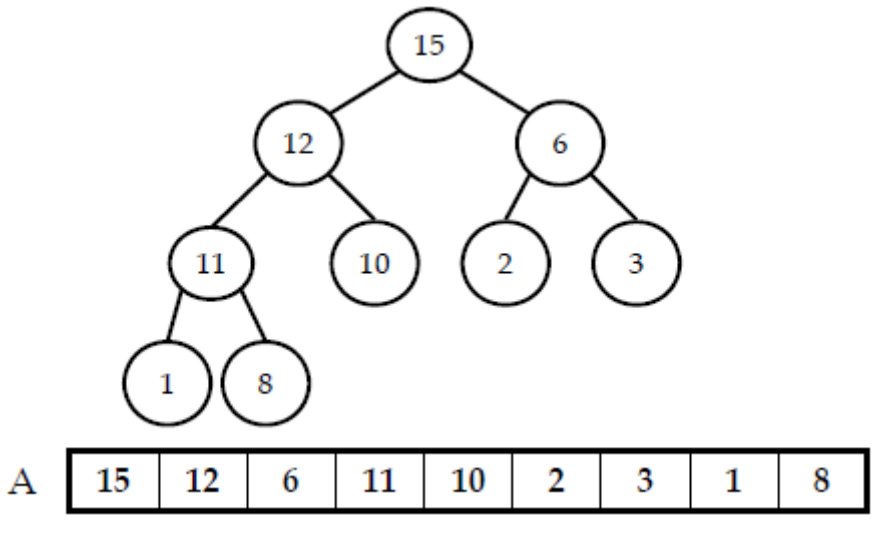
🌗

힙(heap)과 힙정렬(heap sort)

힙 자료구조란?

힙(Heap)

1. 힙(heap)은 1차원 배열 중에서 저장된 값이 힙 조건(모양과 값 조건)을 만족하는 리스트(배열)를 의미한다.
2. 힙의 모양 조건: 리스트를 이진트리로 해것했을 때,
    a. 마지막 레벨을 제외한 각 레벨엔 빠짐없이 노드가 존재한다.
    b. 마지막 레벨의 노드는 왼쪽부터 차례대로 빈틈없이 채워진다.
3. 힙의 값 조건 (heap property):
    a. 루트 노드를 제외한 모든 노드의 값은 부모 노드의 값보다 크지 않아야 한다. (또는 각 노드의 값은 자신의 자손 노드들의 값보다 같거나 커야 한다.)
4. 노드 A[k]의 왼쪽/오른쪽 자식노드의 인덱스와 부모노드의 인덱스를 O(1) 시간에 계산 가능
    a. 왼쪽 자식 노드는 **A[2k+1]**, 오른쪽 자식노드는 **A[2k+2]**
    b. 부모노드는 A[(k-1)//2]



힙 만들기(make_heap)

힙 class

```
class Heap:
  def __init__(self, L=[]): # 입력 리스트가 없으면 빈 리스트를 default 값으로 지정
    self.A = L
  def __str__(self):
    return str(self.A)
  def __len__(self):
    return len(self.A)
```

```
def heapify_down(self, k, n):
# n = 힙의 전체 노드수 [heap_sort를 위해 필요함]
# A[k]가 힙 성질을 위배한다면, 밑으로
# 내려가면서 힙성질을 만족하는 위치를 찾는다
while 2*k+1 < n:           # [?] 조건문이 어떤 뜻인가?
  L, R = 2*k + 1, 2*k + 2  # [?] L, R은 어떤 값?
  if L < n and self.A[L] > self.A[k]:
    m = L
```

```
  else:
    m = k
  if R < n and self.A[R] > self.A[m]:
    m = R # m = A[k], A[L], A[R] 중 최대값의 인덱스
  if m != k:  # A[k]가 최대값이 아니라면 힙 성질 위배
    self.A[k], self.A[m] = self.A[m], self.A[k]
    k = m
  else: break # 왜 break할까?

def make_heap(self):
  n = len(self.A)
  for k in range(n-1, -1, -1): # A[n-1] → ... → A[0]
    self.heapify_down(k, n)
```

