
파이썬
자료구조

CHAPTER

힙 - 이진트리 응용

힙

- 힙(Heap)이란?
- 힙을 저장하는 효과적인 자료구조는 배열이다.
- 우선순위 큐의 가장 좋은 구현 방법은 힙이다.
  - 우선순위 큐:
    원소들의 모임
    연산들:
      원소의 삽입
      최대키의 원소의 삭제

Priority Queue

우선순위가 가장 큰 원소를 삭제함. ⇒ heap 구조임.

힙(Heap)이란?

- 힙(Heap)이란?
  - 완전이진트리 기반의 자료 구조
  - 가장 큰(또는 작은) 값을 빠르게 찾아내도록 만들어진 자료 구조
  - 최대 힙, 최소 힙

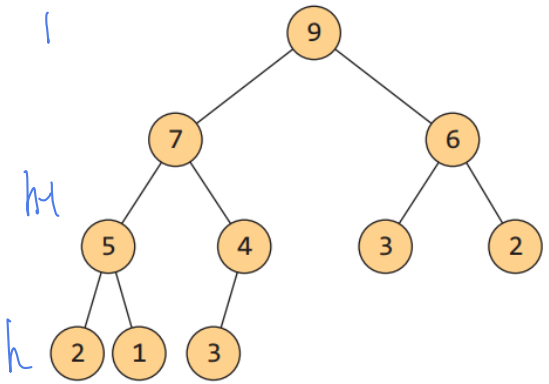
정의 8.2 최대 힙, 최소 힙의 정의

- 최대 힙(max heap): 부모 노드의 키 값이 자식 노드의 키 값보다 크거나 같은 완전이진트리
  (key(부모노드) $\geq$ key(자식노드))
- 최소 힙(min heap): 부모 노드의 키 값이 자식 노드의 키 값보다 작거나 같은 완전이진트리
  (key(부모노드) $\leq$ key(자식노드))

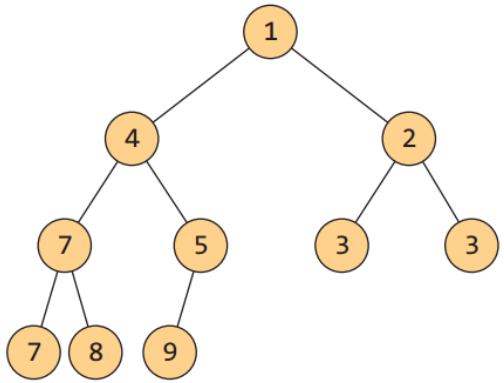
n 노드수

최대힙의 연산 : ① 원소삽입 O(log n)
② 최대키의 원소 찾기 O(1)
③ 최대키의 원소 삭제 O(log n)

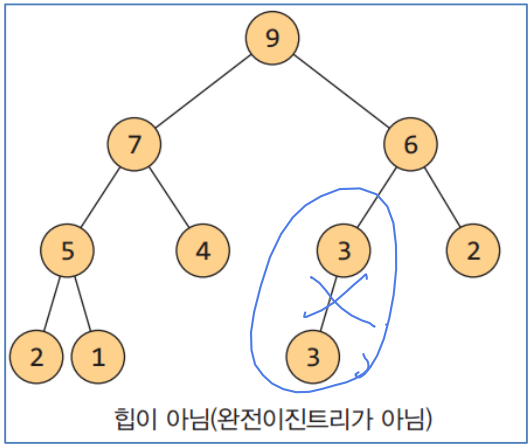
힙의 예



최대 힙(Max Heap)



최소 힙(Min Heap)



힙이 아님(완전이진트리가 아님)

root에 최대값이 들어가있음

최대키 원소 : O(1)

root에 최소값이 들어가 있음.

최소키 원소 : O(1)

