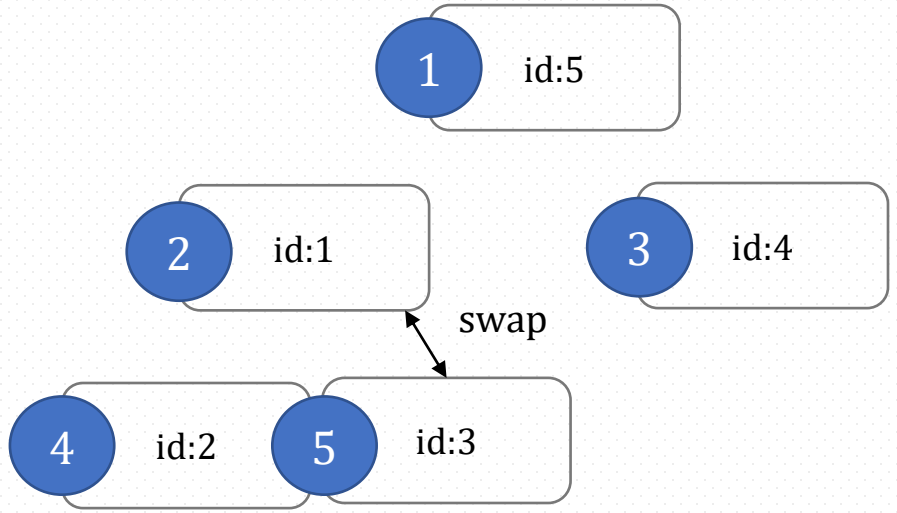
S	1	2	3	4	5
	~~7~~	4	2	1	9

id의 heap index 번호

Pos	1	2	3	4	5
	2	4	5	3	1

swap

Max Heap



Real-Time Update Example

실제 유효한 정보 저장

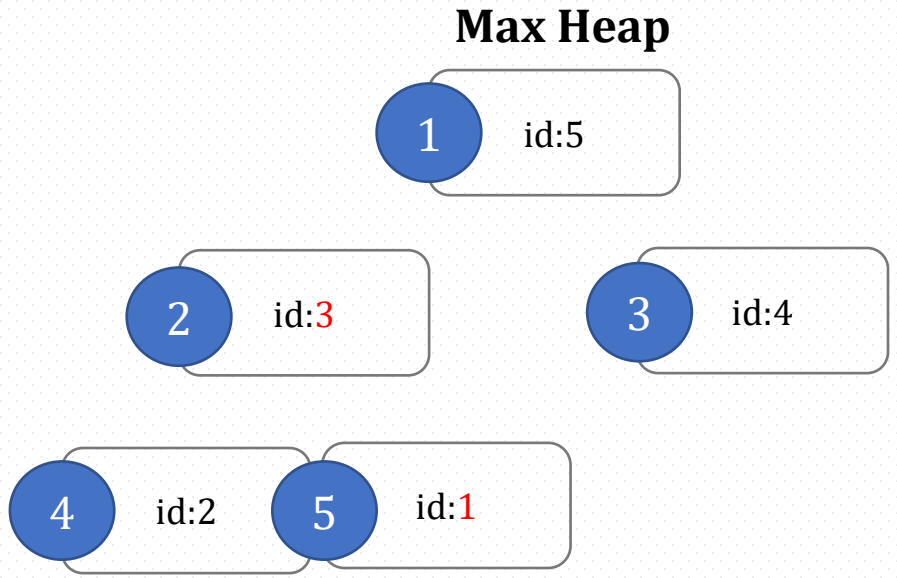
S	1	2	3	4	5
	~~7~~	4	2	1	9

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, 힙의 성질을 만족하기 위한 자리를 찾아가야 한다.



