🕶️

11주차 : Heap

> 힙(heap)은 **최댓값 및 최솟값**을 찾아내는 연산을 빠르게 하기 위해 고안된 **완전이진트리**(complete binary tree)를 기본으로 한 자료구조(tree-based structure)

ref. https://ko.wikipedia.org/wiki/힙_(자료_구조)

- **A가 B의 부모노드(parent node) 이면, A의 키(key)값과 B의 키값 사이에는 대소관계가 성립한다.**

완전 이진 트리

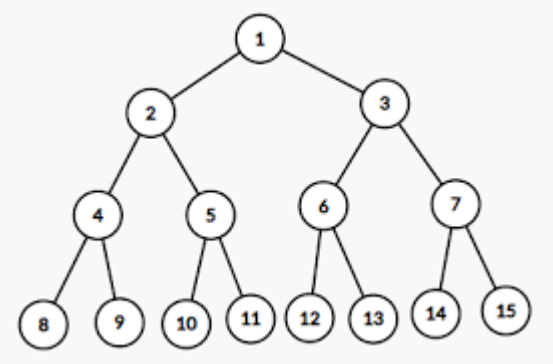
완전 + 이진 + 트리

- 트리
- 이진

  부모 노드가 최대 두 개의 자식 노드를 가지는 트리
- 완전

  잎 노드를 제외한 다른 노드들은 최대 갯수의 자식을 가지고 있고, 잎 노드들은 한쪽부터 채워지는 트리

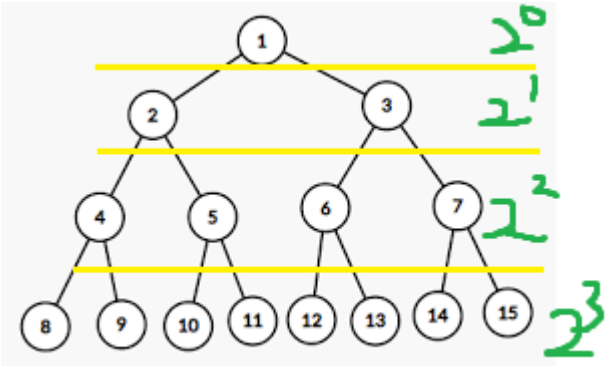
예시) 아래 그림에서 노드 번호 순서대로 노드를 채운다.

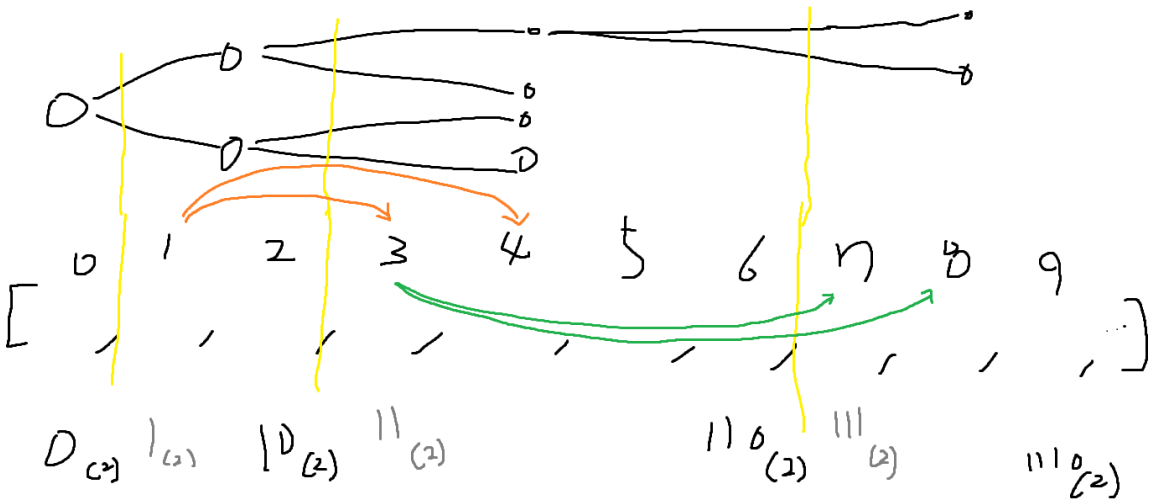


따라서, 최솟값 및 최댓값은 알지만, 번호 순서대로 정렬되지는 않는다.

