2025 겨울방학 알고리즘 스터디

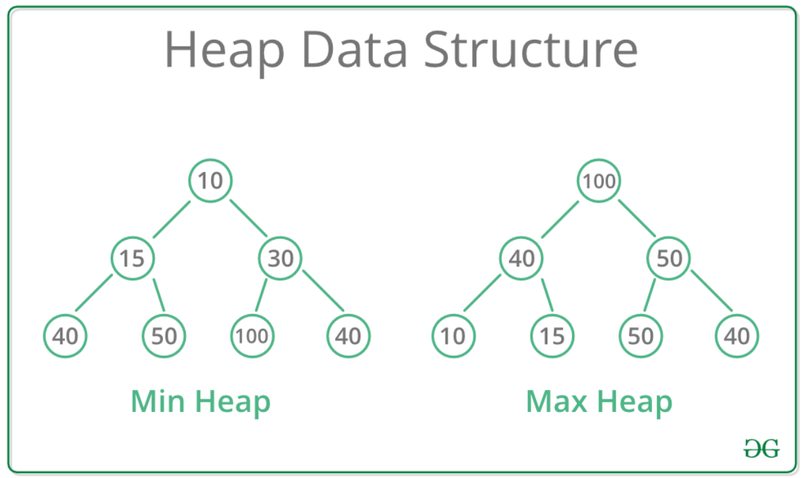
힙

컴퓨터 공학과 20230546 서보경

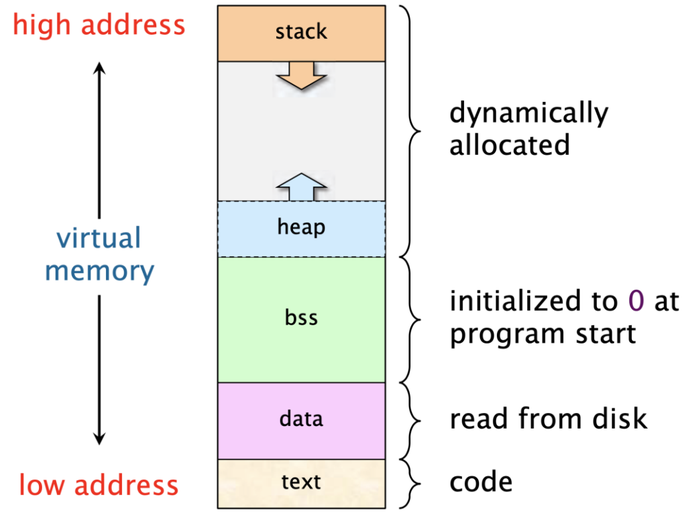
목차

힙이란?

힙(Heap)은 특정 순서에 따라 데이터를 정렬하여 저장하는 완전 이진 트리 기반의 자료구조이다. 힙의 가장 큰 특징은 최댓값(Max-Heap) 또는 최솟값(Min-Heap)을 빠르게 찾을 수 있다는 점이다. 최댓값이나 최솟값은 항상 트리의 맨 위(루트 노드)에 위치한다. 힙에서는 데이터를 삽입하거나 삭제할 때도 항상 이 규칙을 유지하며, 삽입과 삭제는 트리의 높이에 비례하는O(logN)의 시간 복잡도로 처리된다.



전형적인 힙의 모습

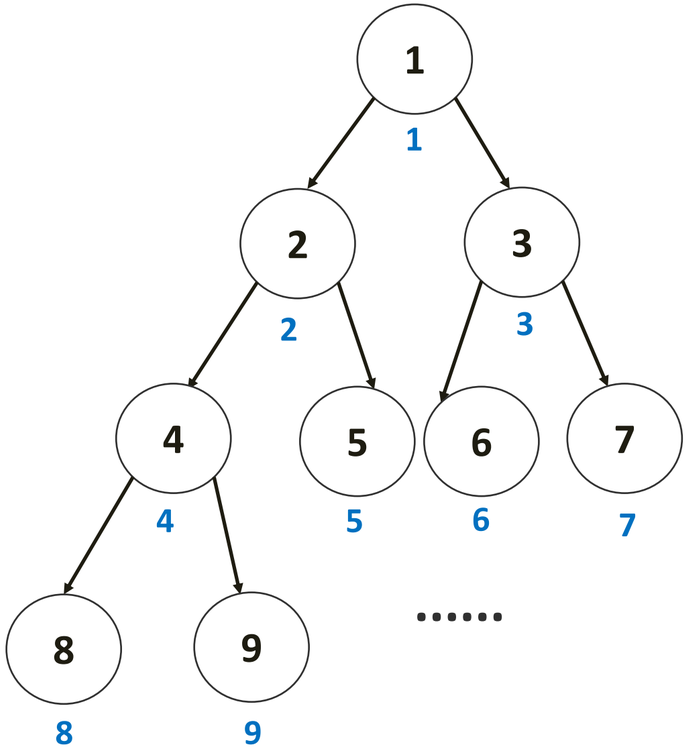


메모리 구조에서 주로 동적 데이터를 저장한다.

배열로 구현한 완전 이진 트리

1	2	3	4	5	6	7	8	9

배열(인덱스)



완전 이진 트리는 마지막 레벨을 제외한 모든 레벨이
완전이 채워진 형태의 이진 트리를 의미한다.

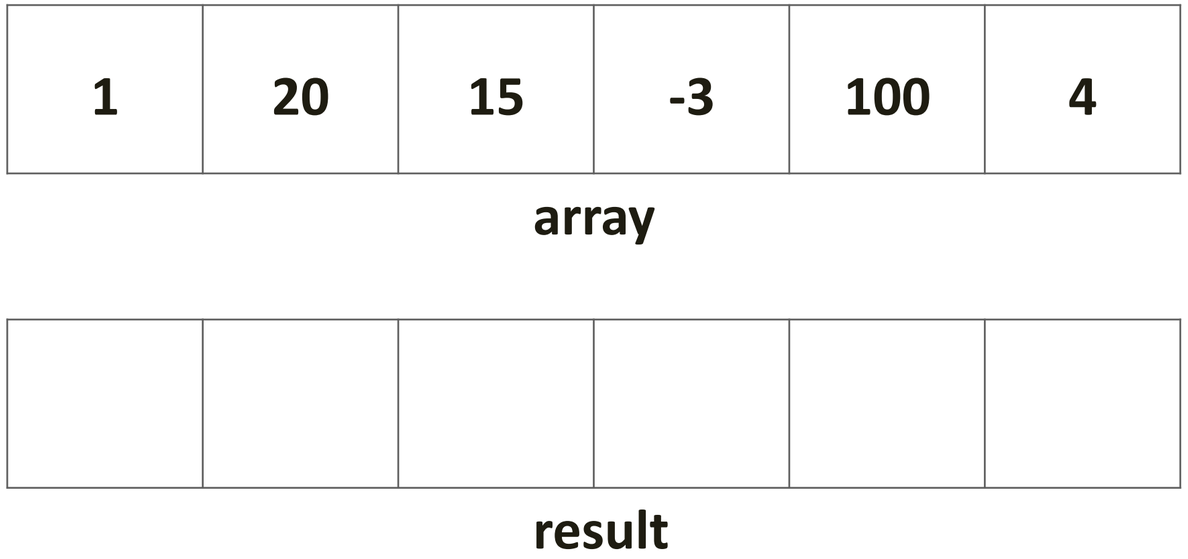
주로 왼쪽부터 차례대로 채워지며 배열로 구현된다.

현재 노드의 자식 : node * 2 (왼쪽) / node * 2 + 1 (오른쪽)
현재 노드의 부모 : node / 2 (정수 나눗셈)

최소힙이나 최대힙을 구현하기 위해선 이 구조가 필요하다!

힙의 삽입

size : 0



힙은 주로 완전 이진 트리로 구현하게 된다.

루트에는 제일 작은 원소가 오며(최소 힙) 나머지 원소들이 자리를 바꾸면서 힙 자체의 균형을 유지하는 구조이다.

부모가 자식보다 작거나 같아야 한다는 조건을 지켜야한다!

힙의 삽입

size : 1

1	20	15	-3	100	4

array

result

query : push 1

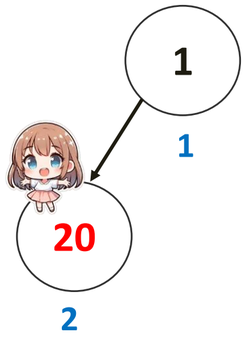
힙에 1을 넣는 연산이다.

현재 힙의 크기는 0이다. 크기를 1 늘려주고(size : 1) 그곳에 원소 1을 넣는다!

(편의를 위해, 힙 테이블의 시작은 1로 고정하겠음)

힙의 삽입

size : 2



	20	15	-3	100	4

array

result

query : push 20

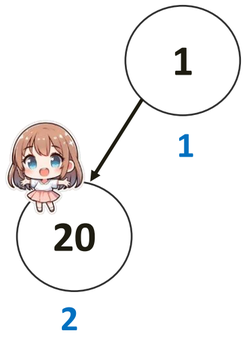
힙에 20을 넣는 연산이다.

현재 힙의 크기는 1이다. 크기를 1 늘려주고(size : 2) 그곳에 원소 20을 넣는다!

부모와 값을 비교하면서 위치를 바꿔야 할 수도 있으므로 확인 작업을 한다.

힙의 삽입

size : 2



	20	15	-3	100	4

array

result

node : (20, 2) | parent : (1, 1)

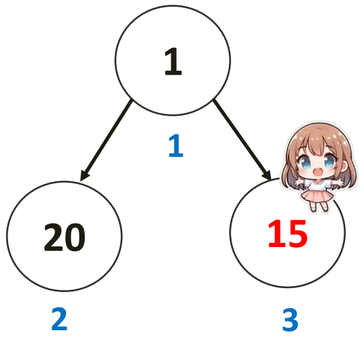
현재 노드의 위치는 2이다. 고로 부모의 위치는 1이다.

현재 노드의 값(20) 보다 부모의 값(1)이 더 작으므로 교체해줄 필요가 없다.

다음 원소 삽입으로 넘어간다.

힙의 삽입

size : 3



		15	-3	100	4

array

result

query : push 15

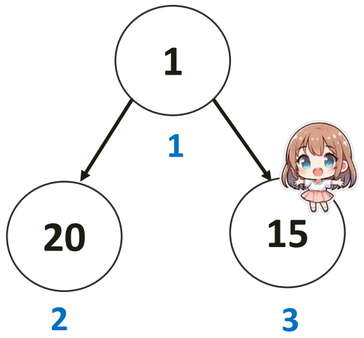
힙에 15을 넣는 연산이다.

현재 힙의 크기는 2이다. 크기를 1 늘려주고(size : 3) 그곳에 원소 15을 넣는다!

부모와 값을 비교하면서 위치를 바꿔야 할 수도 있으므로 확인 작업을 한다.

