Priority Queues

서승현 교수
Prof. Anes Seung-Hyun Seo
(seosh77@hanyang.ac.kr)

Division of Electrical Engineering
Hanyang University, ERICA Campus

Where we are ?

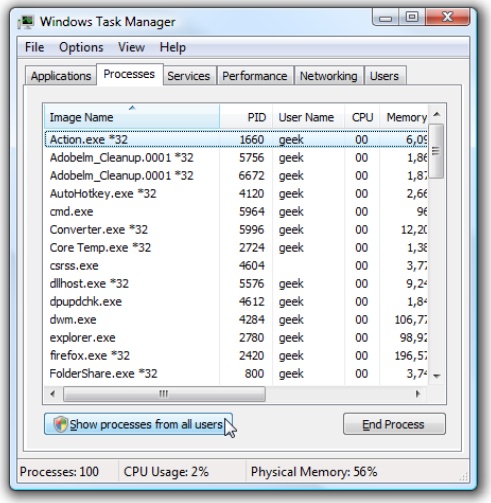

Graph
Tree
Binary Tree
Binary Search Tree
AVL Search Tree
Heap

Queue

■ FIFO queue

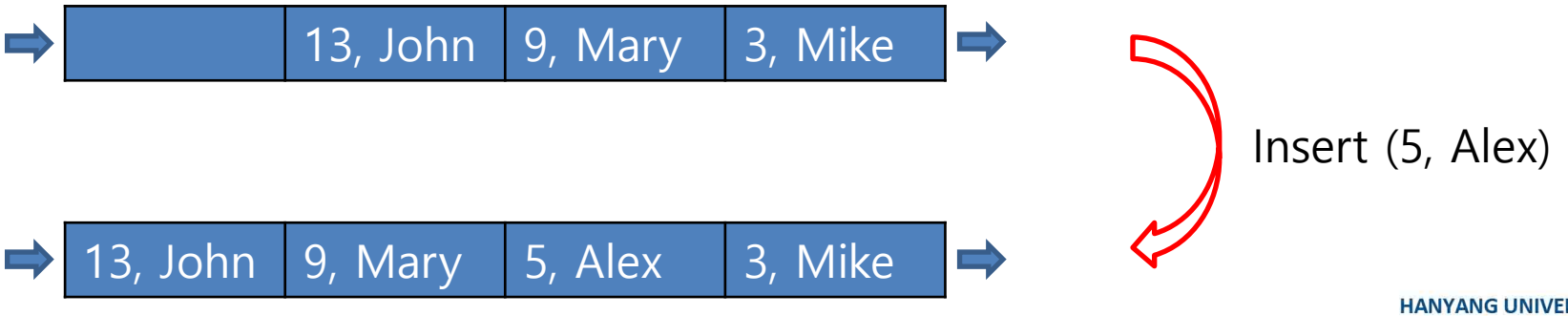
- ▶ Elements are served by order of arrival
- ▶ No priority among elements

■ What if there are priorities?



Priority Queues

- **Priority queue**
  - A collection of elements such that each element has an associated priority.
- **(priority = key, content)**
  - Similar to dictionary
  - Lower/higher keys(min-heap/max-heap) represent higher priorities
- **Operations**
  - Insert
  - Find/Remove min or max key



Example – Priority Queues

■ Emergency room

- What if patients are served based on a FIFO queue?
- (key = waiting time)

- Average waiting time
- (90 + 115 + 155 + 170)/4 = 132.5

Example – Priority Queues

■ Emergency room

- Priority queue
- Patients with less waiting time are served first

90, Mike	40, Mary	25, Alex	15, John

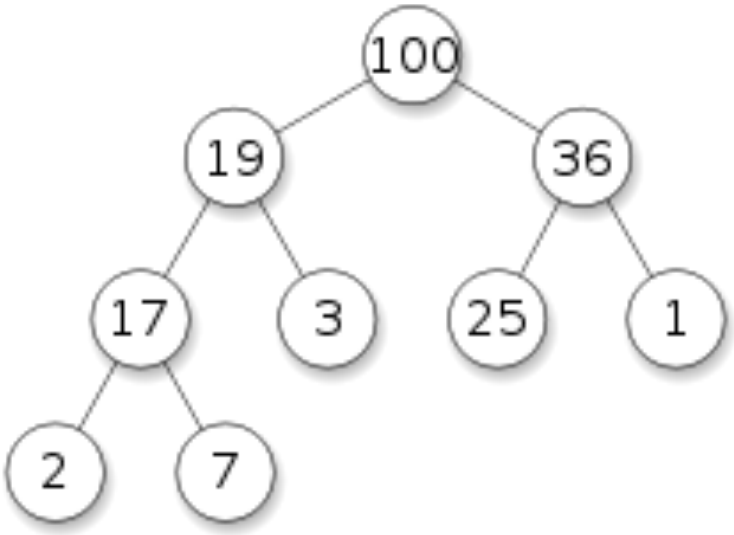
- Average waiting time
- (15 + 40 + 80 + 170)/4 = 76.25
- This can be preferable

Priority Queues

- **Implemented based on "heap"**
  - Heap is a tree-based structure.
  - A heap is a binary tree whose left and right subtrees have values less(max-heap)/greater (min-heap) than or equal to their parents.

- **Min heap**
  - Keys of parents ≤ those of children

- **Max heap**
  - Keys of parents ≥ those of children

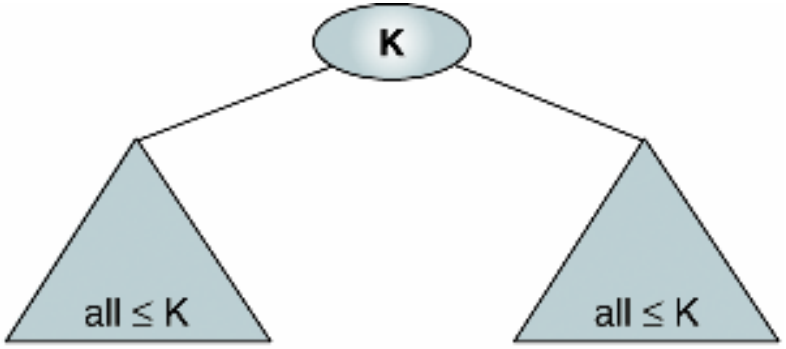


Max heap

Heap

- **Heap: a binary tree structure with the properties …**
  - [Rule 1] The tree is complete.
  - [Rule 2] The key value of each node is **greater than or equal to** the key value in each of its descendants (**MAX heap**)

  Cf. **MIN heap**: key value of each node is less than or equal to the key value in each of its descendants.

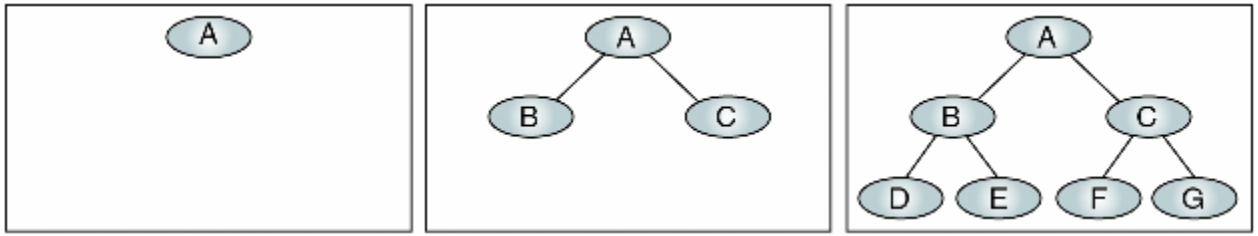


<MAX heap>

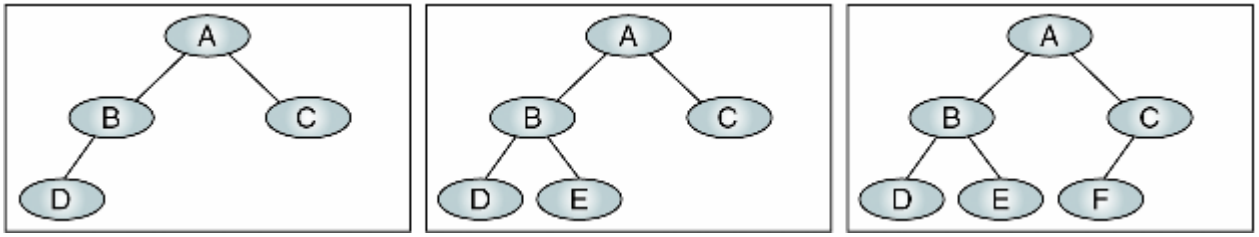
Complete Binary Tree (review)