힙의 연산: 삽입 연산

[1] 힙의 마지막 노드(즉, 리스트의 마지막 항목)의 바로 다음 비어있는 원소에 새로운 항목을 저장

[2] 루트 방향으로 올라가면서 부모의 키와 비교하여 힙 속성이 만족될 때까지 노드를 교환

- [2]의 과정은 이파리로부터 위로 올라가며 수행되므로 upheap이라 부름

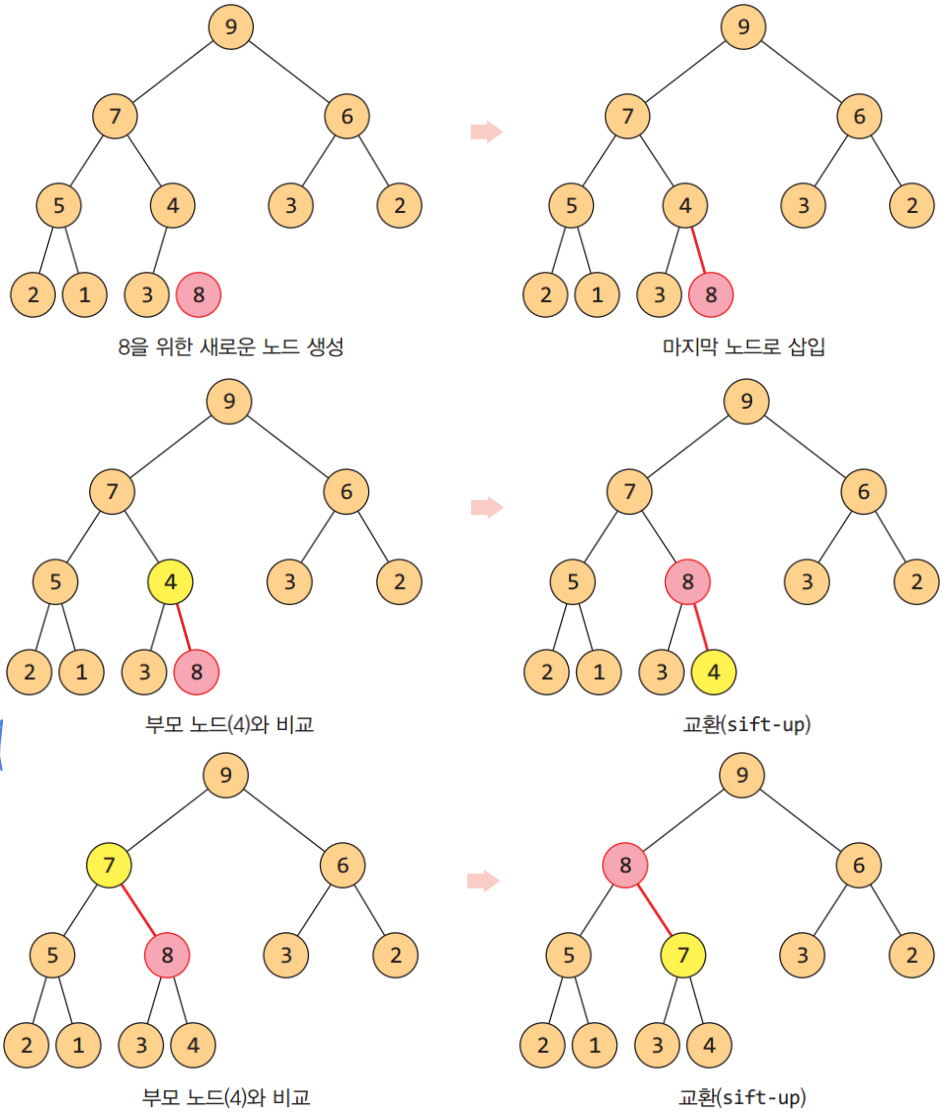
힙의 연산: 삽입 연산

- InsertHeap
  - 시간 복잡도: 높이에 비례한다
    $O(logn)$

힙 생성시간

높이 $\lfloor \log_2 \rfloor + 1$ 루트를 0으로 했을 때

$\lfloor \log_2 \rfloor$ 루트를 1로 했을 때



힙의 연산: 최소키 원소 삭제 연산

- 루트의 키를 삭제

[1] 힙의 가장 마지막 노드, 즉, 리스트의 가장 마지막 항목을 루트로 옮기고,

[2] 힙 크기를 1 감소시킨다.