Real-Time Update Example

실제 유효한 정보 저장

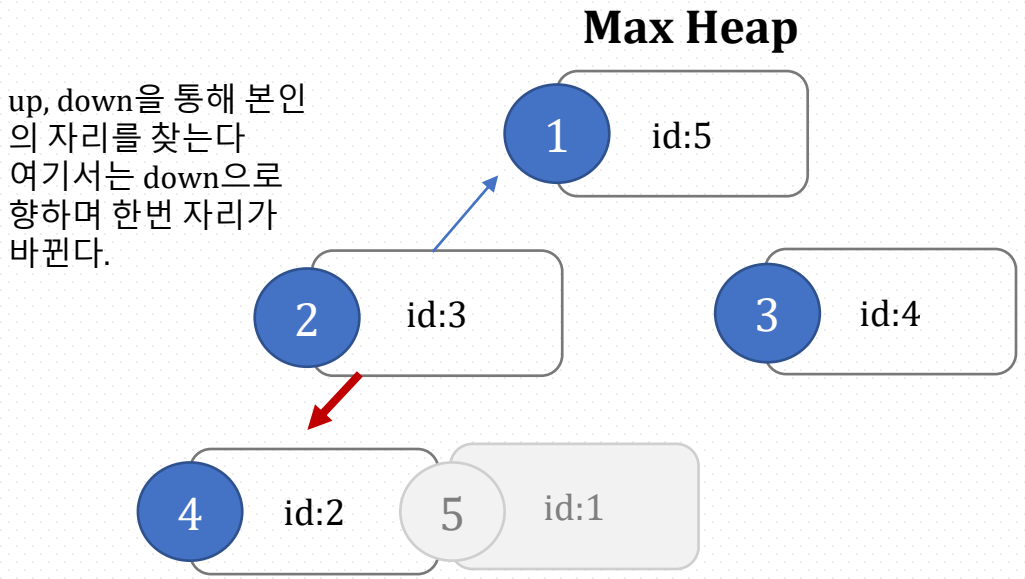
S	1	2	3	4	5
	~~7~~	4	2	1	9

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, 힙의 성질을 만족하기 위한 자리를 찾아가야 한다.



Real-Time Update Example

실제 유효한 정보 저장

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

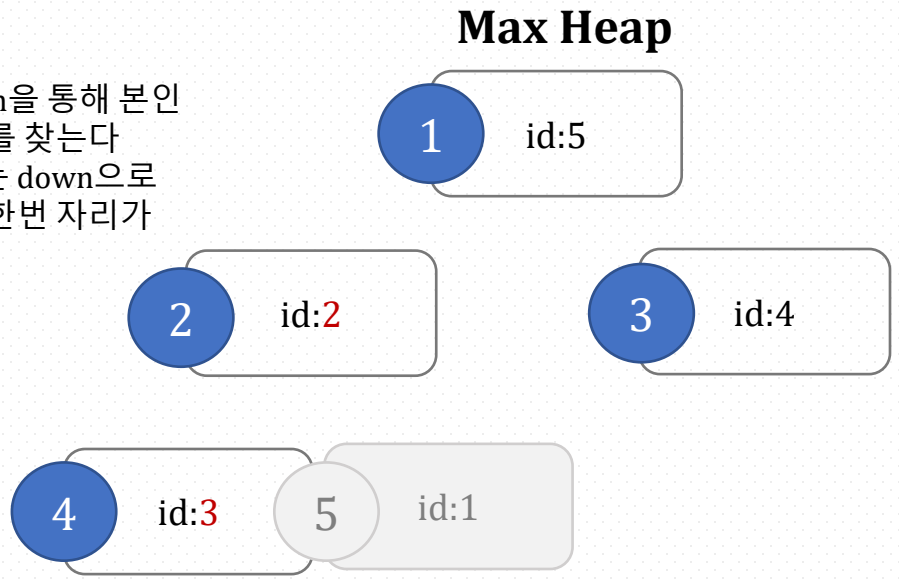
id의 heap index 번호

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, 힙의 성질을 만족하기 위한 자리를 찾아가야 한다.