완전 이진 트리의 배열 표현

일반적인 배열로 사용 가능하다.





$$Child_1 = 2 * Parent + 1$$

$$Child_2 = 2 * Parent + 2$$

따라서, 일반적인 배열을 사용할 수 있다.

힙의 작동

힙의 조건인

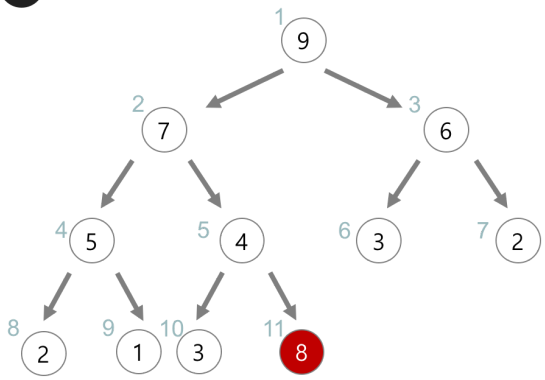
- **A가 B의 부모노드(parent node) 이면, A의 키(key)값과 B의 키값 사이에는 대소관계가 성립한다.**

만 만족시키면서 일반적인 배열 형태의 완전 이진 트리를 사용하면 된다.

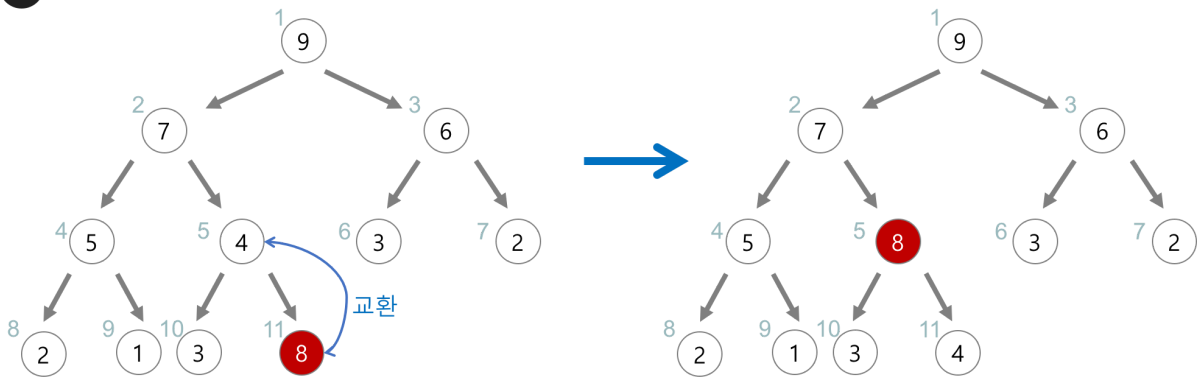
heahppush

1. 배열의 마지막에 원소를 추가한다.
2. 뿌리 노드에 도착하거나, 대소 관계가 옳게 될 때까지 부모 노드와 스왑한다.

