Double-Ended Priority Queue

강지훈
jhkang@cnu.ac.kr
충남대학교 컴퓨터융합학부

Double-Ended Priority Queue

Double-Ended Priority Queue

- It supports the following operations:
  1. Add an element with an arbitrary key.
  2. Remove an element with the largest key.
  3. Remove an element with the smallest key.

Abstract Class "DoubleEndedPriorityQ"

```
public abstract class  DoubleEndedPriorityQ<T>
{
    public  DoubleEndedPriorityQ () ;
    public  DoubleEndedPriorityQ (int givenCapacity) ;

    public  abstract boolean      isEmpty () ;
    public  abstract boolean      isFull () ;
    public  abstract int          size () ;
    public  abstract boolean      add (Element anElement) ;
    public  abstract T            max() ;
    public  abstract T            removeMax () ;

    public  abstract T            min() ;
    public  abstract T            removeMin () ;
}
```

Deap

Double-Ended Priority Queue 의 구현

❑ Deap 을 이용한 구현

```
public class DoubleEndedPriorityQByDeap<T extends Comparable<T>>
    extends DoubleEndedPriorityQ<T>
{
    private int     _capacity ;
    private int     _size ;
    private T[]     _heap ; // Deap or Min_Max Heap
    ......


}
```

Deap

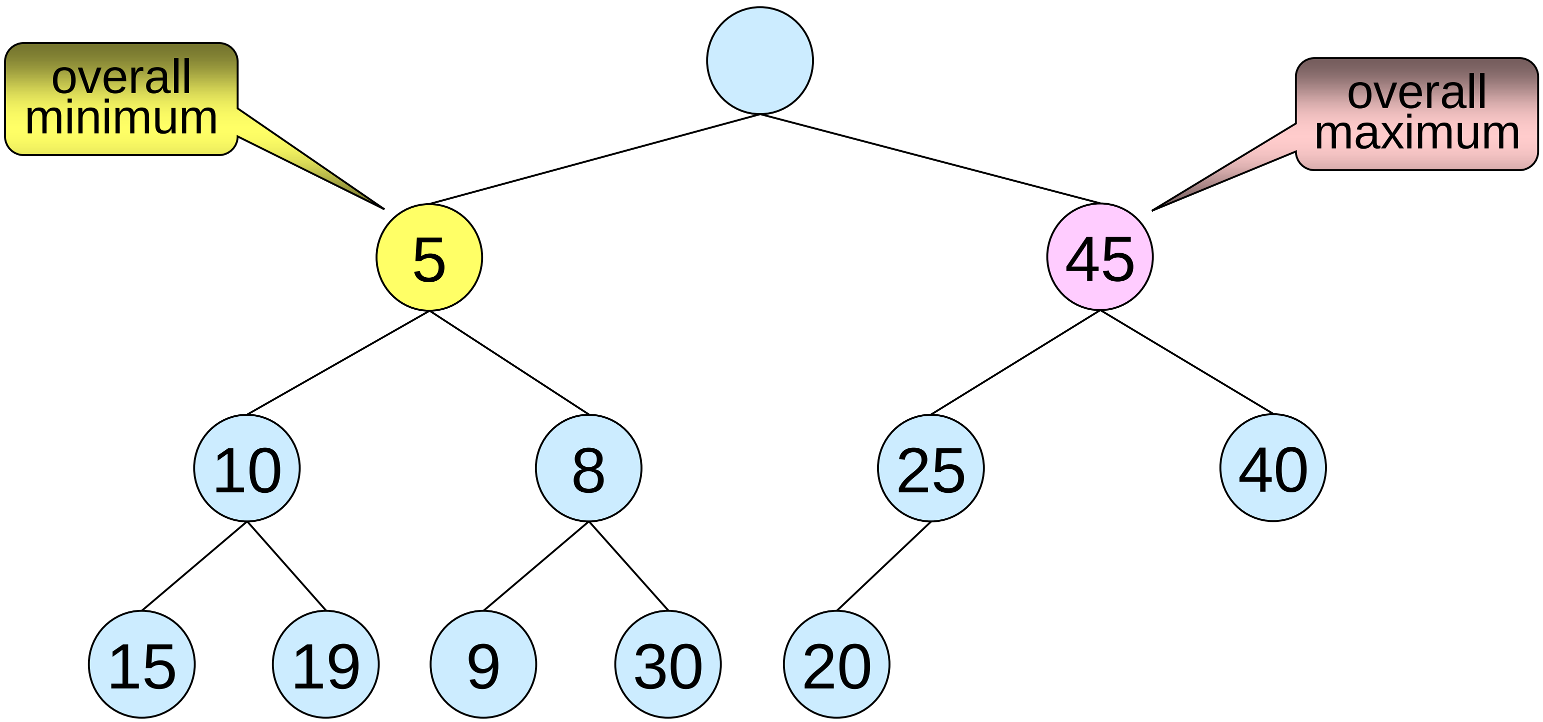
Double-ended heap

- supports the double ended priority queue operations.
  - add
  - remove min
  - remove max

O(log n) for each operation

- n is the size of a deap.

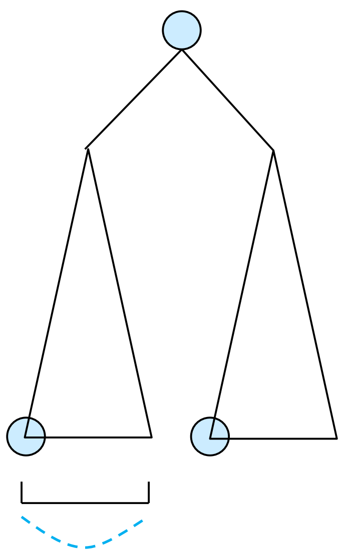
Deaps



Deap

- A deap is a complete binary tree that is either empty or satisfies the following properties:
  1. The root contains no element.
  2. The left subtree is a min-heap.
  3. The right subtree is a max-heap.
  4. If the right subtree is not empty, then let i be any element position in the left subtree.
     Let j be the corresponding element position in the right subtree.
     If such a position j does not exist,
     then let j be the element position in the right subtree that corresponds to the parent of i .
     Then, the key of element at i is less than or equal to the key of element at j.

How to compute the value of j :



2^{k-2} nodes

Level	# of nodes at the level	leftmost node #
1	2^0	2^0
2	2^1	2^1
k-1	2^{k-2}	2^{k-2}
k	2^{k-1}	2^{k-1}

$$j = i + 2^{k-2}$$

$$2^{k-1} \le i < 2^k$$

$$k - 1 \le \log_2 i < k$$

$$k - 1 = \lfloor \log_2 i \rfloor$$

$$k = \lfloor \log_2 i \rfloor + 1$$

$$j = i + 2^{(\lfloor \log_2 i \rfloor + 1) - 2}$$

$$= i + 2^{\lfloor \log_2 i \rfloor - 1}$$

Consequently,

$$j = i + 2^{\lfloor \log_2 i \rfloor - 1} ;$$

$$\text{if } (j > n)\ j \mathrel{/}= 2 ;$$

Functions for Add

- *isMaxHeapPosition(i)*
  - Returns TRUE iff i is a position in the max-heap of the deap.