1. 조건문이 어떤 뜻인가?
    a. (2*k+1) < n: n이 자신 (2*k+1) 자식노드. 자신보다 큰 자식노드가 있으면 밑으로 내려감
2. L, R 각각 왼쪽, 오른쪽 자식노드
3. m = index(A[k], A[L], A[R]) 이 세개의 값 중 가장 큰 값
4. 왜 break할까?
    a. 3개 중의 맥시멈이 자기 자신, 자식 노드가 전부 자기 자신보다 작기 때문에 더 밑으로 내려갈 필요가 없어서 break
5. for k in range(n-1, -1, -1)
    a. 오른쪽 밑에서 왼쪽 위로, 제일 작은 값에서 제일 큰 값으로 거슬러 올라가기 때문

:: **n개 노드: 힙의 높이 h**

1레벨 노드 개수 : 1

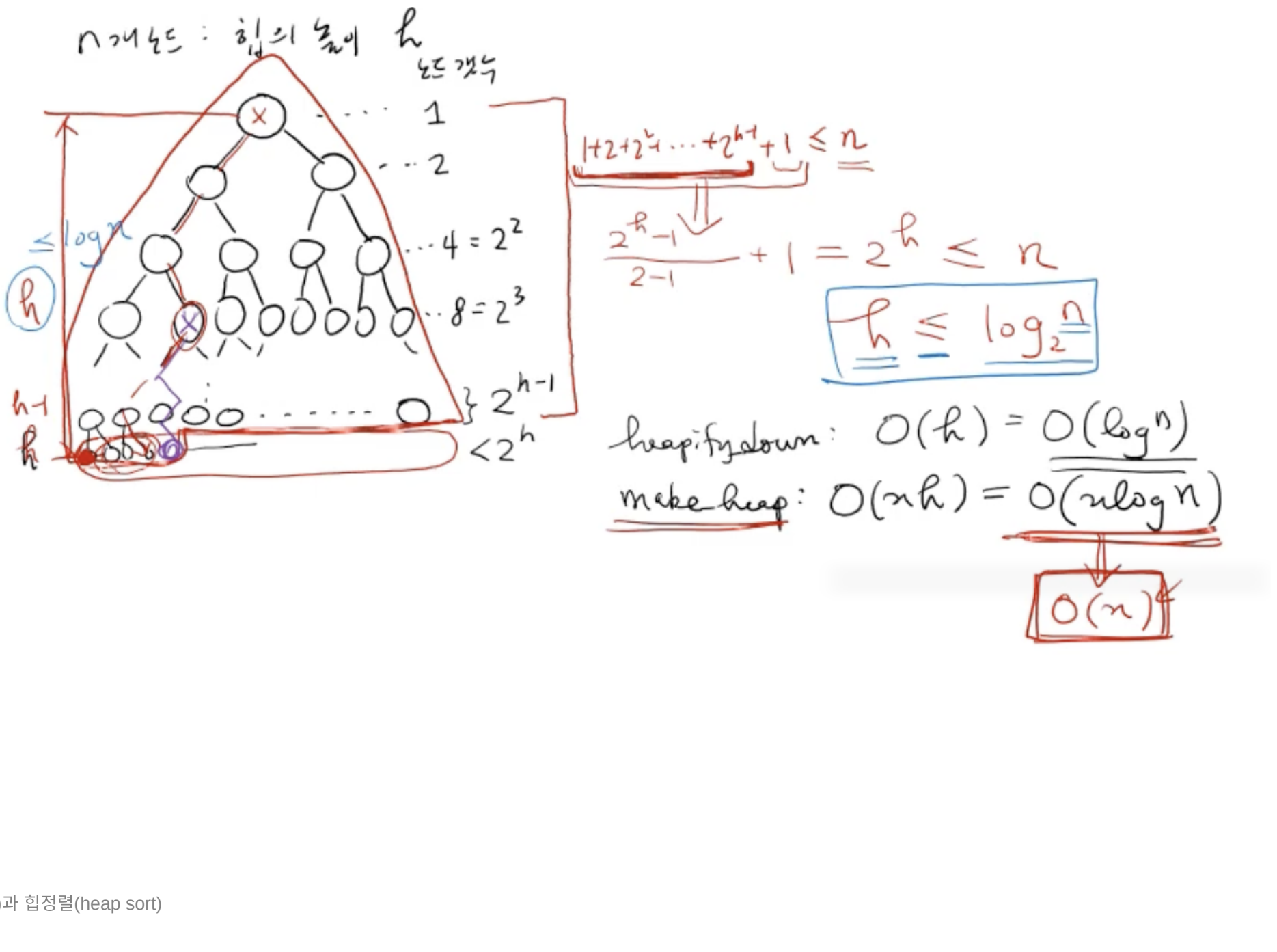
2레벨 노드 개수 : 2

3레벨 노드 개수 : 4 (2의 제곱)

4레벨 노드 개수 : 8 (2의 세제곱)

h-1 레벨 노드 개수 : 2^(h-1)

h 레벨 노드 개수 : 2^h



힙 정렬(heap_sort)

1. 입력으로 주어진 리스트 A를 make_heap을 호출하여 힙으로 만든다.
2. 루트 노드의 값(현재의 최대값(A[0]) 을 현재 리스트의 가장 마지막 값과 바꾼다.
3. 새로 루트노드에 저장된 값은 힙 성질을 만족하지 않을 수 있기 때문에, 자손도드로 내려가면서 힙의 위치를 찾아가야 한다. (heapify_douwn 함수 호출)
4. 위의 과정 2와 3을 (n-1)번 반복하면 (n-1)개의 수가 정렬되어, 결국 모든n개의 수가 정렬된다.

```
def heap_sort(self):
    n = len(self.A)
    for k in range(len(self.A)-1, -1, -1):
      self.A[0],self.A[k] = self.A[k],self.A[0]
      n = n - 1 # A[n-1]은 정렬되었으므로
      self.heapify_down(0, n)
```

힙 삽입(insert) 연산

1. 힙 `A`의 가장 오른쪽에 새로운 값 `x`를 저장하고, 이 값을 힙 성질이 만족하도록 위치를 재조정해야 한다
   - 이 경우엔 `x`가 힙의 리프에 위치하므로, 루트 노드 방향으로 올라가면서 자신의 위치를 조정하면 된다
