Priority Queue and Heap

Il-Chul Moon
Dept. of Industrial and Systems Engineering
KAIST
icmoon@kaist.ac.kr

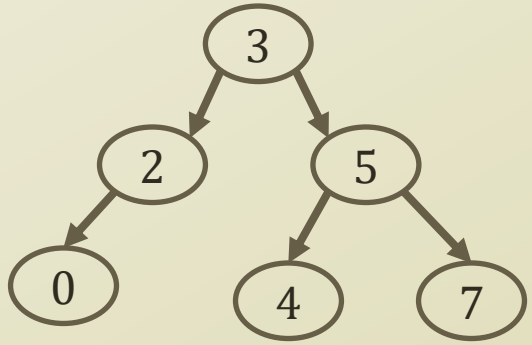
Weekly Objectives

- This week, we study various tree data structures. Particularly, we will focus on the priority queue and the heap.
- Objectives are
  - Understanding the structures and the operations of
    - Priority queues and heaps
    - Insert, delete
      - Structural integrity of the data structures
      - How to maintain the integrity
  - Understanding the performance of
    - Priority queues and heaps

Detour: Performance of binary search tree

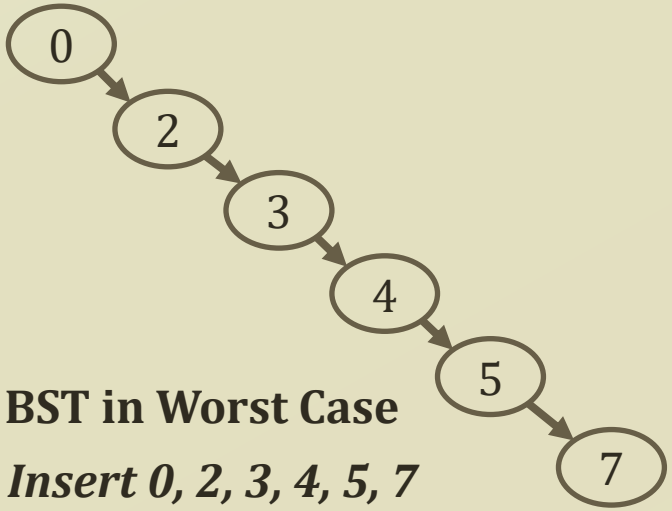
Coming from divide and conquer

	Linked List	BST in Average	BST in Worst Case
Search	O(n)	O(log n)	O(n)
Insert after search	O(1)	O(1)	O(1)
Delete after search	O(1)	O(1)	O(1)
Traverse	O(n)	O(n)	O(n)