    $k = \lfloor \log_2 i \rfloor + 1$; // k is the level of position i in the tree.

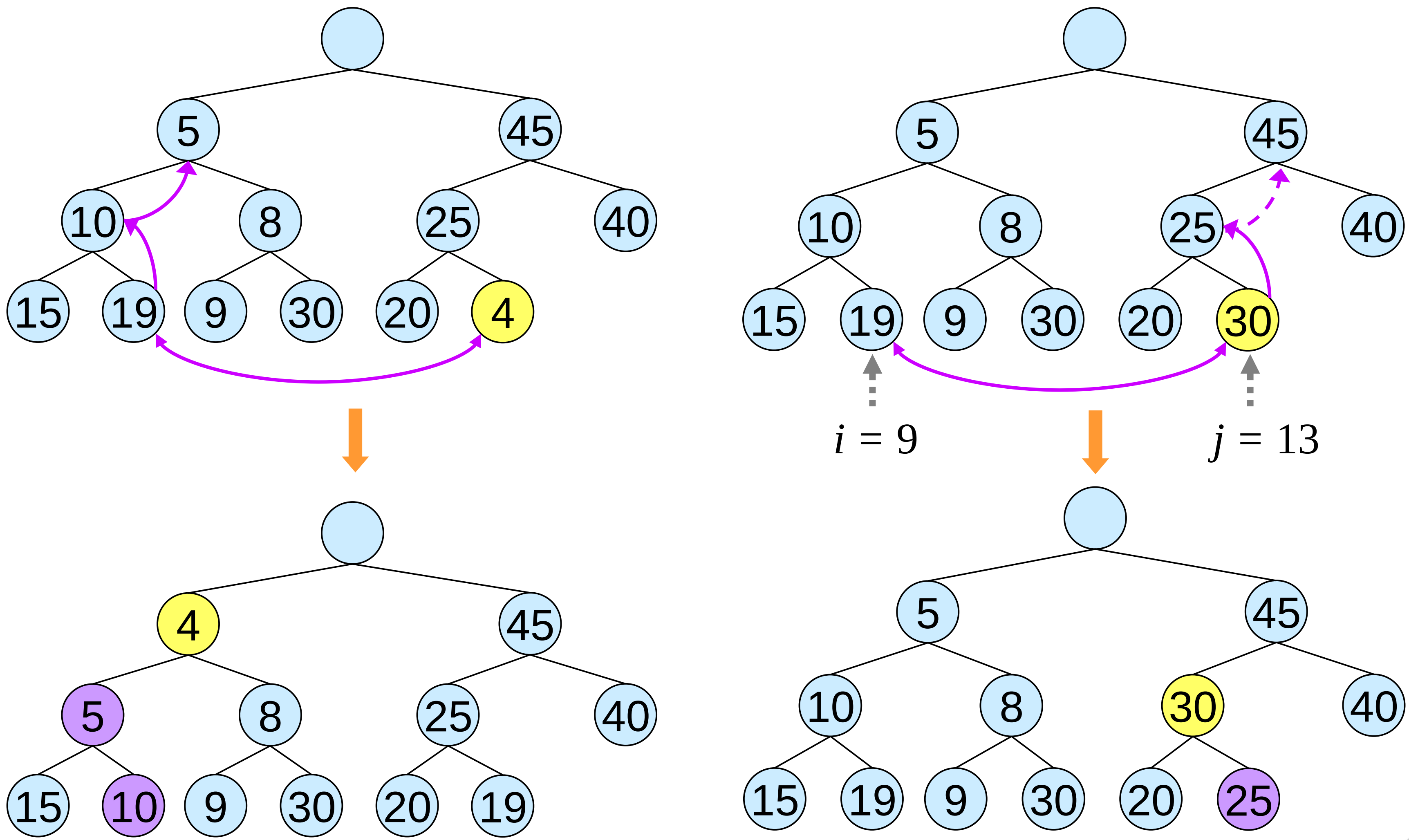
    return $(i \geq (2^{k-1} + 2^{k-2}))$;

    // $(2^{k-1} + 2^{k-2})$ is the smallest number in the max side.
- *maxPartner(i)*
  - Computes the max-heap node that corresponds to the min-heap position i. The value is:

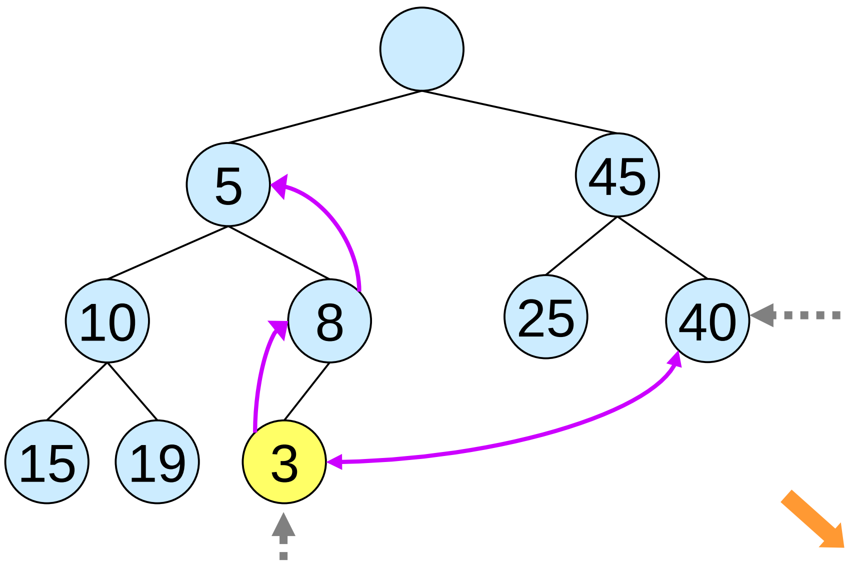
    $j = (i + 2^{\lfloor \log_2 i \rfloor - 1})$;

    if ($j > n$) then $j = j / 2$;
- *minPartner(j)*
  - Computes the min-heap position that corresponds to the max-heap position j. The value is $(j - 2^{\lfloor \log_2 j \rfloor - 1})$.
- *addToMinSide()*, and *addToMaxSide()*

Add to MAX side of a Deap

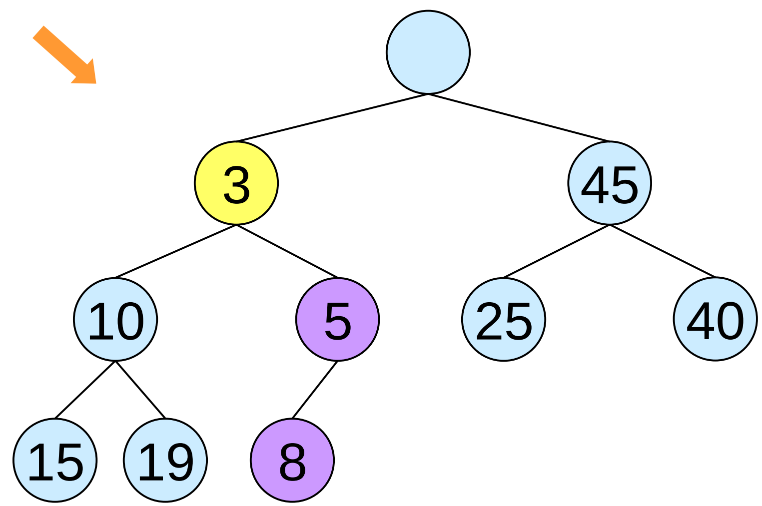


Add to MIN side of a Deap [1]

