Ch. 10 Priority Queues

- Static data sets
  - Data do not change once constructed
- Dynamic data sets
  - Data sets can grow or shrink
  - Dictionary (table)
    - A dynamic data set that support insertion, deletion, and membership test
    - We can use arrays, linked lists, search trees, hash tables, etc
      (inefficient)
  - Priority queue
    - A dynamic data sets that support
      insertion, deletion, and retrieval of max element
    - We can use arrays, linked lists, search trees, **heap**s, etc
      (inefficient)

ADT *Priority Queue* Operations

- Create an empty priority queue
- Determine whether the priority queue is empty
- Add a new item to the priority queue
- Retrieve and then remove the item w/ the highest priority value

비교

- Deletion
  - Table은 search key 제공
  - Priority queue는 search key 사용 안함
    - Priority가 가장 높은 item만 delete 가능함
- Insertion
  - Table과 priority queue 둘 다 key 사용함
- Key 값 중복
  - Table은 불허
  - Priority queue는 허용

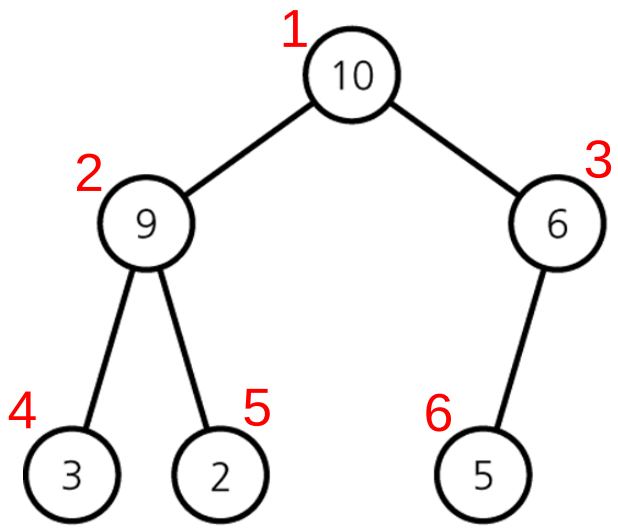
Heap : A Representative Priority Queue

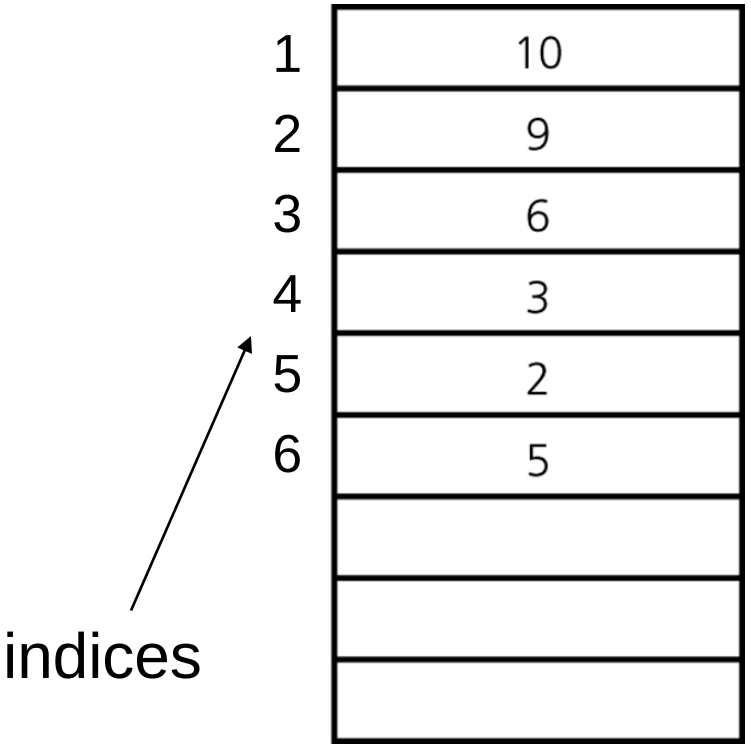
- A heap is a complete binary tree
  that is empty
  or
  the key of each node is greater than or equal to
  the keys of both children (if any)

Heap property라 한다

- The root has the largest key
- Maxheap : minheap
  - The root has max key : min key
- 여기서는 maxheap으로