   - `heapify_down`과 반대방향으로 이동하면서 위치를 조정하므로 `heapify_up` 이라 부른다

```
def heapify_up(self, k): # 올라가면서 A[k]를 재배치
  while k > 0 and self.A[(k-1)//2] < self.A[k] :
    self.A[k], self.A[(k-1)//2] = self.A[(k-1)//2], self.A[k]
    k = (k-1)//2

def insert(self, key):
  self.A.append(key)
  self.heapify_up(len(self.A)-1)
```

예시 → A.heapify_up(9)

여기서 9는 index 값

힙 삭제(delete_max) 연산

1. 힙의 루트노드에 있는 최대값을 삭제하여 값을 리턴하고, 남은 힙의 힙 성질을 그대로 유지되도록 하는 연산
   - 만약 max-heap이 아닌 min-heap(자손노드의 값보다 크지 않은 값이 저장된다는 성질을 만족하는 힙)인 경우엔 루트노드에 있는 최소값을 삭제하는 연산이 됨.

```
def delete_max(self):
  if len(self.A) == 0: return None
  key = self.A[0]
  self.A[0], self.A[len(self.A)-1] = self.A[len(self.A)-1], self.A[0]
  self.A.pop()  # 실제로 리스트에서 delete!
  heapify_down(0, len(self.A)) # len(self.A) = n-1
  return key
```

- 연산 및 수행시간 정리 (n개의 값을 저장한 리스트와 힙에 대해)
  1. heapify_up, heapify_down: **O(log n)**
  2. make_heap = n times * heapify_down = **O(n log n) → O(n)**
  3. insert = 1 * heapify_down: **O(log n)**
  4. delete_max = 1 * heapify_down: **O(log n)**
  5. heap sort = make_heap + n * heapify up = **O(n log n)**