힙의 삽입

size : 3



		15	-3	100	4

array

result

node : (15, 3) | parent : (1,1)

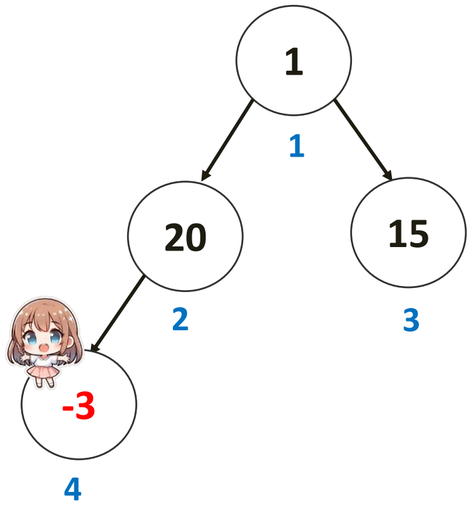
현재 노드의 위치는 3이다. 고로 부모의 위치는 1이다.

현재 노드의 값(15) 보다 부모의 값(1)이 더 작으므로 교체해줄 필요가 없다.

다음 원소 삽입으로 넘어간다.

힙의 삽입

size : 4



			-3	100	4

array

result

query : push -3

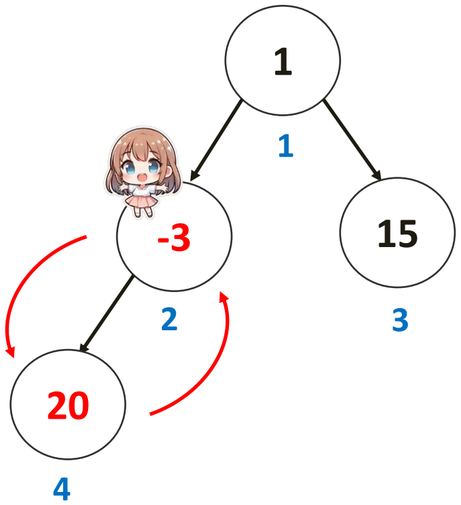
힙에 -3을 넣는 연산이다.

현재 힙의 크기는 3이다. 크기를 1 늘려주고(size : 4) 그곳에 원소 -3을 넣는다!

부모와 값을 비교하면서 위치를 바꿔야 할 수도 있으므로 확인 작업을 한다.

힙의 삽입

size : 4



			-3	100	4

array

result

node : (-3, 4) | parent : (20, 2)

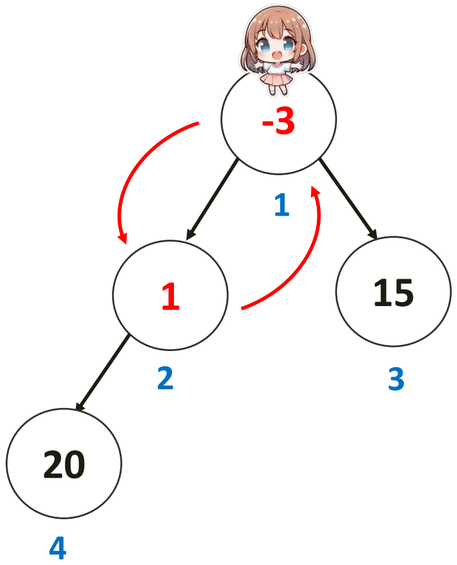
현재 노드의 위치는 4이다. 고로 부모의 위치는 2이다.

현재 노드의 값(-3) 보다 부모의 값(20)이 더 크다!

고로 20과 -3을 교체한다. 바뀐 위치에서도 부모의 값을 확인하자.

힙의 삽입

size : 4



			-3	100	4

array

result

node : (-3, 2) | parent : (1, 1)

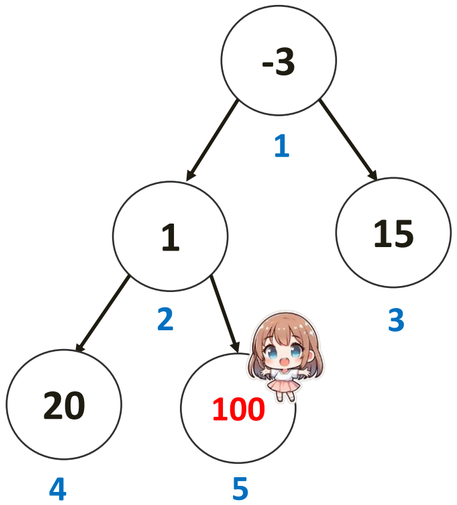
현재 노드의 위치는 2이다. 고로 부모의 위치는 1이다.

현재 노드의 값(-3) 보다 부모의 값(1)이 더 크다!

고로 -3과 1을 교체한다. 루트까지 올라갔으니 교환 작업이 끝났다.

힙의 삽입

size : 5



				100	4

array

result

query : push 100

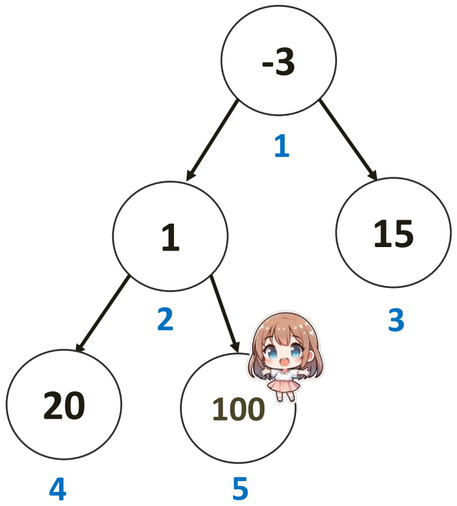
힙에 100을 넣는 연산이다.

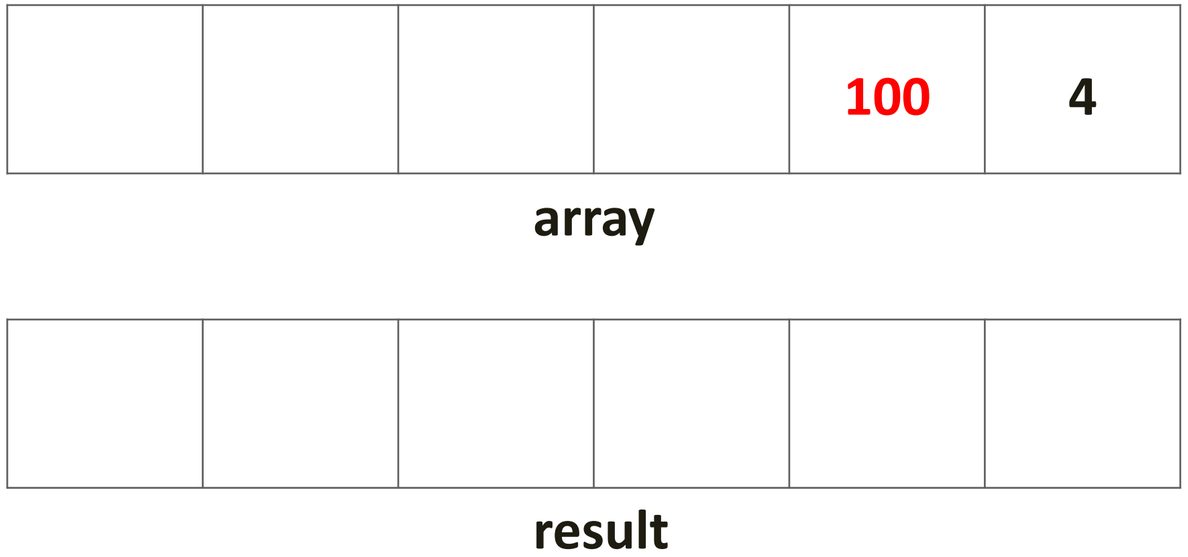
현재 힙의 크기는 4이다. 크기를 1 늘려주고(size : 5) 그곳에 원소 100을 넣는다!

부모와 값을 비교하면서 위치를 바꿔야 할 수도 있으므로 확인 작업을 한다.

힙의 삽입

size : 5





node : (100 , 5) | parent : (1, 2)

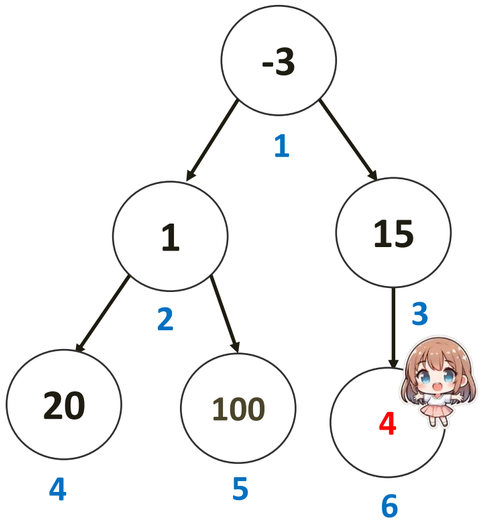
현재 노드의 위치는 5이다. 고로 부모의 위치는 2이다.

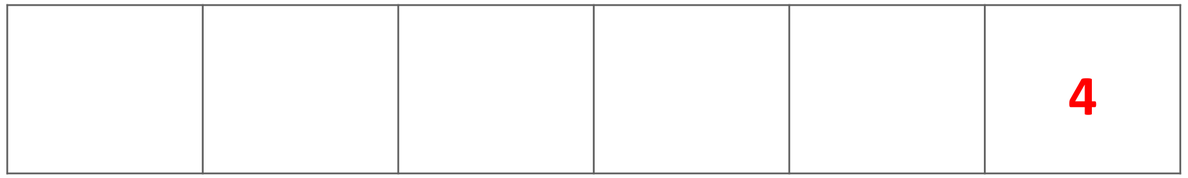
현재 노드의 값(100) 보다 부모의 값(1)이 더 작으므로 교체해줄 필요가 없다.

다음 원소 삽입으로 넘어간다.

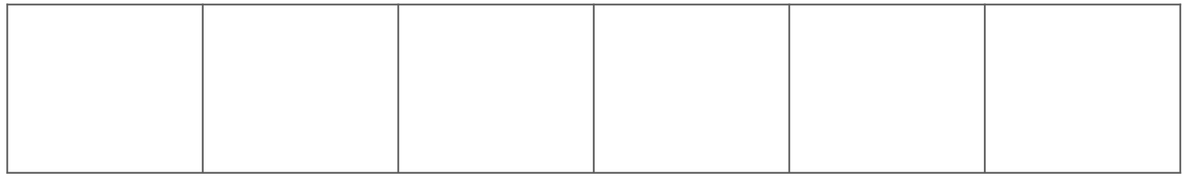
힙의 삽입

size : 6





array



result

query : push 4

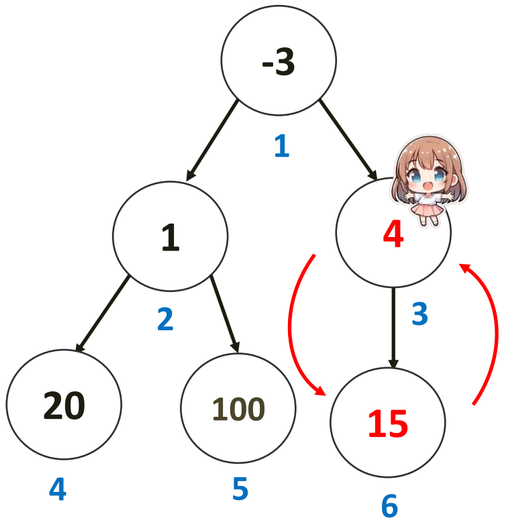
힙에 4을 넣는 연산이다.