BST in Average

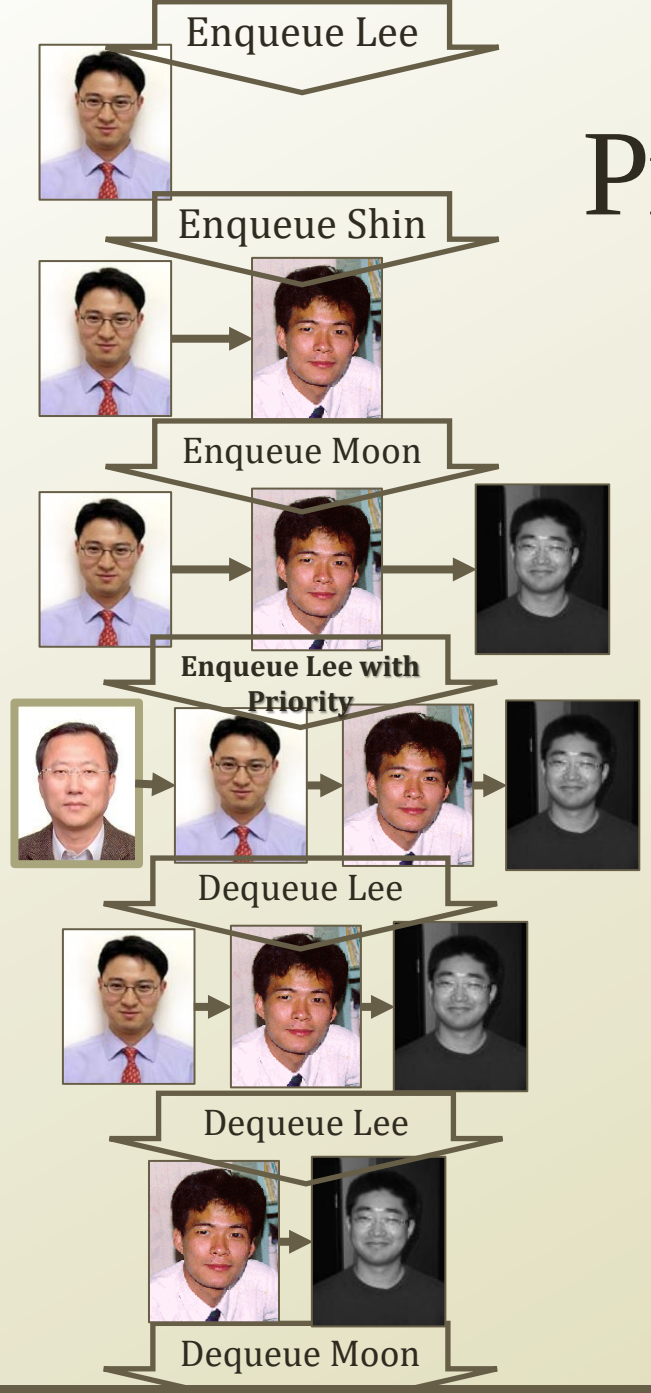
Insert 3, 2, 0, 5, 4, 7



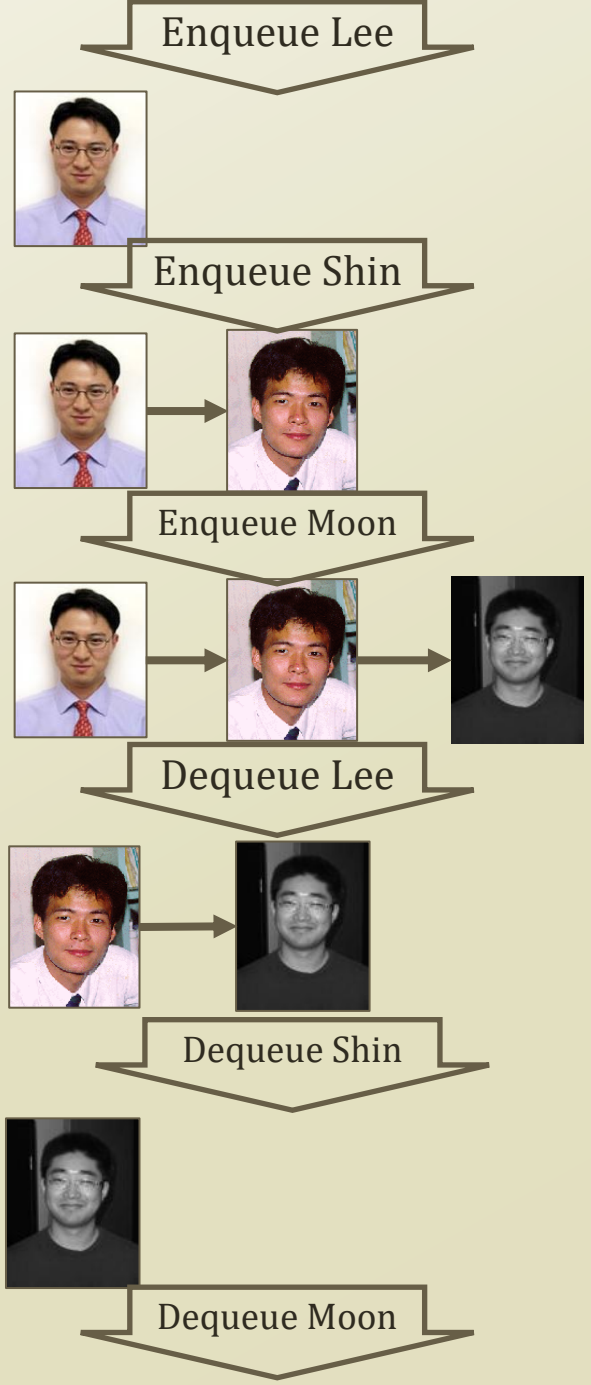
BST in Worst Case

Insert 0, 2, 3, 4, 5, 7

Priority Queue

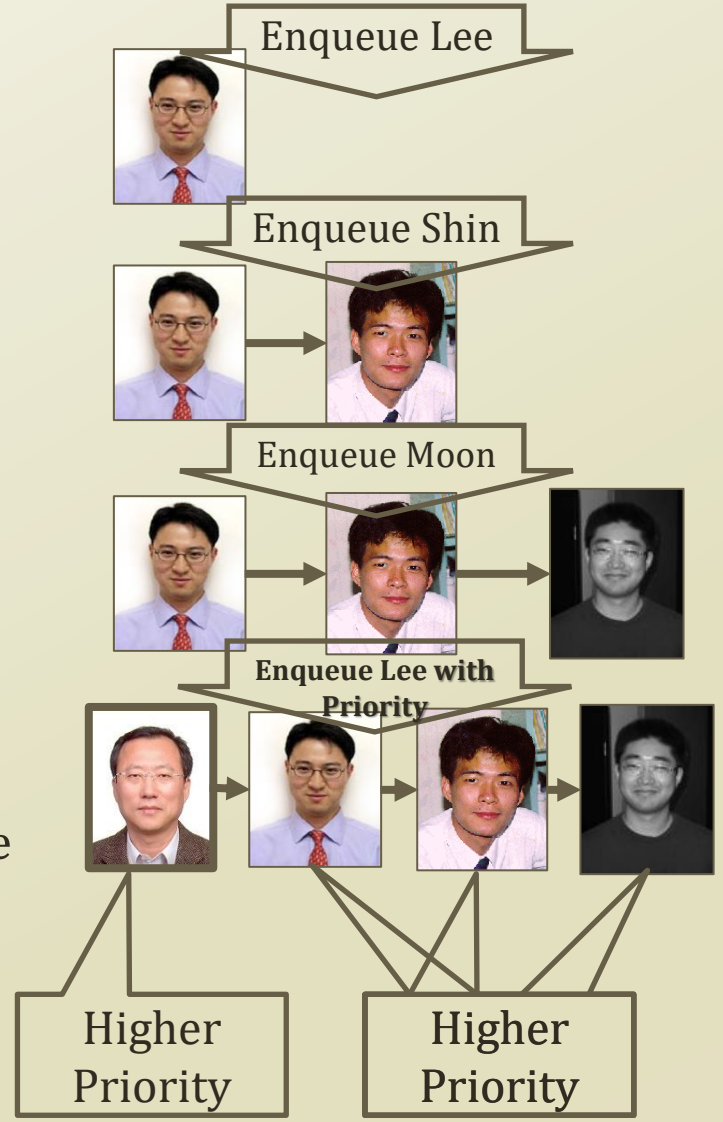


- Queue with priorities
  - Element who has priorities in the enqueue operation
  - Naturally, the element with the priority will be dequeued with the priority as well
- In other words
  - Cutting in the line
  - VIP service