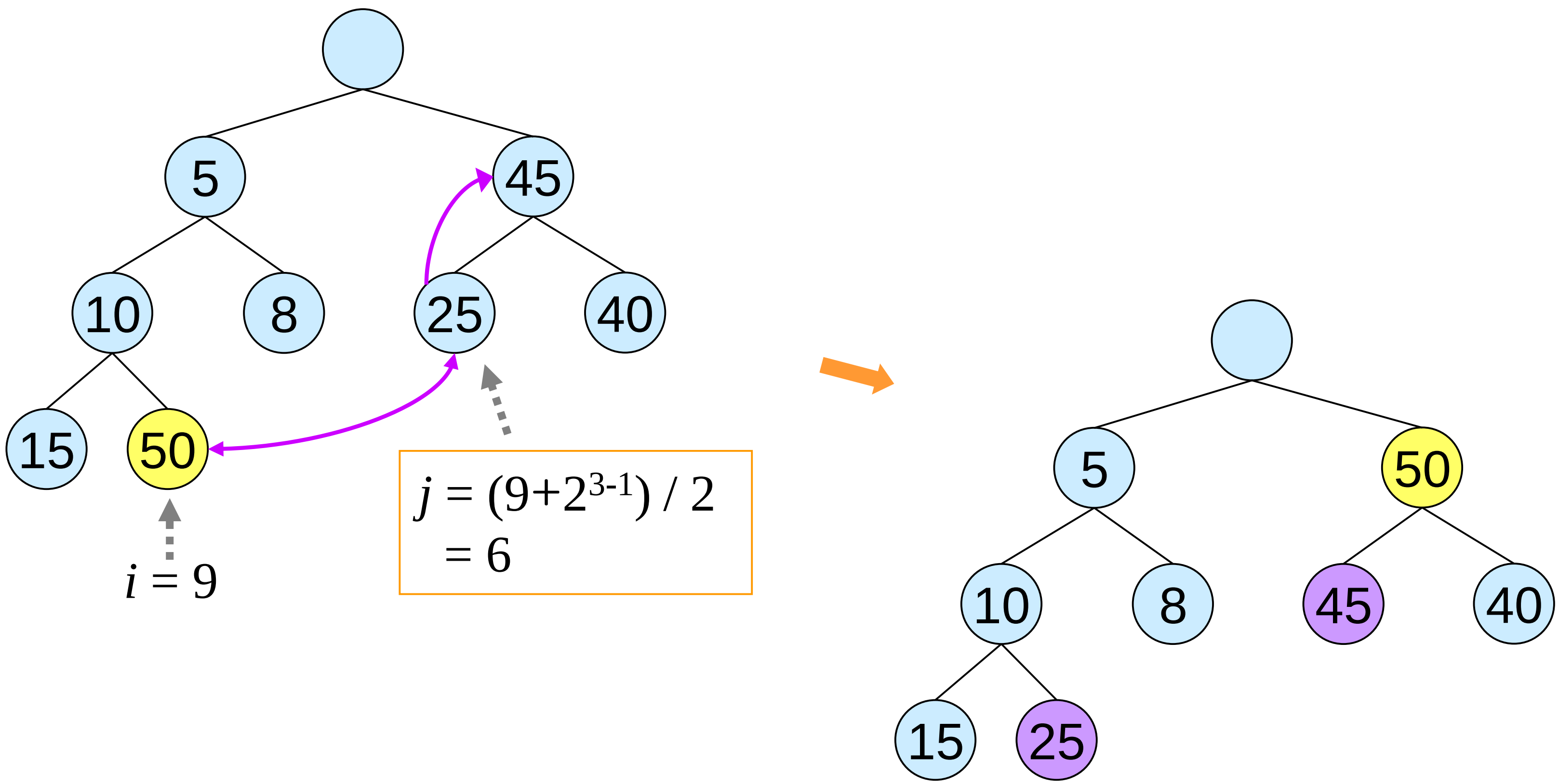


$i = 10$

$$
\begin{aligned}
j &= (i + 2^{\lfloor \log_2 i \rfloor - 1}) / 2 \\
  &= (10 + 2^{3-1}) / 2 \\
  &= 7
\end{aligned}
$$



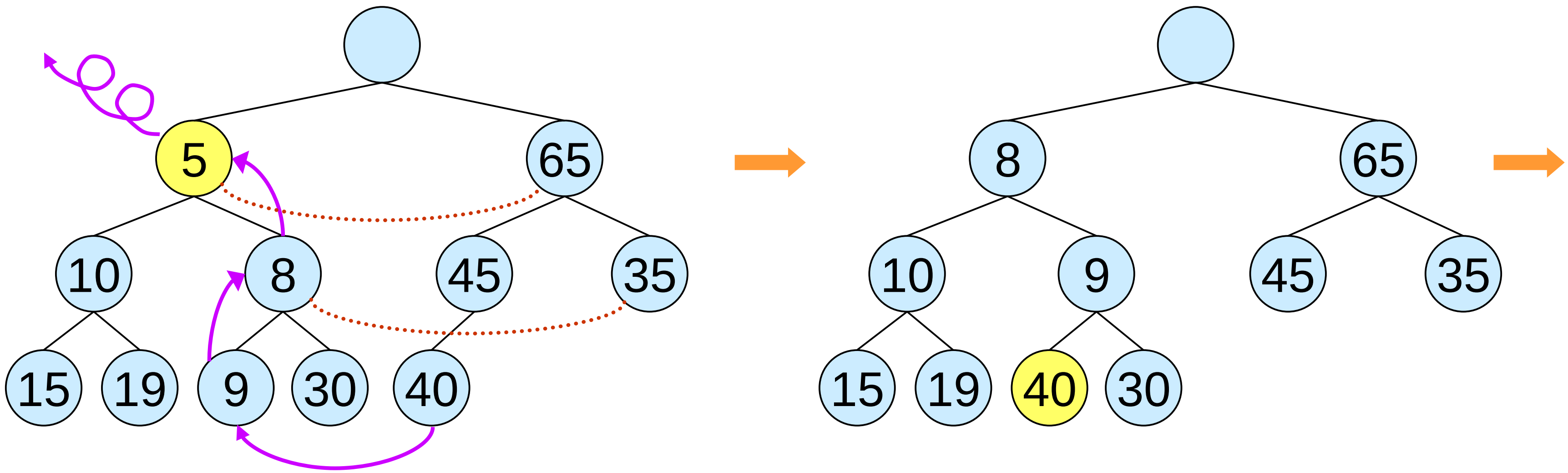
Add to MIN side of a Deap [2]