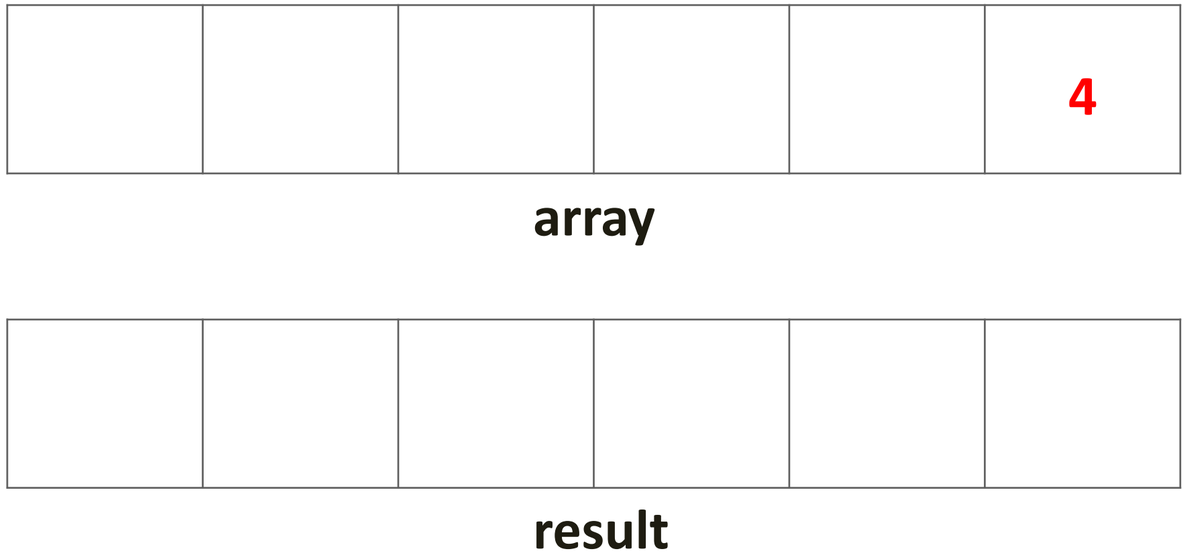
현재 힙의 크기는 5이다. 크기를 1 늘려주고(size : 6) 그곳에 원소 4을 넣는다!

부모와 값을 비교하면서 위치를 바꿔야 할 수도 있으므로 확인 작업을 한다.

힙의 삽입

size : 6





node : (4, 6) | parent : (15, 3)

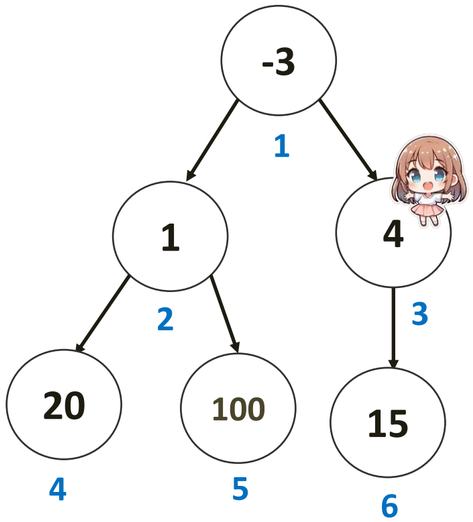
현재 노드의 위치는 6이다. 고로 부모의 위치는 3이다.

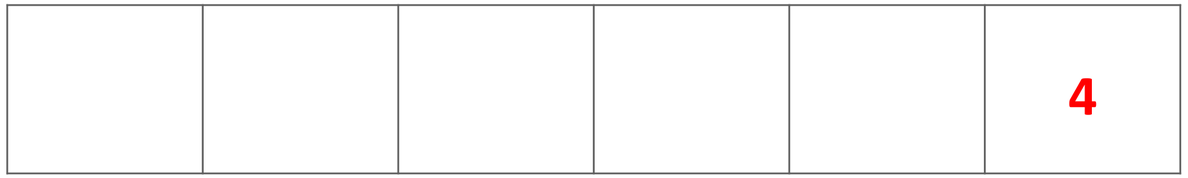
현재 노드의 값(4) 보다 부모의 값(15)이 더 크다!

고로 4과 15을 교체한다. 바뀐 위치에서도 부모의 값을 확인하자.

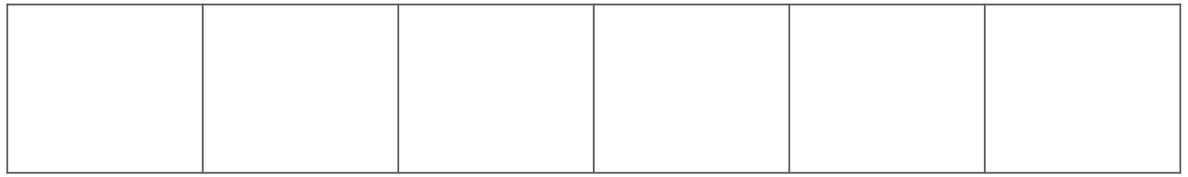
힙의 삽입

size : 6





array



result

node : (4, 3) | parent : (-3, 1)

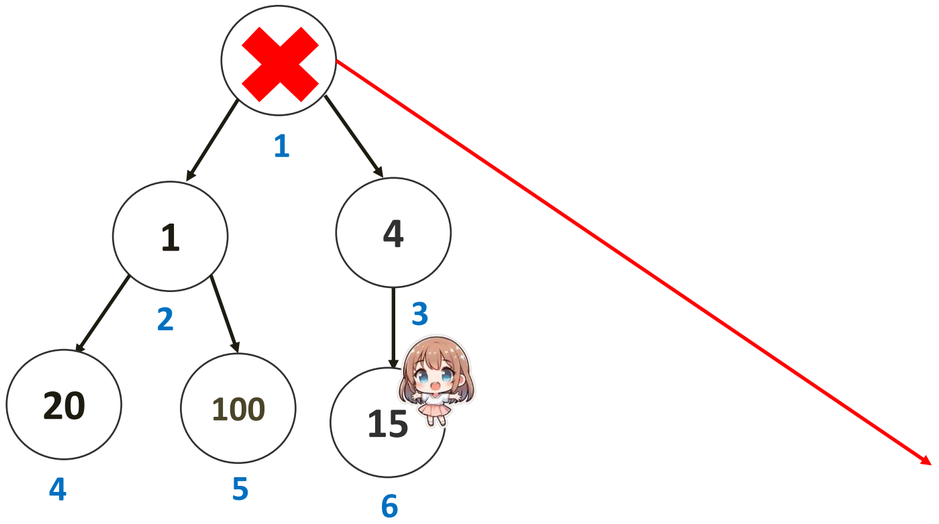
현재 노드의 위치는 3이다. 고로 부모의 위치는 1이다.

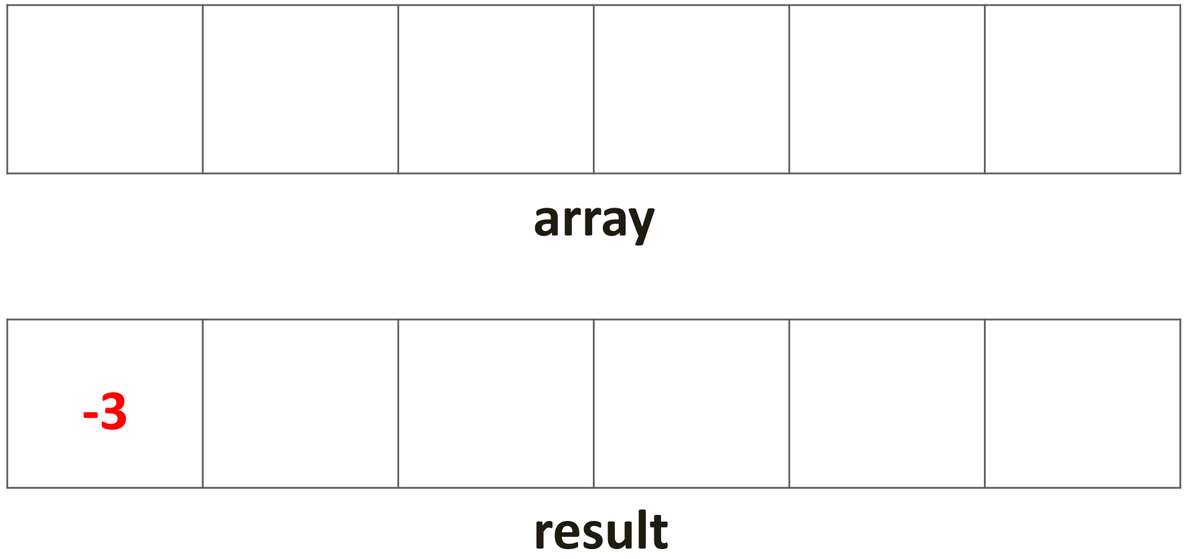
현재 노드의 값(4) 보다 부모의 값(-3)이 더 작으므로 교체해줄 필요가 없다.

원소 삽입 작업이 모두 끝났다.

힙의 삭제

size : 5





query : pop

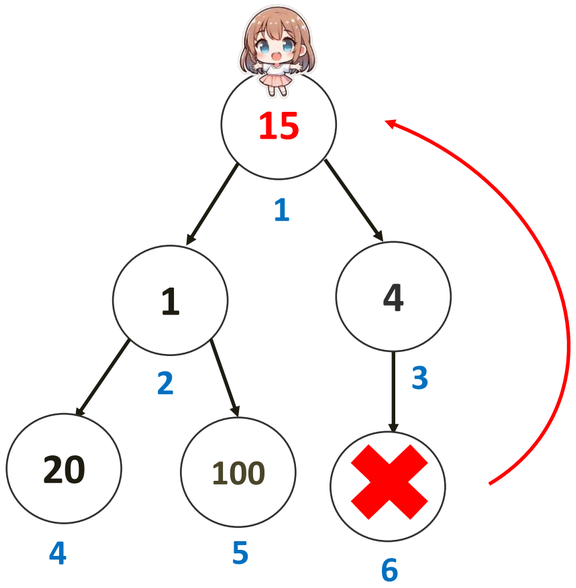
특정 우선순위 기반으로 형성된 힙에서 루트를 제거하는 연산이다.

