Priority Queue (우선순위 큐)

[개념]

우선순위 큐의 필요성

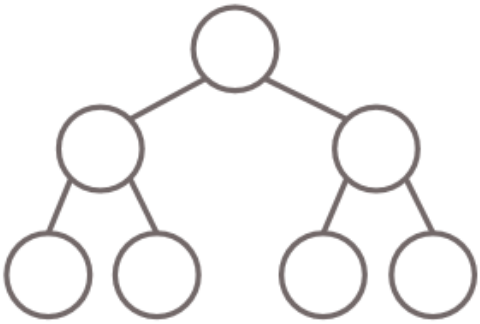
우선순위 큐는 '우선 순위'를 가진 데이터들을 저장하는 큐를 의미합니다. 데이터를 꺼낼 때 우선 순위가 높은 데이터가 가장 먼저 나온다는 특징이 있어 많이 활용되고 있습니다.

우선순위 큐는 운영체제의 작업 스케줄링, 정렬, 네트워크 관리 등의 다양한 기술에 적용되고 있습니다.

우선순위 큐와 큐의 차이점

일반적인 형태의 큐는 선형적인 형태를 가지고 있지만 우선순위 큐는 트리(Tree) 구조로 보는 것이 합리적입니다. 일반적으로 우선순위 큐는 최대 힙을 이용해 구현합니다.

일반적인 큐

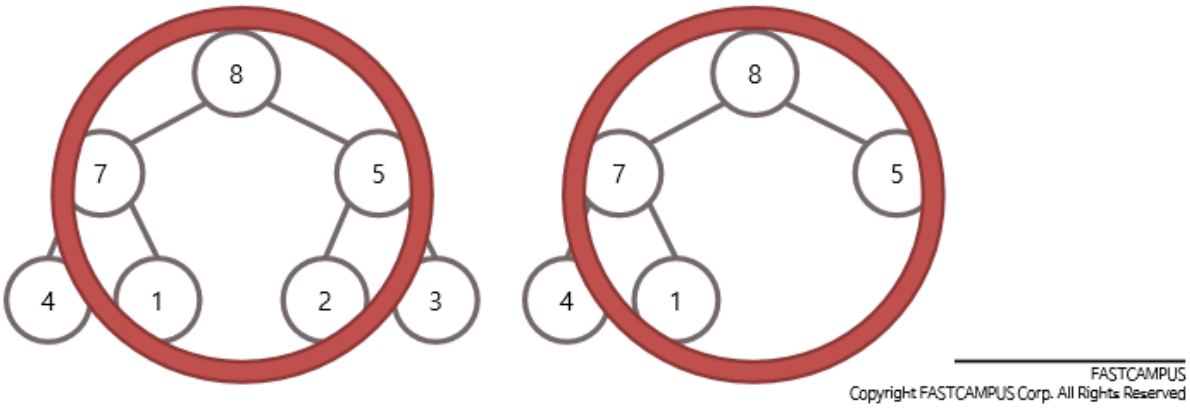


우선순위 큐

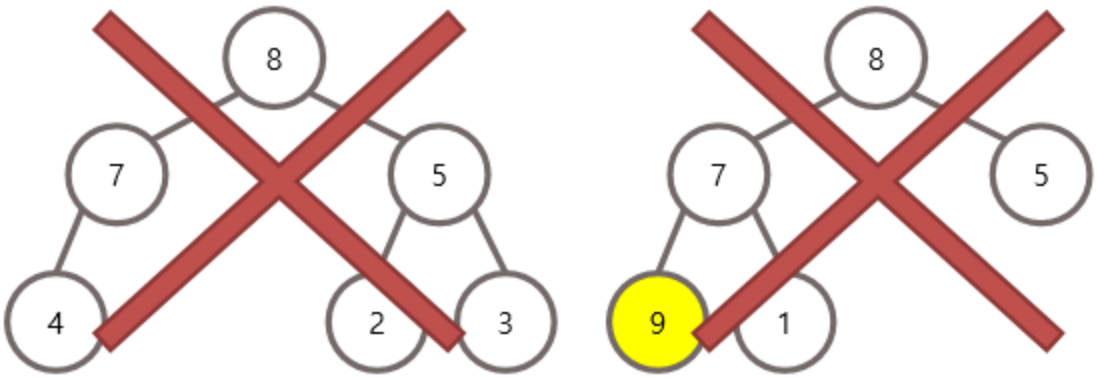
FASTCAMPUS
Copyright FASTCAMPUS Corp. All Rights Reserved

최대 힙(Max Heap)

최대 힙은 부모 노드가 자식 노드보다 값이 큰 완전 이진 트리를 의미합니다.