Binary Search Tree <-> Heap 과의 차이

- **Full binary tree**: a tree with the maximum # of entries for its height

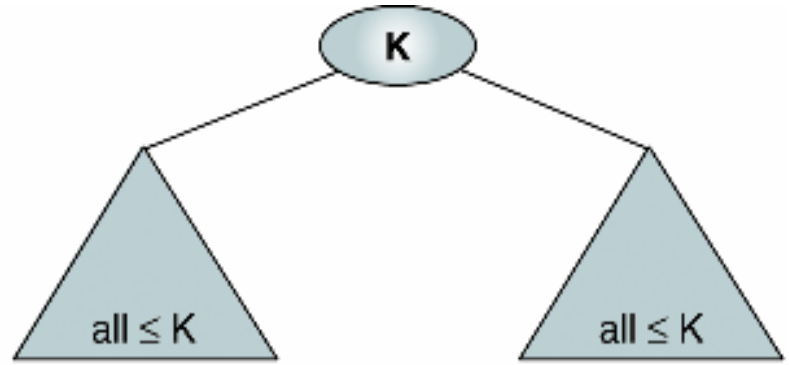


- **Complete binary tree**: a tree that has minimum height for its nodes and all nodes in the last level are found on the left

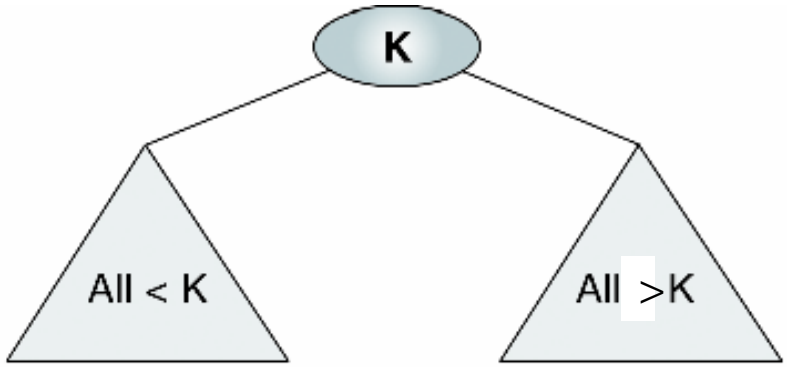


Heap

■ Heap vs. binary search tree



Heap

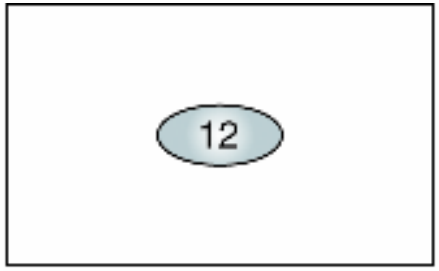


Binary search tree

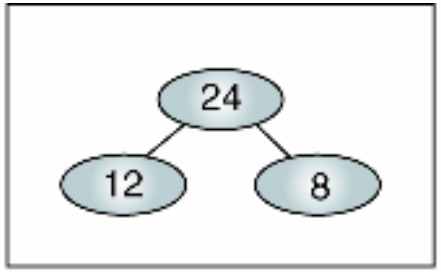
- The lesser-valued nodes of a heap can be placed on either the right or the left subtree.

Examples of Heaps

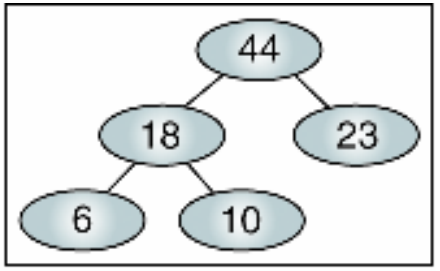
■ Heaps



(a) Root-only heap

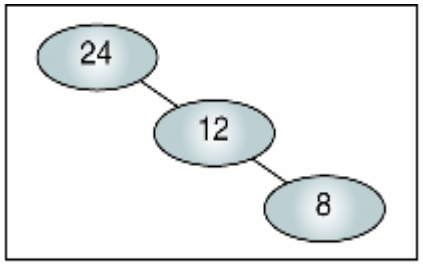


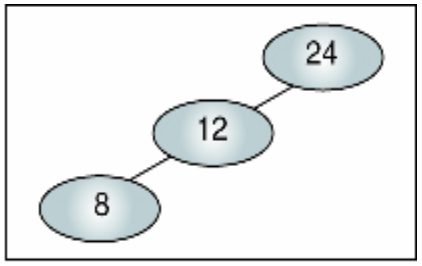
(b) Two-level heap

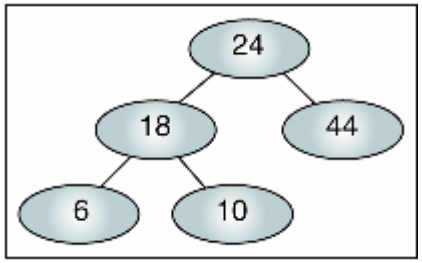


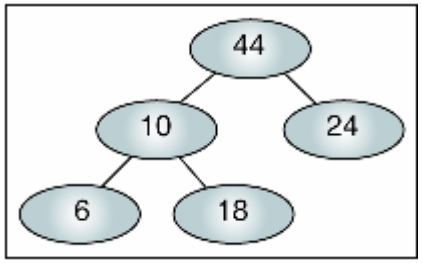
(c) Three-level heap

■ Invalid heaps (Why?)









Why Heap?

- **Heap is efficient for extraction of the largest (or smallest) element.**

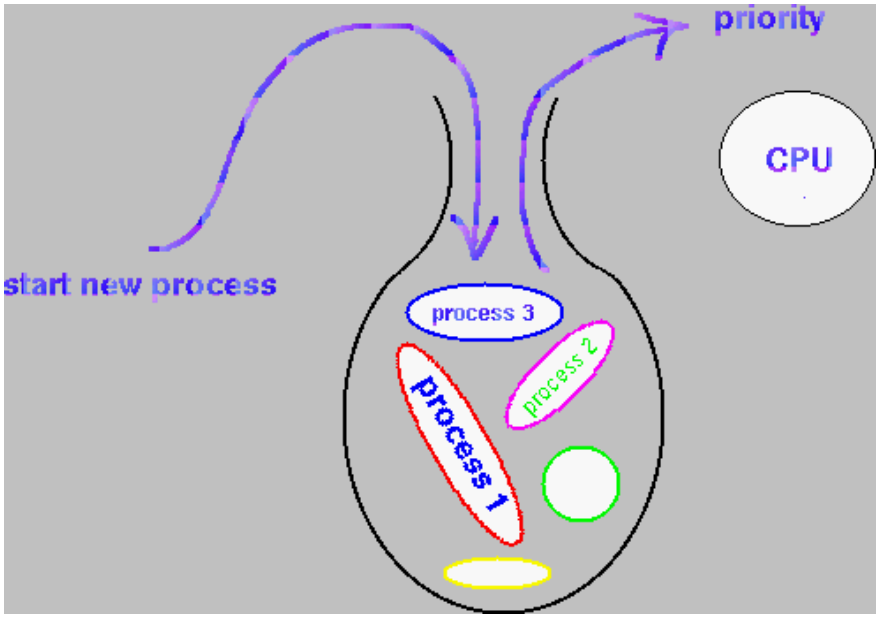
- **Compromise of sorted and unsorted structures**
  - Sorted structures (array or linked-list)
    - **Very efficient in extraction of largest (smallest) element**
    - **Inefficient in insertion**
  - Unsorted structures (array or linked-list)
    - **Inefficient in extraction of largest (smallest) element**
    - **Very efficient in insertion**

Big-Oh (1)

- **Heap is efficient for both insertion and deletion**

Heap and Priority Queue

- **Priority queue: a variation of queue**
  - Add an element to the queue with an associated priority
  - Remove element with the highest priority and return it
- **Heap is an excellent structure to implement priority queue**



Priority Queue ADT

·Object : a collection of elements with a priority of type n elements

·Operation :

