부록 Ⅱ 힙 자료구조

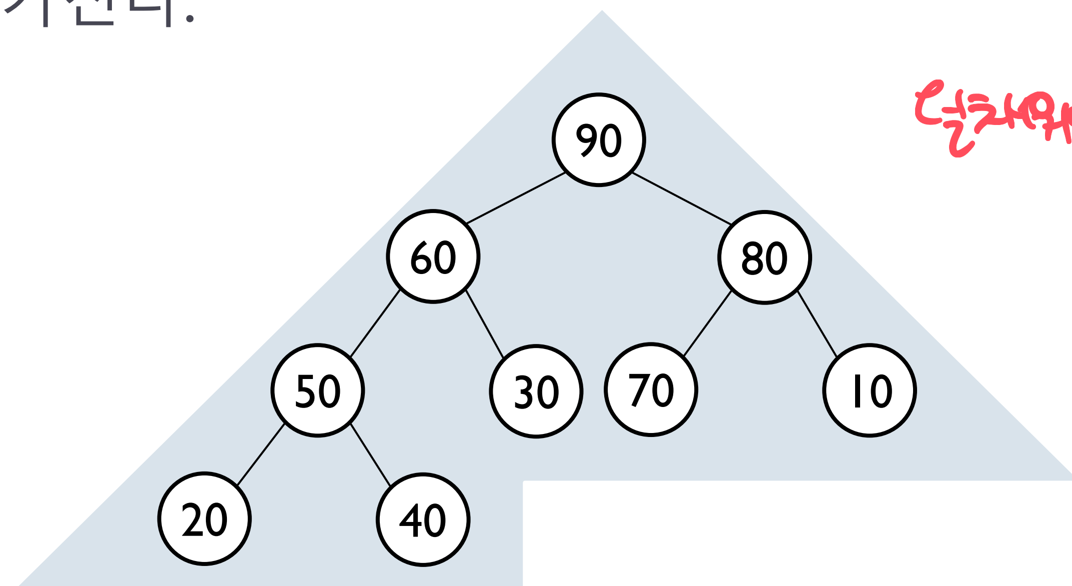
1 힙 자료구조

- 힙(heap)은 최솟값(또는 최댓값)을 빠른 시간에 접근하도록 만들어진 자료구조이다.
  - 최댓값을 빠르게 접근하려면 최대힙(maximum heap)을 사용하여야 하고, 최솟값을 빠르게 접근하려면 최소힙(minimum heap)을 사용하여야 한다.
  - 두 개의 힙이 대칭성을 가지므로, 이 중에서 최대힙 자료구조에 대해서만 설명한다.
- 힙은 다음과 같은 조건을 만족하는 이진트리이다.
  - 각 노드의 값이 자식 노드들의 값들보다 크다.
  - 트리는 완전 이진트리(complete binary tree)이다.

원래정렬 nlogn

완전 이진트리

- 완전 이진트리는 이진트리로서 다음과 같은 특성을 가진다.
  - 트리의 마지막 층에 있는 이파리 노드들은 왼쪽부터 꽉 차 있는 형태를 가진다.

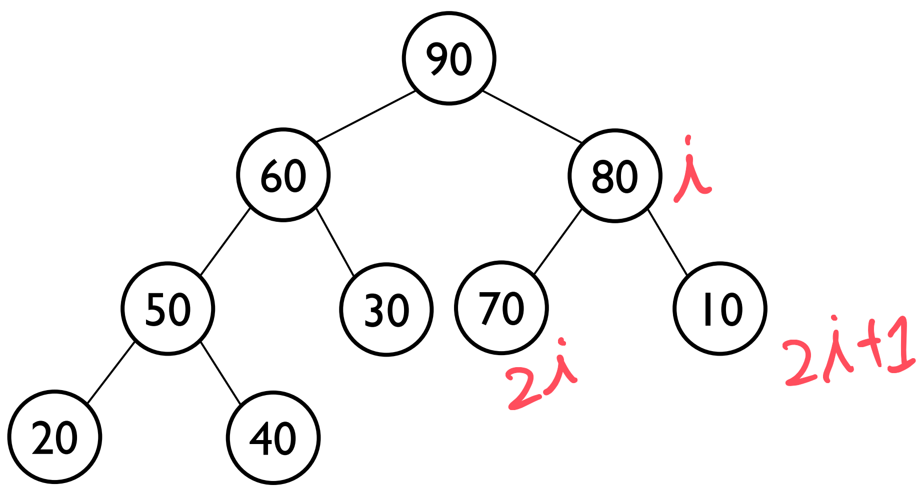


덜채워있어도 왼쪽부터

최대힙

- 최대힙의 루트에는 가장 큰 값이 저장된다. 또한 n개의 노드를 가진 힙은 완전 이진트리이므로, 힙의 높이가 $\log_2 n$이며, 노드들의 값이 빈틈 없이 배열에 저장된다. 다음의 그림은 힙의 노드들이 배열에 저장된 모습을 보여주고 있다.

답에르개 가능



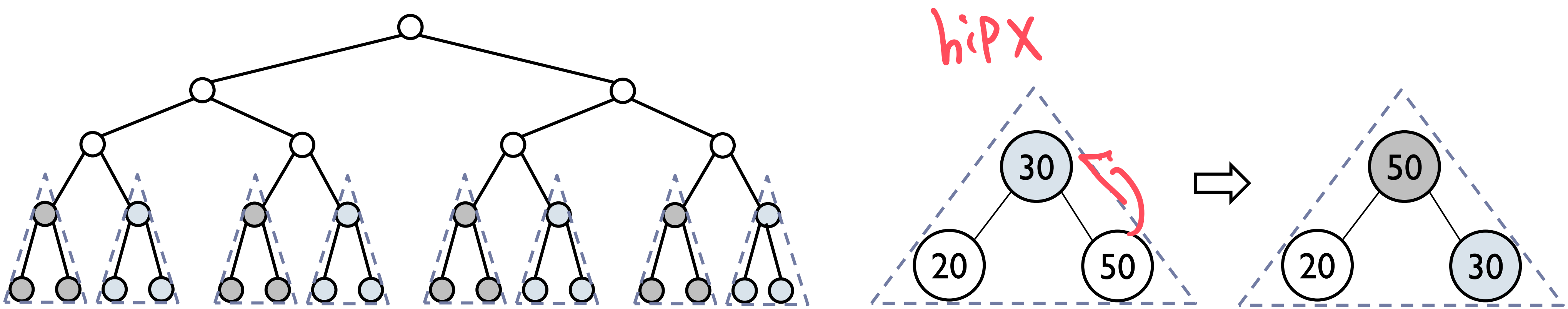
0	1	2	3	4	5	6	7	8	9
-	90	60	80	50	30	70	10	20	40

힙 자료구조

- 배열 A에 힙을 저장한다면, A[0]은 비어 두고, A[1]부터 A[n]까지에 힙 노드들을 트리의 층별로 좌우에 저장한다. 그림에서 보면 루트의 90이 A[1]에 저장되고, 그 다음 층의 60과 80이 각각 A[2]와 A[3]에 저장되며, 그 다음 층의 50, 30, 70, 10이 A[4]에서 A[7]에 각각 저장되고, 마지막으로 20과 40이 A[8]과 A[9]에 저장되어 있다.
- 이와 같이 힙을 배열에 저장하면, 부모 노드와 자식 노드의 관계를 배열의 인덱스로 쉽게 표현할 수 있다. A[i]에 저장된 노드의
  - 부모 노드는 A[i/2]에 저장되어 있다. 단 i가 홀수일 때, i/2에서 정수 부분만을 취한다. 예를 들어 A[7]에 있는 10의 부모 노드는 A[7/2]=A[3]에 저장되어 있다.
  - 왼쪽 자식 노드는 A[2i]에 저장되고, 오른쪽 자식 노드는 A[2i+1]에 저장된다. 예를 들어 A[4]에 있는 50의 왼쪽 자식 노드 20은 A[2i]=A[2×4]=A[8]에 저장되고, 오른쪽 자식 노드 40은 A[2i+1]=A[2×4+1]=A[9]에 저장된다.