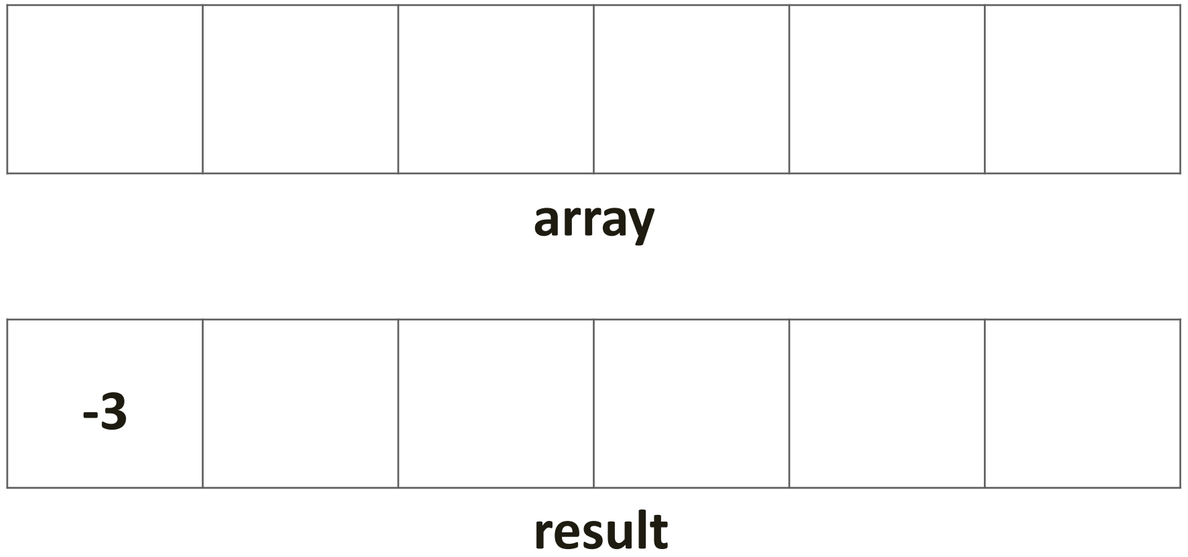
루트인 -3을 제거하고 result 배열에 넣는다.

현재 루트가 비었으니, size를 하나 줄인다. 이 빈 곳은 어떻게 채워야 할까?

힙의 삭제

size : 5





query : pop

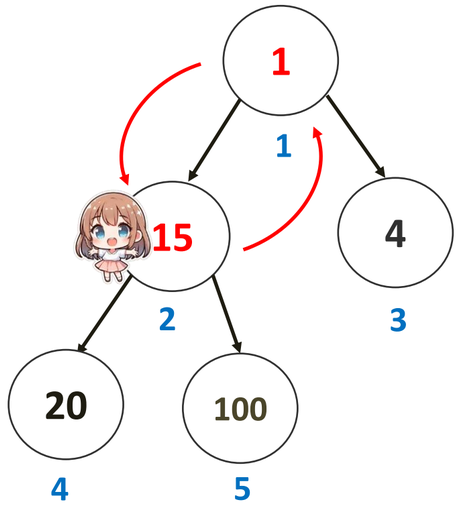
제일 마지막 위치에 있던 원소 (size; 바뀌기 전에는 6)를 루트로 바꿔주면 된다.

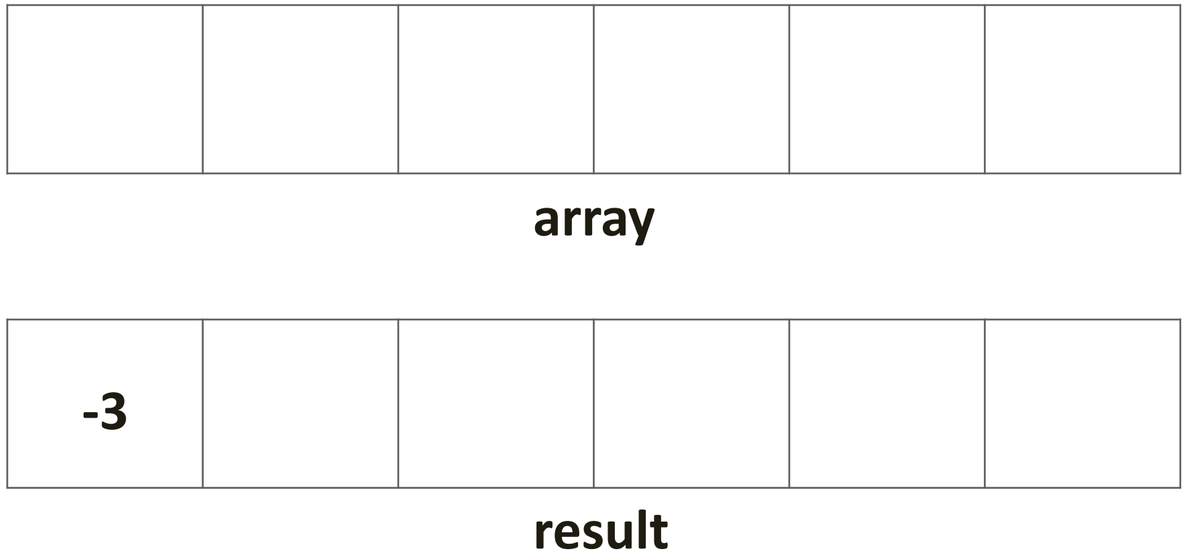
그러나 루트가 제일 작은 값을 유지 해야 하는 특성상 이대로 유지하면 안된다.

고로, 현재 위치에서 왼쪽 자식과 오른쪽 자식을 비교해서 더 작은놈으로 원소를 교환해주자.

힙의 삭제

size : 5





node : (15,1) | left : (1,2) | right : (4,3)

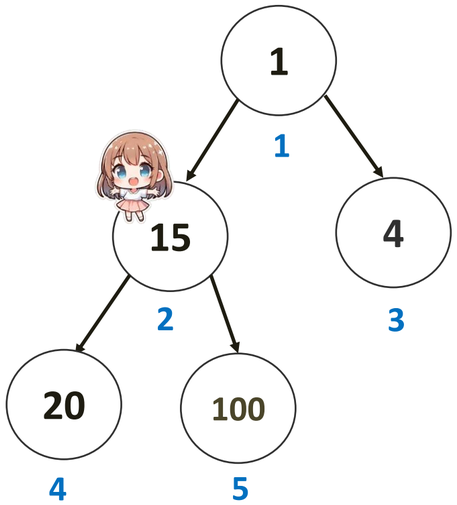
현재 노드의 위치는 1이다. 고로 왼쪽 자식의 위치는 2, 오른쪽 자식의 위치는 3이다.

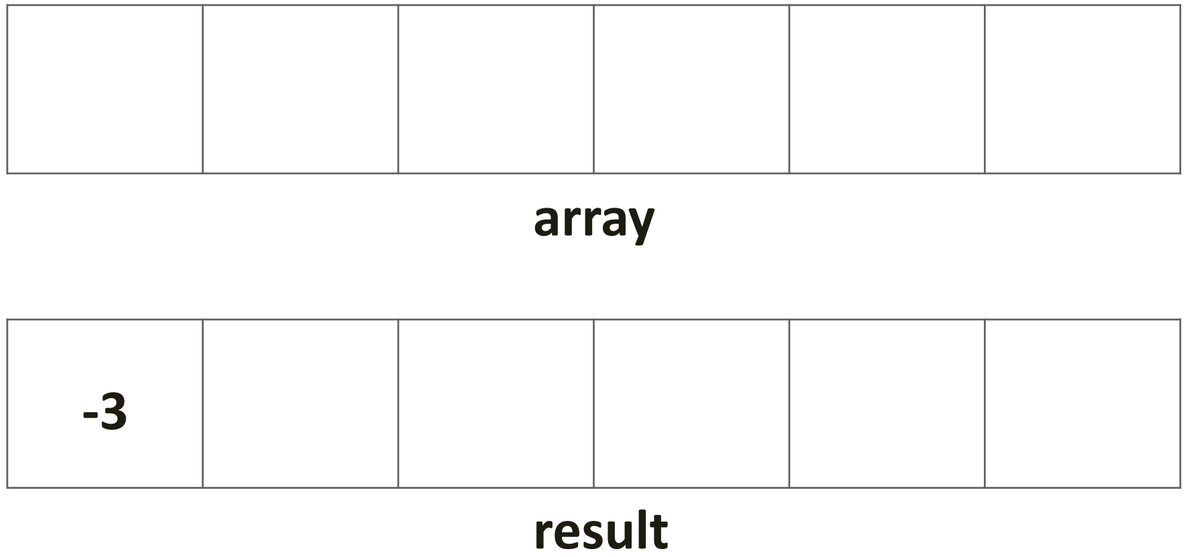
두 값 모두 부모(현재 노드)보다 작지만 왼쪽 자식이 더 작으므로 (1 vs 4)

현재 노드의 값을 왼쪽 자식의 값과 바꿔준다.

힙의 삭제

size : 5





node : (15,2) | left : (20,4) | right : (100,5)

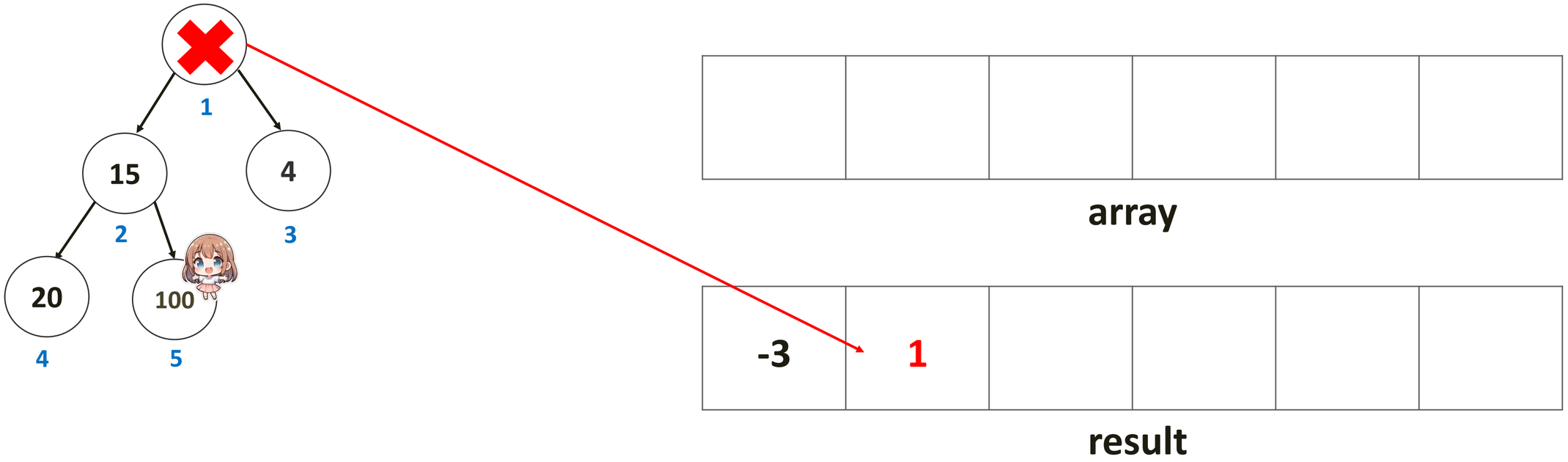
현재 노드의 위치는 2이다. 고로 왼쪽 자식의 위치는 4, 오른쪽 자식의 위치는 5이다.