A Heap w/ Array Representation





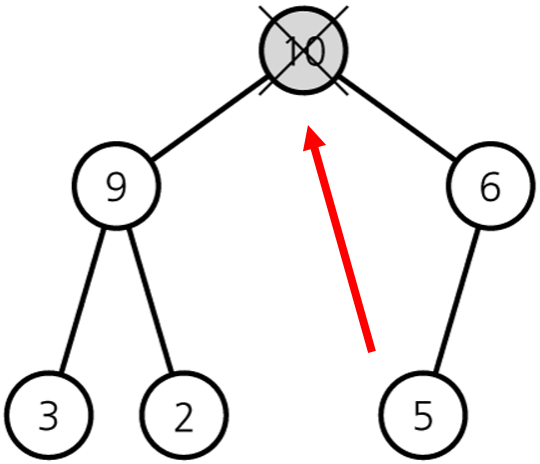
Node i's children: 2i, 2i+1
Node i's parent: ⌊i/2⌋

* Array는 그냥 complete binary tree로 볼 수 있다

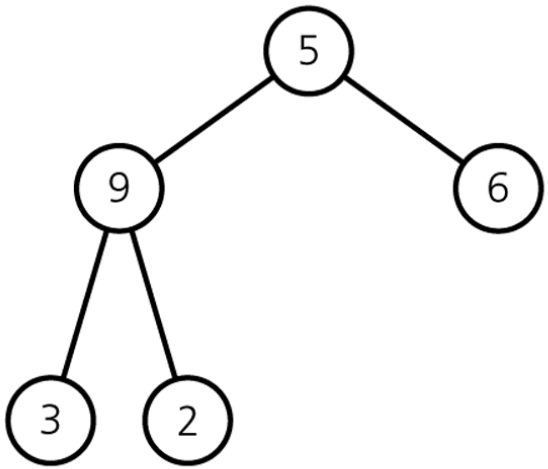
Deletion

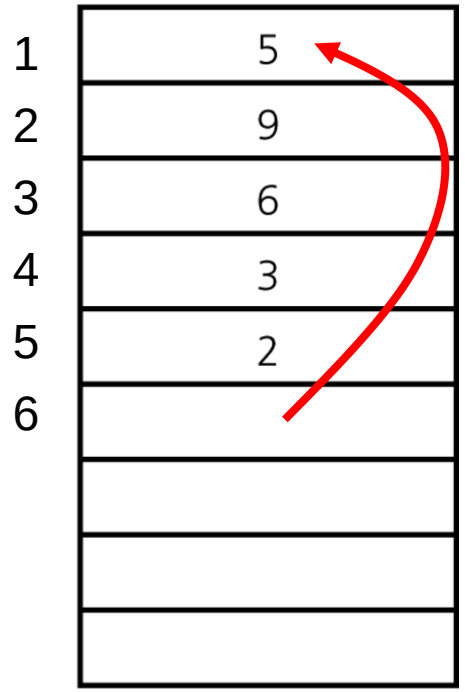
1. Return the root item
2. Remove the last node and move it to the root
3. Percolate down until the heap is valid