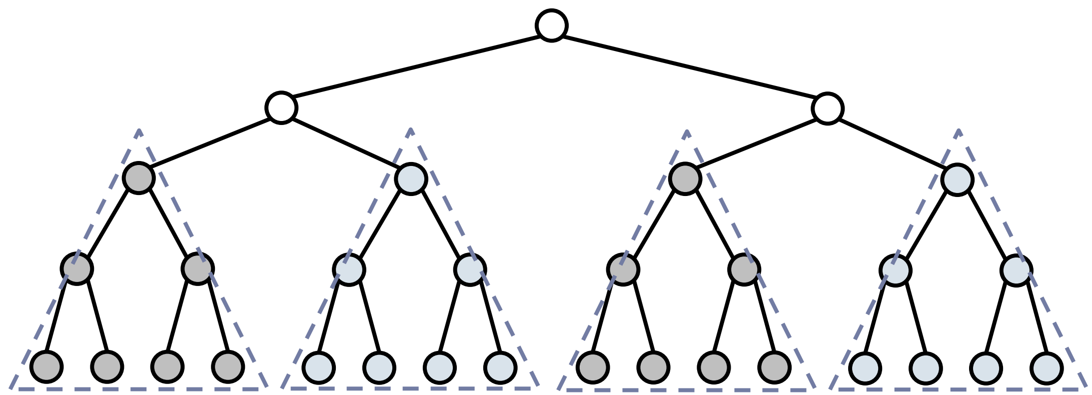
2. 힙 만들기

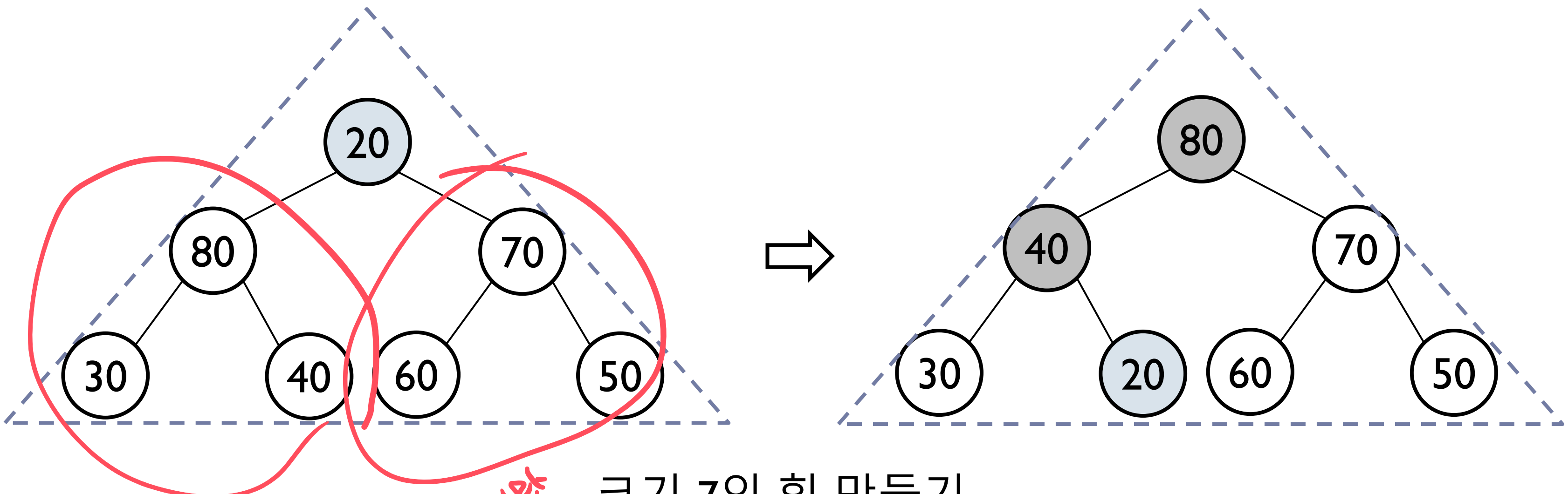
- 크기 n인 배열 A의 힙을 O(n) 시간에 만드는 알고리즘을 소개한다. 단, n개의 숫자들이 A[1]~A[n]에 저장되어 있다.
- 이 알고리즘은 bottom-up 방식으로 힙을 만든다. 다음과 같이 크기가 3인 힙을 만들고, 그 다음에는 크기가 7인 힙을 만들고, 크기가 15인 힙을 만들어 계속 힙 크기를 (2^i-1)로 키워가며 완전한 힙을 만든다.