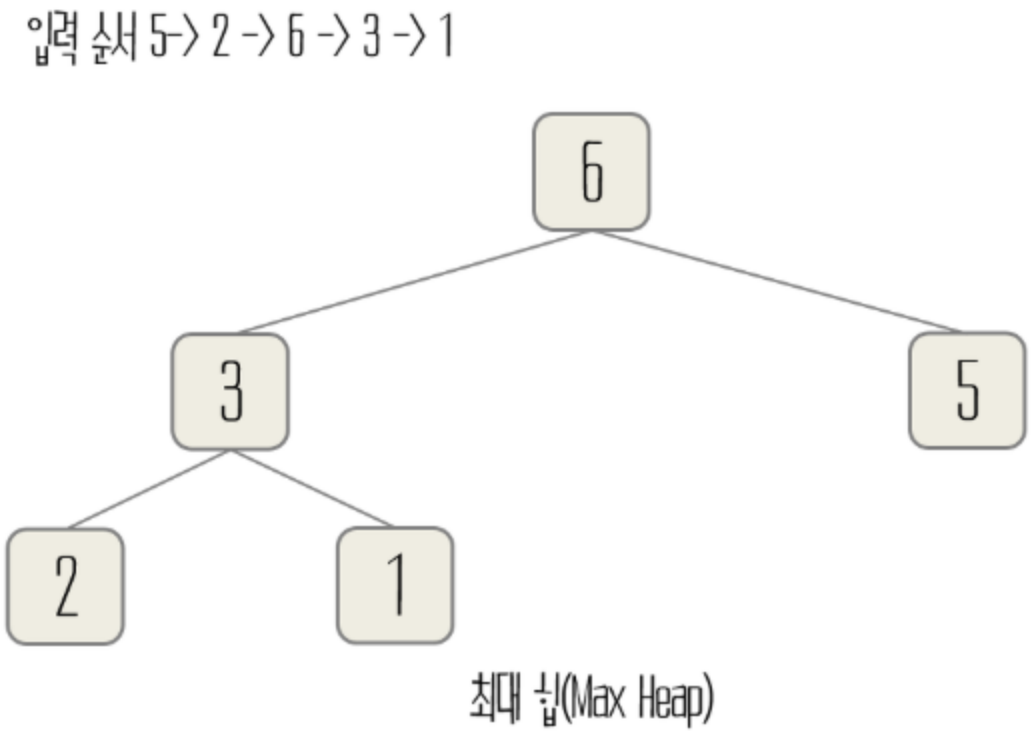


FASTCAMPUS
Copyright FASTCAMPUS Corp. All Rights Reserved



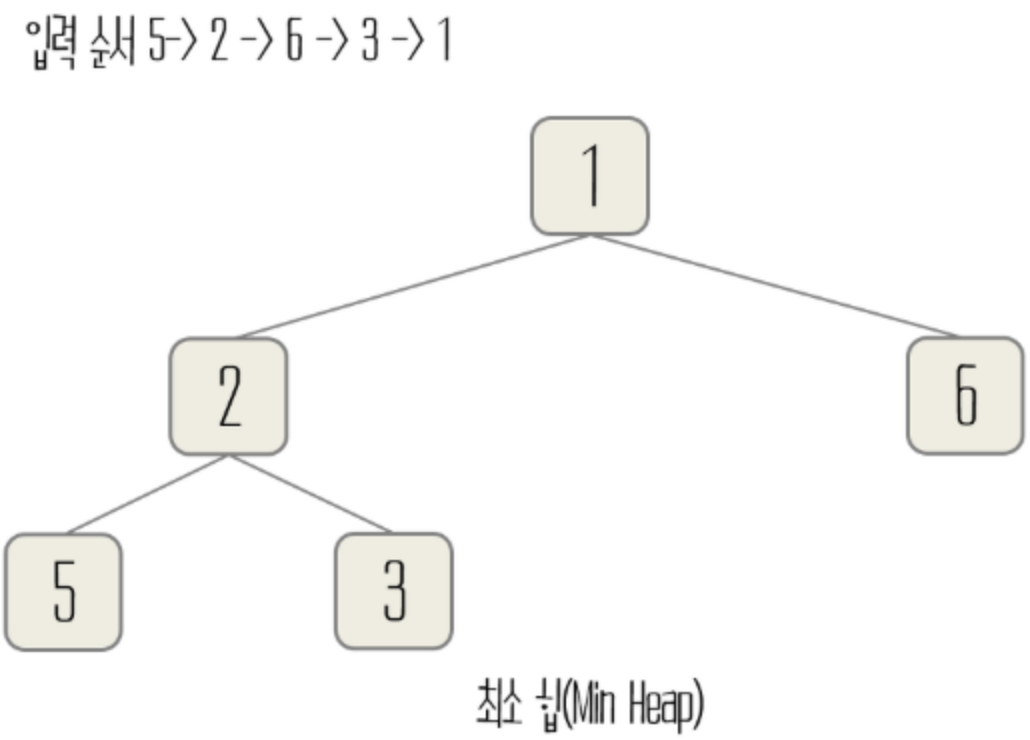
> *최대 힙(Max Heap) :*

- 데이터 중 가장 큰 값이 루트에 위치
- 모든 부모 노드는 자식보다 크거나 같다.