두 값 모두 부모(현재 노드)보다 크다. 고로 바꿀 대상이 없다.

다음 삭제 작업으로 넘어간다.

힙의 삭제

size : 4



query : pop

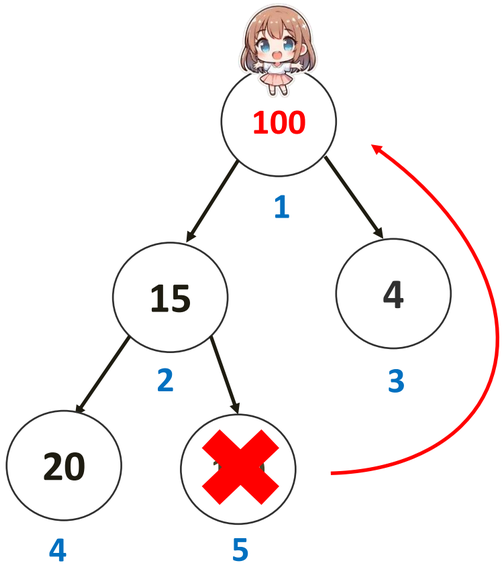
특정 우선순위 기반으로 형성된 힙에서 루트를 제거하는 연산이다.

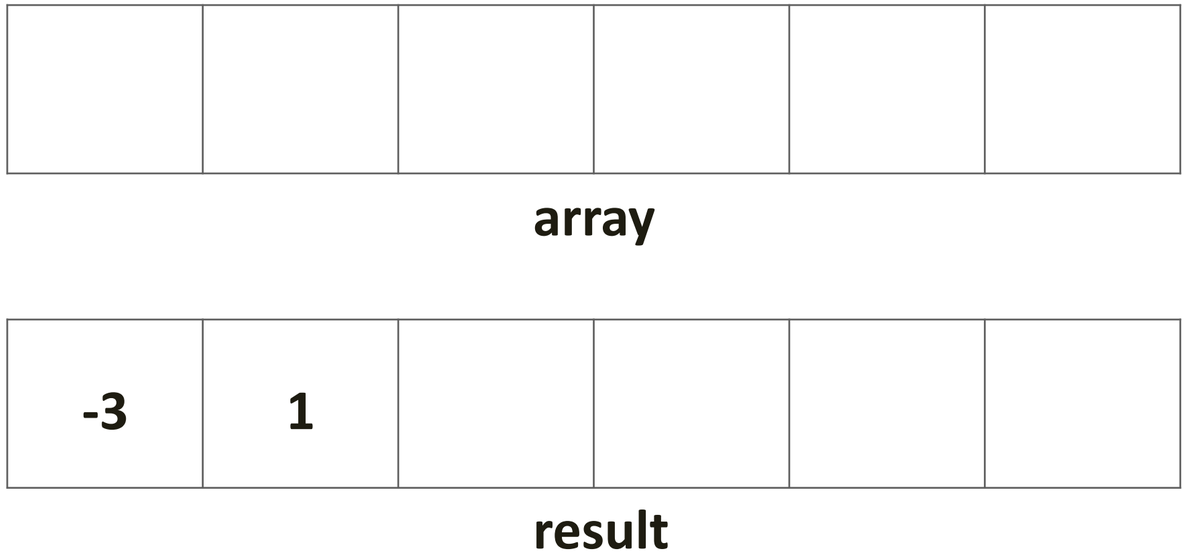
루트인 1을 제거하고 result 배열에 넣는다.

현재 루트가 비었으니, size를 하나 줄인다.

힙의 삭제

size : 4





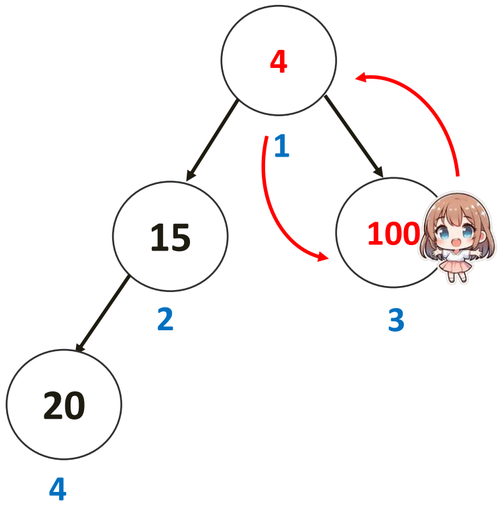
query : pop

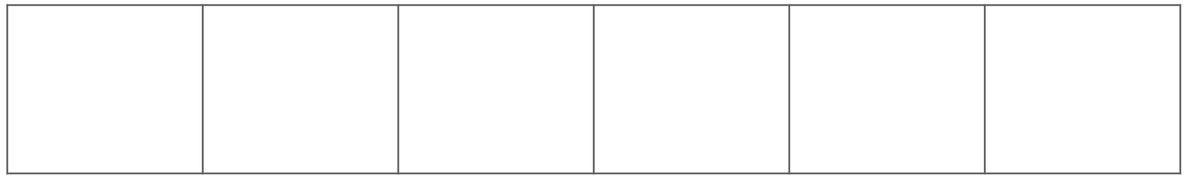
제일 마지막 위치에 있던 원소인 100을 루트로 바꿔준다.

고로, 현재 위치에서 왼쪽 자식과 오른쪽 자식을 비교해서 더 작은놈으로 원소를 교환해주자.

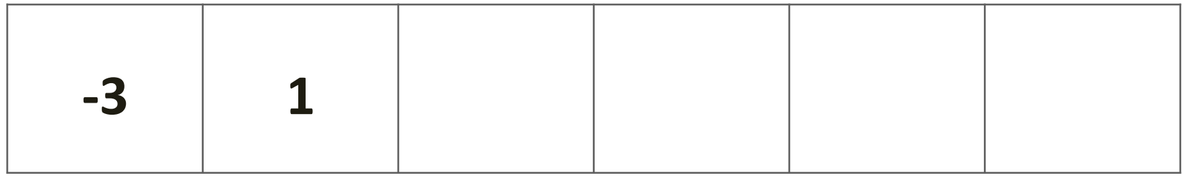
힙의 삭제

size : 4





array



result

node : (100,1) | left : (15,2) | right : (4,3)

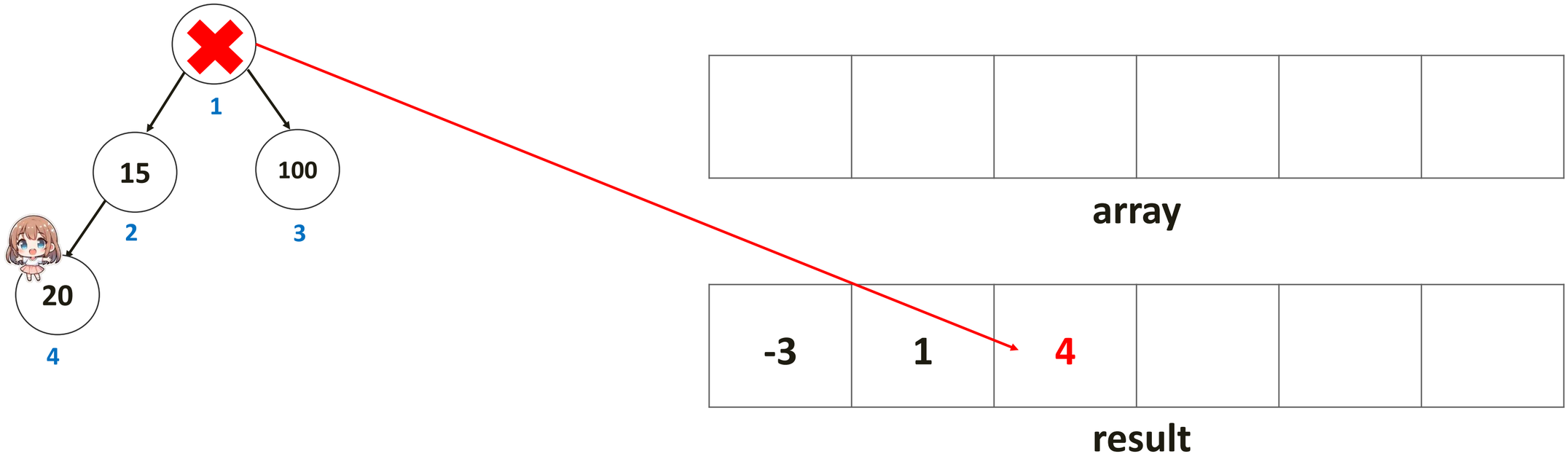
현재 노드의 위치는 1이다. 고로 왼쪽 자식의 위치는 2, 오른쪽 자식의 위치는 3이다.

두 값 모두 부모(현재 노드)보다 작지만 오른쪽 자식이 더 작으므로 (15 vs 4)

현재 노드의 값을 오른쪽 자식의 값과 바꿔준다. 이동한 위치에서 자식이 없으므로 교환작업 끝.

힙의 삭제

size : 3



query : pop

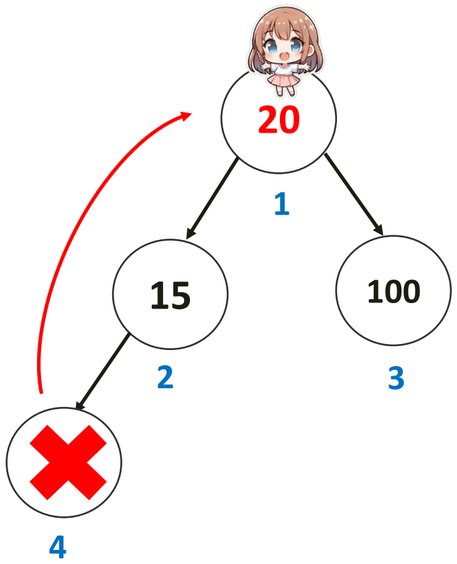
특정 우선순위 기반으로 형성된 힙에서 루트를 제거하는 연산이다.