크기 3인 힙 만들기

힙 만들기



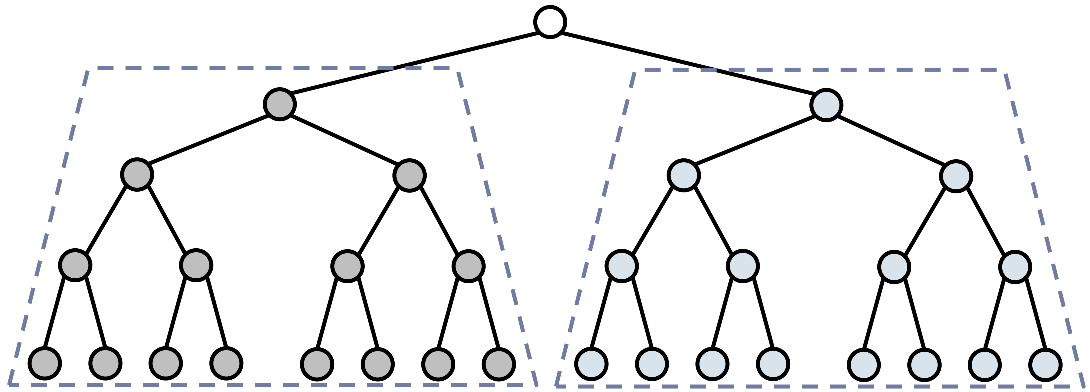


크기 7인 힙 만들기

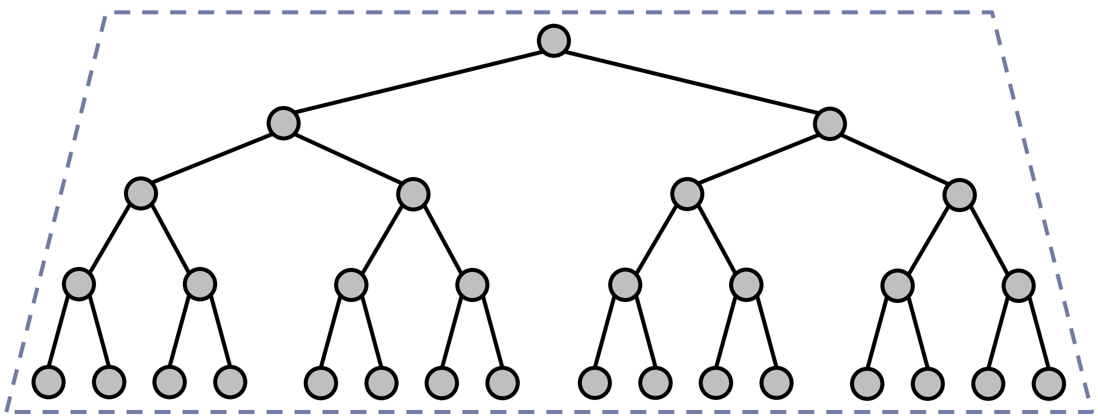
각각 hip 이라면

20을 내려줘야함

힙 만들기



크기 15인 힙 만들기



크기 31인 힙 만들기

힙 만들기 알고리즘

```
BuildHeap(A){
    for(i=⌊n/2⌋ to 1)
        DownHeap(i)
}
```

배열 입력받음

index를 큰거부터 보름. 배열 오른쪽부터 보름 index 점점 작아짐.

트리를 작은것부터 만드는것.

DownHeap(i) ex 15

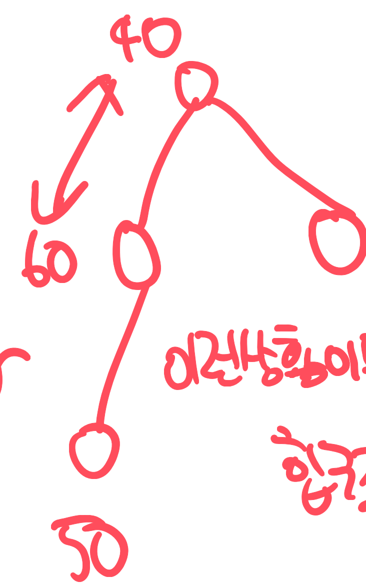
```
leftChild  = 2i          // i의 왼쪽 자식 노드
rightChild = 2i+1        // i의 오른쪽 자식 노드
if ((leftChild ≤ n) and (A[leftChild]>A[i]))
    bigger = leftChild
else
    bigger = i
if ((rightChild ≤ n) and (A[rightChild]>A[i]))
    bigger = rightChild
if (bigger != i) {
    A[i] ↔ A[bigger]
    DownHeap(bigger)
}
```

30 31

30 30 15

참이 된다면

bigger

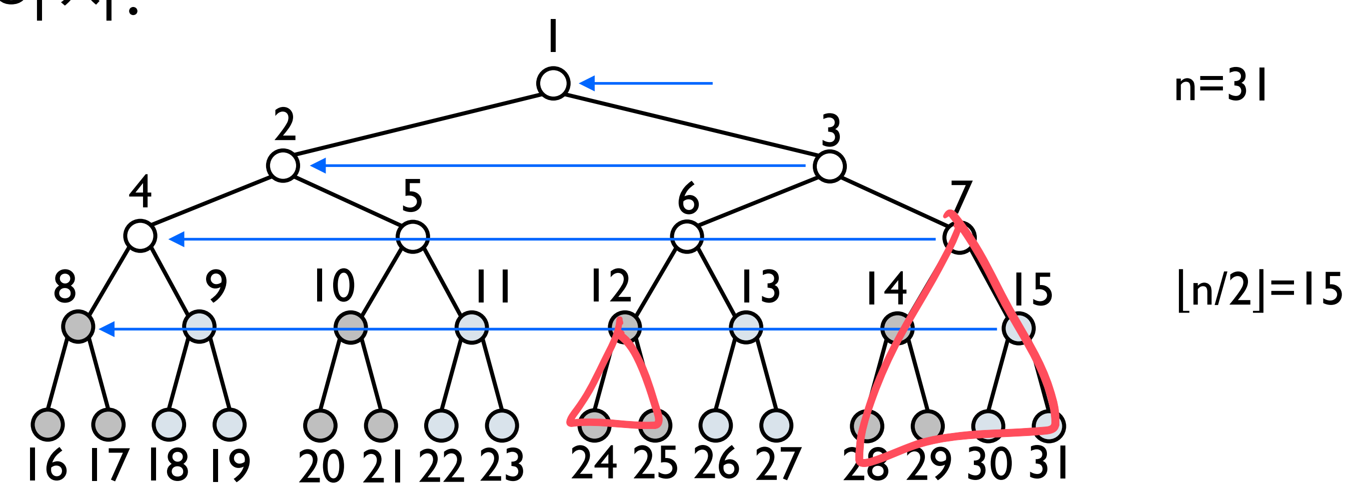


이런상황이면 아래의 힙구조를 다시 만듦

O(n) 상수시간

힙 만들기 알고리즘

- 그림은 BuildHeap 알고리즘이 배열 A[1]~A[31]에 대해서 수행되는 순서를 화살표로 나타내고 있다. 각 노드 옆의 숫자는 노드의 값이 저장된 배열 원소의 인덱스이다. 가장 먼저 $i=\lfloor n/2\rfloor=\lfloor 31/2\rfloor=15$에서 DownHeap을 호출하여, i=1 일 때까지 DownHeap을 호출한다. 즉, for-루프에서 15번의 DownHeap이 호출되는데, 이때 i에 대응되는 노드를 '시작 노드'라고 하자.



힙 만들기 알고리즘

Line 1~2	DownHeap은 시작 노드의 값을 자식 노드들의 값과 비교하여 힙 조건이 만족될 때까지 시작 노드의 값을 아래쪽으로 자리를 바꾸며 이동시킨다. 시작 노드(인덱스 i인 노드)의 왼쪽과 오른쪽 자식 노드들의 인덱스를 각각 leftChild=2i, rightChild=2i+1로 놓는다.
Line 3~6	시작 노드의 값과 왼쪽 자식 노드의 값 중에서 큰 값을 가진 노드의 인덱스를 bigger라는 변수에 저장한다. Line 3의 if-조건에서 (leftChild≤n)의 검사는 '왼쪽 자식 노드가 힙에 포함되는지'를 검사하는 것이다. 만일 조건이 '거짓'이 되면, 왼쪽 자식 노드가 힙에 없는 것이므로 시작 노드가 왼쪽 자식 노드를 가지고 있지 않다는 의미이다.