- Analysis of Add : O(log n)

Remove Min/Max [1]

Remove Min

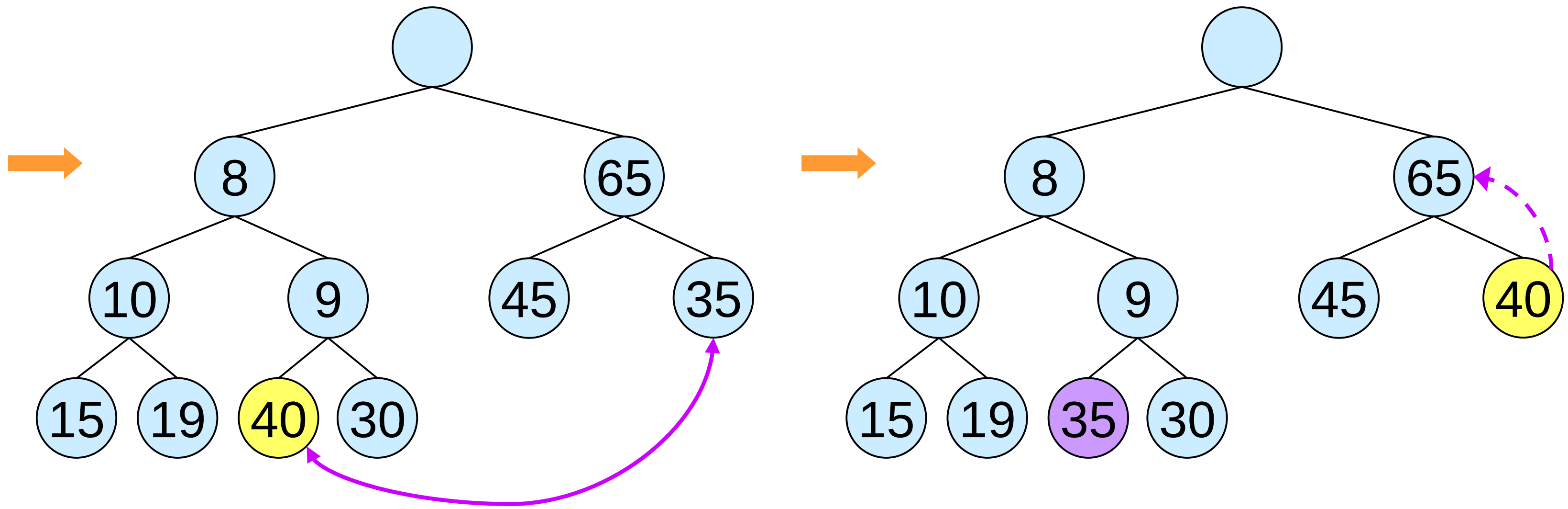


- Element 8 can be moved up since 8 ≤ 35 ≤ 65.
- Element 9 can be moved up since 9 ≤ 35.

At this time, we should add 40 on the MIN side.

Remove Min/Max [2]

Remove Min (Continued): Modified Insertion



- Analysis of Remove : O(log n)

Remove Max

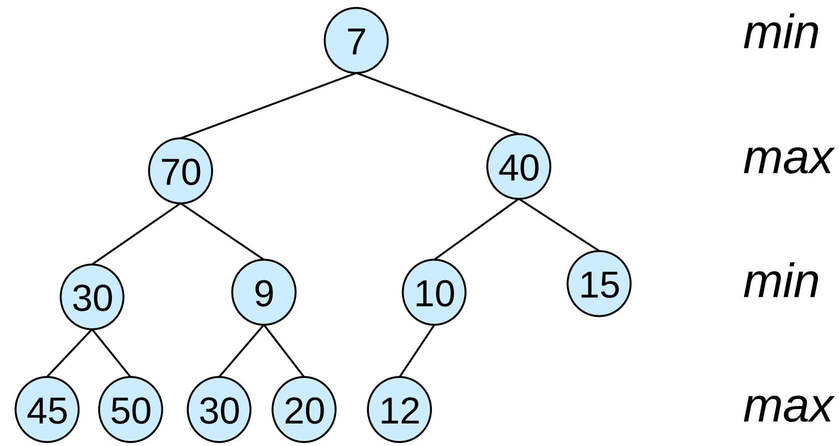
- It is performed in a similar manner.

Min-Max Heap

Doubled-Ended Priority Queue 의 구현

Min-Max Heap [1]

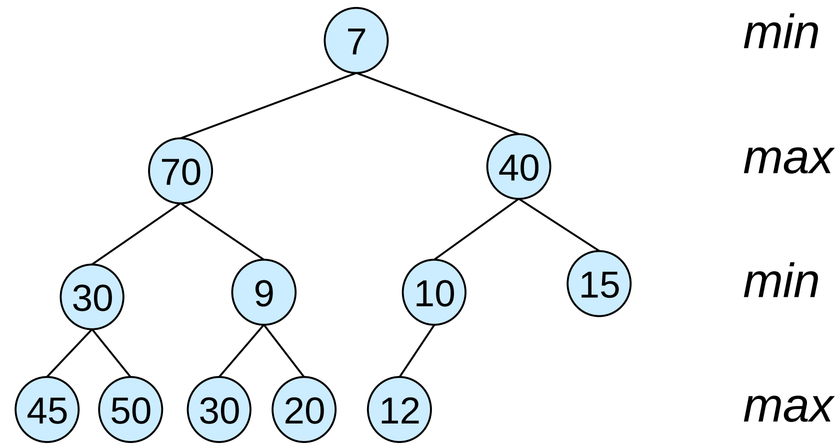
- A complete binary tree such that if it is not empty, each element has a field, called *key*.
- Alternating levels of this tree are min levels and max levels, respectively.
- The root is on a min level.



- Let x be a node in a min-max heap.
  - If x is on a min level, then the element in x has the minimum key from among all elements in the subtree with root x.
    - We call this node a min node.
  - Similarly, if x is on a max level, then the element in x has the maximum key from among all elements in the subtree with root x.
    - We call this node a max node.