[3] 루트로부터 자식들 중에서 작은 값을 가진 자식(승자)과 키를 비교하여 힙속성이 만족될 때까지 키를 교환하며 이파리 방향으로 진행

- [3]의 과정은 루트로부터 아래로 내려가며 진행되므로 downheap이라 부름

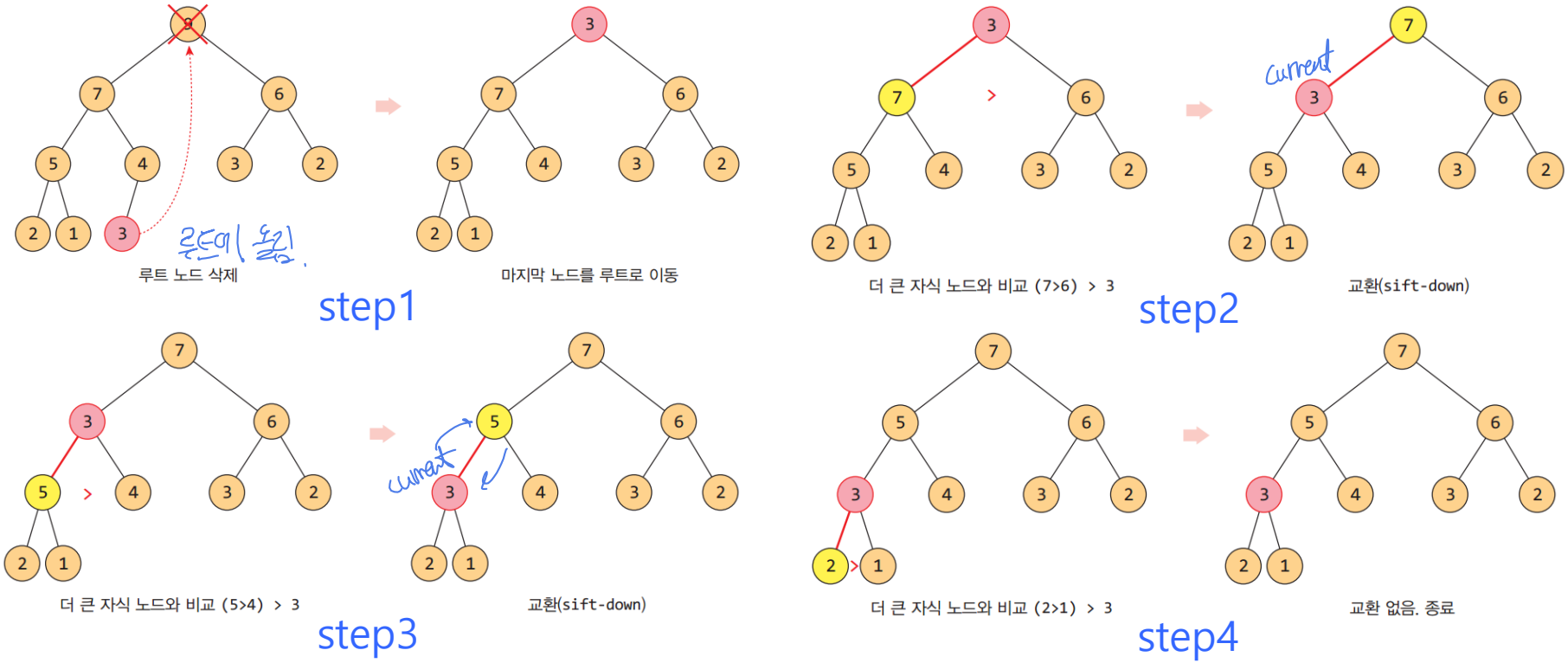
힙의 연산: 삭제 연산

(최대 키 원소)

- deleleMax
  - 시간 복잡도:

    높이에 비례한다 $O(logn)$

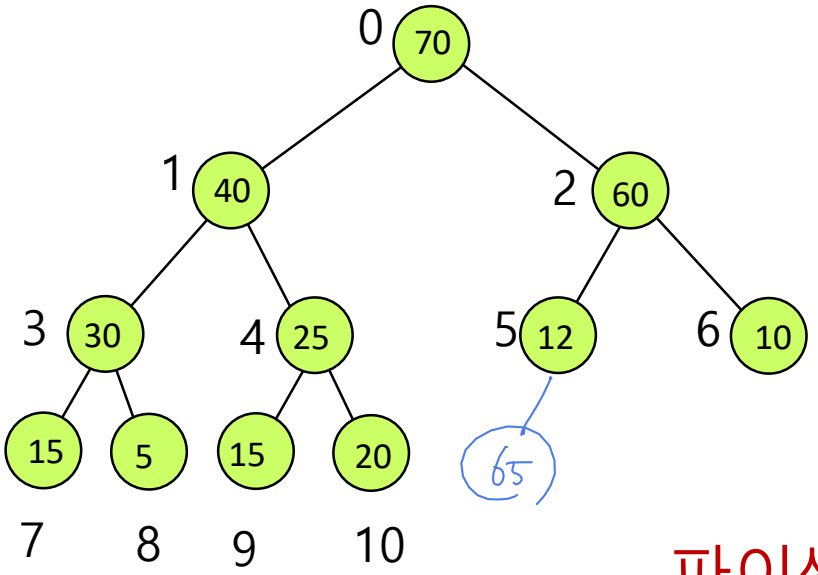
    완전이진트리 유리



힙의 구현: 배열 구조

- 힙은 보통 배열을 이용하여 구현
  - 완전이진트리 ➔ 각 노드에 번호를 붙임 ➔ 배열의 인덱스

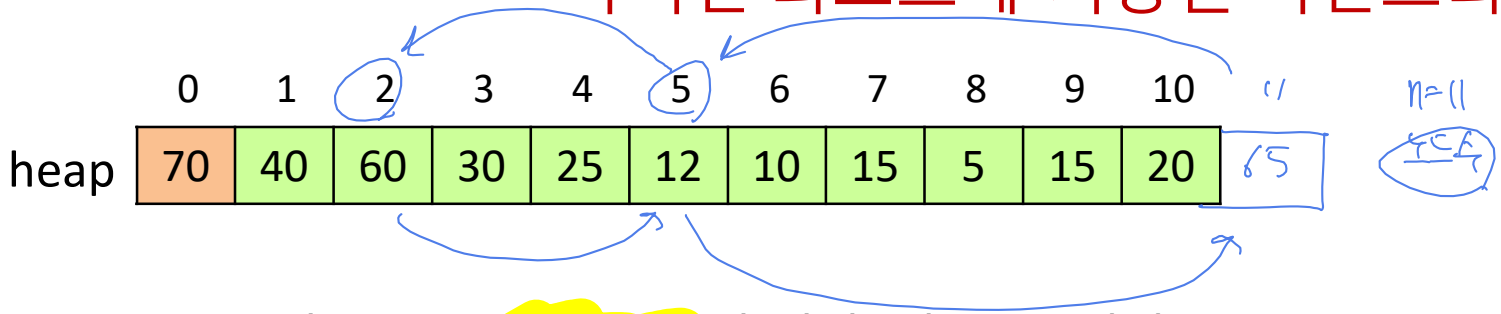
이진트리의 표현: 배열표현법(2)



- 노드에 번호를 0부터 n-1까지 붙힌다:
- root는 0번

  Level by level, 같은 level은 왼쪽부터 오른쪽으로

파이썬 리스트에 저장된 이진트리



heap[i]의 부모는 a[(i-1)//2]에 있다. 단, i > 0이다.
heap[i]의 왼쪽자식은 a[2i+1]에 있다. 단, 2i+1 < n이다.
heap[i]의 오른쪽자식은 a[2i+2]에 있다. 단, 2i + 2 < n이다.

최대 힙의 구현

```
class MaxHeap:
    def __init__(self):
        self.heap = []

    def print(self):
        print(self.heap)
```

리스트로 구현

최대 힙의 원소 삽입

```
def insert(self, value):
    self.heap.append(value)
    i = len(self.heap) - 1
    while (i != 0 and value > self.heap[(i-1)//2]):
        self.heap[i] = self.heap[(i-1)//2]
        i = (i-1)//2
    self.heap[i] = value
```

맨 마지막에 append

0번 ~ n-1번. 마지막 위치는 n-1이므로

부모노드와 값을 변경한 횟수

while loop $O(\log_2 n)$

최대 힙에서 원소(최대 원소) 삭제

```
def delete(self):
    n = len(self.heap)
    if n == 0:
        return None
    current = 0
    maxValue = self.heap[0] # 최대값 저장
    value = self.heap[n-1] # 마지막 원소를 value에 저장
    self.heap.pop()  # 마지막 원소 삭제
    n = n - 1        # 원소 하나가 삭제되어 n을 1 줄임
    while (2*current+1 < n):   # current가 leaf가 아니면
# 두 자식 노드 중 큰 값의 노드를 largerChild
        largerChild = 2*current + 1
        if (largerChild + 1) < n and self.heap[largerChild + 1] > self.heap[largerChild]:
            largerChild += 1

        if value < self.heap[largerChild]: # largerChild의 값이 크면
            self.heap[current] = self.heap[largerChild]
            current = largerChild          # current를 largerChild로 내림
        else:
            break

    self.heap[current] = value
    return maxValue
```

최대 힙에서 원소(최대 원소) 삭제

```
def delete(self):  // down heap
    n = len(self.heap)
    if n == 0:
        return None
    current = 0
    maxValue = self.heap[0] # 최대값 저장
    value = self.heap[n-1]  # 마지막 원소를 value에 저장
    self.heap.pop()  # 마지막 원소 삭제
    n = n – 1        # 원소 하나가 삭제되어 n을 1 줄임
    while (2*current+1 < n):   # current가 leaf가 아니면
        largerChild = 2*current + 1
        rightChild = leftChild + 1
        # 두 자식 노드 중 큰 값의 노드를 largerChild
        if rightChild < n and self.heap[rightChild] > self.heap[leftChild]:
            largerChild = rightChild
        else:
            largerChild = leftChild

        if value < self.heap[largerChild]: # largerChild의 값이 크면
            self.heap[current] = self.heap[largerChild]
            current = largerChild          # current를 largerChild로 내림
        else:
            break
    self.heap[current] = value
    return maxValue
```

|파이썬 heapq|

파이썬은 우선순위큐를 위한 heapq를 라이브러리로 제공

Import heapq

heapq에 선언된 메소드

- heapq.heappush(*heap, item*) # insert() 메소드와 동일
- heapq.heappop(*heap*) # delete_min() 메소드와 동일
- heapq.heappushpop(*heap, item*) # item 삽입 후 delete_min() 수행
- heapq.heapify(*h*) # 리스트 h를 힙으로 만듬