힙 만들기 알고리즘

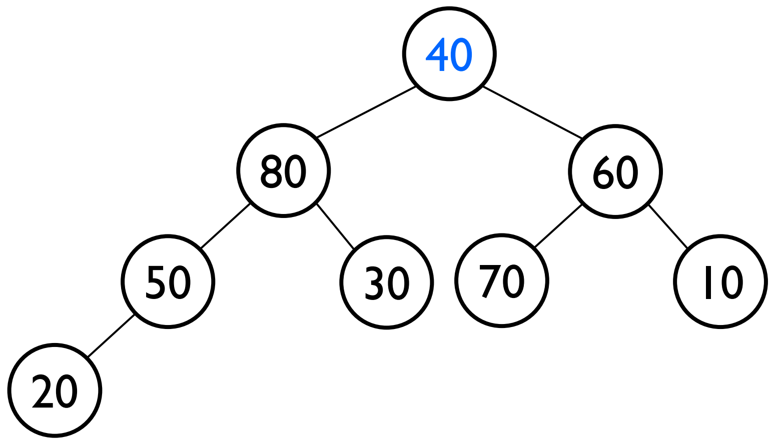
Line 7~8	bigger를 인덱스로 가지는 노드의 값(즉, 시작 노드의 값과 왼쪽 자식 노드의 값 중에서 큰 값을 가진 노드의 값)과 오른쪽 자식 노드의 값 중에서 큰 값을 가진 노드의 인덱스를 bigger에 저장한다. if-조건에서 (rightChild≤n)의 검사는 역시 '오른쪽 자식 노드가 힙에 포함되는지'를 검사하는 것이다. 만일 조건이 '거짓'이 되면, 오른쪽 자식 노드가 힙에 없는 것이므로 시작 노드가 오른쪽 자식 노드를 가지고 있지 않다는 것을 의미한다.

힙 만들기 알고리즘

Line 9~11	bigger가 시작 노드의 인덱스(i)와 같으면, 시작 노드의 값이 힙 조건을 만족하여 더 이상의 자리바꿈이나 DownHeap 호출이 필요 없다. 그러나 그렇지 않으면, 두 자식 중에서 bigger를 인덱스로 가지는 노드의 값이 시작 노드의 값보다 커서 bigger가 i와 다르므로, 시작 노드의 값과 bigger를 인덱스로 가지는 노드의 값을 교환한 후에, 즉 시작 노드의 값을 자식 노드로 내려 보냈으니 다시 이전과 같이 힙 조건을 만족시키기 위해 DownHeap을 재귀 호출한다. 여기서 재귀 호출될 때의 인자는 bigger인데, bigger를 인덱스로 가지는 노드에 시작 노드의 값이 저장되어 DownHeap이 호출되는 것이다. 따라서 이 노드를 계속해서 '시작 노드'로 놓으면, 재귀 호출 후 수행 과정을 이해하기 쉬울 것이다.

BuildHeap 예제

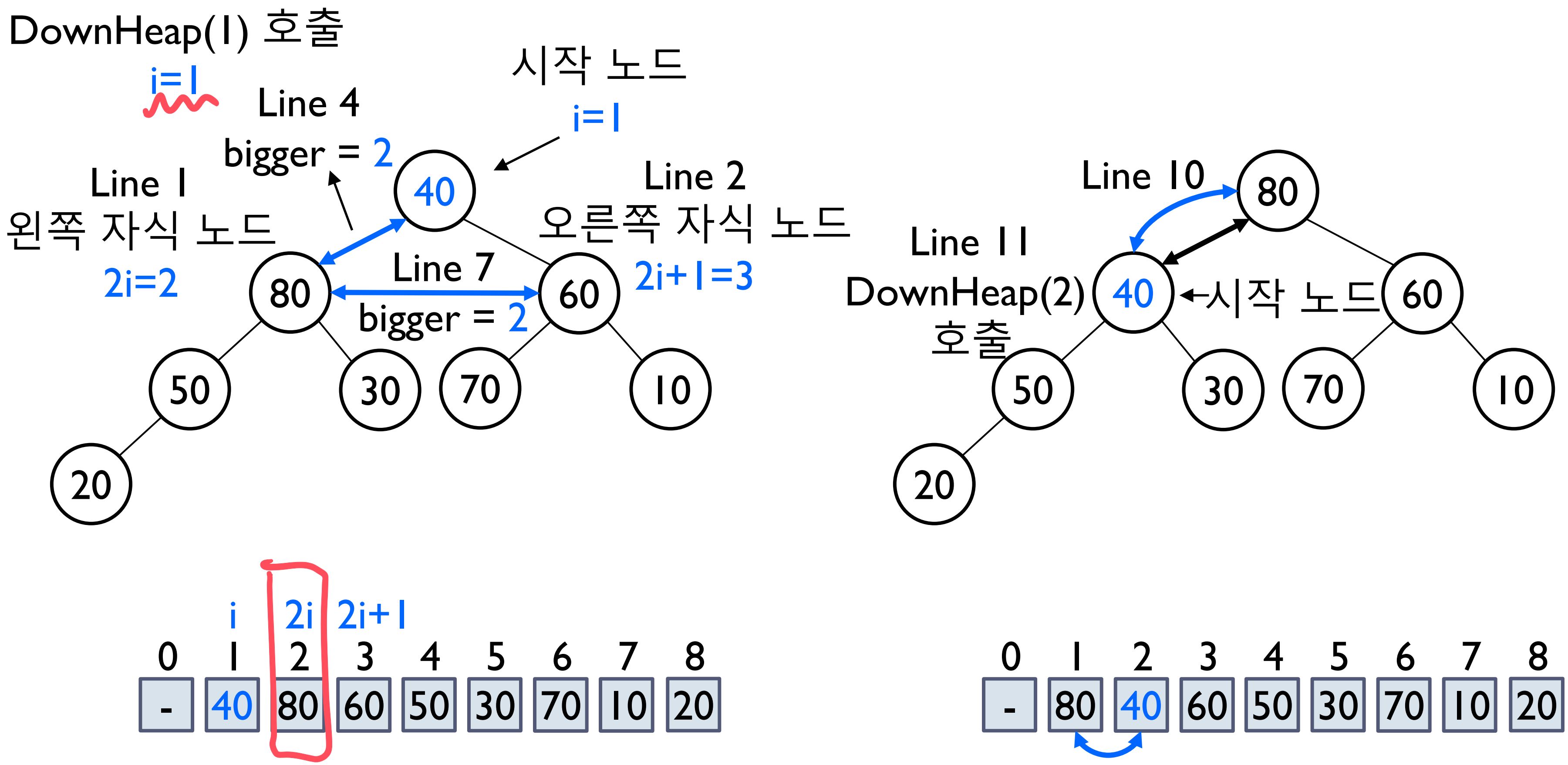
- DownHeap의 수행과정을 다음의 예제를 통해서 살펴보자.