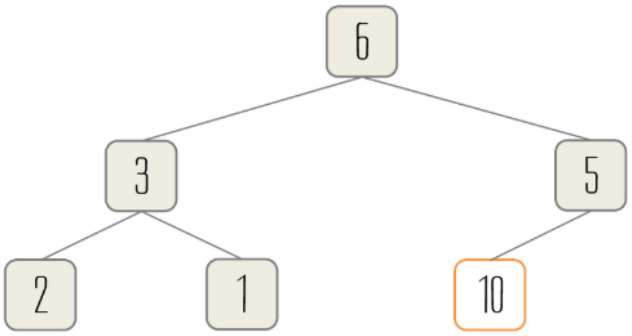


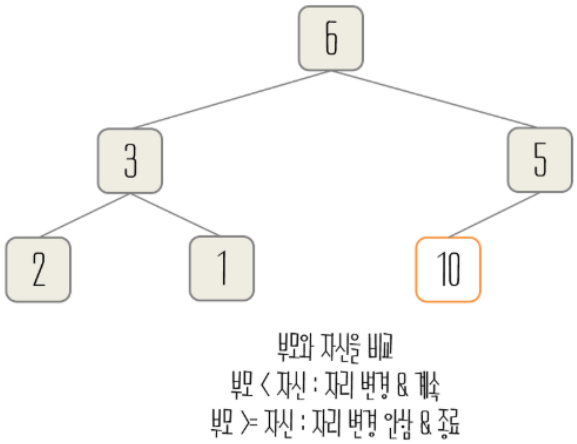
> *최소 힙(Min Heap) :*

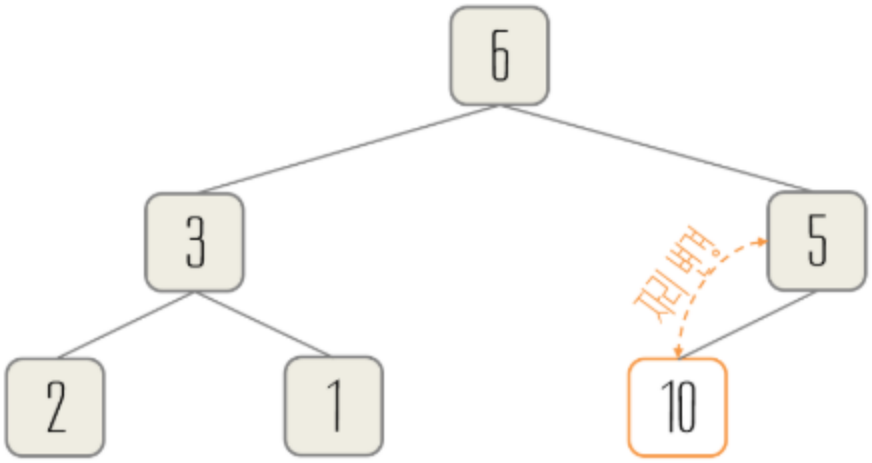
- 데이터 중 가장 작은 값이 루트에 위치
- 모든 부모 노드는 자식보다 작거나 같다.