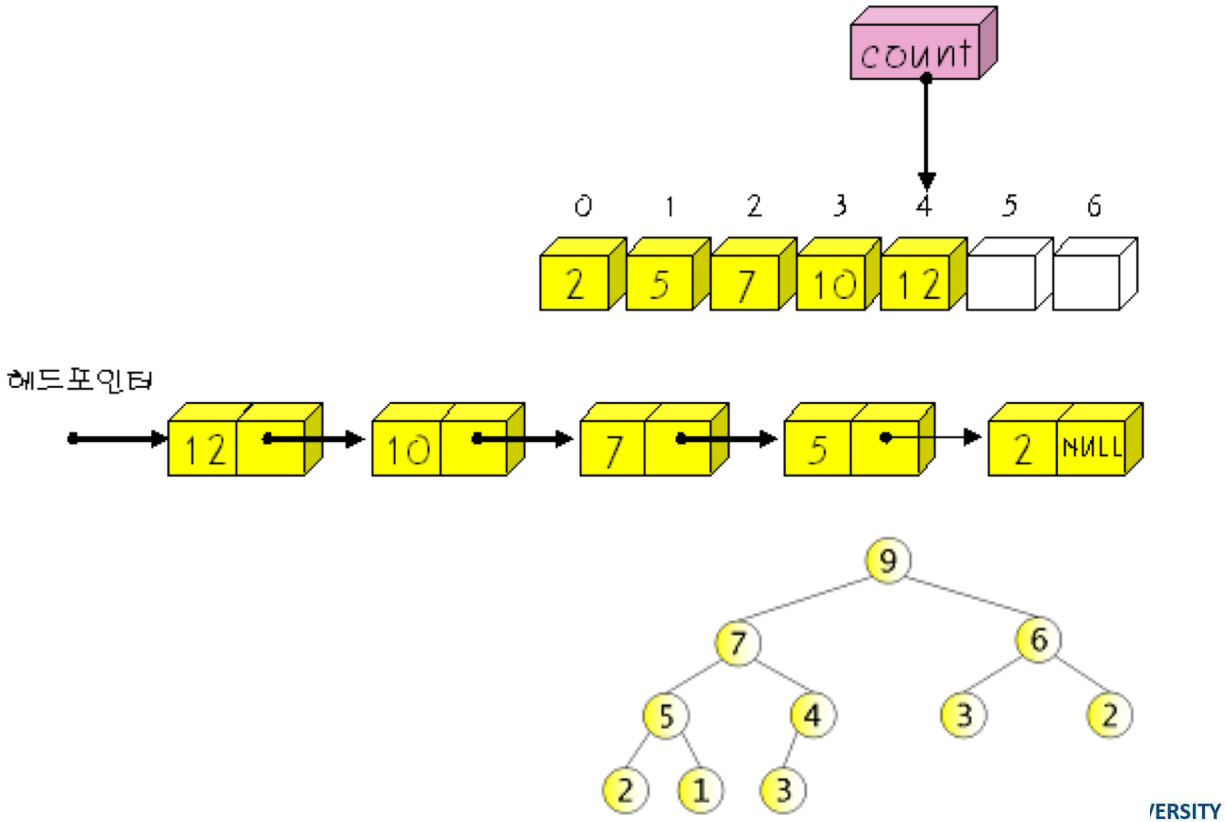
- create() ::= create a priority queue
- init(q) ::= initialize a priority queue q
- is_empty(q) ::= check if a priority queue q is empty
- is_full(q) ::= check if a priority queue q is full.
- insert(q, x) ::= insert an element x in a priority queue q
- delete(q) ::= delete the highest priority element and return it
- find(q) ::= Return the highest priority element

Priority Queue ADT

- **The most important operation: insert, delete**
- **Priority Queue**
  - Min priority queue
  - Max priority queue

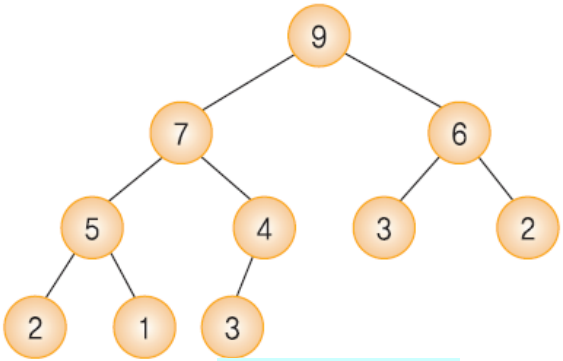
Representation for priority queue

- Array
- Linked list
- Heap

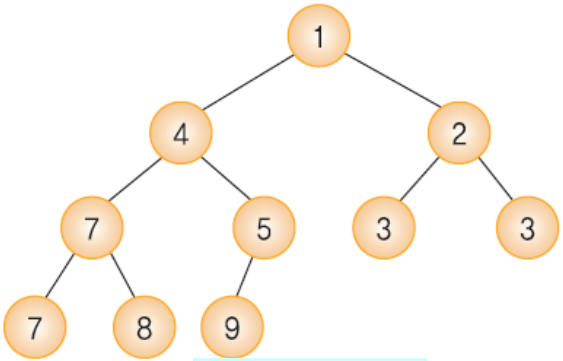


Heap

- **A heap is a complete binary tree in which the keys stored by the nodes satisfy the following properties:**
- **Max heap**
  - A complete binary tree where the key value of the parent node is greater than or equal to the key value of the child node
  - key(parent node) ≥ key(child node)
- **Min heap**
  - A complete binary tree where the key value of the parent node is less than or equal to the key value of the child node
  - key(parent node) ≤ key(child node)



Max heap



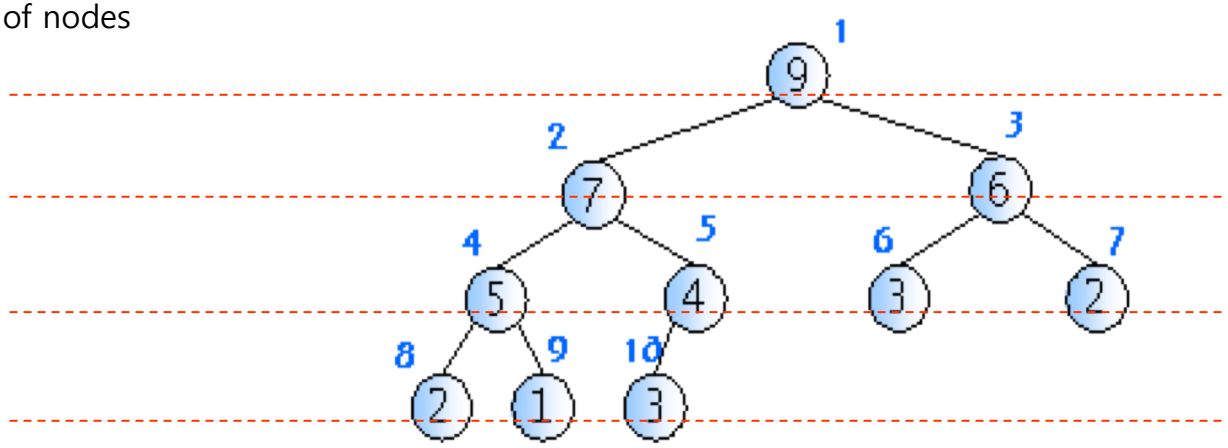
Min heap

Heap

■ **The height of heap with n nodes: O(logn)** 왜 logn인가?

- Heap is a complete binary tree.
- Except the last level, h, each i level has 2^{i-1} nodes.

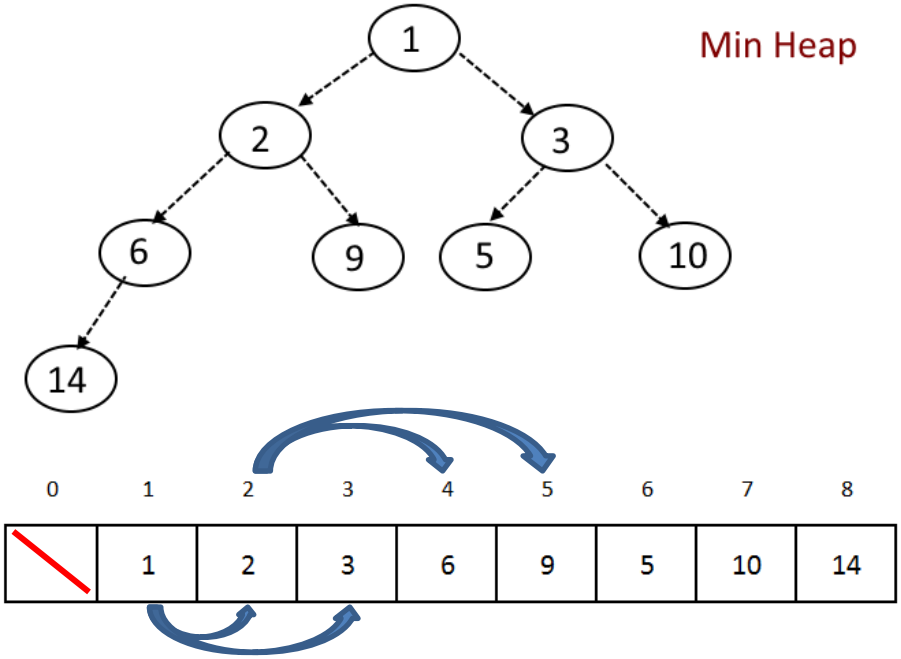
Level	the number of nodes
1	$1=2^0$
2	$2=2^1$
3	$4=2^2$
4	3



Binary heap as array

How to access parents/children?