0	1	2	3	4	5	6	7	8
-	40	80	60	50	30	70	10	20

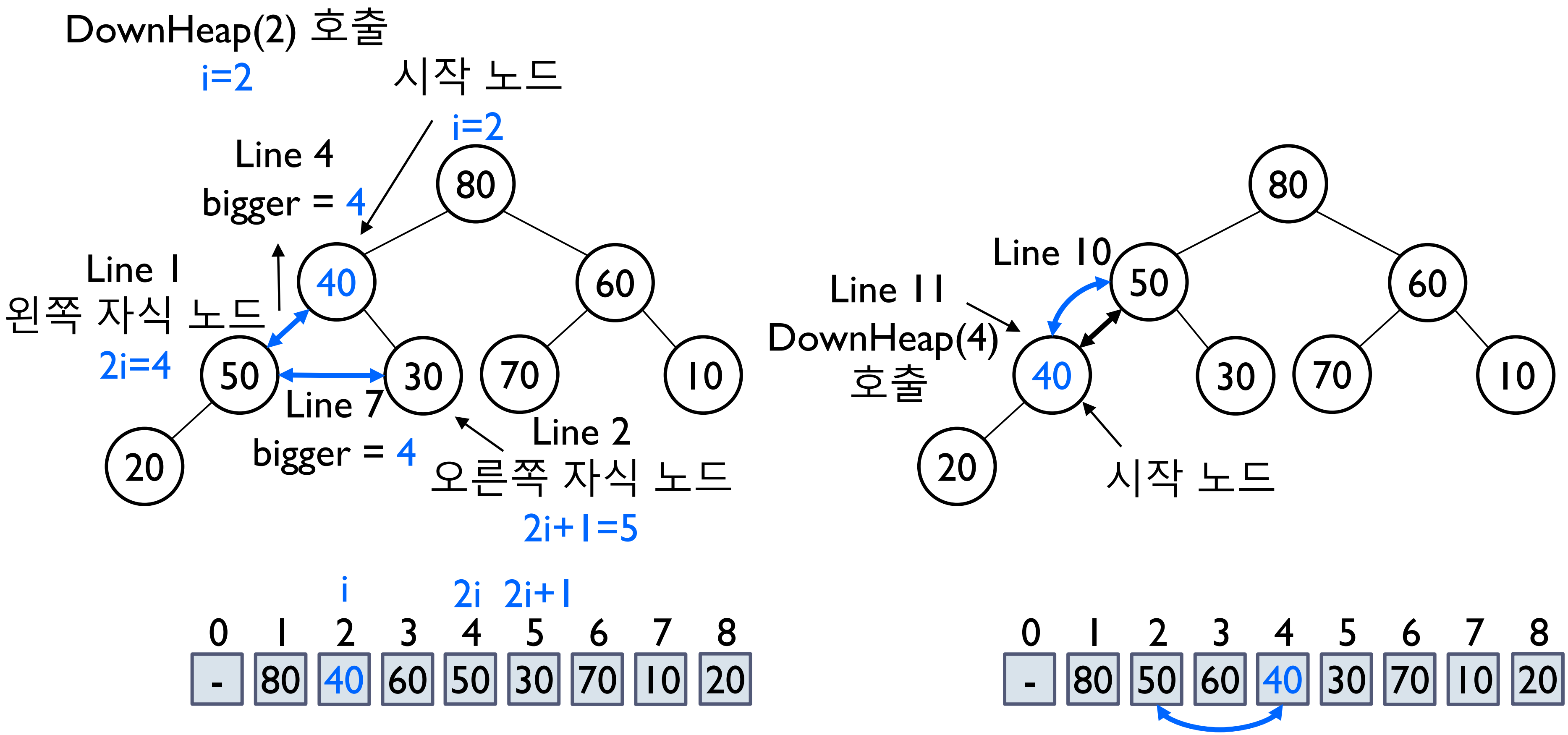
BuildHeap 예제

- DownHeap(1)이 호출 되었을 때의 수행과정:



BuildHeap 예제

- DownHeap(2)이 호출 되었을 때의 수행과정:

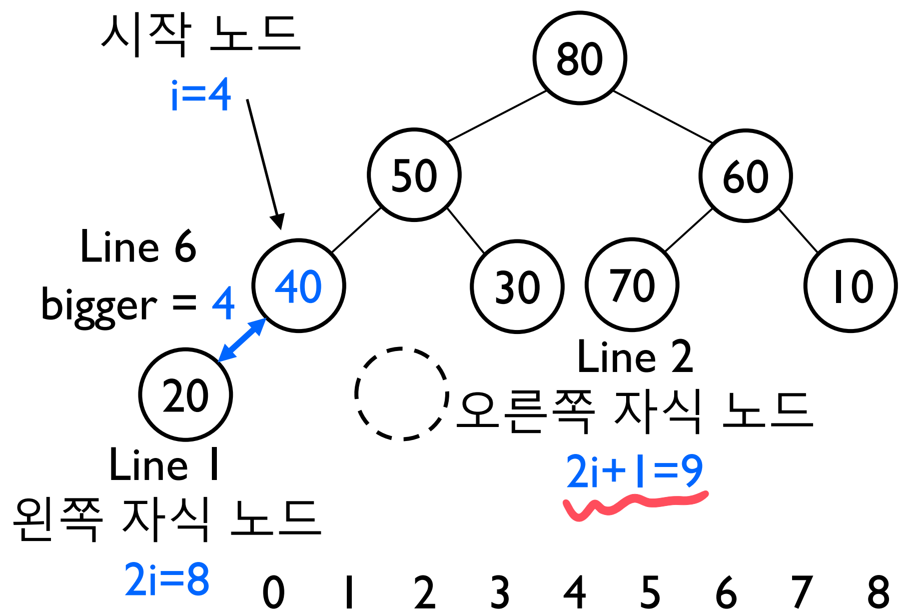


BuildHeap 예제

- DownHeap(4)이 호출 되었을 때의 수행과정:

DownHeap(4) 호출

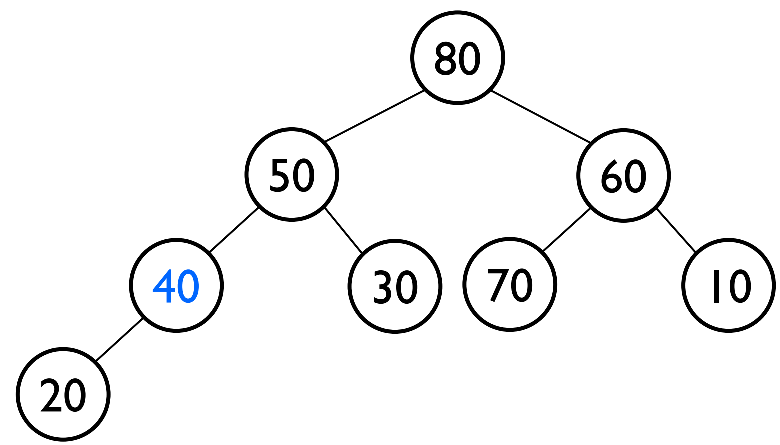
i=4



0	1	2	3	4	5	6	7	8
-	80	50	60	40	30	70	10	20

n=8 이었음. 오른꺼 없음.