Deletion Example

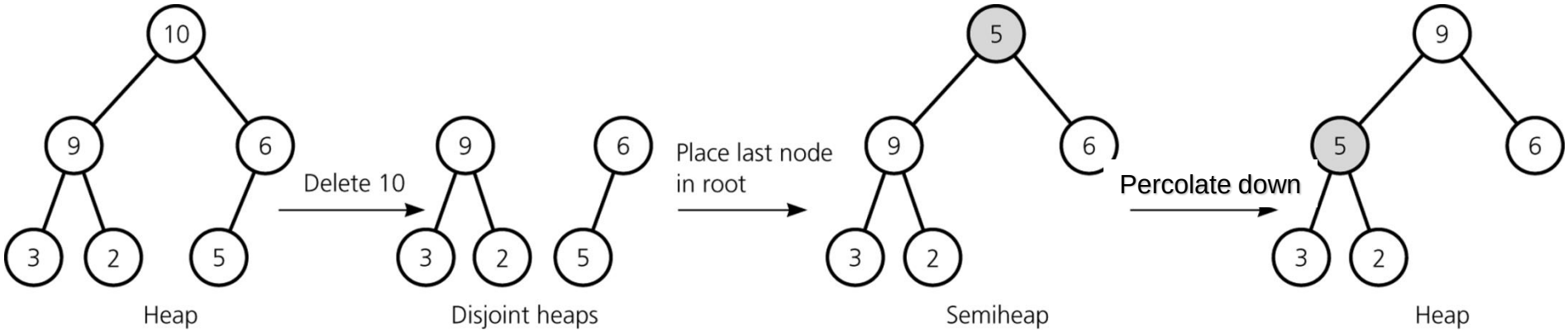


1	10
2	9
3	6
4	3
5	2
6	5

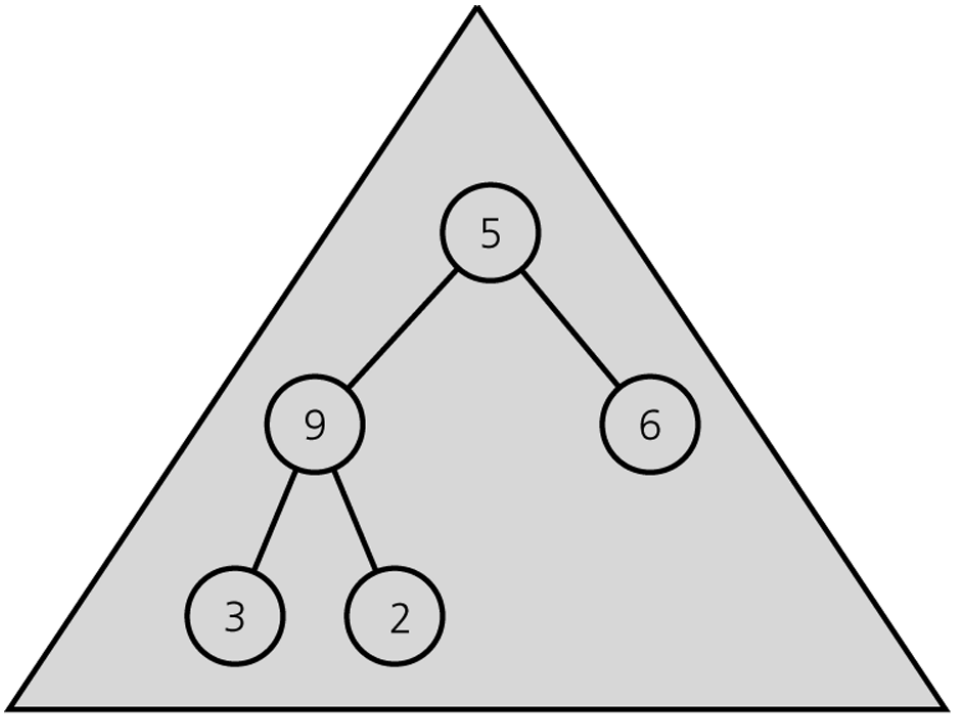




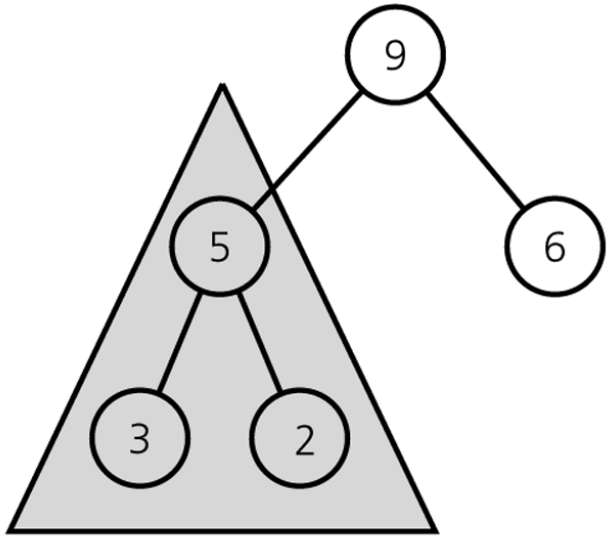
Heap property가 깨졌다. 수선해야 한다.