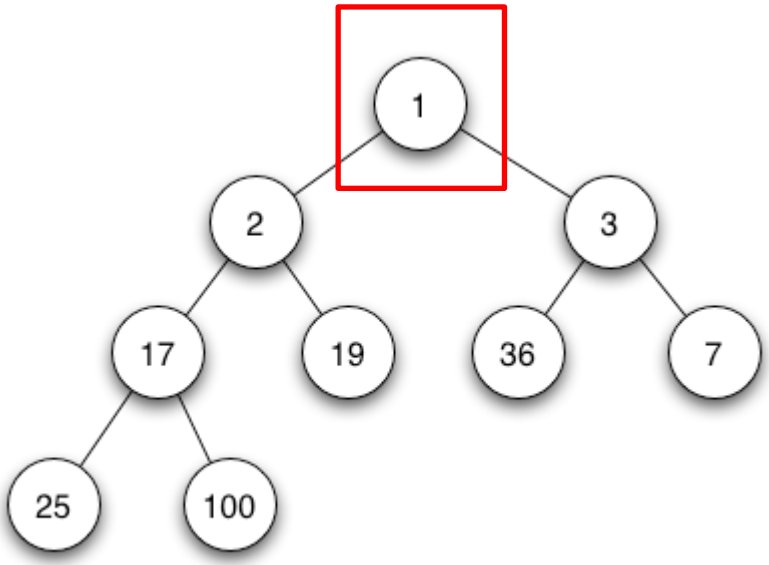
- First element of the array is empty
- The i^{th} node: Left child will be 2i
  Right child will be 2i+1
  Parent will be $\lfloor i/2 \rfloor$



Binary Heap

■ Get the minimum element

- ▶ Simply output the root node



Min-heap operation

■ Important operations

- ▶ Insertion
- ▶ Deletion

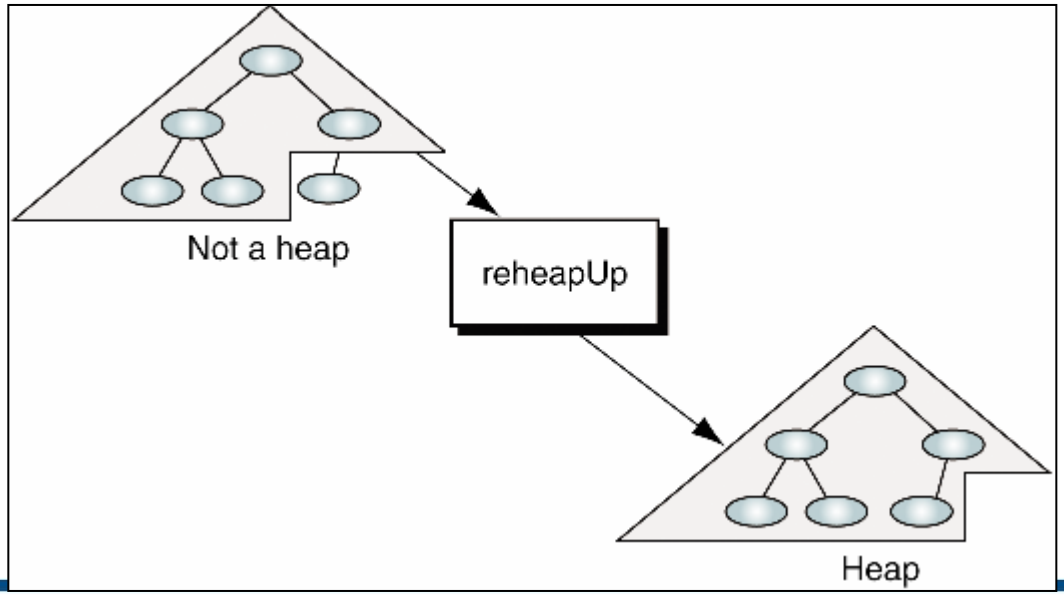
■ Problem: preserving conditions of min heap

1. Complete tree
2. Key value of each node should be no bigger than those of children

Insertion into heap

■ Inserting a node into heap

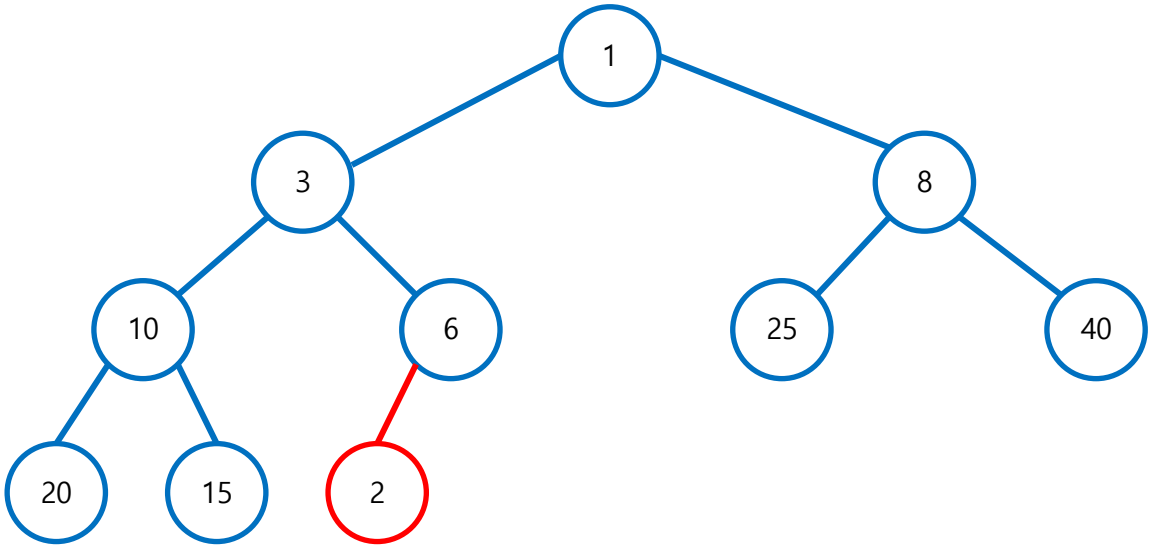
1. Insert the node to heap
   - To preserve 1st condition, the new node should be added at the last leaf level at the first empty position
   - 2nd condition could be broken
2. Repair the structure so that 2nd condition is satisfied
   - **ReheapUp**



Binary Heap : Operations

■ Insert a new element

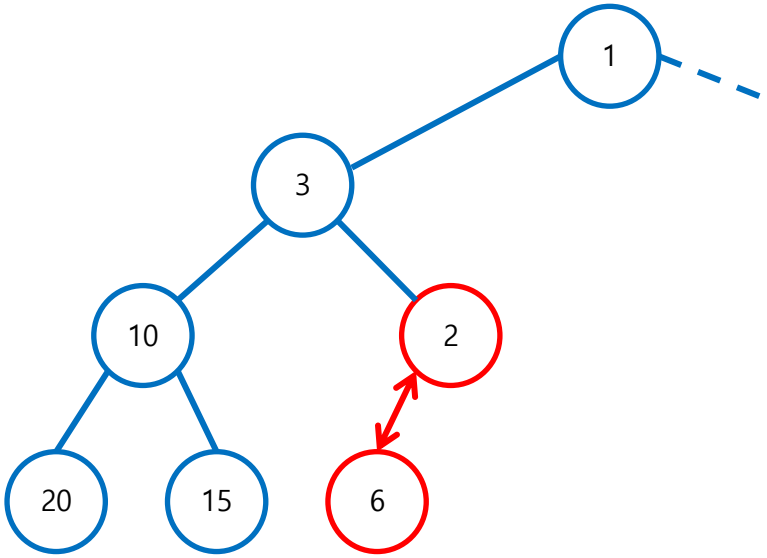
- ▶ 1. place it at the bottom level, as left as possible
- ▶ (why?)