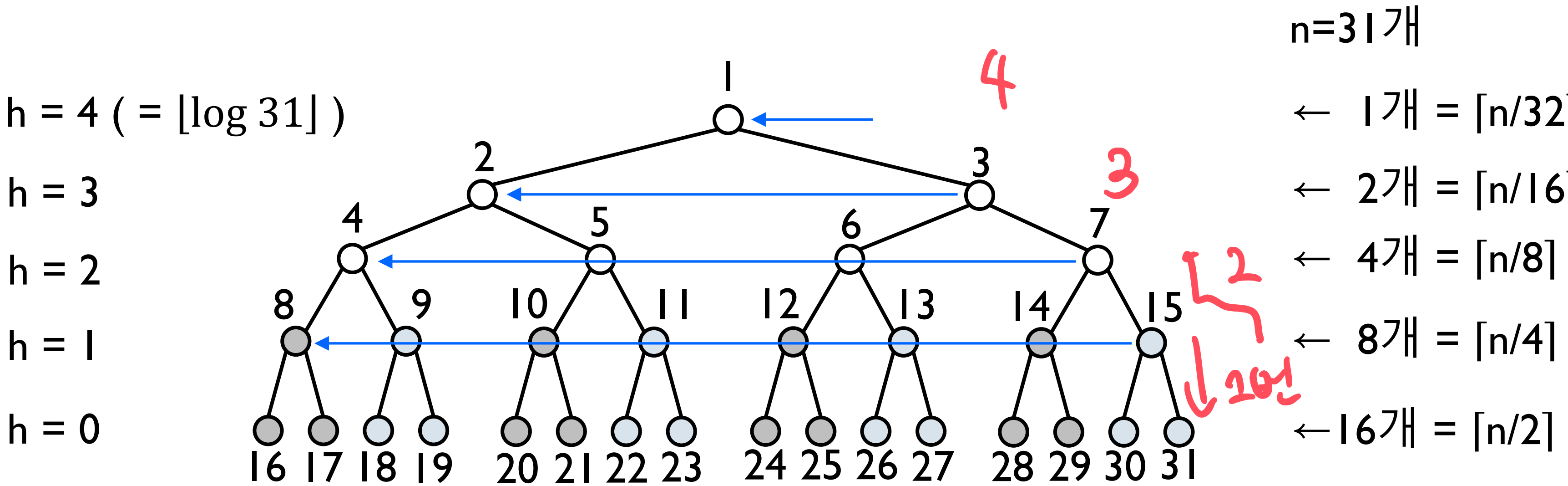
DownHeap 종료



0	1	2	3	4	5	6	7	8
-	80	50	60	40	30	70	10	20

힙 만들기: 시간복잡도

- 그림은 노드 수가 31개일 때 BuildHeap이 수행되는 과정을 보여준다. h=1일 때, 최악의 경우 시작 노드가 1층 내려가게 되고, h=2일 때, 최악의 경우 시작 노드가 2층 내려가게 되며, ..., h=4일 때, 최악의 경우 시작 노드가 4층 내려가서 이 파리 노드에 저장된다.



힙 만들기: 시간복잡도

- 그런데 한 개 층을 내려가는 일은 DownHeap을 호출하여 이루어지는데 DownHeap의 수행 시간은 루프의 반복 없이 상수 시간에 수행된다. 따라서 BuildHeap의 시간복잡도는 (각 층에 있는 노드 수)×(층 높이)이다.

시간복잡도

$$= \sum_{h=0}^{\lfloor \log n \rfloor} h \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

$$= O\left(\sum_{h=0}^{\lfloor \log n \rfloor} h \frac{n}{2^h}\right)$$

$$= O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} h \frac{1}{2^h}\right)$$

$$= O\left(n \sum_{h=0}^{\infty} h \frac{1}{2^h}\right)$$

$$= O(2n) = O(n)$$

for $|x| < 1$,

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

미분

$$\sum_{k=0}^{\infty} k x^k = \frac{1}{(1-x)^2}$$

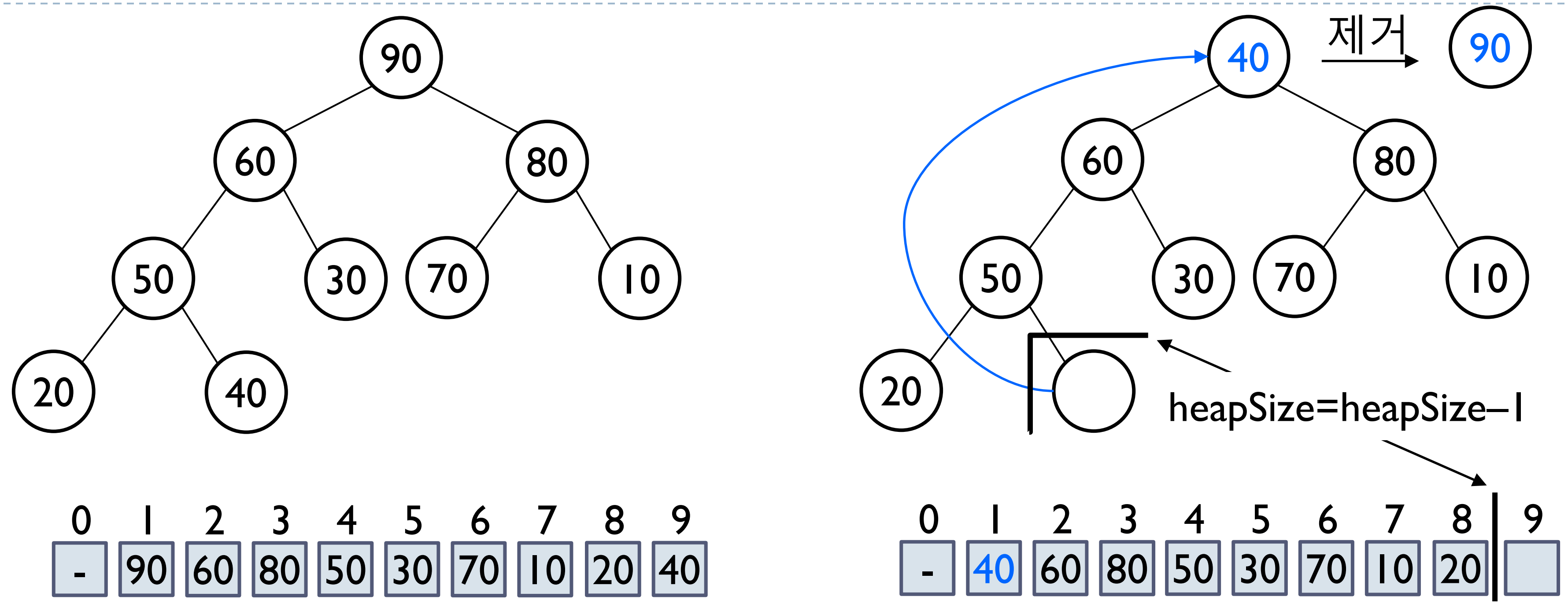
$$\therefore \sum_{k=0}^{\infty} k \frac{1}{2^k} = \frac{1}{(1-\frac{1}{2})^2} = 2$$