10
9
6
3
2
5
Heap
Delete 10
9
6
3
2
5
Disjoint heaps
Place last node
in root
5
9
6
3
2
Semiheap
Percolate down
9
5
6
3
2
Heap

Recursive Calls to *percolateDown*



First semiheap passed tc **percolateDown**



Second semiheap passed to **percolateDown**

```
heapDelete(A[], size) {
        // Keys are in A[1 … size]
        if (!heapIsEmpty( )) {
                rootItem ← A[1];
                A[1] ← A[size]; // move the last node
                size--;
                percolateDown(A, 1, size);
                return rootItem;
        } else {
                return null;
        }
}
```

```
percolateDown (A[], i, n) {
        // Percolate down w/ A[i] as the root
        // A[n]: last item, boundary
        child ← 2*i ; // left child
        rightChild ← 2*i + 1; // right child
        if (child <= n) {
                if ((rightChild <= n) && (A[child] < A[rightChild])) {
                        child ← rightChild;  // index of larger child
                }
                if (A[i] < A[child]) {
                        Swap A[i] and A[child];
                        percolateDown(A, child, n);
                }
        }
}
```

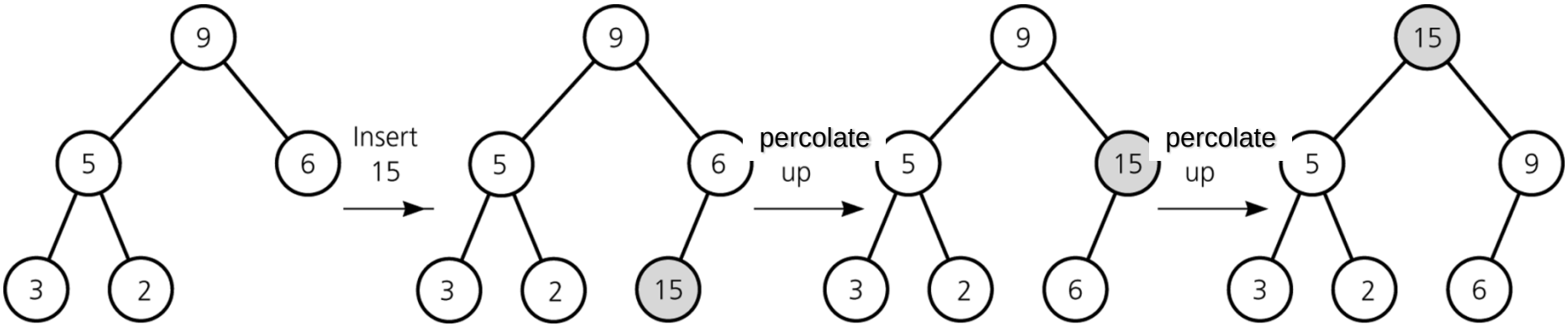
Insertion

1. Insert an item into the bottom of the complete tree
2. Percolate up until the heap is valid

```
heapInsert(A[], x, n) { // Insert item x on a heap A[1 … n]
        i ←n+1;
        A[i] ←x;
        parent ←i/2;
        while ((parent >= 1) && (A[i] > A[parent])) {
                Swap A[i] and A[parent];
                i ←parent;
                parent ←i/2;
        }
}
```

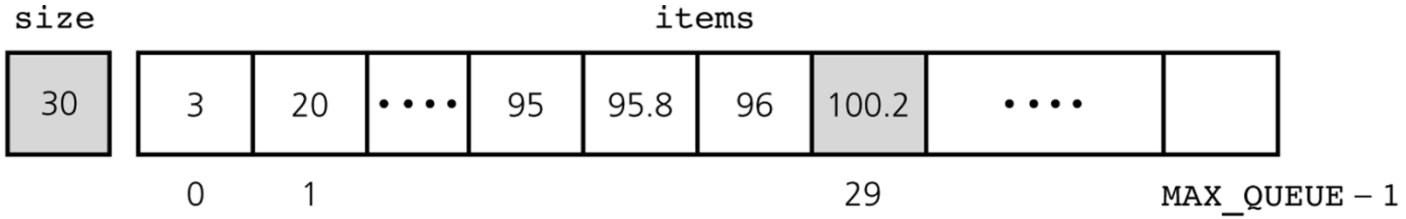
Recursive Version

```
heapInsert(A[], x, n) { // Insert item x on a heap A[1 … n]
        A[n+1] ←x;
        percolateUp(n+1);
}
percolateUp(A[], i) {
        parent ←i/2;
        if ((parent >= 1) && (A[i] > A[parent])) {
                Swap A[i] and A[parent];
                percolateUp(parent);
        }
}
```

