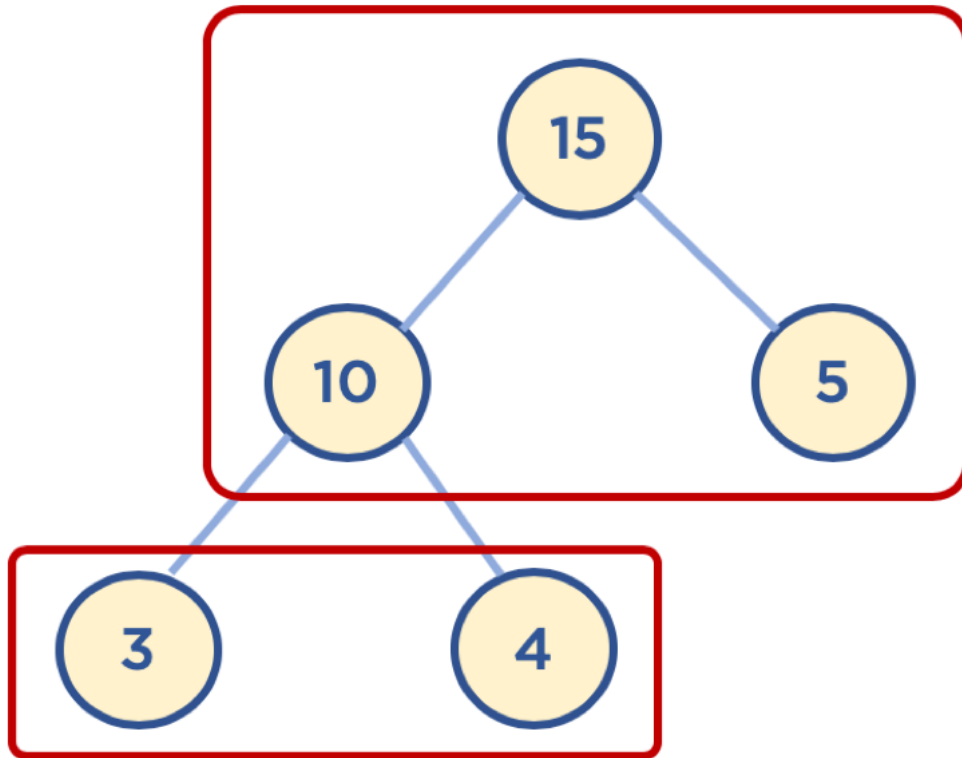


힉(Heap)

1. 힉(Heap)이란?

- 힉 : 데이터에서 최대값과 최소값을 빠르게 찾기 위해 고안된 완전 이진 트리(Complete Binary Tree)
 - 완전 이진 트리 : 노드를 삽입할 때 최하단 왼쪽 노드부터 차례대로 삽입하는 트리



- 힉을 사용하는 이유
 - 배열에 데이터를 넣고, 최대값과 최소값을 찾으려면 $O(n)$ 이 걸림
 - 이에 반해, 힉에 데이터를 넣고, 최대값과 최소값을 찾으려면, $O(\log n)$ 이 걸림
 - 우선순위 큐와 같이 최대값 또는 최소값을 빠르게 찾아야 하는 자료구조 및 알고리즘 구현 등에 활용됨