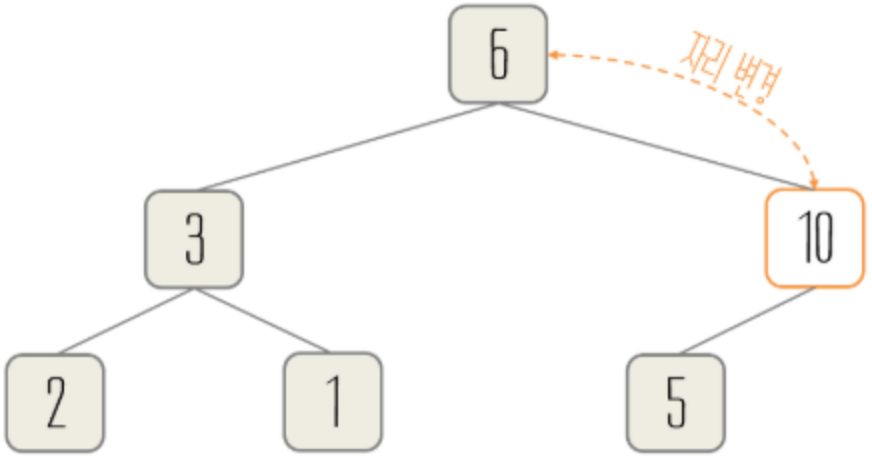

입력 순서 5-> 2 -> 6 -> 3 -> 1
1
2
6
5
3
최소 힙(Min Heap)

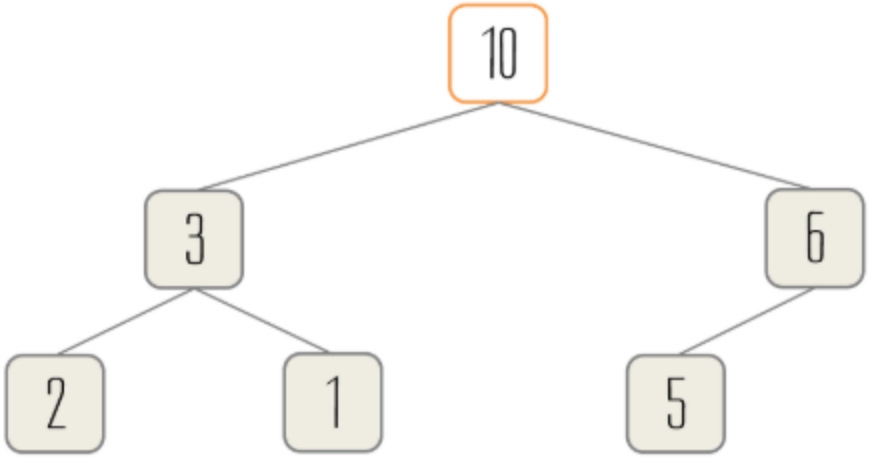
[삽입]