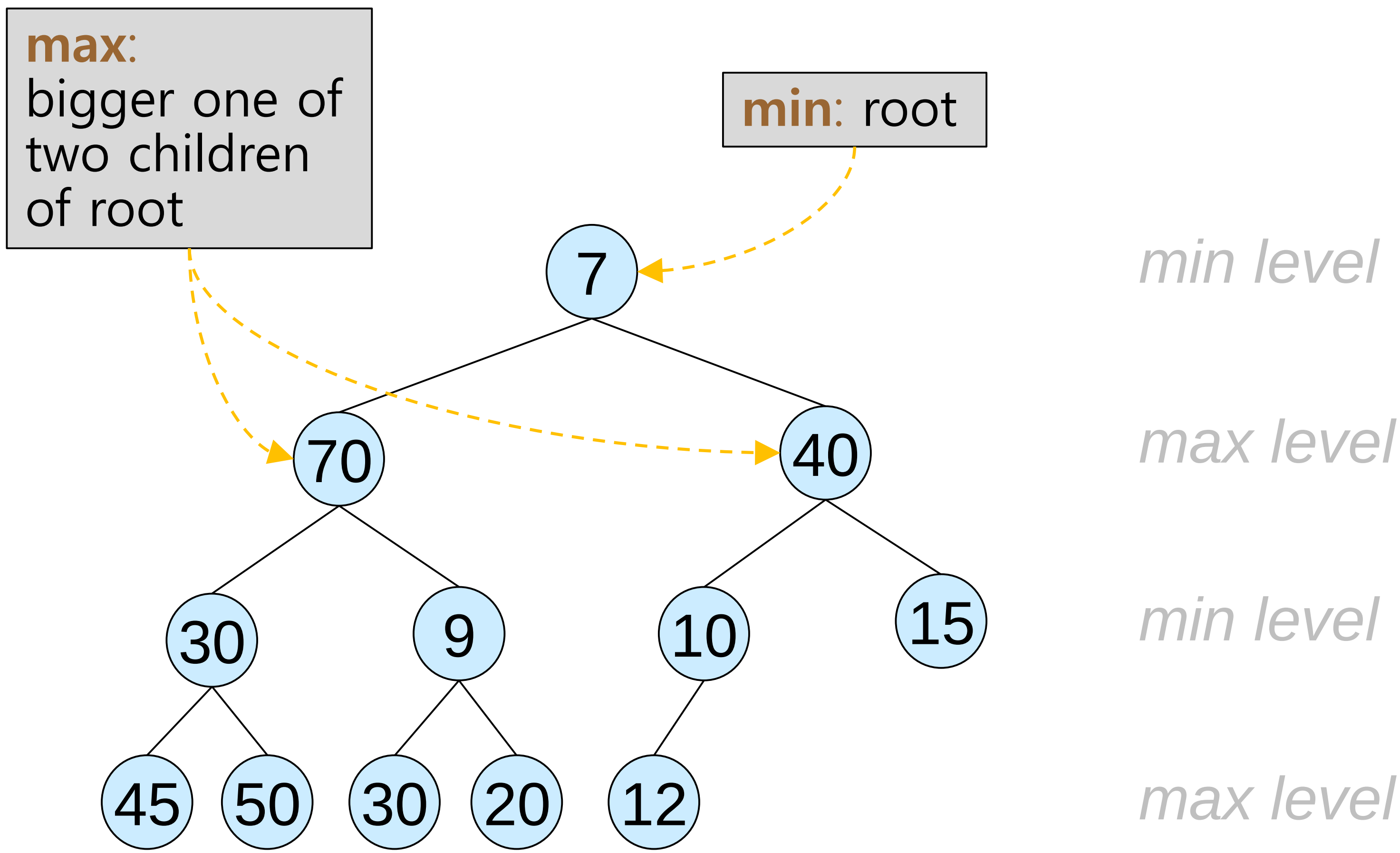
Min-Max Heap [2]

- A complete binary tree such that if it is not empty, each element has a field, called *key*.
- Alternating levels of this tree are min levels and max levels, respectively.
- The root is on a min level.



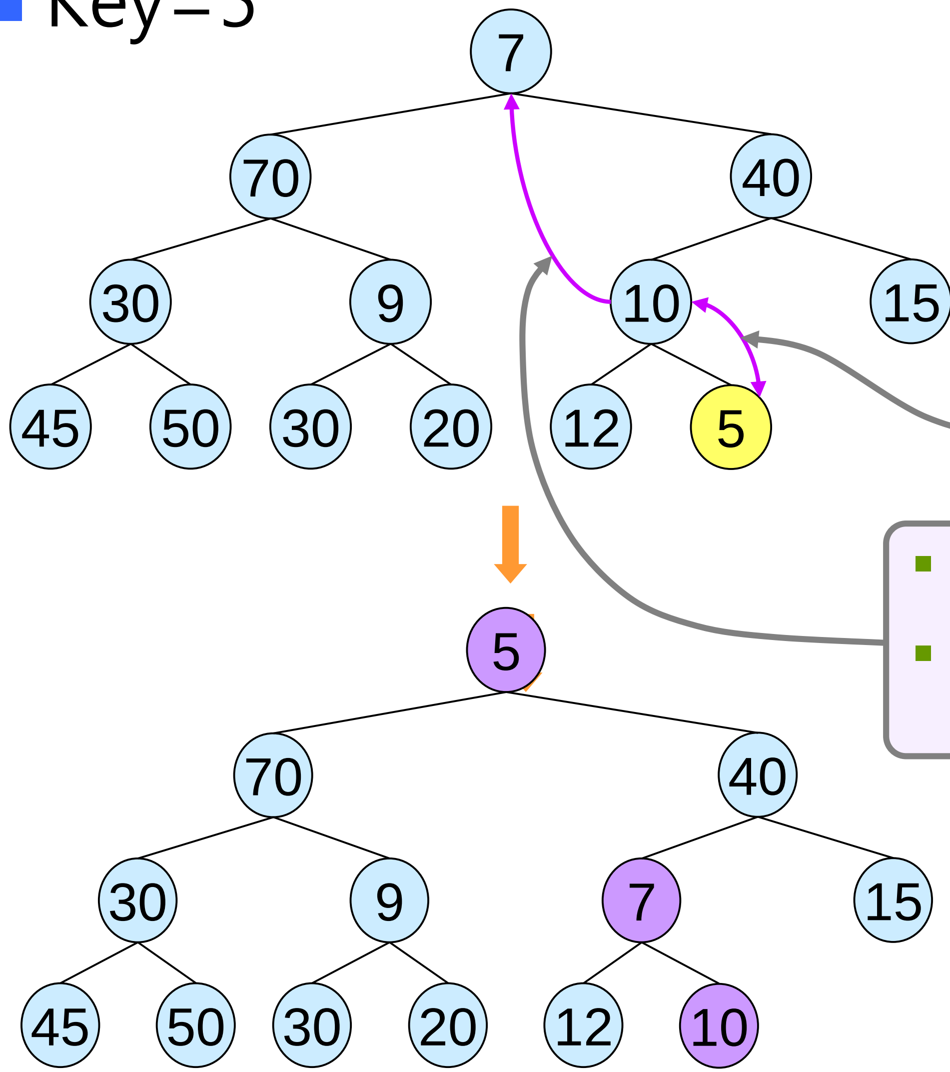
- Let x be a node in a min-max heap.
  - If x is on a min level, then the element in x has the minimum key from among all elements in the subtree with root x.
    - We call this node a min node.
  - Similarly, if x is on a max level, then the element in x has the maximum key from among all elements in the subtree with root x.
    - We call this node a max node.