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



Real-Time Update Example

S[2] 을 10로 업데이트 하는 경우

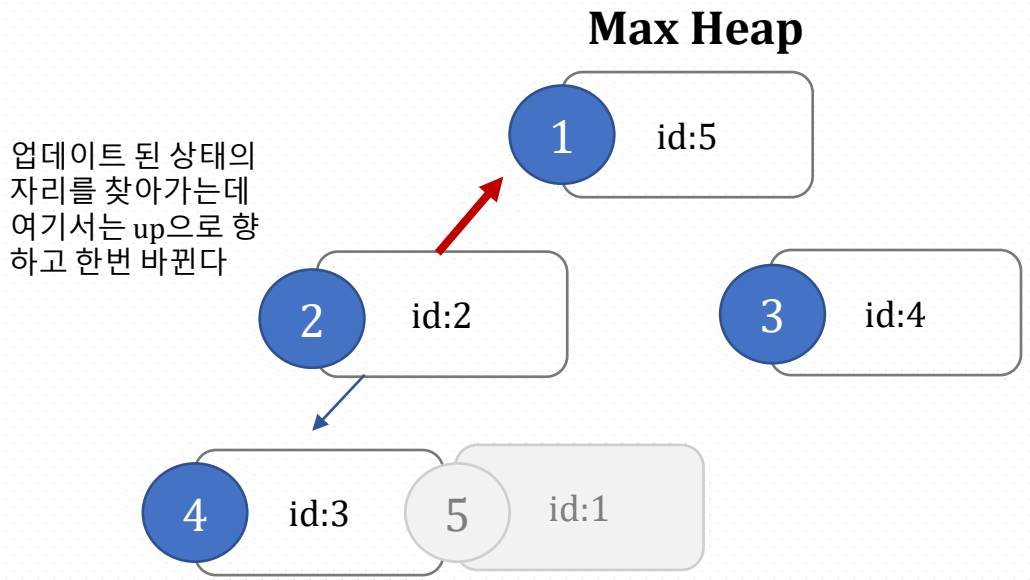
- pos[2]를 통해 heap 내의 index를 접근 한다.
- 값이 update 되었으므로 up, down을 통 해 본인 자리를 찾아간다.