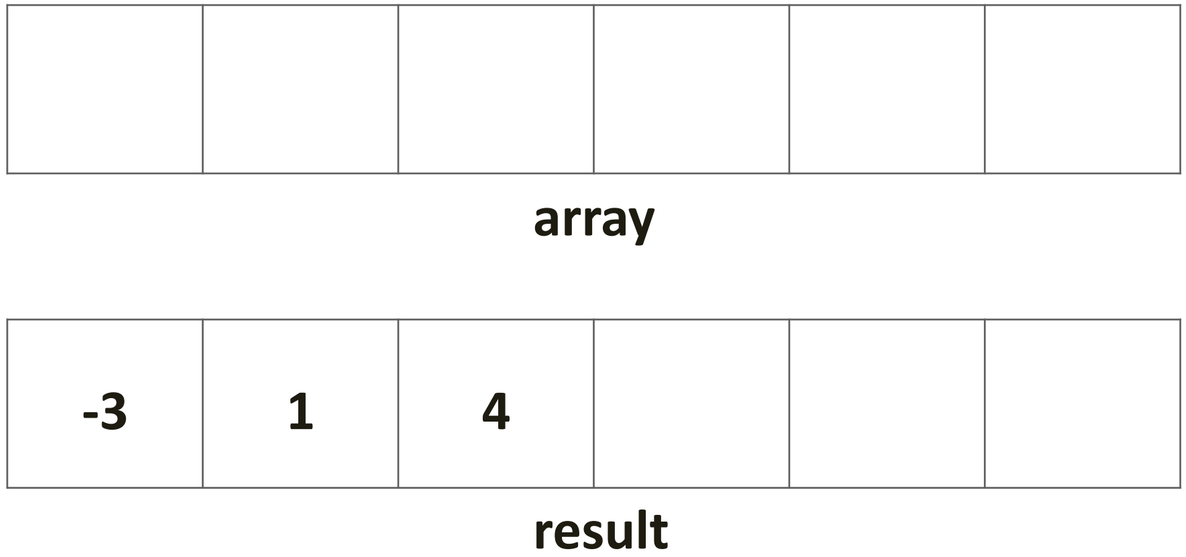
루트인 4을 제거하고 result 배열에 넣는다.

현재 루트가 비었으니, size를 하나 줄인다.

힙의 삭제

size : 3





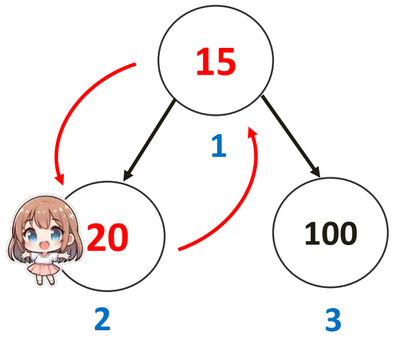
query : pop

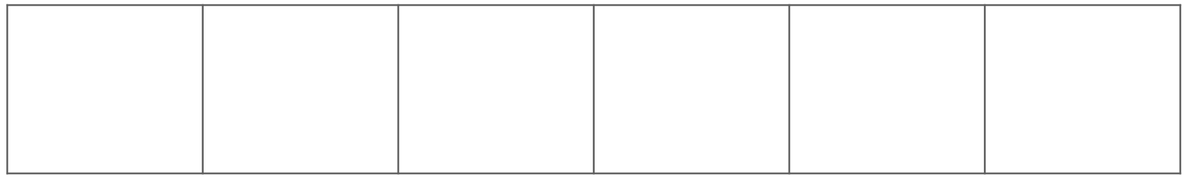
제일 마지막 위치에 있던 원소인 20을 루트로 바꿔준다.

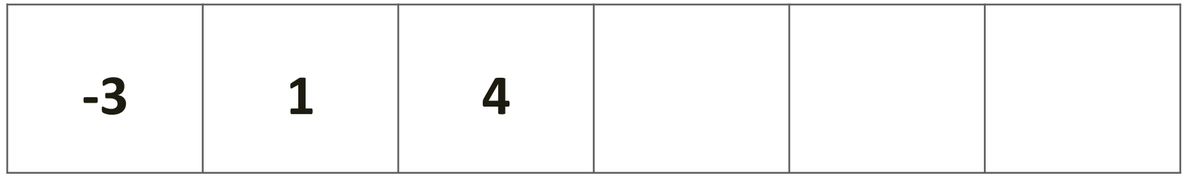
고로, 현재 위치에서 왼쪽 자식과 오른쪽 자식을 비교해서 더 작은놈으로 원소를 교환해주자.

힙의 삭제

size : 3







node : (20,1) | left : (15,2) | right : (100,3)

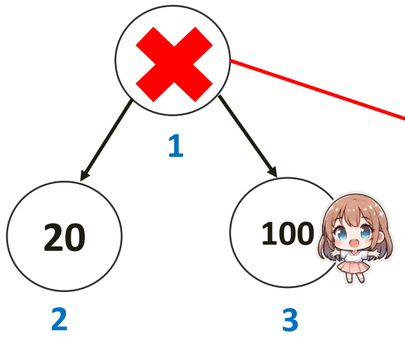
현재 노드의 위치는 1이다. 고로 왼쪽 자식의 위치는 2, 오른쪽 자식의 위치는 3이다.

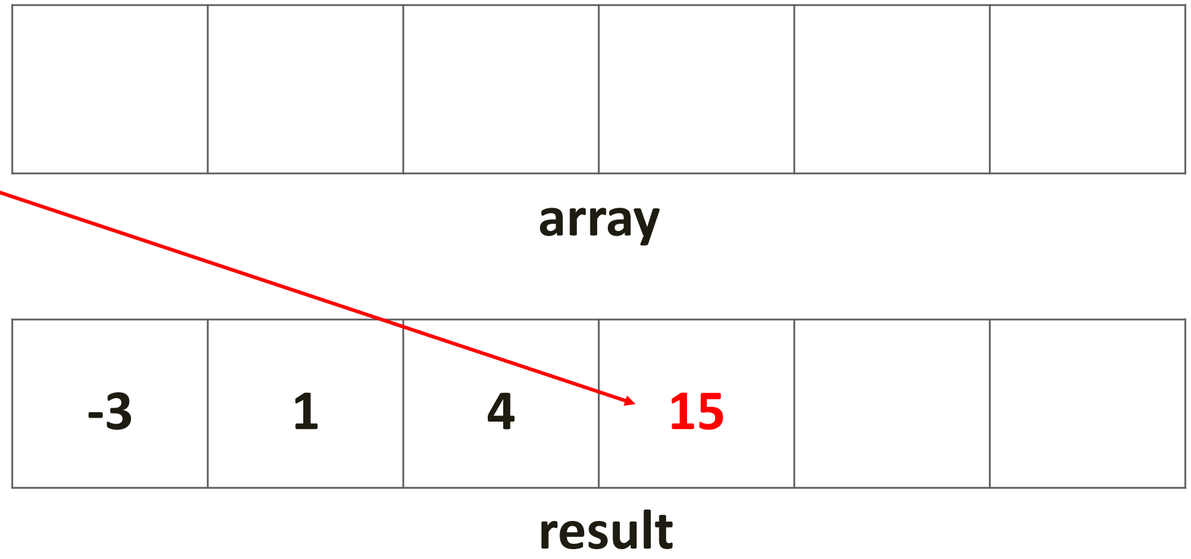
왼쪽 값만이 현재 노드의 값보다 작다 (15 vs20)

현재 노드의 값을 왼쪽 자식의 값과 바꿔준다. 이동한 위치에서 자식이 없으므로 교환작업 끝.

힙의 삭제

size : 2





query : pop

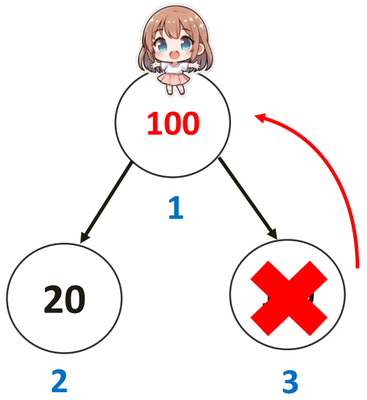
특정 우선순위 기반으로 형성된 힙에서 루트를 제거하는 연산이다.

루트인 15을 제거하고 result 배열에 넣는다.

현재 루트가 비었으니, size를 하나 줄인다.

힙의 삭제

size : 2



array

-3	1	4	15		

result

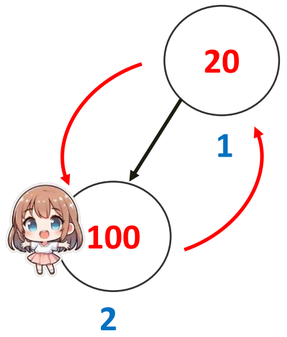
query : pop

제일 마지막 위치에 있던 원소인 20을 루트로 바꿔준다.

고로, 현재 위치에서 왼쪽 자식과 비교해서 더 작은놈으로 원소를 교환해주자.

힙의 삭제

size : 2



array

-3	1	4	15		

result

node : (20,100) | left : (20,2) | right : NONE

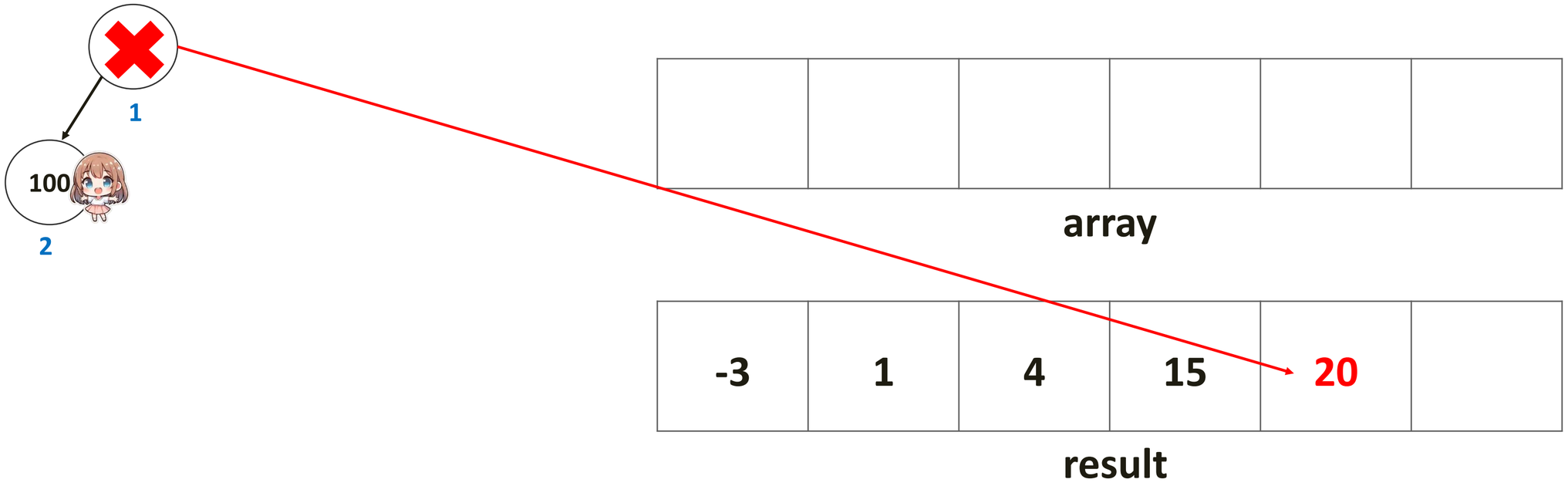
현재 노드의 위치는 1이다. 고로 왼쪽 자식의 위치는 2, 오른쪽 자식은 없다.