6
3
5
2
1
10
제일 먼저 삽입 데이터를
가장 뒤에 위치 시킨다

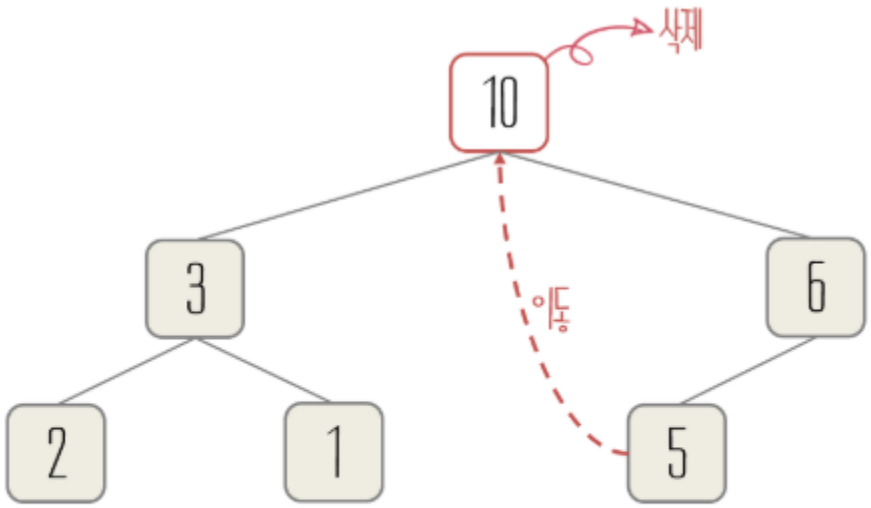

6
3
5
2
1
10
부모와 자신을 비교
부모 < 자신 : 자리 변경 & 계속
부모 >= 자신 : 자리 변경 안함 & 종료


6
3
5
2
1
10
교환!

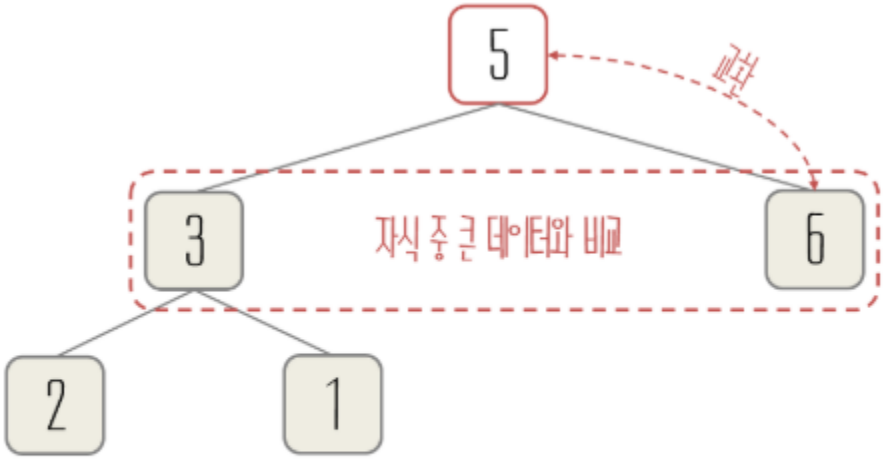

6
교환!
3
10
2
1
5


10
3
6
2
1
5

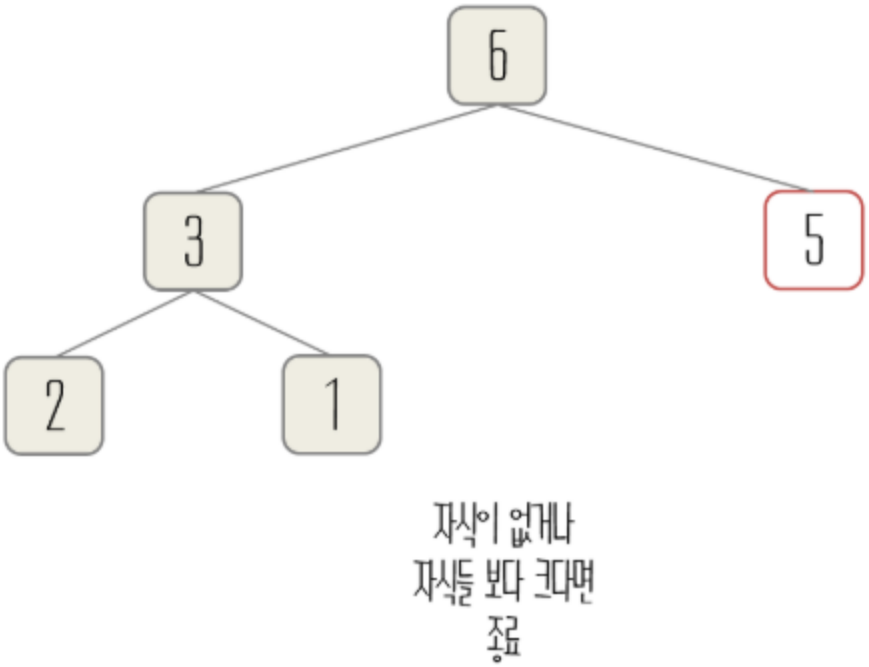
[출력]