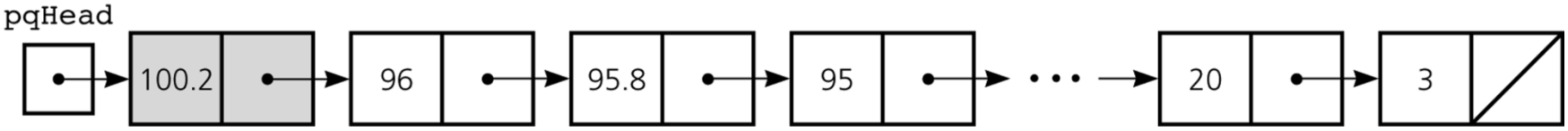

9
5
6
3
2
Insert
15
9
5
6
3
2
15
percolate
up
9
5
15
3
2
6
percolate
up
15
5
9
3
2
6

Some Non-efficient Implementations of an ADT Priority Queue

Array based

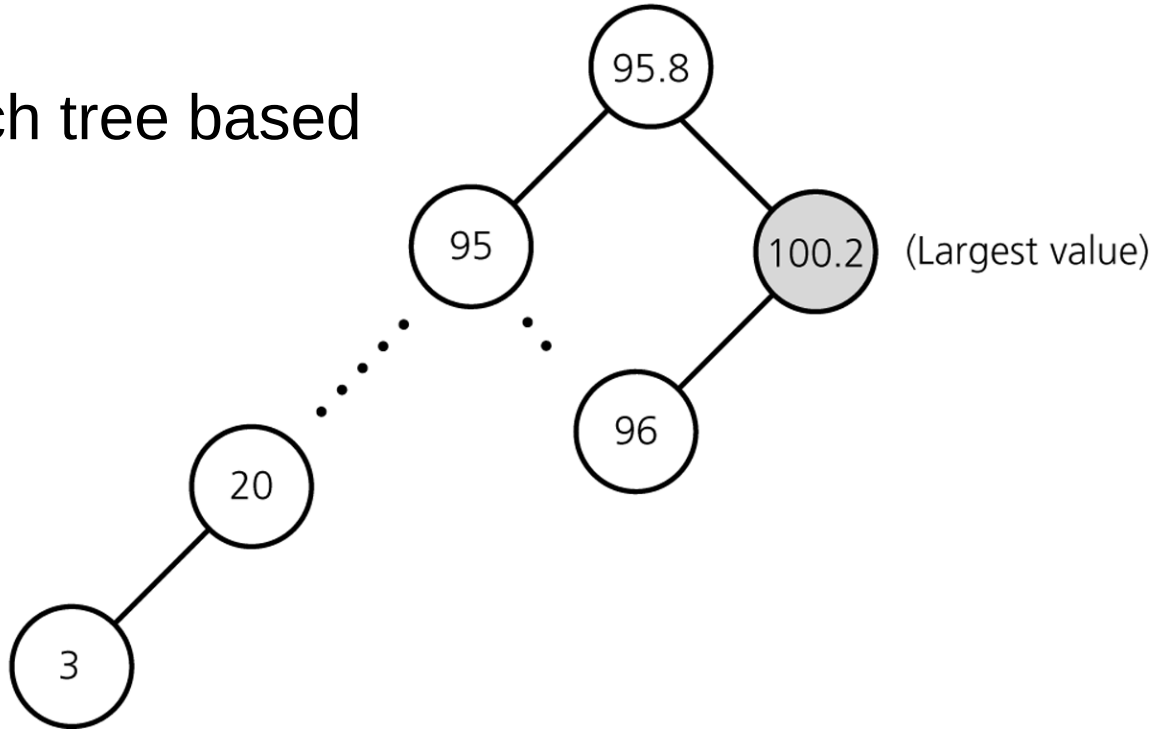


Reference based



A Non-Desirable Implementation of the ADT Priority Queue

Binary search tree based



동일한 key가 2개 이상일 때 별도로 처리를 해주어야 한다
이게 아니라도 priority queue 용도로는 너무 과하다

Heapsort

- Heap을 이용한 sorting
- 먼저 heap을 만든 다음, “Root node의 key를 빼내고, last node의 key를 root로 옮긴 다음 percolate down”하는 작업을 반복한다.

```
heapsort(A[], n) {
        // build array A[1…n] to heap
        for i ← n/2 downto 1 {
                percolateDown(A, i, n);
        }
        // delete one by one
        for size ← n downto 2 {
                Swap A[1] and A[size];