❑ 전체의 Min 과 Max 의 위치



Add [1]

- Key=5

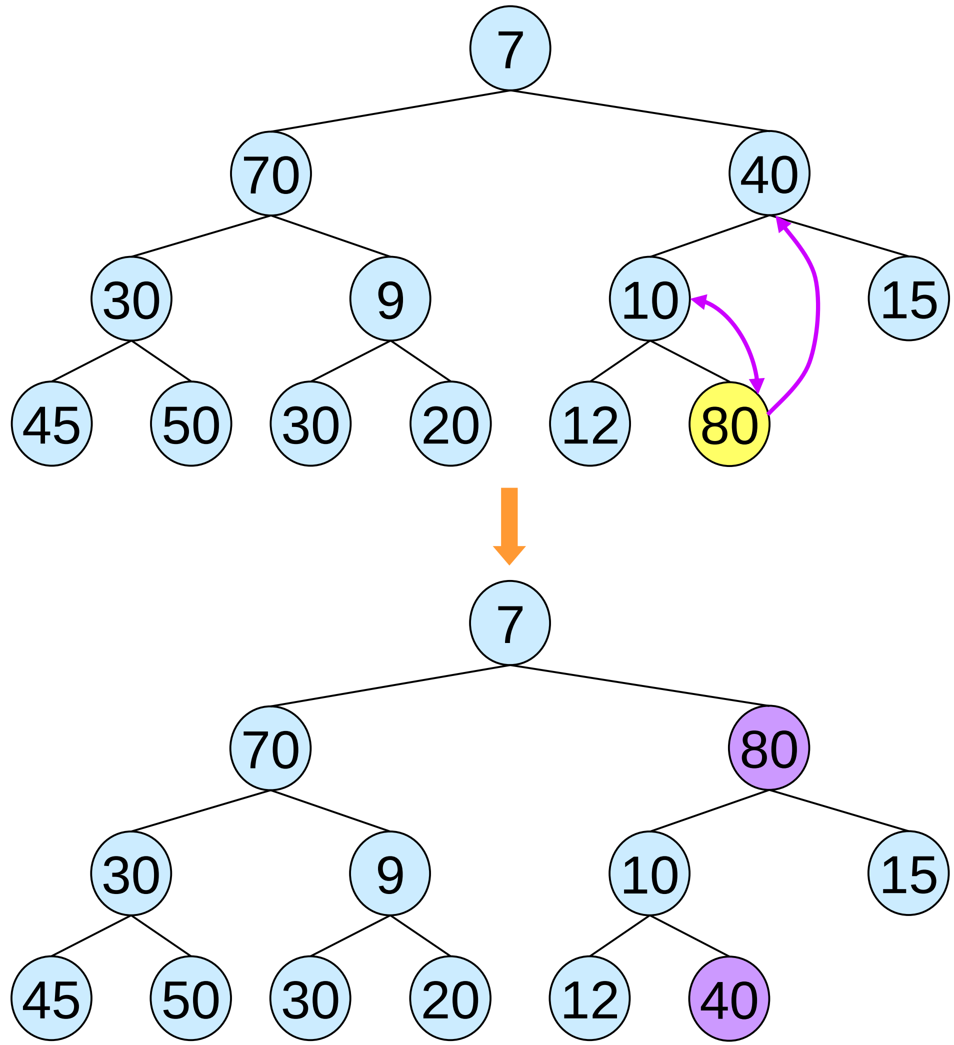


- Determine whether the added node will be located on the min level or on the max level by comparing its parent node.
- In this case, '5' will be put to the min level.

- Then, '5' is exchanged with '10'.
- '5' will go up only through the min levels.

Add [2]