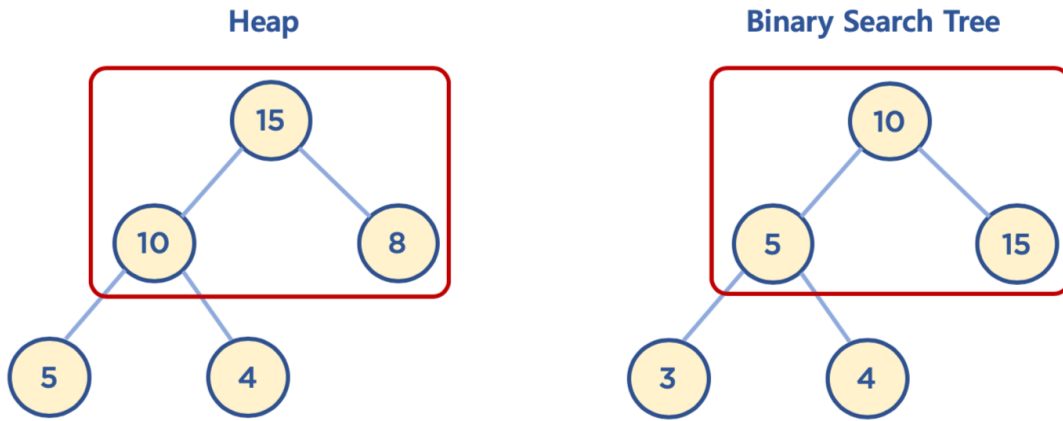
2. 힉(Heap)구조

- 힉은 최대값을 구하기 위한 구조(최대 힉, Max Heap)와 최소값을 구하기 위한 구조(최소 힉, Min Heap)로 분류할 수 있음
- 힉은 다음과 같이 두 가지 조건을 가지고 있는 자료구조임
 1. 각 노드의 값은 해당 노드의 자식 노드가 가진 값보다 크거나 같다. (최대 힉의 경우)
 - 최소 힉의 경우는 각 노드의 값은 해당 노드의 자식 노드가 가진 값보다 작거나 같음
 2. 완전 이진 트리 형태를 가짐

힉과 이진 탐색 트리의 공통점과 차이점

- 공통점 : 힉과 이진 탐색 트리는 모두 이진 트리임
- 차이점:

- 힙은 각 노드의 값이 자식 노드보다 크거나 같음(Max Heap의 경우)
- 이진 탐색 트리는 왼쪽 자식 노드의 값이 가장 작고, 그 다음 부모 노드, 그 다음 오른쪽 자식 노드 값이 가장 큼
- 힙은 이진 탐색 트리의 조건인 자식 노드에서 작은 값은 왼쪽, 큰 값은 오른쪽이라는 조건은 없음
 - 힙의 왼쪽 및 오른쪽 자식 노드의 값은 오른쪽이 클 수도 있고, 왼쪽이 클 수도 있음
- 이진 탐색 트리는 탐색을 위한 구조, 힙은 최대/최소값 **검색**을 위한 구조 중 하나로 이해하면 됨