최대값, 최소값

3 삭제 연산

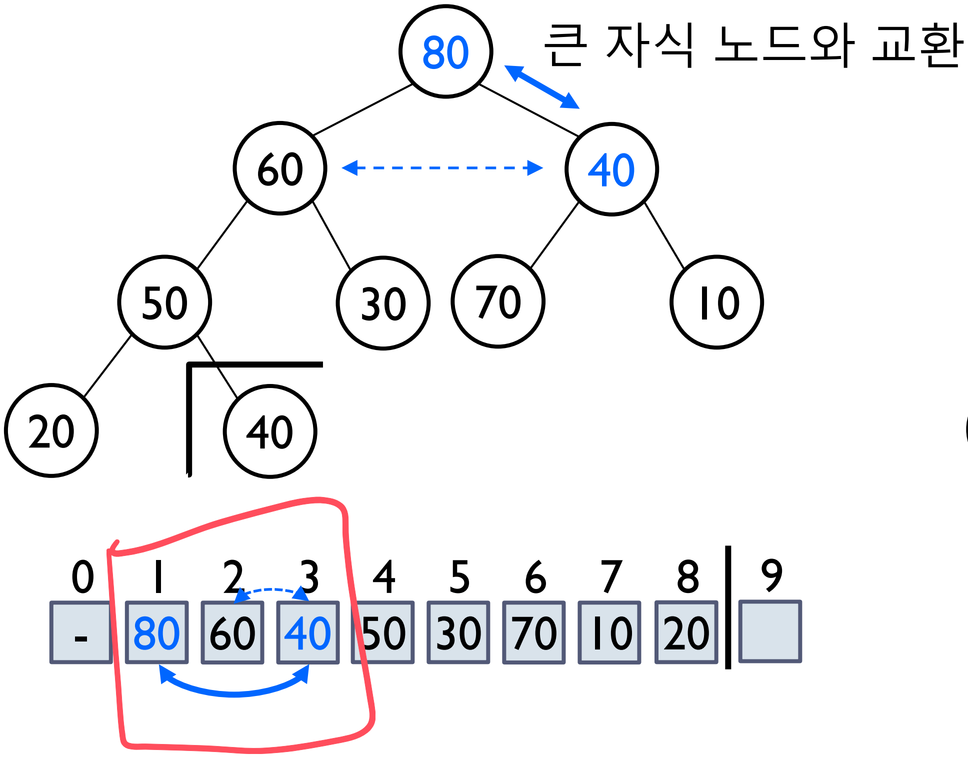
- 힙에서의 삭제 연산은 루트 노드를 제거하는 것을 뜻한다. 루트를 제거한 후에 힙의 마지막 노드를 루트로 옮긴다. 이때 힙 조건이 위배되므로 다음과 같이 노드들의 위치를 바꾼다.
  - 루트 값과 자식 노드들의 값들 중에서 큰 것을 비교하여 큰 자식 노드와 루트를 바꾼다.
  - 새로 자식 노드로 이동된 노드의 값은 다시 자식들의 값들 중에서 큰 것과 비교하여 힙 조건이 위배되면 큰 값을 가진 자식 노드와 자리를 바꾼다. 이와 같은 과정을 힙 조건이 만족될 때까지 반복한다.

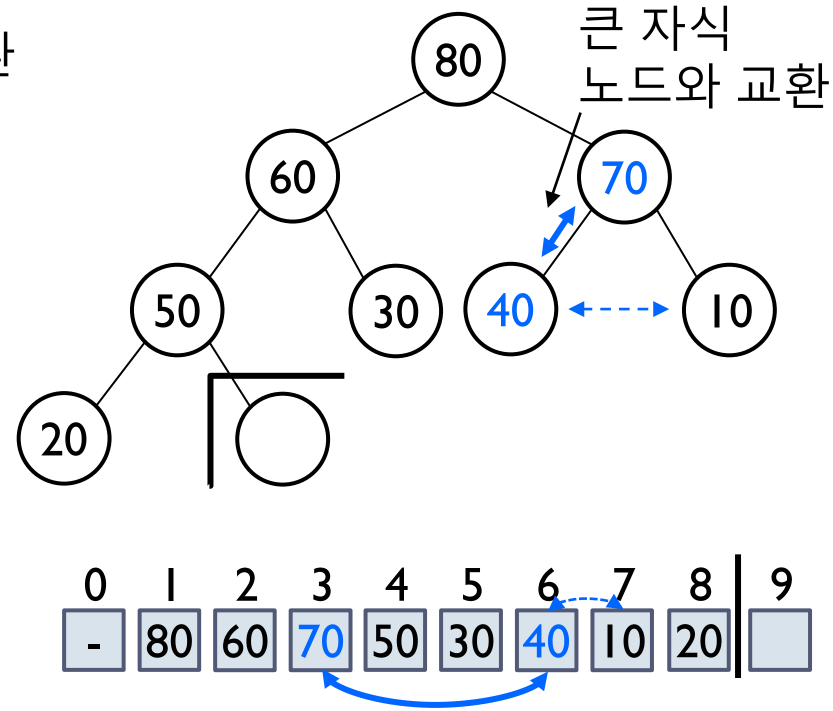
삭제연산 예제



- 루트의 90을 제거하고, 힙의 마지막 노드인 40을 루트로 옮긴다. 이때 힙의 크기(노드 수)를 1개 감소시킨다.

삭제연산 예제





- 루트로 이동한 40이 자식 노드(60과 80)보다 작으므로 자식 노드 중에서 큰 80과 루트의 40이 교환되었다.
- 40이 다시 자식 노드들(70과 10)과 비교되고, 이들 중 70이 40보다 크므로 70과 40을 서로 바꾼다.

삭제 연산: 시간 복잡도

- 삭제 연산은 최악의 경우 힙의 마지막 노드의 숫자가 힙의 이파리 노드에 저장될 때이므로 힙의 높이는 $\log_2 n$이기 때문에 O(logn) 시간이 걸린다.

비교하면서 내려옴.

O(logn)

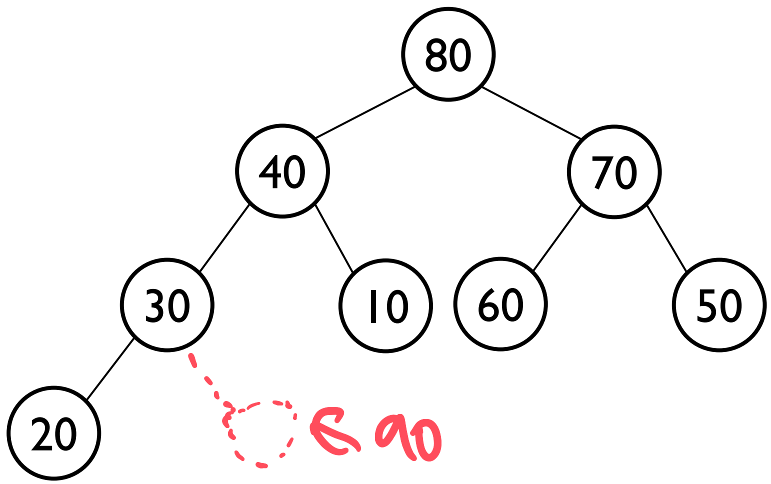
4 삽입 연산

- 힙에 새로운 값을 삽입하는 것은 다음과 같은 단계로 이루어진다.
  - 힙의 마지막 노드의 다음 노드에 새로운 값을 저장한다.
  - 힙 크기를 1 증가시킨다.
  - 새로운 값이 자신의 부모 노드의 값보다 작을 때까지 부모의 값과 서로 바꾼다. 새로운 값이 힙에 있는 모든 값들보다 크면, 새로운 값은 루트에 저장된다.