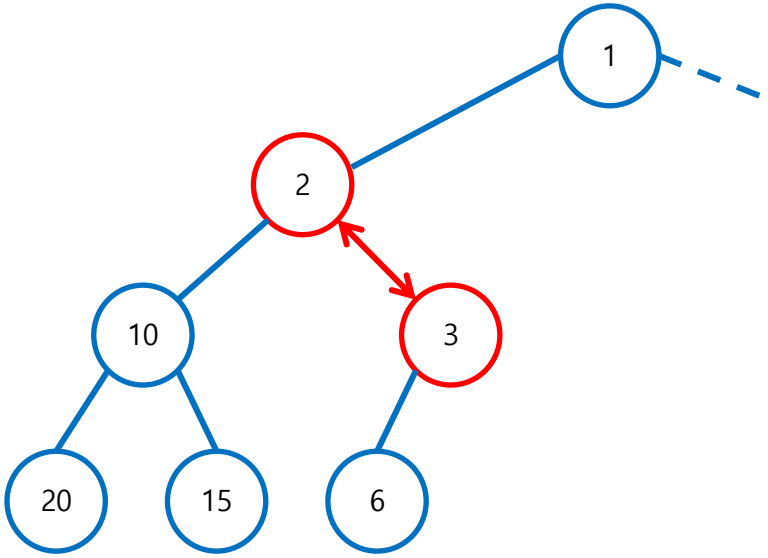
Binary Heap : Operations

■ Insert a new element

- ▶ 2. exchange the nodes until the minimum heap property is satisfied



Exchange 1

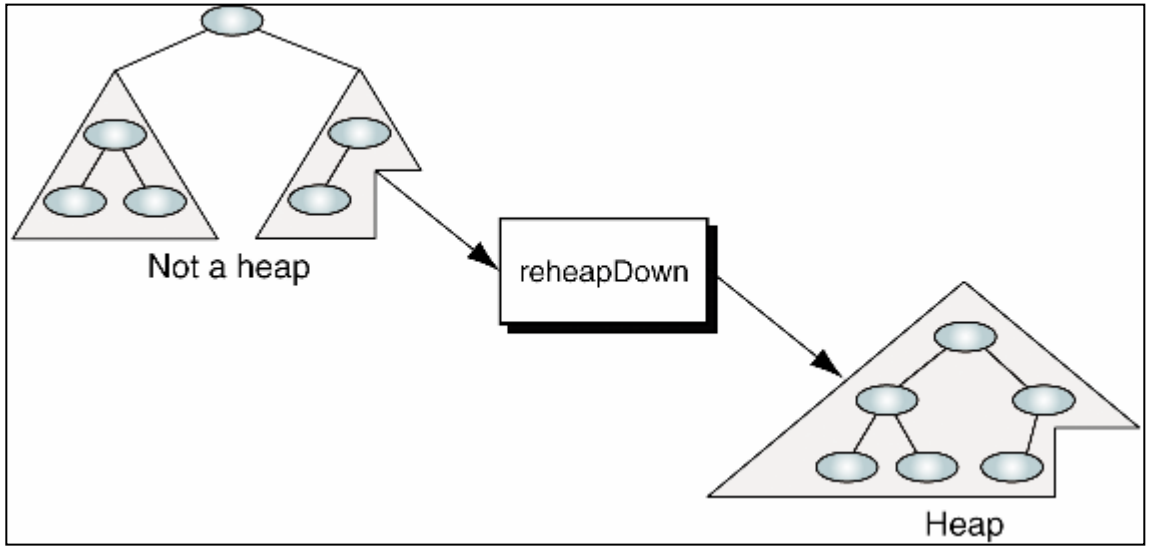


Exchange 2

Deletion from heap

■ Deleting a node from heap

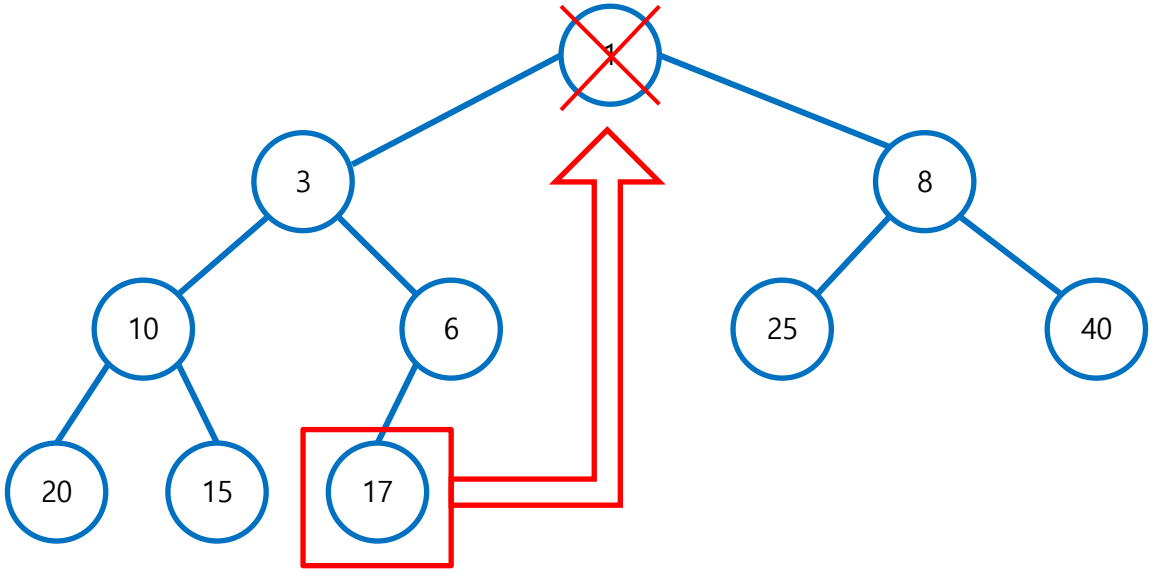
1. Deletion always occurs at the root
   - To preserve 1st condition, the last node should move to the root
   - 2nd condition could be broken
2. Repair the structure so that 2nd condition is satisfied
   - **ReheapDown**



Binary Heap : Operations

■ Remove the minimum element

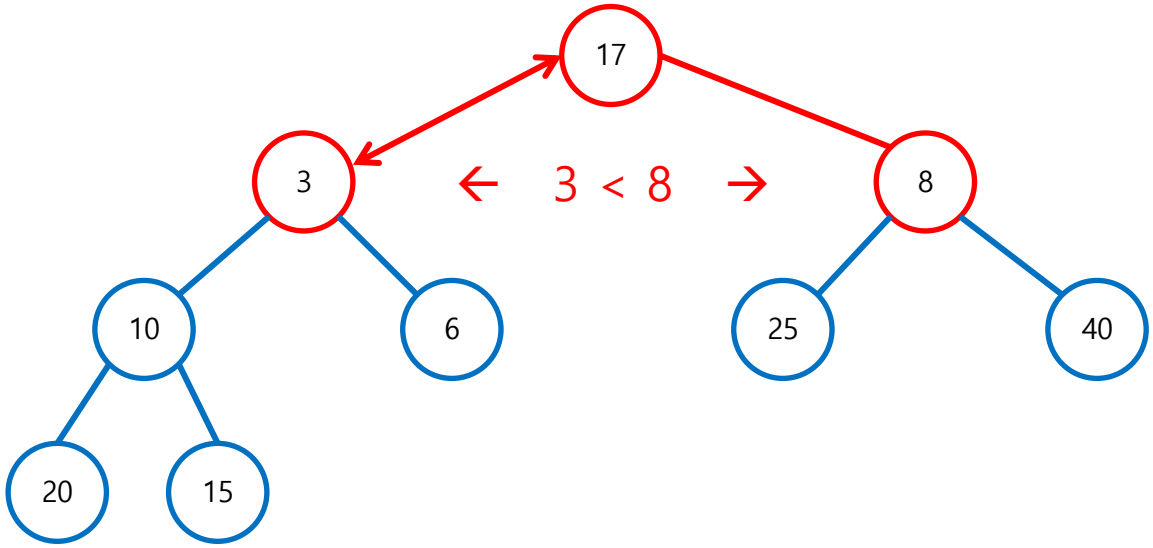
- ▶ 1. exchange the root with the right-most node in the bottom level (why?)



Binary Heap : Operations

■ Remove the minimum element

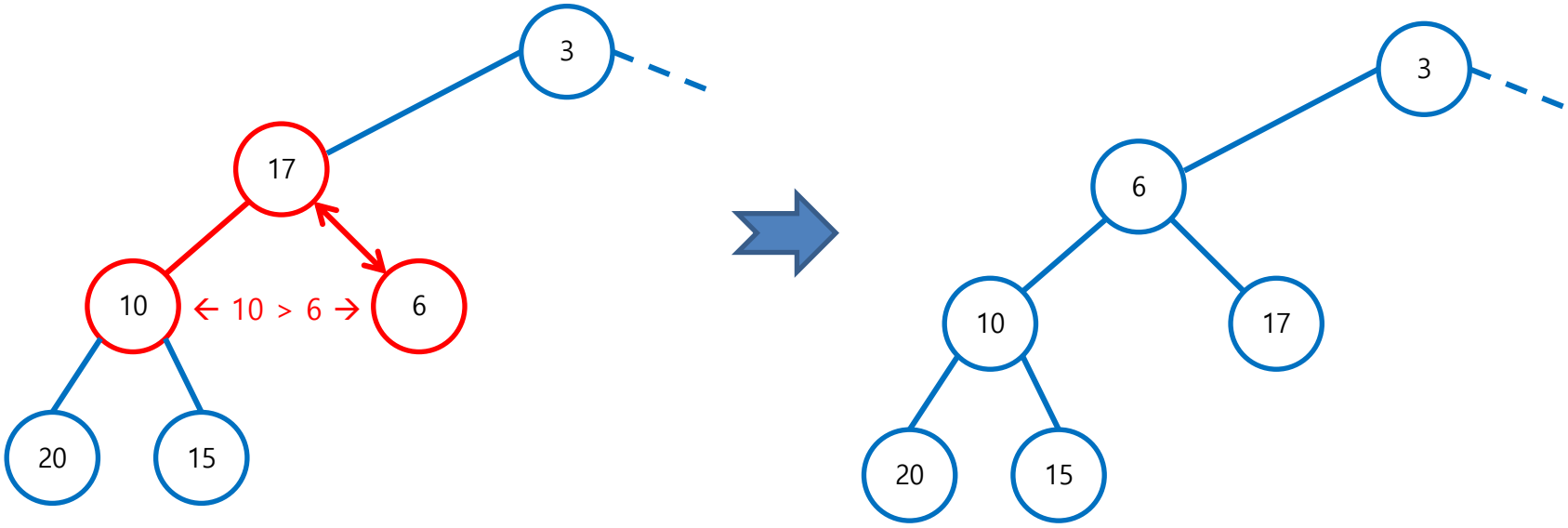
- ▶ 2. exchange the root with the minimum of its children
- ▶ (why not the entire subtrees but the children?)