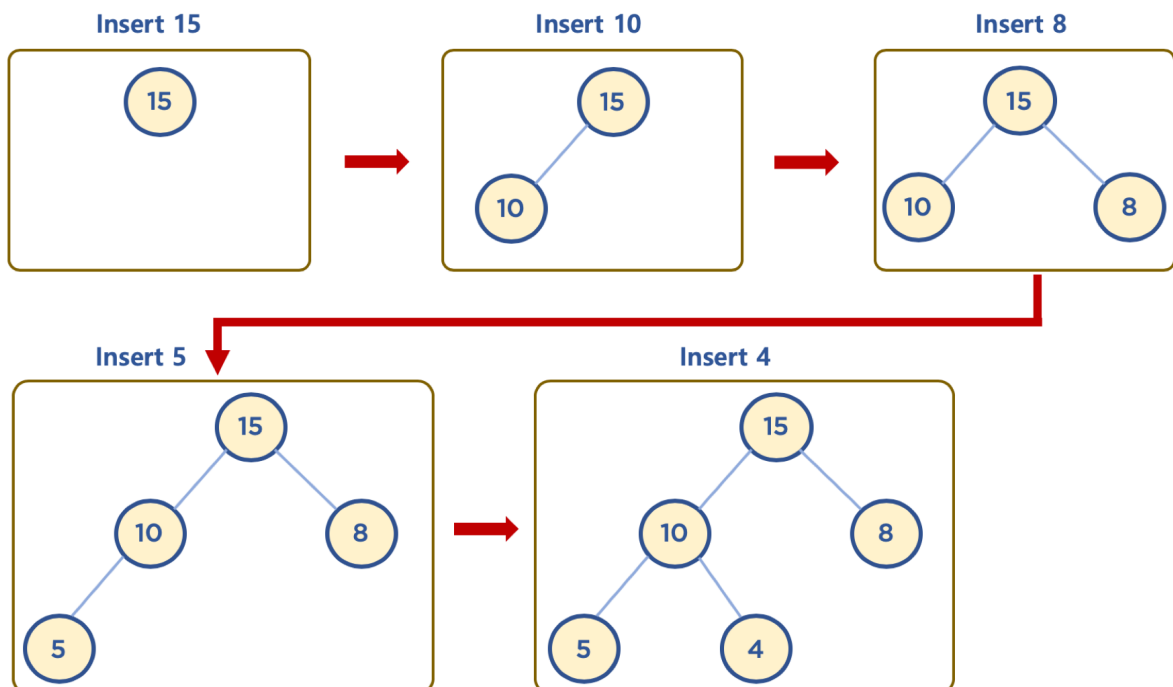


3. 힙(Heap) 동작

- 데이터를 힙 구조에 삽입, 삭제하는 과정을 그림을 통해 선명하게 이해하기

힙에 데이터 삽입하기 - 기본동작

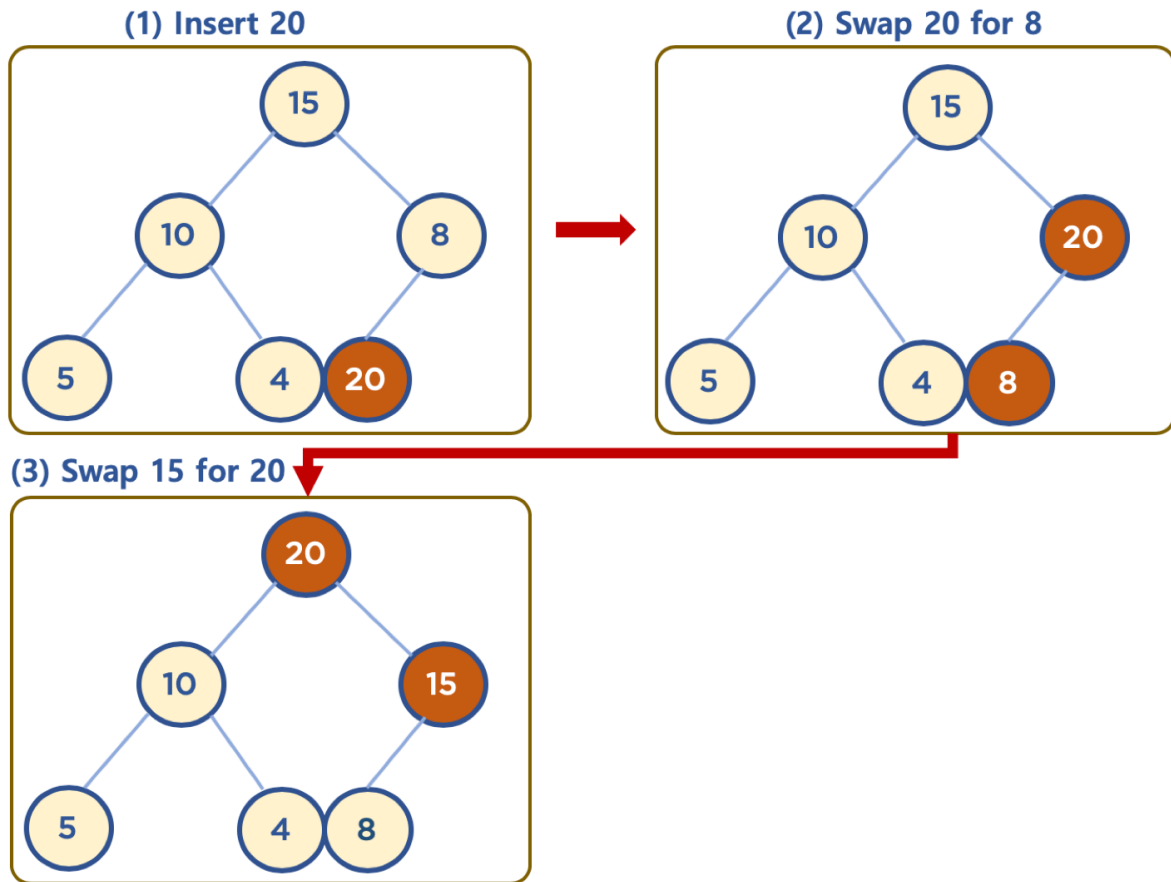
- 힙은 완전 이진 트리이므로, 삽입할 노드는 기본적으로 왼쪽 최하단 노드부터 채워지는 형태로 삽입



힙에 데이터 삽입하기 - 삽입할 데이터가 힙의 데이터보다 클 경우 (Max Heap의 예)