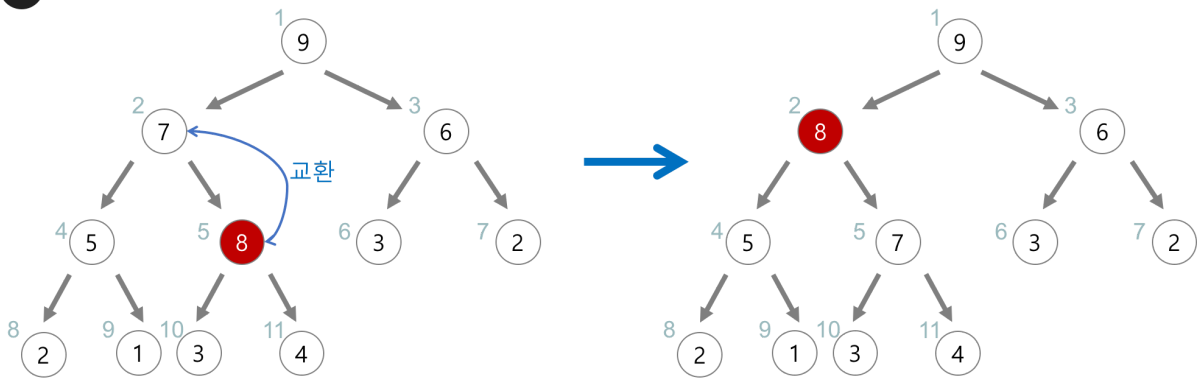
1 인덱스순으로 가장 마지막 위치에 이어서 새로운 요소 8을 삽입



2 부모 노드 4 < 삽입 노드 8 이므로 서로 교환



3 부모 노드 7 < 삽입 노드 8 이므로 서로 교환



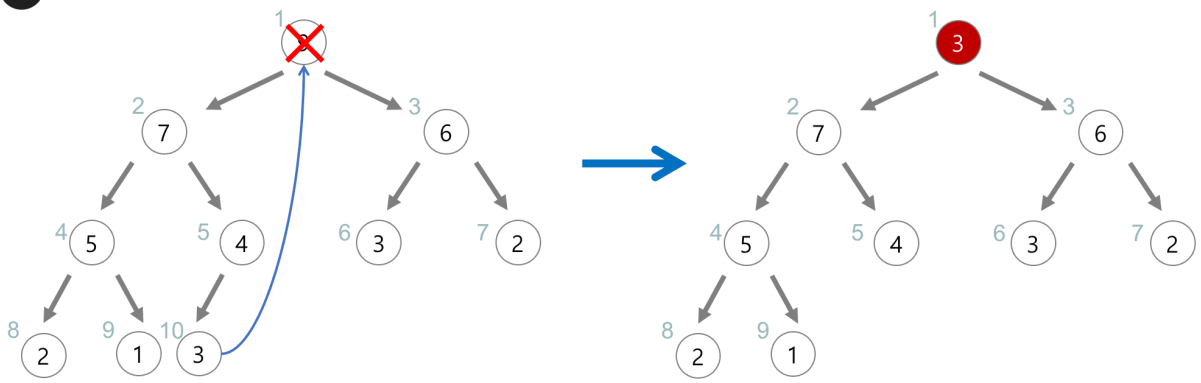
4 부모 노드 9 > 삽입 노드 8 이므로 더 이상 교환하지 않는다.

ref. https://gmlwjd9405.github.io/2018/05/10/data-structure-heap.html

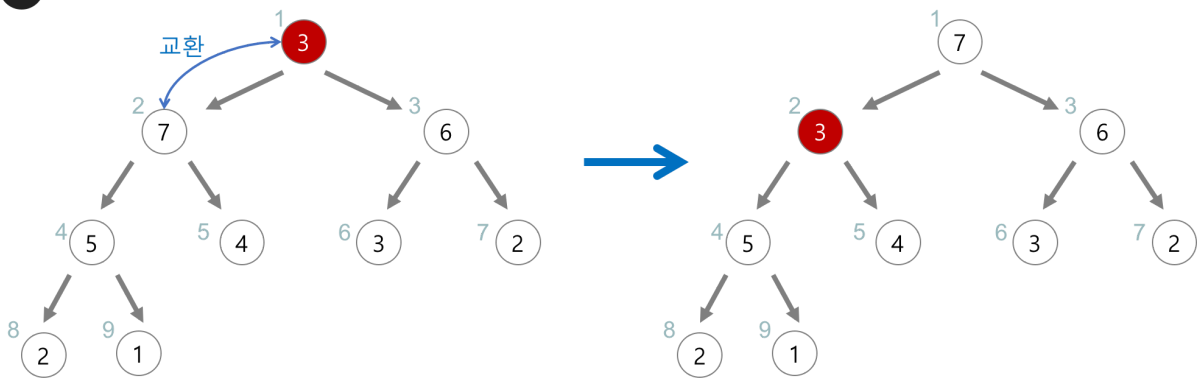
heappop

1. 배열의 마지막 요소와 루트를 스왑한다.
2. 배열의 마지막 요소를 pop 후 리턴한다.
3. 루트 요소가 잎 노드에 도착하거나, 대소 관계를 만족할 때까지 자식 노드와 스왑한다.