Binary Heap : Operations

■ Remove the minimum element

- ▶ 3. Repeat the process until the min-heap property is satisfied



Heap implementation

- Heap can be represented as the one dimensional array.
  - Elements of heap: one dimensional array of structure
  - heap_size: the number of elements

```
#define MAX_ELEMENT 200
typedef struct {
        int key;
 } element;
typedef struct {
    element heap[MAX_ELEMENT];
    int heap_size;
 } HeapType;
```

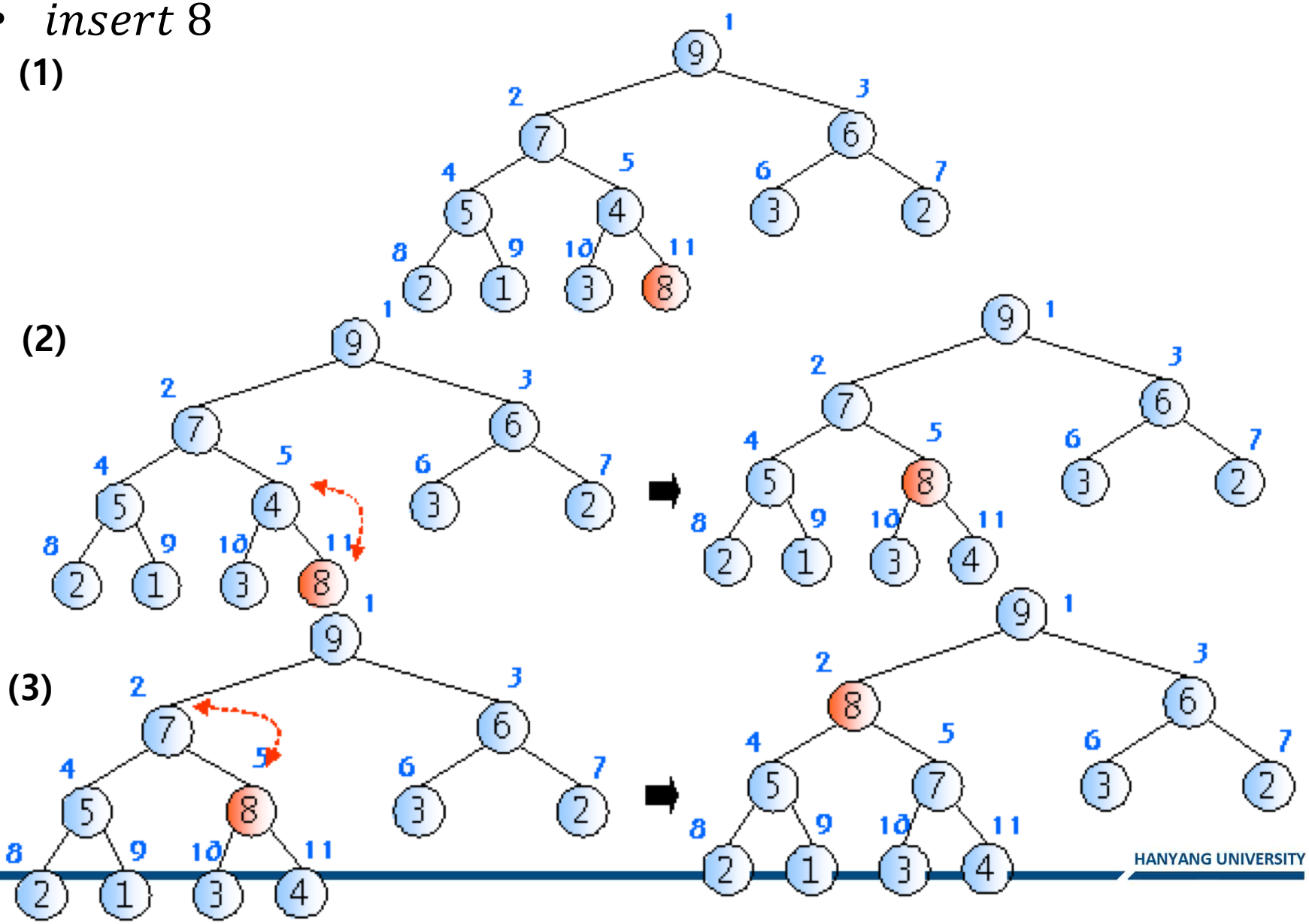
Example of insertion

- $insert\ 8$

(1)

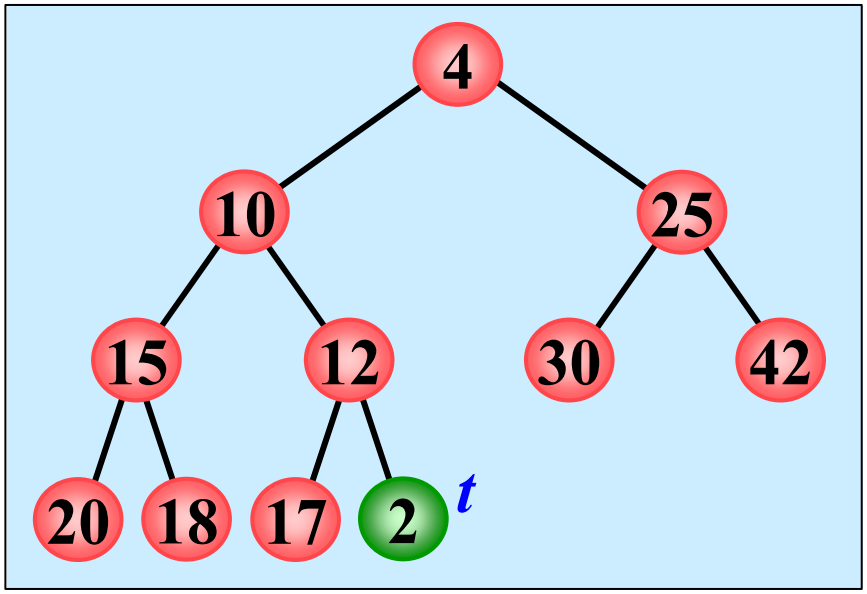
(2)

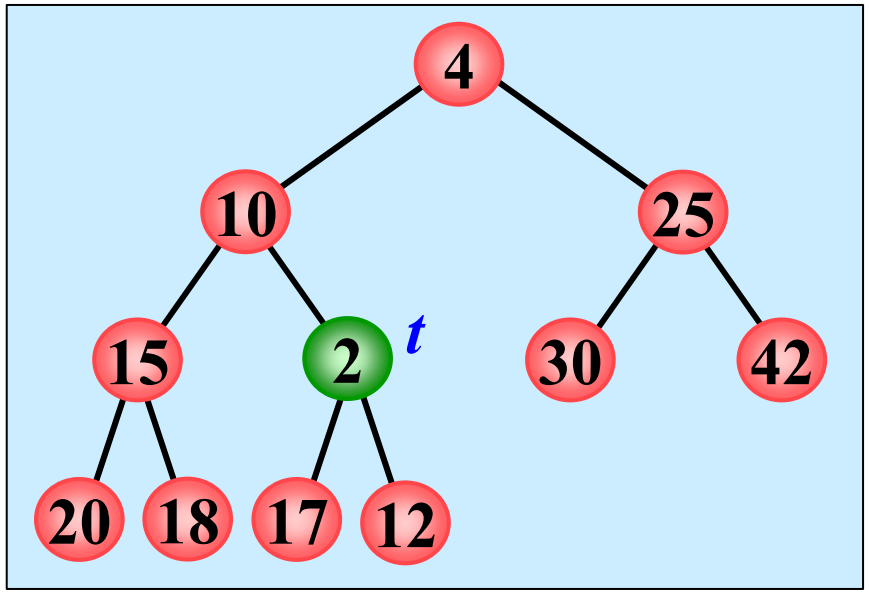
(3)