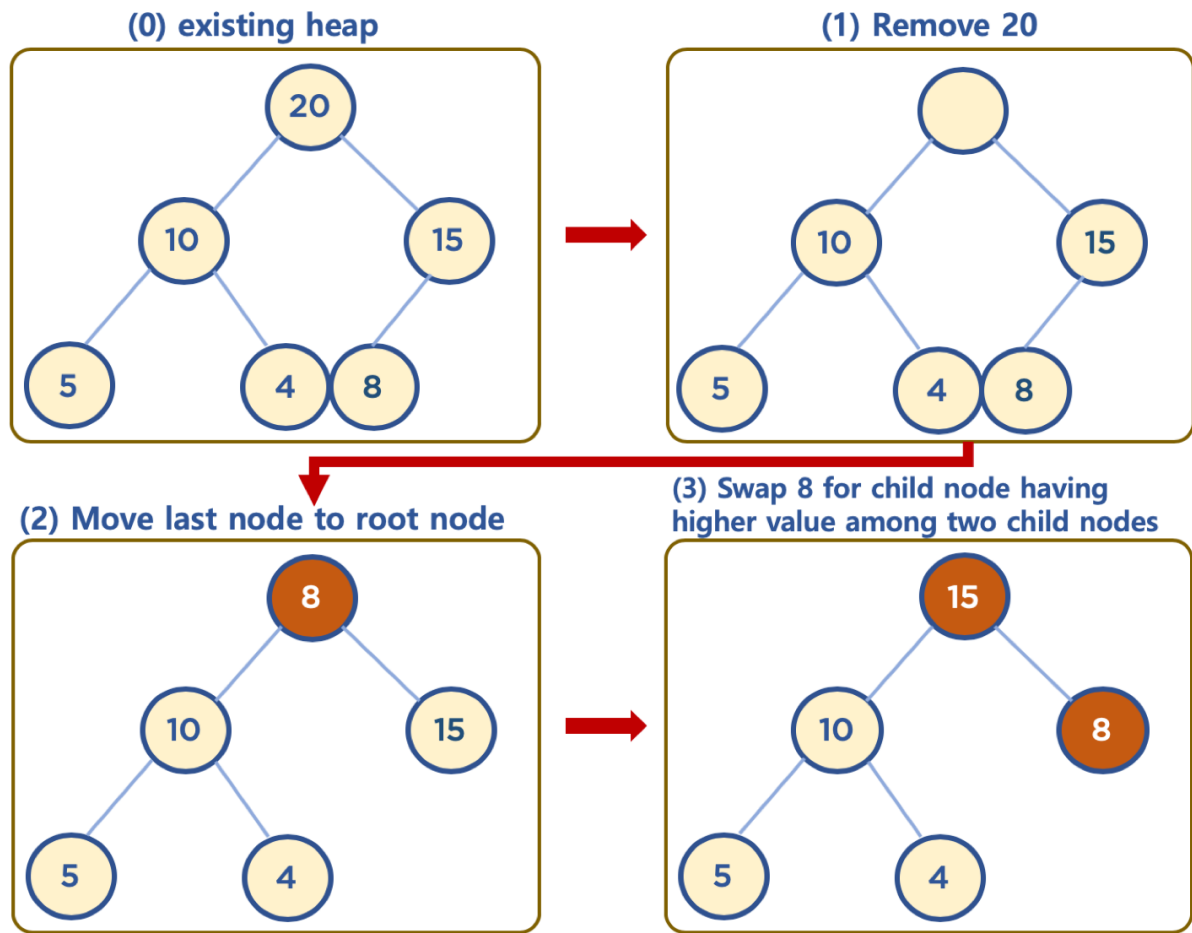
- 먼저 삽입된 데이터는 완전 이진 트리 구조에 맞추어, 최하단부 왼쪽 노드부터 채워짐

- 채워진 노드 위치에서, 부모 노드보다 값이 클 경우, 부모 노드와 위치를 바꿔주는 작업을 반복함(Swap)



힉의 데이터 삭제하기 (Max Heap 의 예)

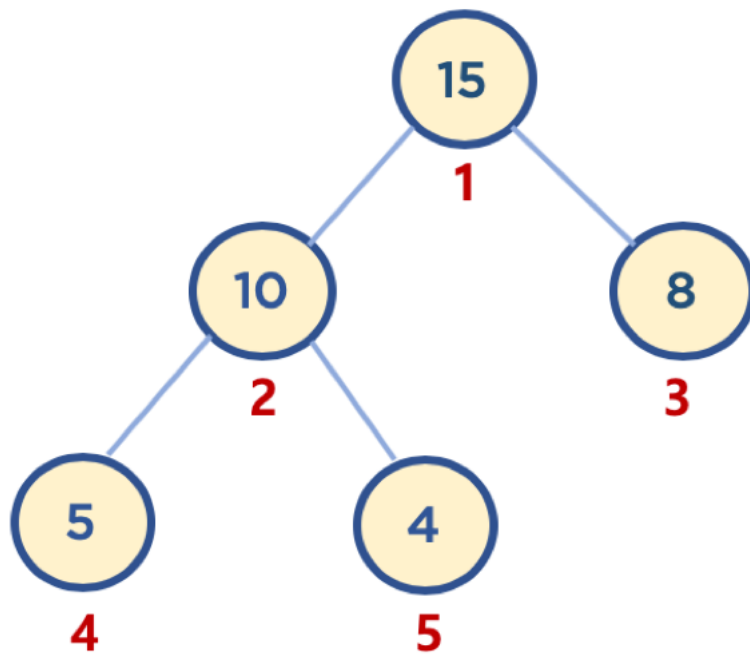
- 보통 삭제는 최상단 노드(root노드)를 삭제하는 것이 일반적임
 - 힉의 용도는 최대값 또는 최소값 root노드에 놓아서, 최대값과 최소값을 바로 꺼내 쓸 수 있도록 하는 것임
- 상단의 데이터 삭제시, 가장 최하단부 왼쪽에 위치한 노드(일반적으로 가장 마지막에 추가한 노드)를 root노드로 이동
- root노드의 값이 child node보다 작을 경우, root노드의 child node 중 가장 큰 값을 가진 노드와 root노드 위치를 바꿔주는 작업을 반복함 (swap)