왼쪽 이 현재 노드의 값보다 작다 (20 vs 100)

현재 노드의 값을 왼쪽 자식의 값과 바꿔준다. 이동한 위치에서 자식이 없으므로 교환작업 끝.

힙의 삭제

size : 1



query : pop

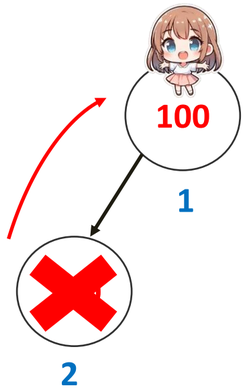
특정 우선순위 기반으로 형성된 힙에서 루트를 제거하는 연산이다.

루트인 20을 제거하고 result 배열에 넣는다.

현재 루트가 비었으니, size를 하나 줄인다.

힙의 삭제

size : 1



array

-3	1	4	15	20	

result

query : pop

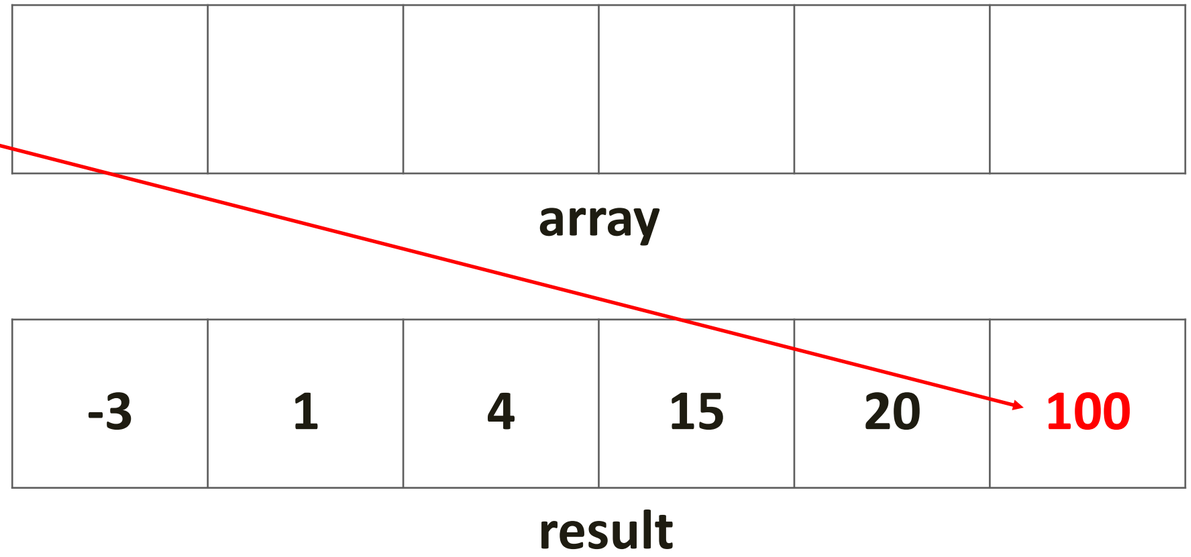
제일 마지막 위치에 있던 원소인 20을 루트로 바꿔준다.

고로, 현재 위치는 아무 자식을 가지지 않으므로 아무것도 하지않는다.

힙의 삭제

size : 0

1



query : pop

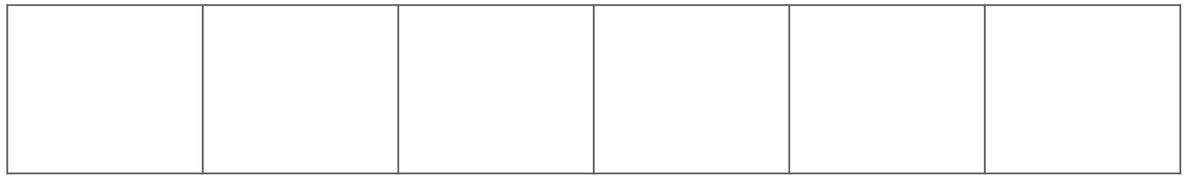
특정 우선순위 기반으로 형성된 힙에서 루트를 제거하는 연산이다.

루트인 100을 제거하고 result 배열에 넣는다.

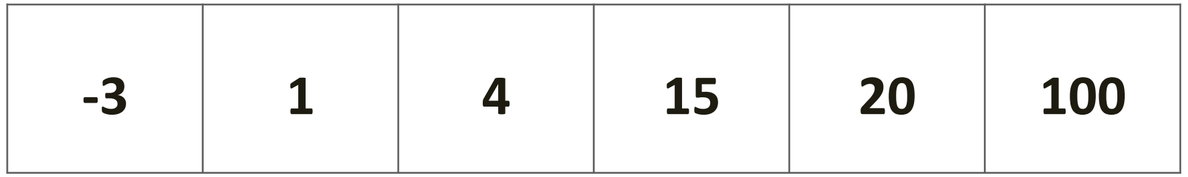
현재 루트가 비었으니, size를 하나 줄인다. 힙이 비었으므로 모든 원소를 삭제하였다.

힙의 삭제

size : 0



array



result

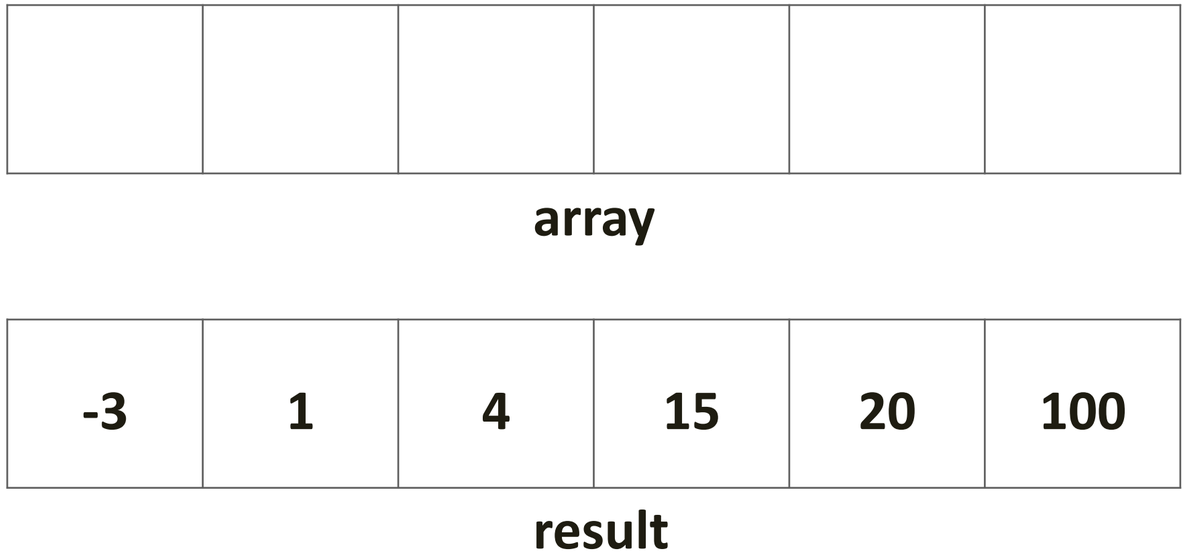
query : pop

특정 우선순위 기반으로 형성된 힙에서 루트를 제거하는 연산이다.

루트인 100을 제거하고 result 배열에 넣는다.

현재 루트가 비었으니, size를 하나 줄인다. 힙이 비었으므로 모든 원소를 삭제하였다.

힙의 삭제



지금까지 최소 힙의 삽입과 삭제를 보았다.

삽입 할 땐 제일 마지막에 넣은 다음 부모와 비교, 삭제할 땐 제일 마지막을 루트에 넣고 자식과 비교한다.
최대 힙도 논리는 같다. 삽입시 부모가 작다면 바꾸고, 삭제시 자식이 크면 바꿔주는 식이다.

힙의 활용 – 중앙값 관리

백준이는 동생에게 "가운데를 말해요" 게임을 가르쳐주고 있다. 백준이가 정수를 하나씩 외칠때마다 동생은 지금까지 백준이가 말한 수 중에서 중간값을 말해야 한다. 만약, 그동안 백준이가 외친 수의 개수가 짝수개라면 중간에 있는 두 수 중에서 작은 수를 말해야 한다.

예를 들어 백준이가 동생에게 1, 5, 2, 10, -99, 7, 5를 순서대로 외쳤다고 하면, 동생은 1, 1, 2, 2, 2, 2, 5를 차례대로 말해야 한다. 백준이가 외치는 수가 주어졌을 때, 동생이 말해야 하는 수를 구하는 프로그램을 작성하시오.

입력

첫째 줄에는 백준이가 외치는 정수의 개수 N이 주어진다. N은 1보다 크거나 같고, 100,000보다 작거나 같은 자연수이다. 그 다음 N줄에 걸쳐서 백준이가 외치는 정수가 차례대로 주어진다. 정수는 -10,000보다 크거나 같고, 10,000보다 작거나 같다.

출력

한 줄에 하나씩 N줄에 걸쳐 백준이의 동생이 말해야 하는 수를 순서대로 출력한다.

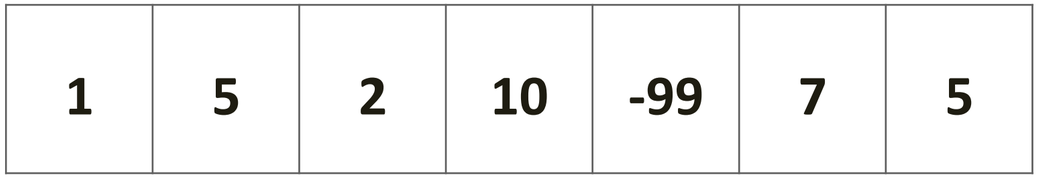
10만개의 정수를 받으면서 일일히 정렬시키면서 중앙값을 찾을까?
근데 시간초과 날거같은데..
어떻게 해야하지?

➔ **최소 힙과 최대 힙으로 구현을 하면 되겠구나! 최대 힙의 루트 원소는 최소 힙의 루트 원소보다 작거나 같게 유지해주는 식으로 하면 되지 않을까?**

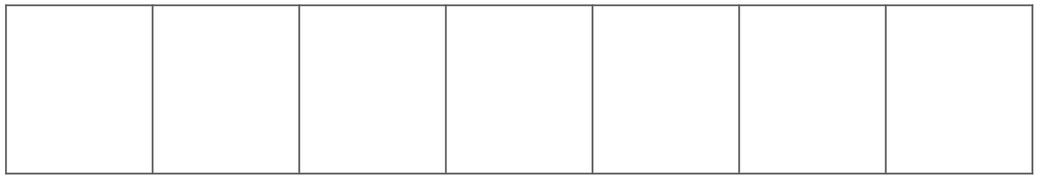
힙의 활용 – 중앙값 관리

max | size : 0

min | size : 0



array



result

최대 힙과 최소 힙을 사용 하기에 앞서, 이 테크닉을 어떻게 쓰는지 간략하게 설명하겠다.