더크다면 올라가야함

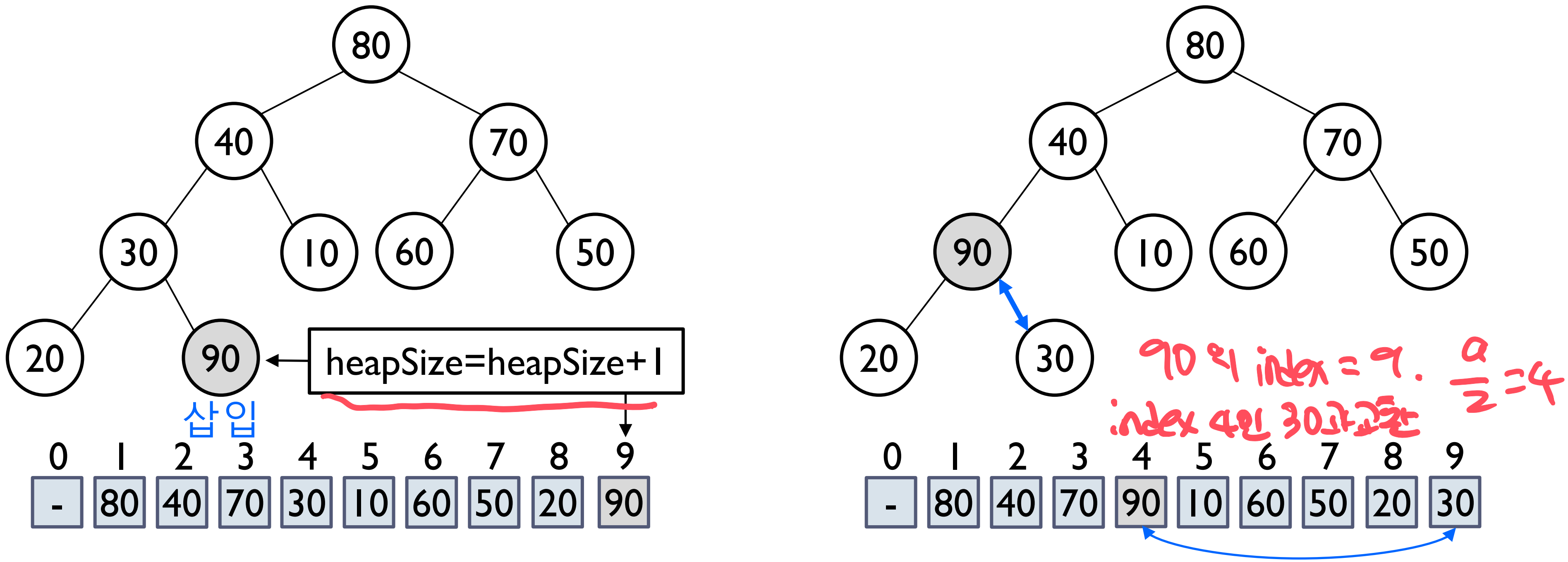
삽입 연산 예제

- 다음의 힙에 새로운 값 **90**을 삽입하는 과정을 살펴보자.



0	1	2	3	4	5	6	7	8
-	80	40	70	30	10	60	50	20

삽입 연산 예제

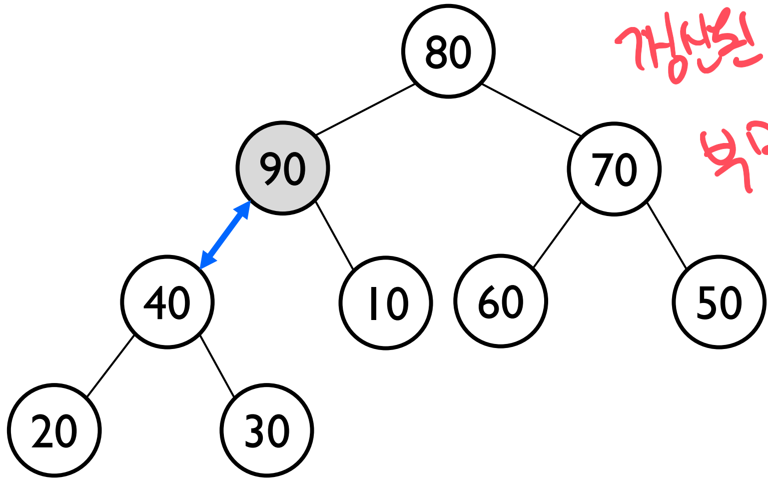


- 새로운 값 90은 힙의 마지막 노드의 다음 노드에 삽입되고 힙의 크기를 1 증가시킨다.
- 새로 삽입된 값 90과 부모 노드의 값 30을 비교해보면 최대힙의 조건이 위배되므로 이 두 값을 서로 교환한다. 여기서 배열을 살펴보면 90이 있는 배열의 인덱스가 9이므로, 부모 노드는 인덱스가 9/2=4인 원소이다. 따라서 배열에서 이 두 원소를 서로 바꾸면 된다.

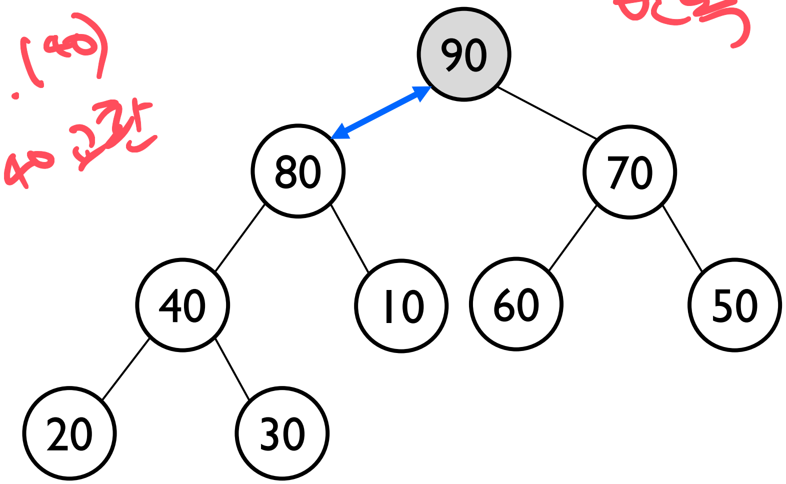
삽입 연산 예제

갱신된 90의 index : 4.
부모 = $\frac{4}{2}$ = 2 (40)
90 ↔ 40 교환

반복



0	1	2	3	4	5	6	7	8	9
-	80	90	70	40	10	60	50	20	30



0	1	2	3	4	5	6	7	8	9
-	90	80	70	40	10	60	50	20	30

- 앞과 같이 새로운 값 **90**은 계속해서 부모 노드와 교환되고 마지막에는 루트에 자리 잡는다.

삽입연산: 시간복잡도

- 삽입 연산은 최악의 경우 새로 삽입되는 값이 힙의 루트에 저장될 때이고, 힙의 높이는 $\log_2 n$이기 때문에 O(logn) 시간이 걸린다.

힙 만들때 O(n)

삭제, 삽입 O(logn)