4. 힙 구현

힙과 배열

- 일반적으로 힙 구현시 배열 자료구조를 활용함
- 배열은 인덱스가 0번부터 시작하지만, 힙 구현의 편의를 위해, root노드 인덱스번호를 1로 지정하면, 구현이 좀 더 수월함
 - 부모 노드 인덱스 번호(parent node's index) = 자식 노드 인덱스 번호(child node's index) // 2
 - 왼쪽 자식 노드 인덱스 번호(left child node's index) = 부모 노드 인덱스 번호(parent node's index) * 2
 - 오른쪽 자식 노드 인덱스 번호(right child node's index) = 부모 노드 인덱스 번호(parent node's index) * 2 + 1



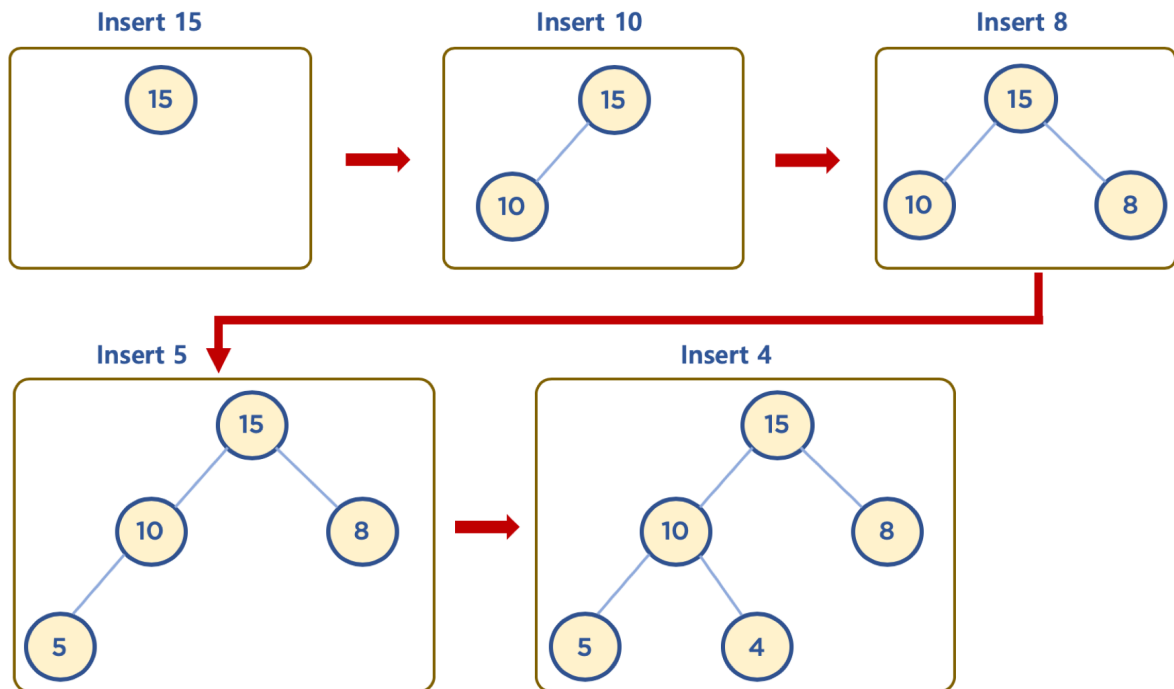
힉과 데이터 삽입 구현(Max Heap예)