                percolateDown(A, 1, size-1);
        }
        // 이 지점에서 A[1…n]은 sorting되어 있다
}
```

```
percolateDown (A[], i, n) {
        child ← 2*i ; // left child
        rightChild ← 2*i + 1; // right child
        if (child <= n) {
                if ((rightChild <= n) && (A[child] < A[rightChild])) {
                        child ← rightChild;  // index of larger child
                }
                if (A[i] < A[child]) {
                        Swap A[i] and A[child];
                        percolateDown(A, child, n);
                }
        }
}
```

수작업으로 따라가 보기

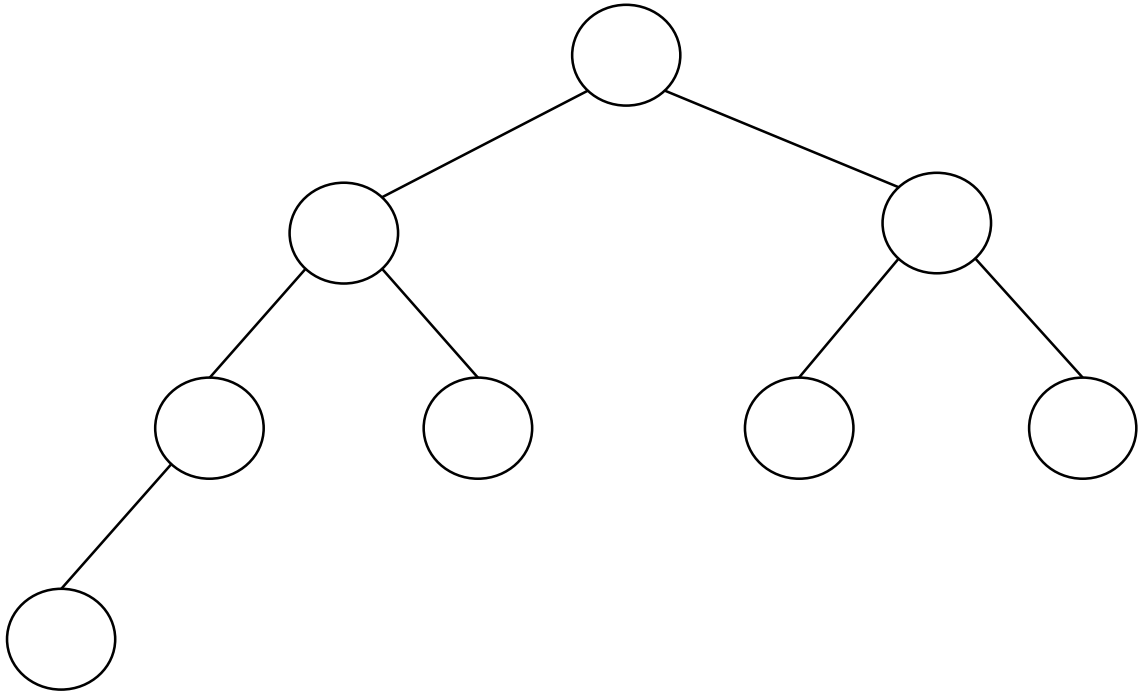
Given an array A[4, 6, 3, 10, 3, 9, 5, 2]

1. Building heap
2. Sorting

} 다음 페이지에…

A[4, 6, 3, 10, 3, 9, 5, 2]

1. Building heap



A[10, 6, 9, 4, 3, 3, 5, 2]

2. Sorting