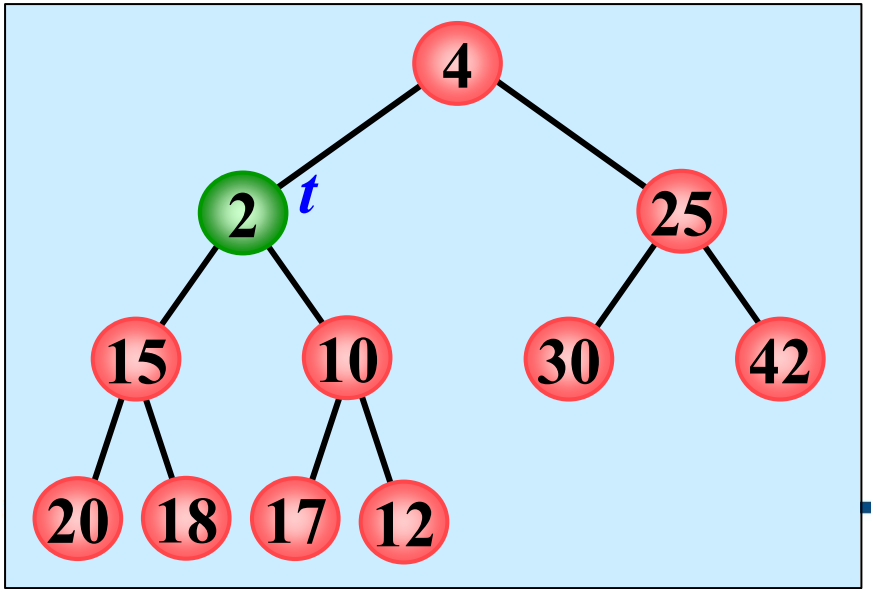


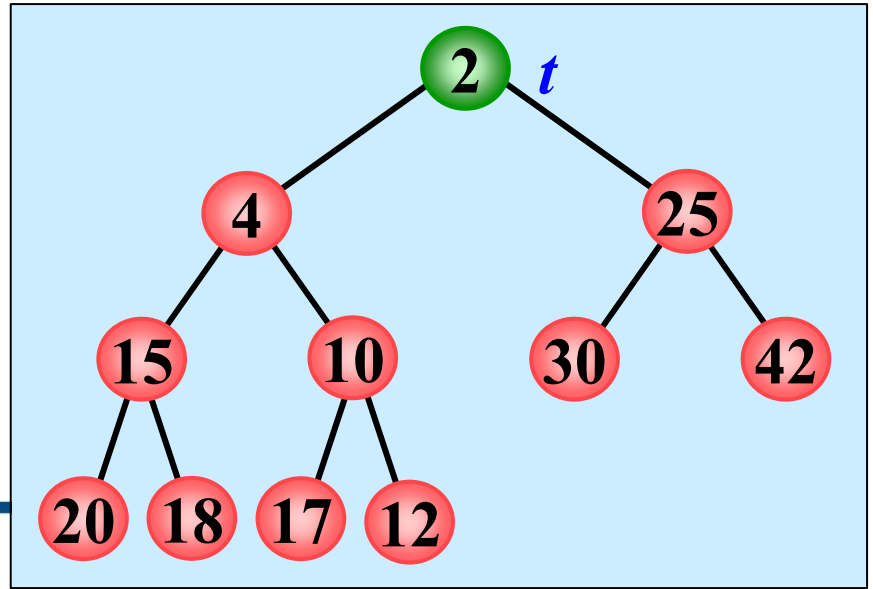
Example of insertion

- Insert 2









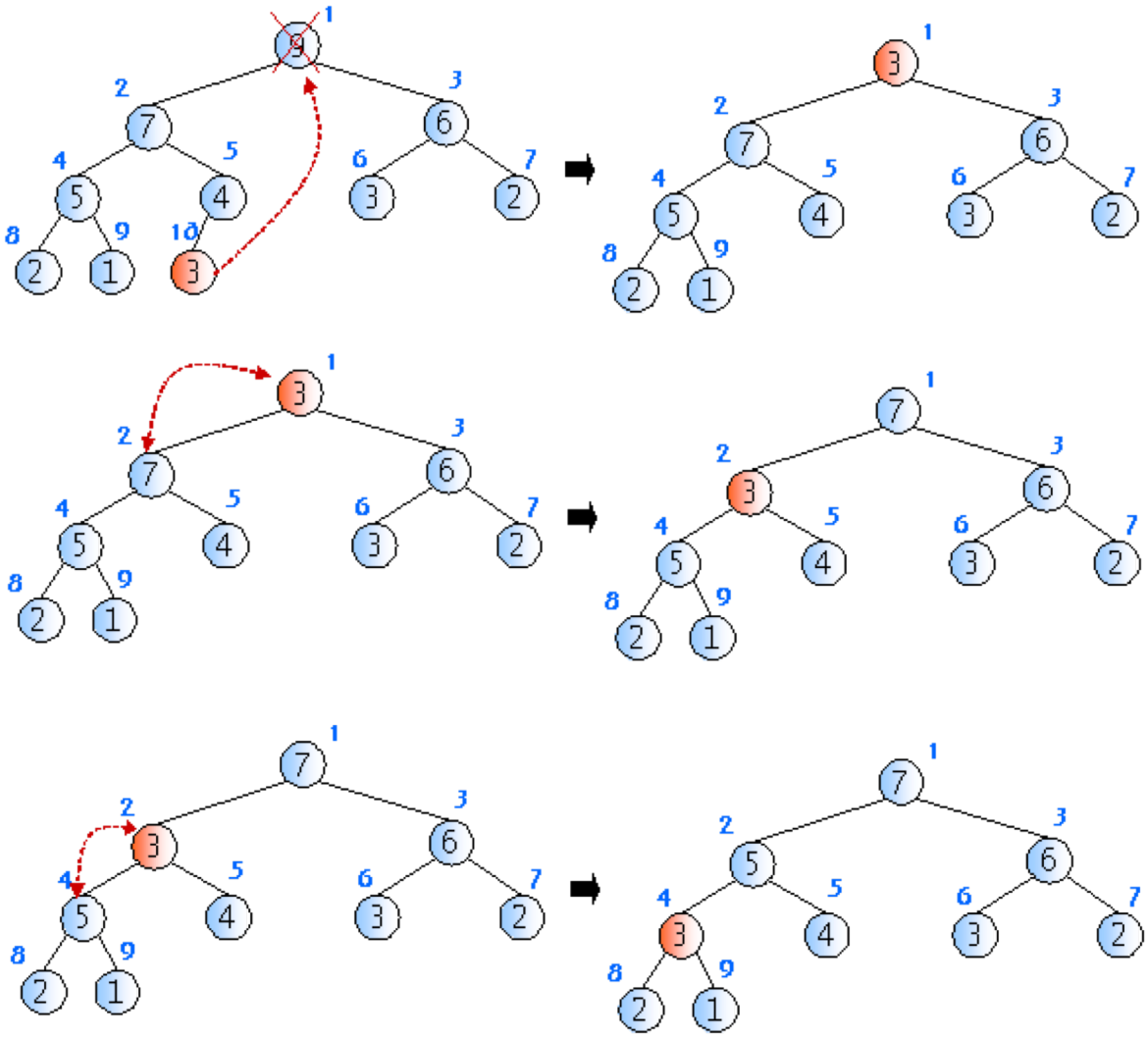
MAX-heap insertion

```
insert_max_heap(A, key)

heap_size ← heap_size + 1;        //힙의 크기 증가
i ← heap_size;                    //증가된 힙 크기 위치에 새로운 노드삽입
A[i] ← key;
while i ≠ 1 and A[i] > A[PARENT(i)] do //i가 루트 노드가 아니고, i번째 노드
                                       //  가 i의 부모 노드보다 크면
        A[i] ↔ A[PARENT];     //i번째 노드와 부모노드 교환
        i ← PARENT(i);        // 한 레벨 위로 올라함
```