- 힉 클래스 구현1

```
class Heap:
    def __init__(self, data):
        self.heap_array = list()
        self.heap_array.append(None)
        self.heap_array.append(data)
```

```
heap = Heap(1)
heap.heap_array
>>[None, 1]
```

- 힉 클래스 구현2 - insert1
 - 인덱스 번호는 1번부터 시작하도록 변경

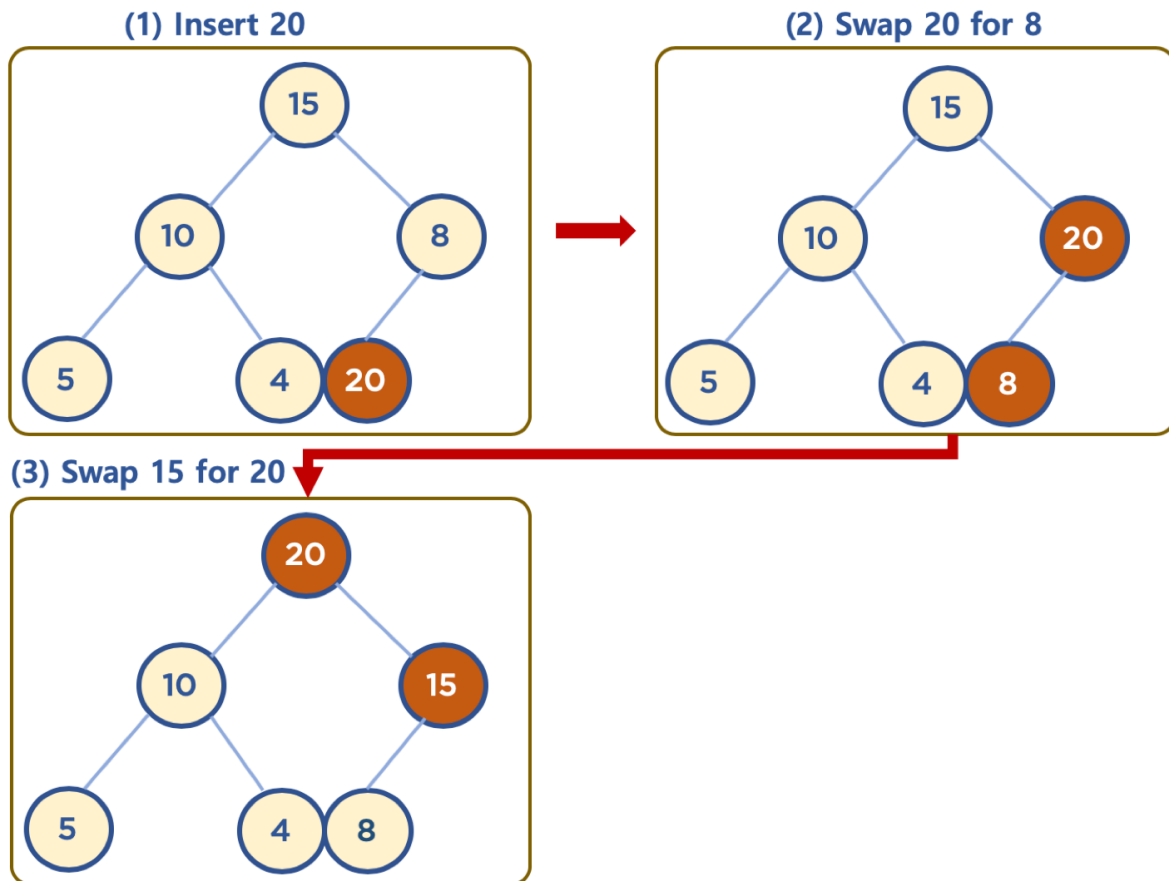


```

class Heap:
    def __init__(self,data):
        self.heap_array = list()
        self.heap_array.append(None)
        self.heap_array.append(data)
    def insert(self,data):
        if len(self.heap_array)==0:
            self.heap_array.append(None)
            self.heap_array.append(data)
            return True
        self.heap_array.append(data)
        return True
  
```

• 힙 클래스 구현3 - insert2

- 삽입한 노드가 부모 노드의 값보다 클 경우, 부모 노드와 삽입한 노드 위치를 바꿈
- 삽입한 노드가 루트 노드가 되거나, 부모 노드보다 값이 작거나 같을 경우까지 반복



```

heap = Heap(15)
heap.insert(10)
heap.insert(8)
heap.insert(5)
heap.insert(4)
heap.insert(20)
heap.heap_array
>>[None, 15, 10, 8, 5, 4, 20]