실제 유효한 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



Real-Time Update Example

실제 유효한 정보 저장

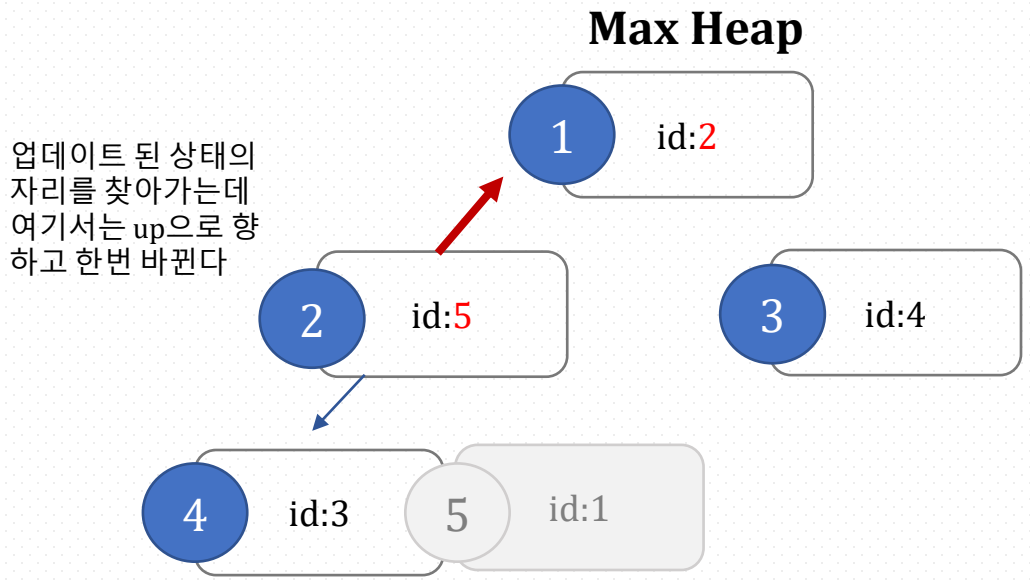
S	1	2	3	4	5
	7	10	2	1	9

id의 heap index 번호

Pos	1	2	3	4	5
	5	1	4	3	2

S[2] 을 10로 업데이트 하는 경우

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



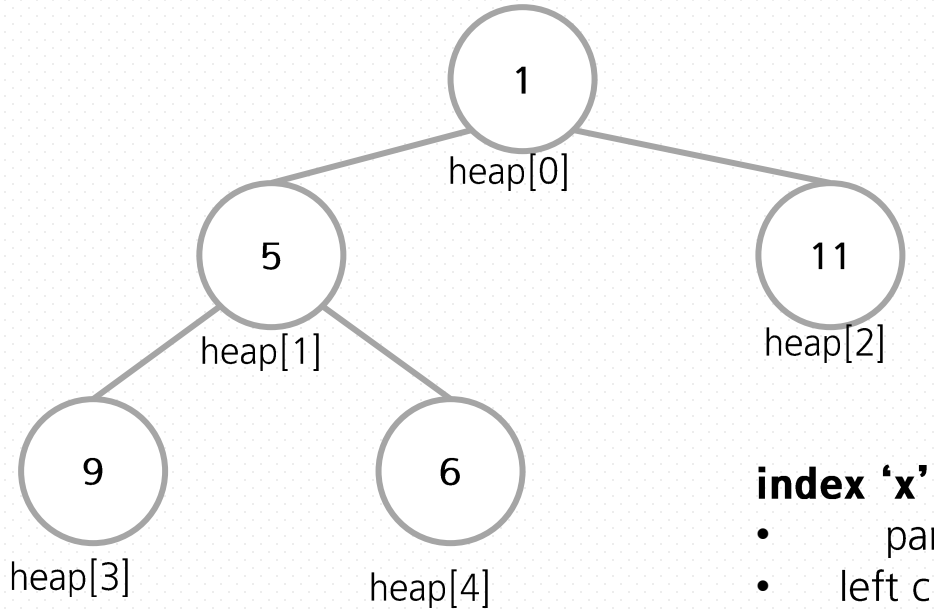
특징 비교

	Lazy Update	Real-time Update
heap 구성	유효성 검증을 위해 실제 data의 index와 data 필요	실제 data를 따로 기록한다면 해당 index만 필요 (or heap 자체로만 실제 data 기록 가능) heap의 index를 기록하는 배열 필요
heap size	heap size는 update 중복 push를 고려하여 실제 data 수 + update 되는 횟수	실제 data 수만큼 잡는다
update	실제 data 업데이트 후, 중복 push	heap의 필요한 index에 접근하여 up, down
erase	실제 data 삭제 표시 외 별도 작업 없음	heap의 필요한 index에 접근하여 마지막 위치와 변경 후 up, down
주의사항	heap에는 유효하지 않은 값들이 들어가 있으므로 heap의 개수가 실제 data수를 나타내지는 않음	

Reference Priority_Queue

reference code

1. min_heap
2. [0~heapSize-1] index 사용
3. 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
value	1	5	11	9	6	

Reference Priority_Queue

```c
#define MAX_SIZE 100

int heap[MAX_SIZE];
int heapSize = 0;

void heapInit(void)
{
    heapSize = 0;
}

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

    heap[heapSize] = value;

    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;
}
```

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

    *value = 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;
}
```

Reference Priority_Queue

```
#define MAX_SIZE 100

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

    void init(void)
    {
        heapSize = 0;
    }

    void push(int value)
    {
        ①
    }

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

MinPQ minpq;
```

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

    heap[heapSize] = value;

    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;
}
```

Reference Priority_Queue

```
#define MAX_SIZE 100

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

    void init(void)
    {
        heapSize = 0;
    }

    void push(int value)
    {
        ①
    }

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

MinPQ minpq;
```

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

    *value = 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;
}
```

Reference Priority_Queue

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

    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 value 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

Reference 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