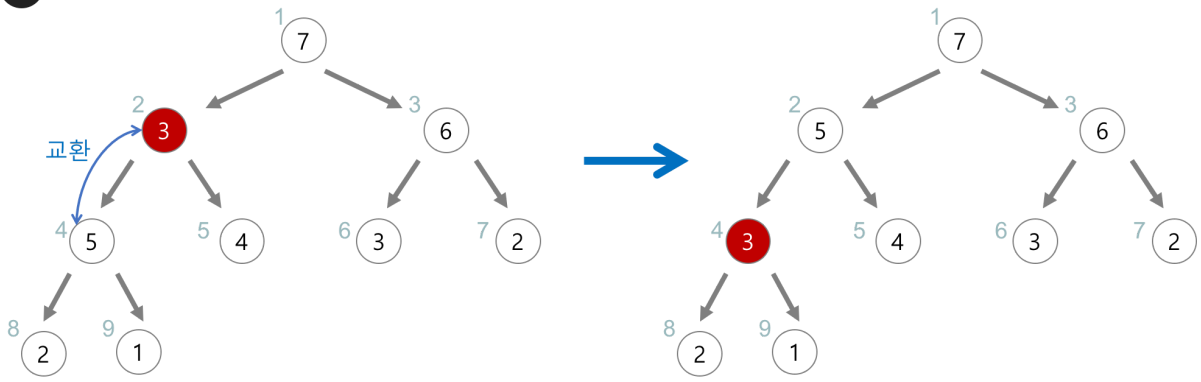
1 최댓값인 루트 노드 9를 삭제. (빈자리에는 최대 힙의 마지막 노드를 가져온다.)



2 삽입 노드와 자식 노드를 비교. 자식 노드 중 더 큰 값과 교환. (자식 노드 7 > 삽입 노드 3 이므로 서로 교환)



3 삽입 노드와 더 큰 값의 자식 노드를 비교. 자식 노드 5 > 삽입 노드 3 이므로 서로 교환



4 자식 노드 1, 2 < 삽입 노드 3 이므로 더 이상 교환하지 않는다.

시간복잡도

1. 하나의 요소를 넣거나 뺄 때
    - O(log(n))

2. 힙 정렬

- n개의 요소를 넣고(힙 구성) 빼는 것으로 볼 수 있음
- 힙 구성시간 + n개의 데이터 삭제 및 재구성 시간

$$
\begin{aligned}
&= (\log n + \log(n-1) + \ldots + log2) \\
&= (\log n + \log(n-1) + \ldots + \log 2) + (\log n + \log(n-1) + \ldots + \log 2) \\
&= (n \log n)
\end{aligned}
$$

- 다른 정렬과 비교

Name	Best	Avg	Worst	Run-time(정수 60,000개) 단위: sec
삽입정렬	n	n^2	n^2	7.438
선택정렬	n^2	n^2	n^2	10.842
버블정렬	n^2	n^2	n^2	22.894
셸 정렬	n	$n^{1.5}$	n^2	0.056
퀵 정렬	$n\log_2 n$	$n\log_2 n$	n^2	0.014
힙 정렬	$n\log_2 n$	$n\log_2 n$	$n\log_2 n$	0.034
병합정렬	$n\log_2 n$	$n\log_2 n$	$n\log_2 n$	0.026

힙 정렬과 관련된 시간 측정 테스트(n=1,000,000)