```

```

class Heap:
    def __init__(self, data):
        self.heap_array = list()
        self.heap_array.append(None)
        self.heap_array.append(data)

    def move_up(self, inserted_idx):
        if inserted_idx <= 1:
            return False
        parent_idx = inserted_idx // 2
        if self.heap_array[inserted_idx] > self.heap_array[parent_idx]:
            return True
        else:
            return False

    def insert(self, data):
        if len(self.heap_array) == 0:
            self.heap_array.append(None)
            self.heap_array.append(data)
            return True
        self.heap_array.append(data)

        inserted_idx = len(self.heap_array) - 1

        while self.move_up(inserted_idx):
            parent_idx = inserted_idx // 2
            self.heap_array[inserted_idx], self.heap_array[parent_idx] = self.heap_array[parent_idx], self.heap_array[inserted_idx]
            inserted_idx = parent_idx
        return True

```

```

heap = Heap(15)
heap.insert(10)
heap.insert(8)
heap.insert(5)
heap.insert(4)
heap.insert(20)
heap.heap_array
>>[None, 20, 10, 15, 5, 4, 8]

```

힅에 데이터 삭제 구현(Max Heap 예)

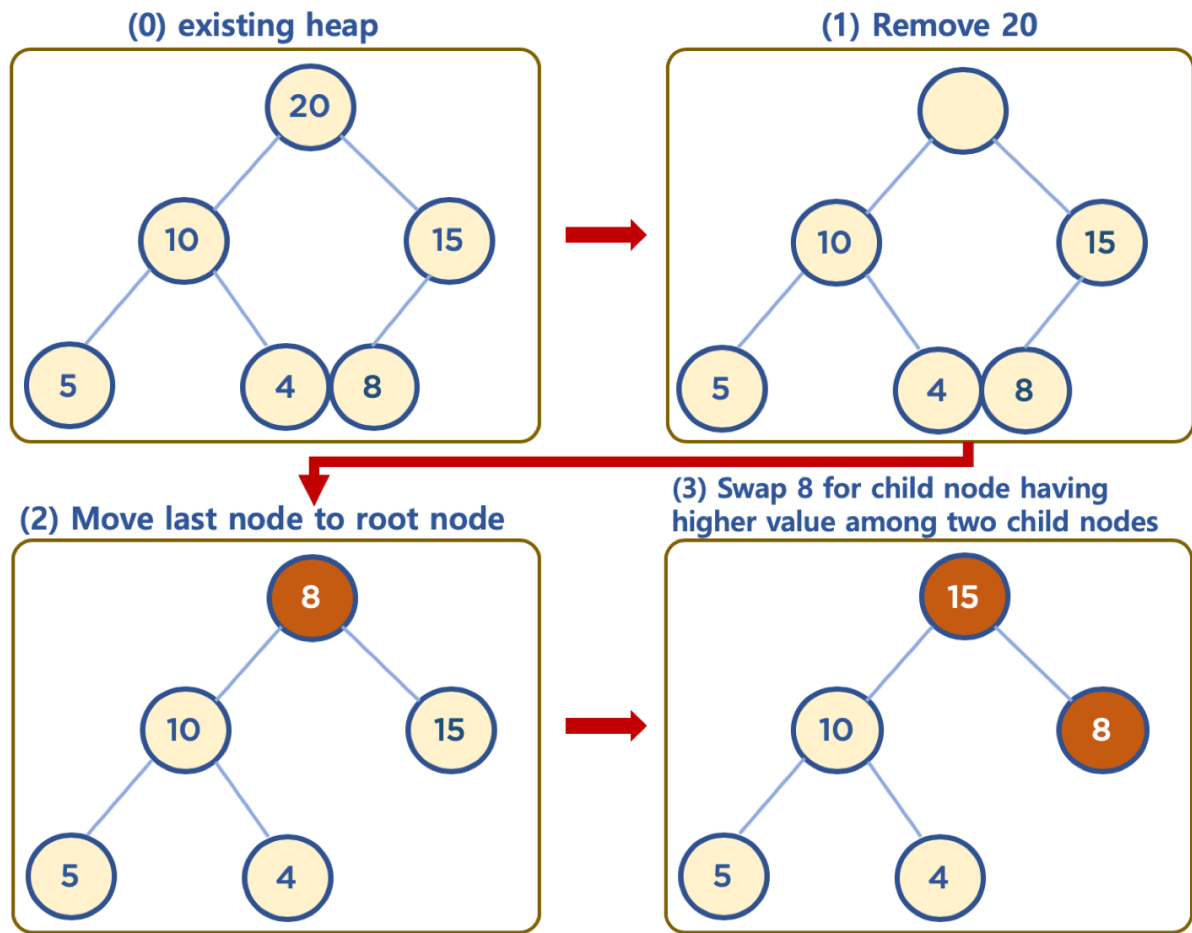
- 힅 클래스 구현4 - delete1
- 보통 삭제는 최상단 노드(root 노드)를 삭제하는 것이 일반적임
 - 힅의 용도는 최대값 또는 최소값을 root노드에 놓아서, 최대값과 최소값을 바로 꺼내 쓸 수 있도록 하는 것임

```

class Heap:
    def __init__(self, data):
        self.heap_array = list()
        self.heap_array.append(None)
        self.heap_array.append(data)
    def pop(self):
        if len(self.heap.array)<=1:
            return None
        returned_data = self.heap_array[1]
        return returned_data