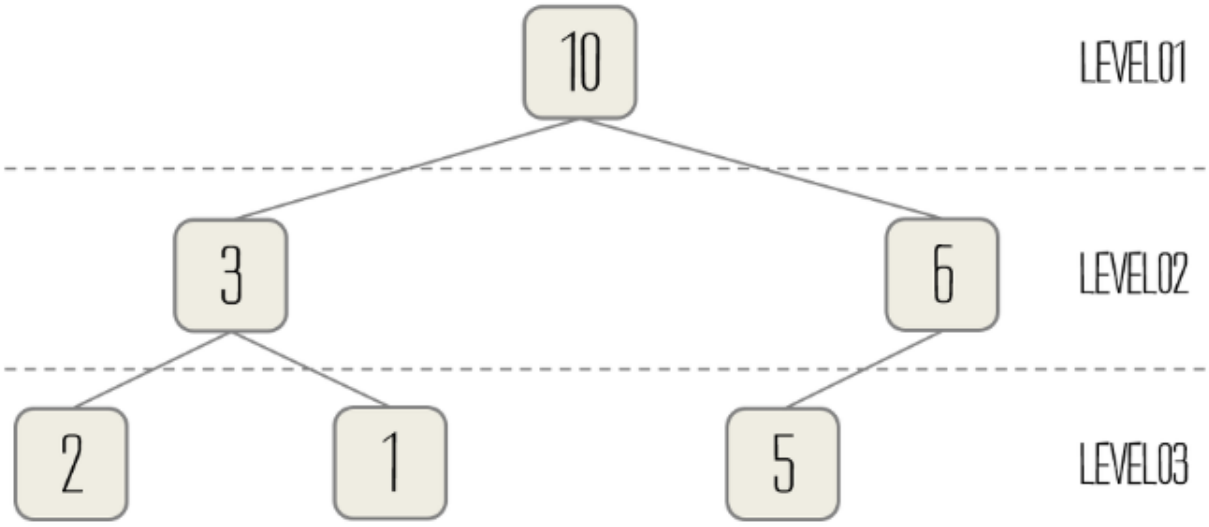
데이터를 삭제하고
제일 뒤에 있는 데이터를
삭제한 위치로 옮긴다



자식 중 큰 데이터와 비교
자식 > 자신 : 자리 변경 & 계속
자식 <= 자신 : 자리 변경 안함 & 종료

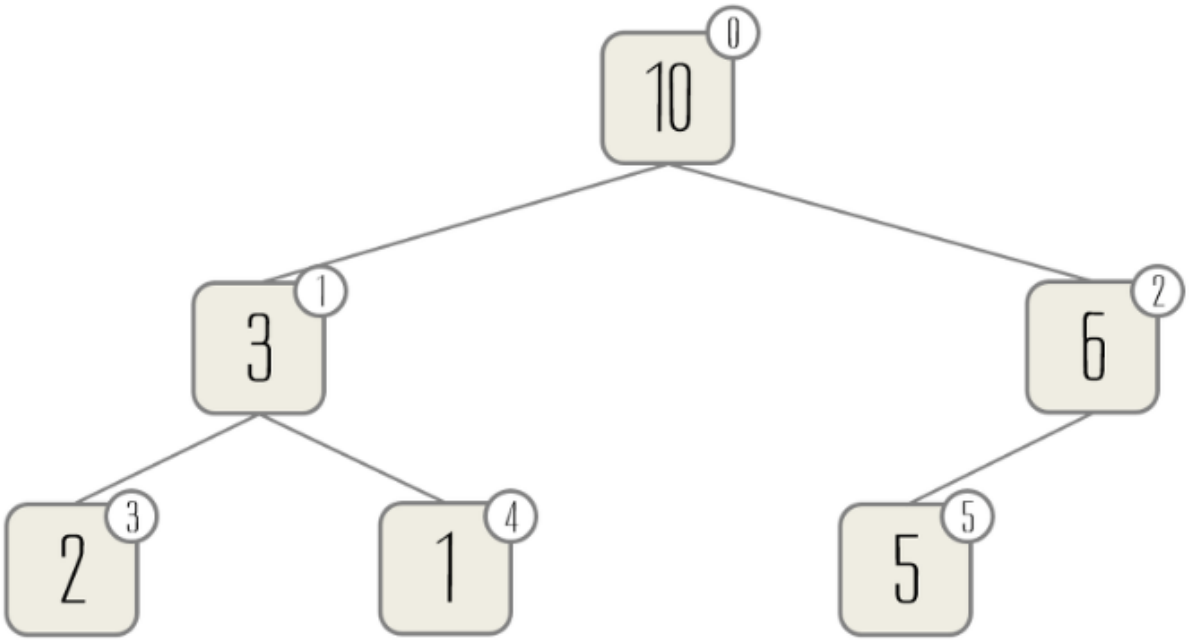


[Code]



배열	위치	0	1	2	3	4	5	…
	데이터	10	3	6	2	1	5	…

LEVEL01 LEVEL02 LEVEL03



배열	위치	0	1	2	3	4	5	…
	데이터	10	3	6	2	1	5	…

위치 계산 법(n = 위치)		
부모	(n - 1) / 2	5의 부모 = (5-1)/2 = 2
왼쪽 자식	2n + 1	2의 왼쪽 자식 = 2*2+1 = 5
오른쪽 자식	2n + 2	1의 오른쪽 자식 = 1*2+2 = 4

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

#define MAX_SIZE 10000

void Swap(int *pFir, int *pSec)
```

```
{
        if (NULL == pFir || NULL == pSec)
                return;

        int nTemp = *pFir;
        *pFir = *pSec;
        *pSec = nTemp;
}

typedef struct ST_PRIORITY_QUEUE
{
        int arrHeap[MAX_SIZE];
        int nCount;
} PriorityQueue;

void Push(PriorityQueue *pQ, int nData)
{
        if (pQ->nCount >= MAX_SIZE) return;
        pQ->arrHeap[pQ->nCount] = nData;

        int nNow              = pQ->nCount;
        int nParent           = (pQ->nCount - 1) / 2;
        int nLeftChild        = (2 * nNow) + 1;
        int nRightChild = nLeftChild + 1;

        while (nNow > 0 && pQ->arrHeap[nNow] > pQ->arrHeap[nPa
rent])
        {
                Swap(&pQ->arrHeap[nNow], &pQ->arrHeap[nParen
t]);
```

```
                nNow = nParent;
                nParent = (nNow - 1) / 2;
        }

        pQ->nCount++;
}

int Pop(PriorityQueue *pQ)
{
        if (pQ->nCount <= 0) return -9999;