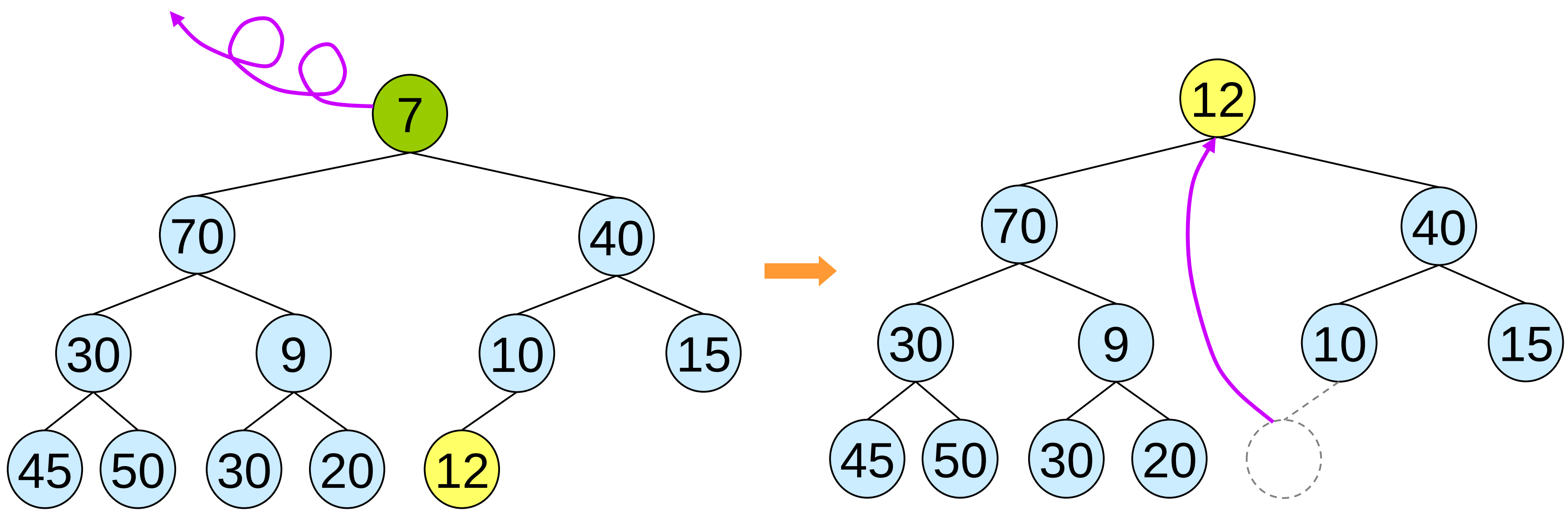
- Key=80



- Analysis of Add : $O(\log n)$

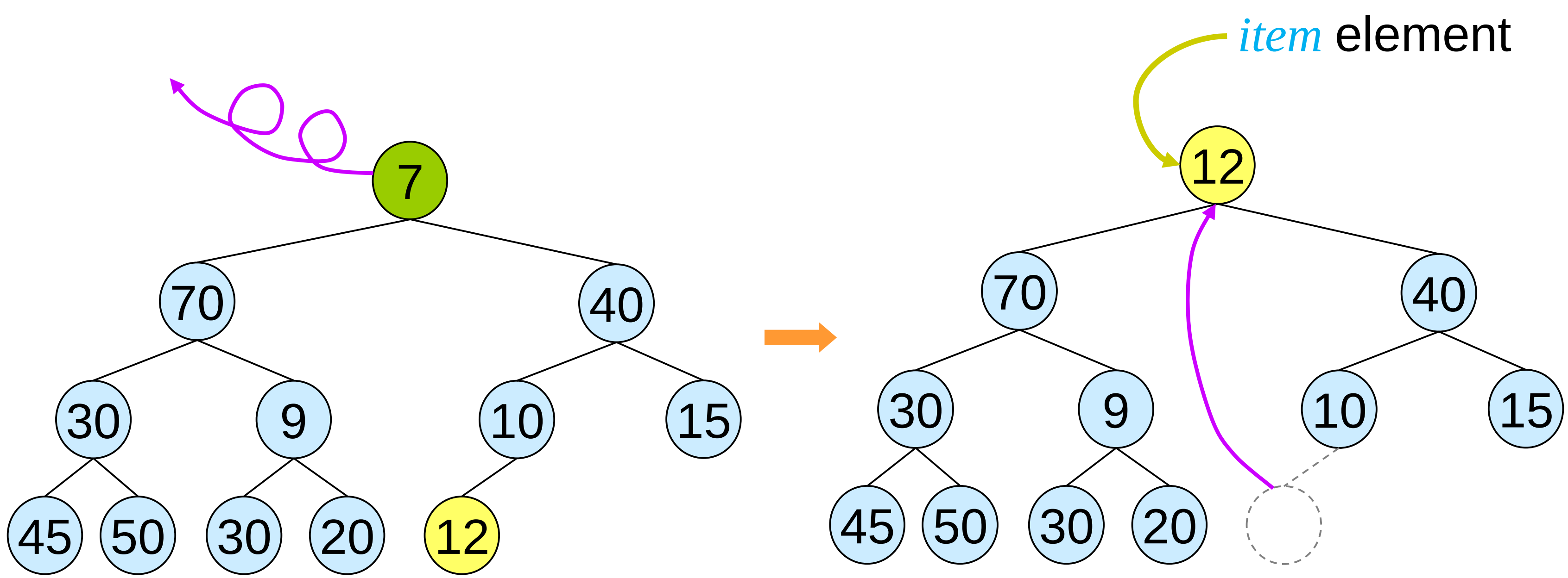
Remove MIN

- The root has the smallest key.
  - So, the root is removed as the min element.
- The last element of the heap is removed and it is added again into the root.
  - We should adjust the heap.



Remove MIN

- The root has the smallest key.
  - So, the root is removed as the min element.
- The last element of the heap is removed and it is added again into the root.
  - We should adjust the heap.



Adjusting the heap after removing MIN

- The root has no children.
  - The tree becomes empty after removing.

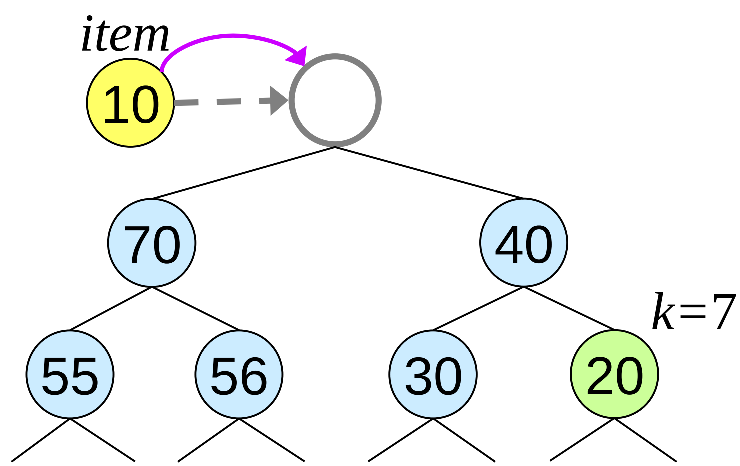
Adjusting the heap after removing MIN

- The root has no children.
  - The tree becomes empty after removing.
- The root has at least one child.
  - The smallest key is in one of the children or grand-children of the root. Let this be node k.

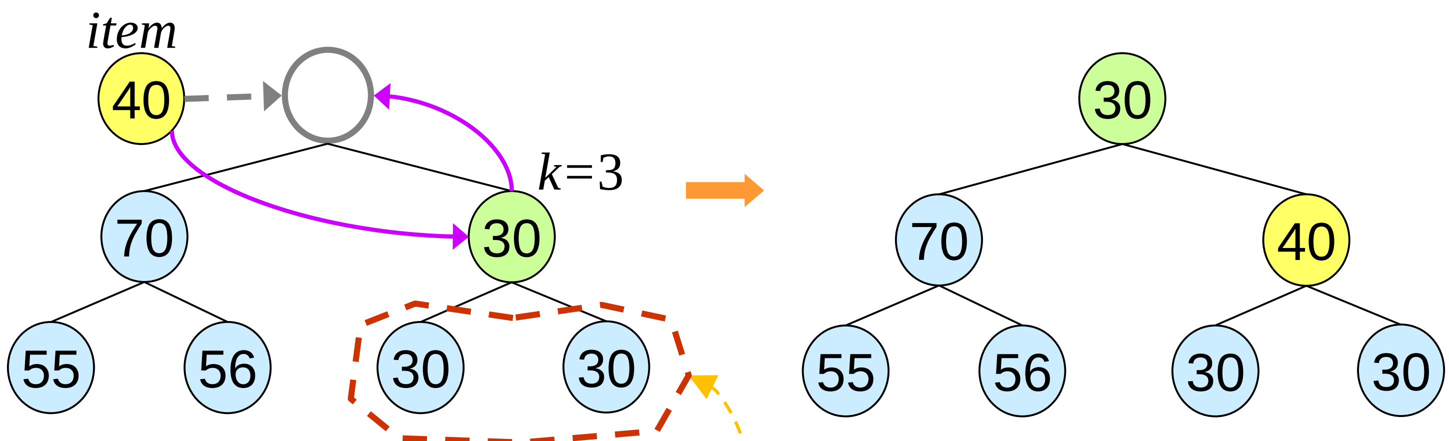
(a) $item.key \leq heap[k].key$

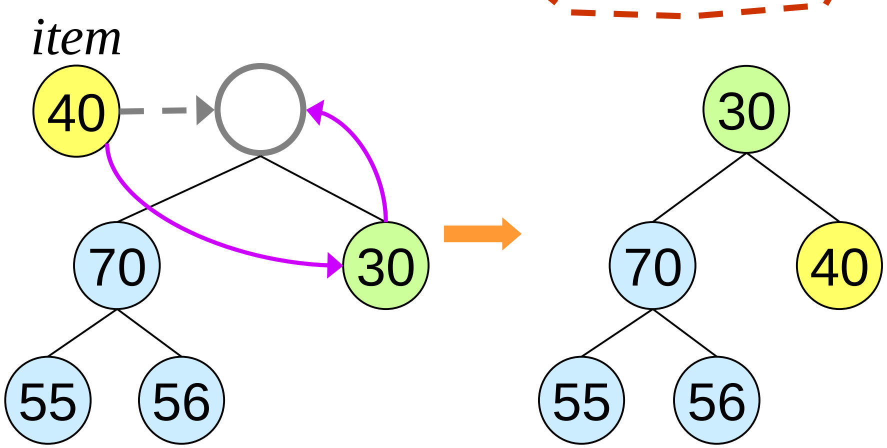
In this case, there is no element in the heap with key smaller than $item.key$.