▼ 코드

```
import heapq
from time import time
import random


def bubble_sort(lst):
    start_time = time()
    for endpoint in range(len(lst)-1, 0, -1):
        for bubble in range(endpoint):
            if lst[bubble] > lst[bubble+1]:
                lst[bubble], lst[bubble+1] = lst[bubble+1], lst[bubble]
    end_time = time()
    print(f'bubble sort: {end_time-start_time}')
    return lst


def heappush_sort(lst):
    start_time = time()
```

```
    heap = []
    for component in lst:
        heapq.heappush(heap, component)
    middle_time = time()
    rtn = list()
    while heap:
        rtn.append(heapq.heappop(heap))
    end_time = time()
    print(f'heappush sort: {end_time-start_time}')
    print(f'heappush: {middle_time-start_time}')
    return rtn


def heapify_sort(lst):
    start_time = time()
    heapq.heapify(lst)
    middle_time = time()
    rtn = list()
    while lst:
        rtn.append(heapq.heappop(lst))
    end_time = time()
    print(f'heapify sort: {end_time - start_time}')
    print(f'heapify: {middle_time - start_time}')
    return rtn


RANGE = 100000000
number = 1000000
sample_start_time = time()
sample = random.sample(range(RANGE), number)
sample_end_time = time()
print(f'sample: {sample_end_time - sample_start_time}')
# bubble_sort(sample[:])
heappush_sort(sample[:])
heapify_sort(sample[:])
multiply_start_time = time()
m = [10000000] * 1000000
multiply_end_time = time()
print(f'multiply: {multiply_end_time - multiply_start_time}')
for_start_time = time()
f = [10000000 for _ in range(1000000)]
for_end_time = time()
print(f'for: {for_end_time - for_start_time}')
append_start_time = time()
a = []
for _ in range(1000000):
    a.append(10000000)
append_end_time = time()
print(f'append: {append_end_time - append_start_time}')
pop_start_time = time()
for _ in range(1000000):
    a.pop()
pop_end_time = time()
print(f'pop: {pop_end_time - pop_start_time}')
copy_start_time = time()
b = a[:]
copy_end_time = time()
print(f'copy: {copy_end_time - copy_start_time}')
```

- 결과

```
heappush sort: 1.3549683094024658
heappush: 0.10805344581604004
heapify sort: 1.259857416152954
heapify: 0.03408694267272949
multiply: 0.0020003318786621094
for: 0.03356814384460449
append: 0.07365608215332031
pop: 0.056151390075683594
copy: 0.0
```

- list 조작

  multiply ≥ copy >>(약 15배)>> for >(약 2배)> pop ≥ append

- heap 조작

  heapify >(약 3배)> heappush >>(약 10배)>> heappop

힙의 장단점

장점

- 전체를 정렬할 때가 아닌 몇 개의 최댓값(최대 힙) 혹은 최솟값(최소 힙)을 컨트롤 할 때 유용함
- 힙에 요소를 추가할 때는 배열의 마지막에 추가하게 되므로 append 시 시간 소모가 적음(pop의 경우도 마찬가지)

단점

- 완전히 정렬된 상태가 아니므로 전체를 정렬하기 위해서는 추가적인 시간이 필요함
- 값을 뽑아올 때 항상 log(n)의 시간이 고정으로 필요함

오늘의 문제

가운데를 말해요(Gold2): https://www.acmicpc.net/problem/1655

▼ 힌트

고려해야 할 요소

- 계속해서 요소를 추가하는 문제이므로, 여러 번 정렬하는 것은 효율이 좋지 않음
- 또한 정렬된 상태에서, 배열의 중간에 요소를 추가하는 방식은 효율이 좋지 않음
- 따라서 중간보다 작은 값들과 큰 값들을 어느 정도 정렬된 상태로 유지하되, 요소 추가 시 시간 소모가 적은 방식을 사용해야 함

결론

- 따라서 중간 값을 기준으로 큰 값들을 보관하는 최소 힙과 작은 값들을 보관하는 최대 힙, 총 두 개의 힙을 사용하며 두 힙의 길이를 균일하게 유지

자율 학습

- Python heapq library 뜯어보기
  - push, pop정도만 뜯어봐도 충분

추가 자료

- Min-max heap: 최댓값과 최솟값 컨트롤을 동시에!