        int nRoot = pQ->arrHeap[0];
        pQ->nCount--;
        pQ->arrHeap[0] = pQ->arrHeap[pQ->nCount];

        int nNow = 0;
        int nLeftChild = nNow + 1;
        int nRightChild = nLeftChild + 1;

        int nTarget = nNow;
        while (nLeftChild < pQ->nCount)
        {
                if (pQ->arrHeap[nTarget] < pQ->arrHeap[nLeftCh
ild])
                        nTarget = nLeftChild;

                if (pQ->arrHeap[nTarget] < pQ->arrHeap[nRightC
hild] && nRightChild < pQ->nCount)
                        nTarget = nRightChild;
```

```
                if (nTarget == nNow)
                        break;
                else
                {
                        Swap(&pQ->arrHeap[nNow], &pQ->arrHeap
[nTarget]);
                        nNow = nTarget;
                        nLeftChild = (2 * nNow) + 1;
                        nRightChild = nLeftChild + 1;
                }
        }

        return nRoot;
}

int main(void)
{
        int n, data;
        scanf("%d", &n);

        PriorityQueue pQ;
        pQ.nCount = 0;
        for (int i = 0; i < n; i++)
        {
                scanf("%d", &data);
                Push(&pQ, data);
        }
        printf("--0-0-0-0-0-0-0-0-0--\n");
        for (int i = 0; i < n; i++)
        {
```

```
                int data = Pop(&pQ);
                printf("[%d]Pop : %d\n", i, data);
        }

        system("pause");
        return 0;
}
```