So, $item$ may be added into the root.



(b) *item.key* > *heap*[*k*].*key* and node *k* is a child of the root.
Then *k* is a max node. Hence, node *k* has no descendants with key larger than *heap*[*k*].*key*.
So, the element *heap*[*k*] may be moved to the root and item added into node *k*.





Note: Actually if node *k* has descendents, then they have the same key value as the *heap*[*k*].*key*.

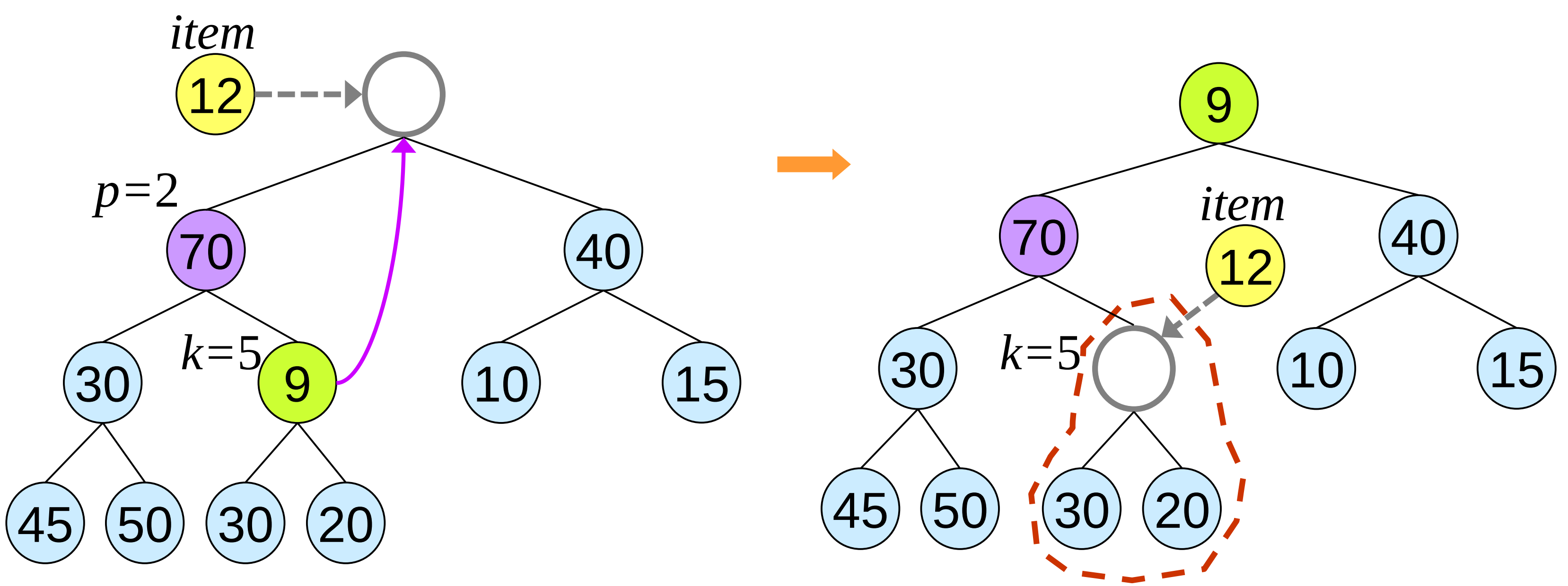
(c) *item.key* > *heap*[*k*].*key* and *k* is a grandchild of the root.

heap[*k*] may be moved to the root.

Let *p* be the parent of *k*. (i.e., $p = \lfloor k/2 \rfloor$)

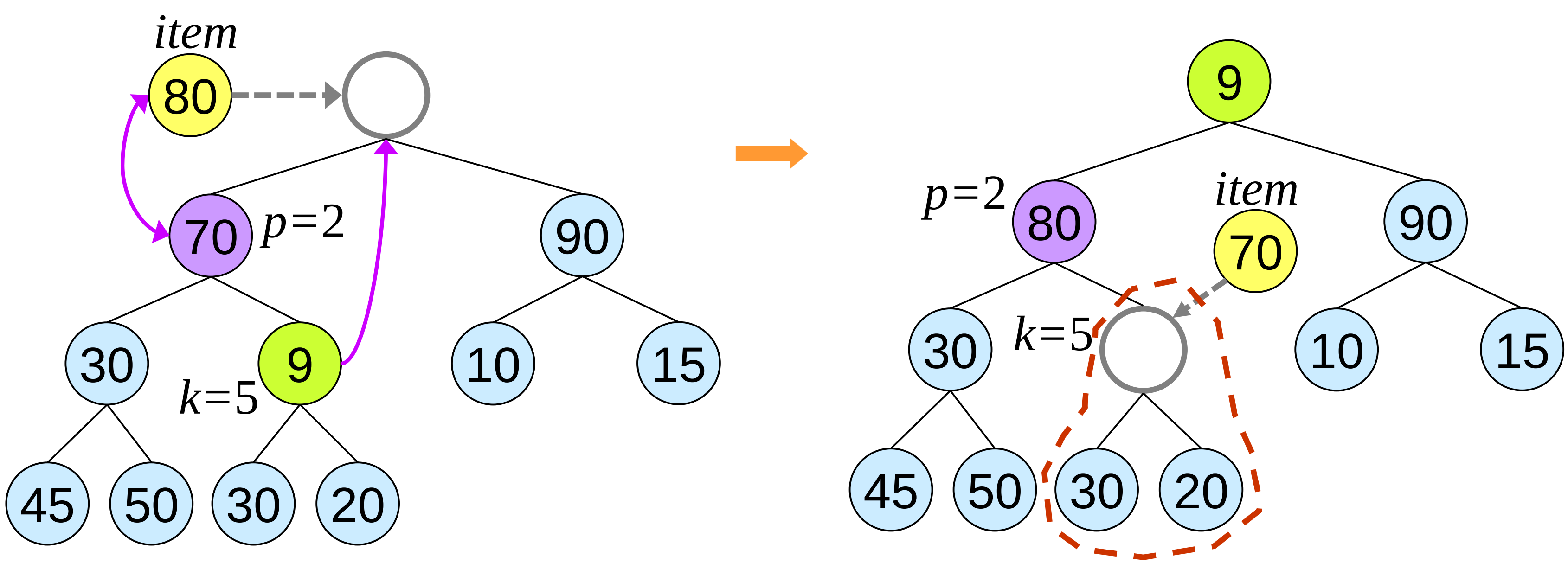
① *item.key* ≤ *heap*[*p*].*key*

Repeat the above adjusting process for the sub min-max heap with root *k*.



② *item.key* > *heap*[*p*].*key*

item and *heap*[*p*] are interchanged.
The max node *p* contains the largest key in the sub-heap with the root *p*.
Repeat the above adjusting process for the sub min-max heap with root *k*.



- Analysis of Remove MIN : O(log *n*).

Remove Max

- Similar to Remove MIN

- Analysis of Remove MAX : $O(\log n)$.

End of "Double-Ended Priority Queue"