```

- 힅 클래스 구현4 - delete2
 - 상단의 데이터 삭제시, 가장 최하단부 왼쪽에 위치한 노드(일반적으로 가장 마지막에 추가한 노드)를 root노드로 이동
 - root노드의 값이 child node보다 작을 경우, root노드의 child node중 가장 큰 값을 가진 노드와 root노드 위치를 바꿔주는 작업을 반복함(swap)



```
class Heap:
    def __init__(self, data):
        self.heap_array = list()
        self.heap_array.append(None)
        self.heap_array.append(data)

    def move_down(self, popped_idx):
        left_child_popped_idx = popped_idx * 2
        right_child_popped_idx = popped_idx * 2 + 1

        #case1 : 왼쪽 자식 노드도 없을 때
        if left_child_popped_idx >= len(self.heap_array):
            return False
        #case2 : 오른쪽 자식 노드만 없을 때
        elif right_child_popped_idx >= len(self.heap_array):
            if self.heap_array[popped_idx] < self.heap_array[left_child_popped_idx]:
                return True
            else:
                return False
        #case3 : 왼쪽, 오른쪽 자식 노드 모두 있을 때
        else:
            if self.heap_array[left_child_popped_idx] > self.heap_array[right_child_popped_idx]:
                if self.heap_array[popped_idx] < self.heap_array[left_child_popped_idx]:
                    return True
            else:
                return False
            else:
                if self.heap_array[popped_idx] < self.heap_array[right_child_popped_idx]:
                    return True
                else:
                    return False

    def pop(self):
        if len(self.heap_array) <= 1:
            return None

        returned_data = self.heap_array[1]
        self.heap_array[1] = self.heap_array[-1]
        del self.heap_array[-1]
        popped_idx = 1
```

```

while self.move_down(popped_idx):
    left_child_popped_idx = popped_idx * 2
    right_child_popped_idx = popped_idx * 2 + 1

    #case 2 : 오른쪽 자식 노드만 없을 때
    if right_child_popped_idx >= len(self.heap_array):
        if self.heap_array[popped_idx] < self.heap_array[left_child_idx]:
            self.heap_array[popped_idx], self.heap_array[left_child_popped_idx] = self.heap_array[left_child_popped_idx], self.heap_array[popped_idx]
            popped_idx = left_child_popped_idx
    #case3 : 왼쪽, 오른쪽 자식 노드 모두 있을 때
    else:
        if self.heap_array[left_child_popped_idx] > self.heap_array[right_child_popped_idx]:
            if self.heap_array[popped_idx] < self.heap_array[left_child_popped_idx]:
                self.heap_array[popped_idx], self.heap_array[left_child_popped_idx] = self.heap_array[left_child_popped_idx], self.heap_array[popped_idx]
                popped_idx = left_child_popped_idx
            else:
                if self.heap_array[popped_idx] < self.heap_array[right_child_popped_idx]:
                    self.heap_array[popped_idx], self.heap_array[right_child_popped_idx] = self.heap_array[right_child_popped_idx], self.heap_array[popped_idx]
                    popped_idx = right_child_popped_idx
        return returned_data

```

5. 힙(Heap) 시간 복잡도

- depth(트리의 높이)를 h 라고 표기한다면,
- n 개의 노드를 가지는 heap에 데이터 삽입 또는 삭제 시, 최악의 경우 root노드에서 leaf노드까지 비교해야 하므로 $h = \log n$ 에 가까우므로, 시간복잡도는 $O(\log n)$