먼저, 어떤 원소가 들어오면 최대 힙에 먼저 push 한다.

C#1 : 만약 최대 힙과 최소 힙의 크기 차이가 2 이상인 경우에는 크기가 더 많은 쪽의 루트를 pop 하여 다른 힙 쪽에 push시킨다. (보통 최대 힙이 원소 개수가 더 많음)

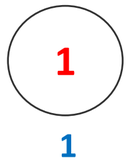
C#2 : 크기 차이가 1이하일 때, 최대 힙의 루트와 최소 힙의 루트를 비교한다. 만약 최대 힙의 루트가 최소 힙의 루트보다 크기가 크다면 두 값을 swap 한다. 이 때, 최대 힙의 원소가 바로 중앙값이다.

중앙값 기준으로 했을 때, 최대 힙에는 작거나 같은 값을 최소 힙에는 큰 값으로 유지된다.

힙의 활용 – 중앙값 관리

max | size : 1

min | size : 0



1	5	2	10

array

1			

result

query : 1

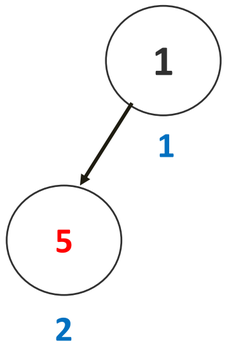
현재 원소는 1이다. 먼저 최대 힙에 넣어준다.

현재 최대 힙과 최소 힙의 크기 차이는 1이다. 고로 최대 힙에 있는 원소가 중앙값이다.

result 배열에 1을 기록한다.

힙의 활용 – 중앙값 관리

max | size : 2　　**min | size : 0**



1	5	2	10

array

1			

result

query : 5

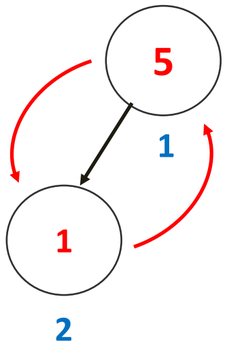
현재 원소는 5이다. 먼저 최대 힙에 넣어준다.

최대 힙에 넣었으니 우선순위에 맞춰 정렬을 해야한다.

힙의 활용 – 중앙값 관리

max | size : 2

min | size : 0



1	5	2	10

array

1			

result

query : 5

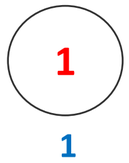
현재 노드의 위치는 2이다. 고로 부모의 위치는 1이다.

부모의 값(1)이 현재 노드(5)보다 작으므로 최대 힙에 맞지 않는다.

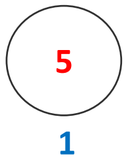
두 값을 교환 해준다. 루트까지 올라갔으므로 교환 작업이 끝났다.

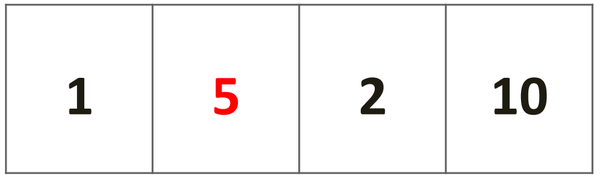
힙의 활용 – 중앙값 관리

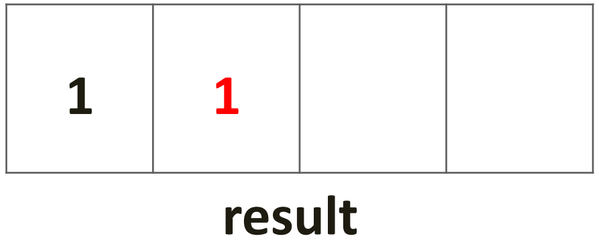
max | size : 1



min | size : 1







query : 5

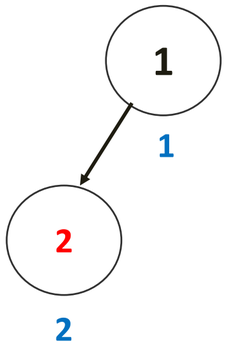
최소 힙과 최대 힙의 크기 차이가 2이므로 최대 힙에서 루트를 pop 하여 최소 힙에 push 한다.

힙의 균형이 맞추어 졌다.

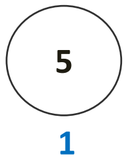
최대 힙의 루트가 항상 중앙값 이므로 루트인 1을 result 배열에 기록한다.

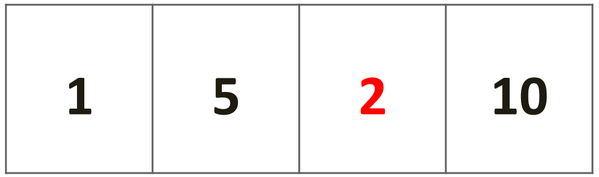
힙의 활용 – 중앙값 관리

max | size : 2

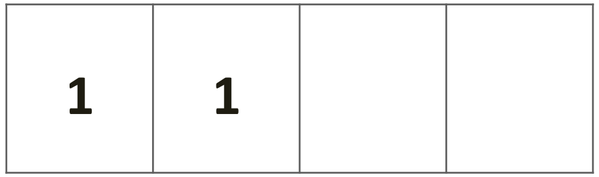


min | size : 1





array



result

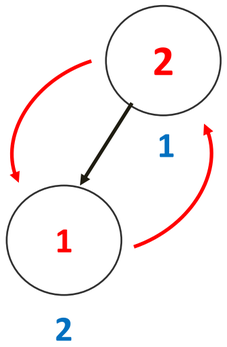
query : 2

현재 원소는 2이다. 먼저 최대 힙에 넣어준다.

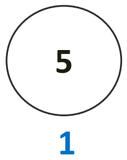
최대 힙에 넣었으니 우선순위에 맞춰 정렬을 해야한다.

힙의 활용 – 중앙값 관리

max | size : 2



min | size : 1



1	5	2	10

array

1	1		

result

query : 2

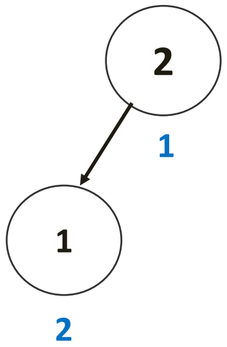
현재 노드의 위치는 2이다. 고로 부모의 위치는 1이다.

부모의 값(1)이 현재 노드(2)보다 작으므로 최대 힙에 맞지 않는다.

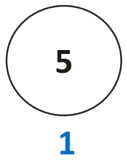
두 값을 교환 해준다. 루트까지 올라갔으므로 교환 작업이 끝났다.

힙의 활용 – 중앙값 관리

max | size : 2



min | size : 1



1	5	2	10

array

1	1	2	

result

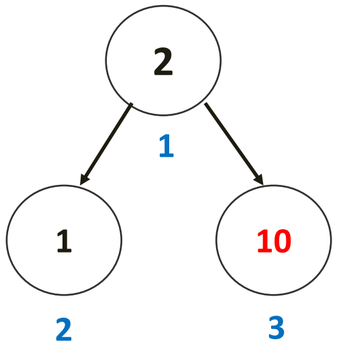
query : 2

현재 최대 힙과 최소 힙의 크기 차이는 1이다.

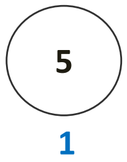
고로 result 배열에 최대 힙의 루트인 2를 기록해준다.

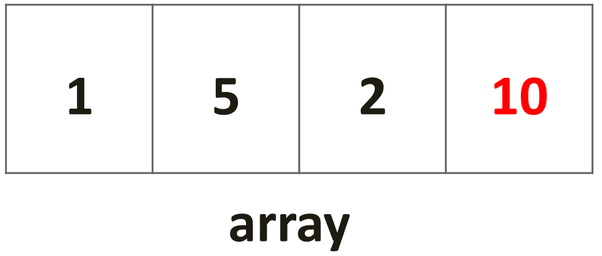
힙의 활용 – 중앙값 관리

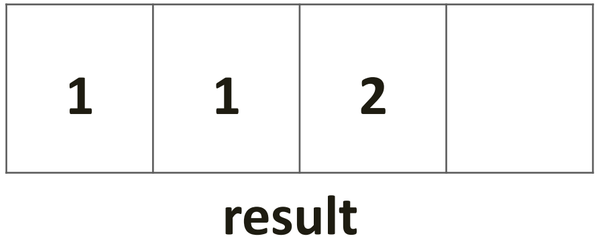
max | size : 3



min | size : 1







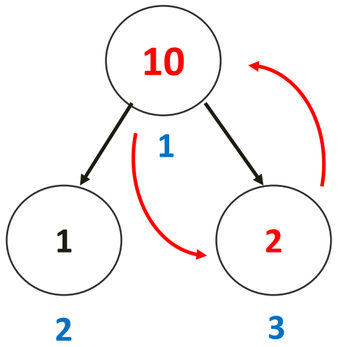
query : 10

현재 원소는 10이다. 먼저 최대 힙에 넣어준다.

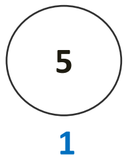
최대 힙에 넣었으니 우선순위에 맞춰 정렬을 해야한다.

힙의 활용 – 중앙값 관리

max | size : 3



min | size : 1



1	5	2	10

array

1	1	2	

result

query : 10

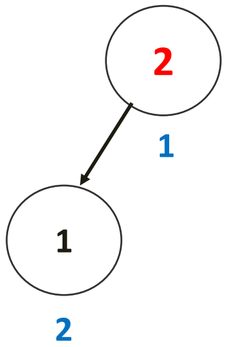
현재 노드의 위치는 3이다. 고로 부모의 위치는 1이다.

부모의 값(2)이 현재 노드(10)보다 작으므로 최대 힙에 맞지 않는다.

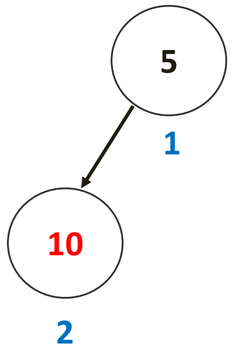
두 값을 교환 해준다. 루트까지 올라갔으므로 교환 작업이 끝났다.

힙의 활용 – 중앙값 관리

max | size : 2



min | size : 2



1	5	2	10

array

1	1	2	2

result

query : 10

최소 힙과 최대 힙의 크기 차이가 2이므로 최대 힙에서 루트를 pop 하여 최소 힙에 push 한다.

힙의 균형이 맞추어 졌다.

최대 힙의 루트가 항상 중앙값 이므로 최대 힙의 루트인 2을 result 배열에 기록한다.