Max-heap insertion

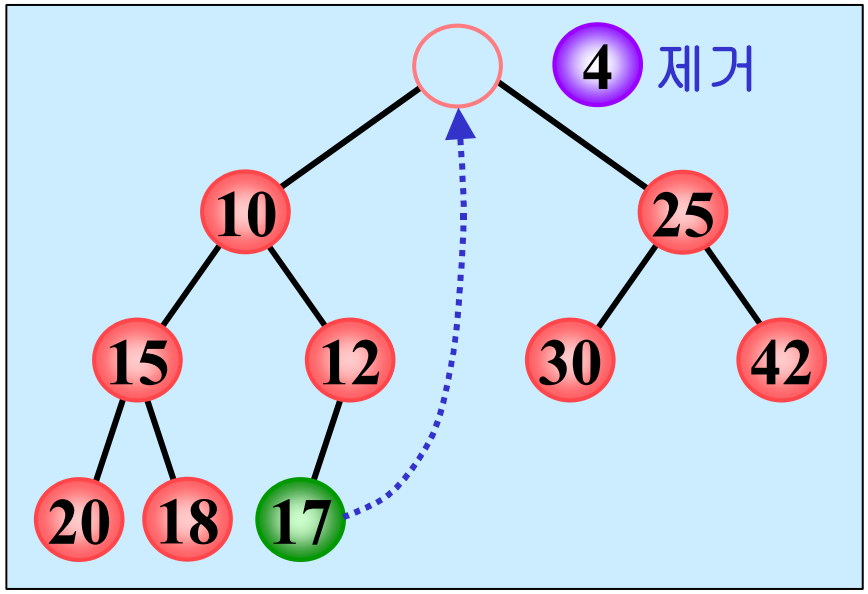
```
// 현재 요소의 개수가 heap_size인 힙 h에 item을 삽입한다.
// 삽입 함수
void insert_max_heap(HeapType *h, element item)
 {
    int i;
    i = ++(h->heap_size);

    //  트리를 거슬러 올라가면서 부모 노드와 비교하는 과정
    while((i != 1) && (item.key > h->heap[i/2].key)) {
                h->heap[i] = h->heap[i/2];
                        i /= 2;
    }
              h->heap[i] = item;      // 새로운 노드를 삽입
 }
```

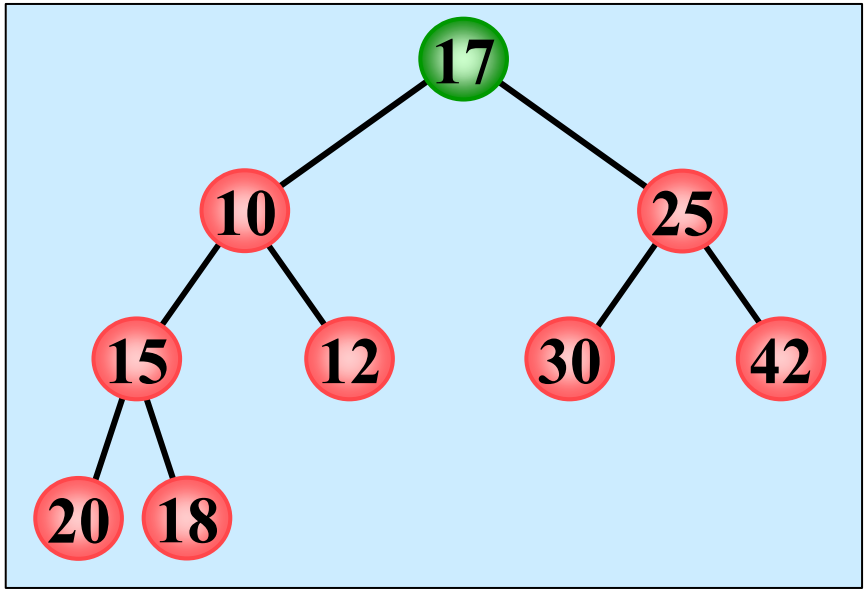
Heap delete

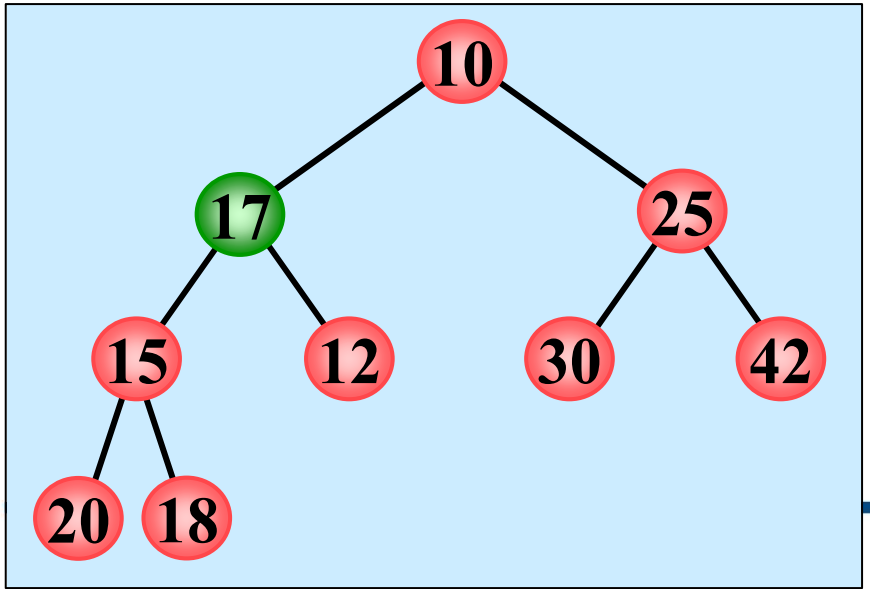


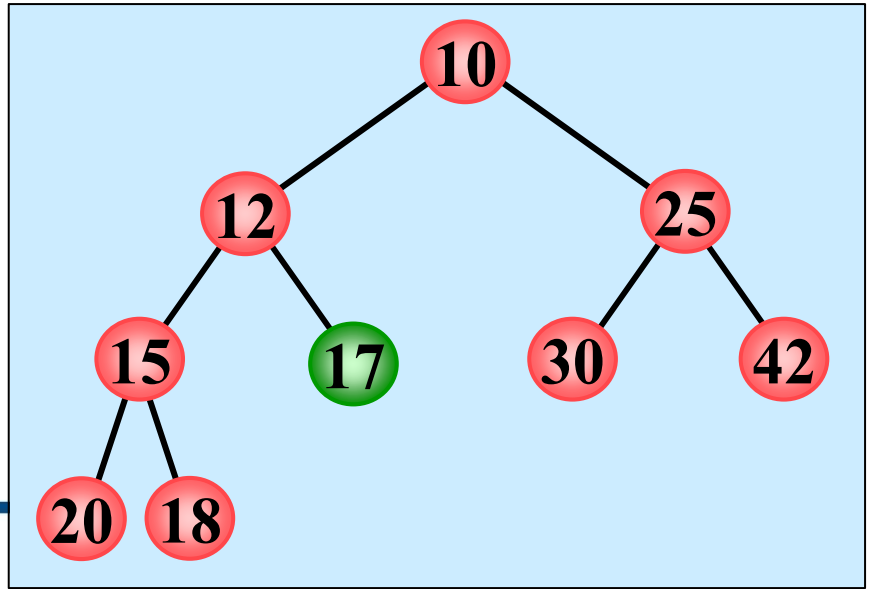
- **Time complexity of heap delete operation :**
  - → O(logn)

Heap delete


4
제거
10
25
15
12
30
42
20
18
17


17
10
25
15
12
30
42
20
18


10
17
25
15
12
30
42
20
18


10
12
25
15
17
30
42
20
18

MAX-Heap delete

```
delete_max_heap(A)

item ← A[1];
A[1] ← A[heap_size];
heap_size←heap_size-1;
i ← 2;
while i ≤ heap_size do
        if i < heap_size and A[i+1] > A[i] //(왼쪽자식과 오른쪽자식
비교)
                then largest ← i+1;   //오른쪽 자식이 더 크면
                else largest ← i;   // 왼쪽 자식이 더 크면
        if A[PARENT(largest)] > A[largest]//largest의 부모 노드가
                                          //largest보다 더 크면
                then break;
        A[PARENT(largest)] ↔ A[largest]; //서로 교환
        i ← CHILD(largest); //한 레벨 밑으로 내려감
return item; // 최대값 반환
```

MAX-heap delete

```
// 삭제 함수
element delete_max_heap(HeapType *h)
{
    int parent, child;
    element item, temp;

    item = h->heap[1];
    temp = h->heap[(h->heap_size)--];
    parent = 1;
    child = 2;
    while( child <= h->heap_size ) {
        // 현재 노드의 자식노드중 더 작은 자식노드를 찾는다.
        if( ( child < h->heap_size ) &&
            (h->heap[child].key) < h->heap[child+1].key)
            child++;
        if( temp.key >= h->heap[child].key ) break;
        // 한단계 아래로 이동
        h->heap[parent] = h->heap[child];
        parent = child;
        child *= 2;
    }
    h->heap[parent] = temp;
    return item;
 }
```

Heap complexity

- In the worst case of insert operations, we need to go up to the root node, so we need comparison and shift operations corresponding to the height of the tree. → O(logn)

- In the worst case, the deletion also has to go down to the lowest level, so it takes time as much as the height of the tree. →O(logn)

Complexity Comparison

- **Fast max or min operation**
- **Logarithmic complexity for other operations**

	Array (unsorted)	Linked list	Array (sorted)	Binary heap
Search	O(n)	O(n)	O(log n)	O(log n)
Insert	O(1)	O(1)	O(n)	O(log n)
Remove	O(n)	O(n)	O(n)	O(log n)
Max/min	O(n)	O(n)	O(1)	O(1)