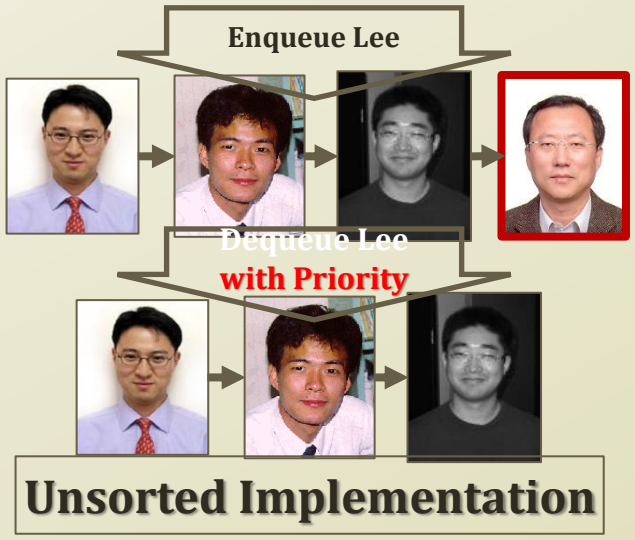
Operations of priority queues

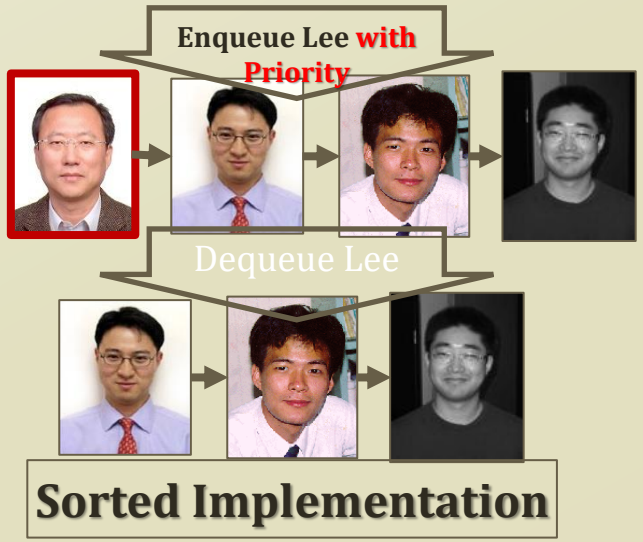
- Previously in queues
  - Enqueue an element
- Now in priority queues
  - Enqueue an element with a priority
  - Priority in the priority queue context
    - In our definition, let's say
      - Higher value = higher priority
      - Lower value = lower priority
    - Then, consider the previous example
      - Prof. Tae Eog Lee
        - Priority 2
      - Prof. Hayong Shin, Taesik Lee, Il-Chul Moon
        - Priority 1
  - Therefore, the interface of the priority queue will be
    - *enqueue(element, key)*
    - Rather, *enqueue(element)* in queue



How to implement priority queues

- Using the linked list as the basis of the priority queue
  - Store the element as well as the priority
  - Then, two approaches to implement the priority queue
    - Lazy approach == Unsorted implementation
      - When there is an enqueue event, just insert the element and the priority value at the end of the queue
      - When there is a dequeue event, remove the element with the highest priority by searching the queue from the beginning to the end
    - Early-bird approach == Sorted implementation
      - When there is an enqueue event, insert the element and the priority at the position that starts a sequence of elements with lower priorities
      - When there is a dequeue event, remove the element at the front of the queue





```
tae eog lee
hayong shin
taesik lee
il-chul moon
```

```python
from src.edu.kaist.seslab.ie362.week3.SinglyLinkedList import SinglyLinkedList

class PriorityNode:
    priority = -1
    value = ''
    def __init__(self, value, priority):
        self.priority = priority
        self.value = value
    def getValue(self):
        return self.value
    def getPriority(self):
        return self.priority

class PriorityQueue:

    list = ''

    def __init__(self):
        self.list = SinglyLinkedList()

    def enqueueWithPriority(self, value, priority):
        idxInsert = 0
        for itr in range(self.list.getSize()):
            node = self.list.get(itr)
            if node.getValue() == '':
                idxInsert = itr
                break
            if node.getValue().getPriority() < priority:
                idxInsert = itr
                break
            else:
                idxInsert = itr + 1
        self.list.insertAt( PriorityNode(value,priority), idxInsert )

    def dequeueWithPriority(self):
        return self.list.removeAt(0).getValue()

pq = PriorityQueue()
pq.enqueueWithPriority('il-chul moon', 1)
pq.enqueueWithPriority('taesik lee', 2)
pq.enqueueWithPriority('hayong shin', 3)
pq.enqueueWithPriority('tae eog lee', 99)

print(pq.dequeueWithPriority())
print(pq.dequeueWithPriority())
print(pq.dequeueWithPriority())
print(pq.dequeueWithPriority())
```

Implementation of priority queues

- Sorted implementation
  - Enqueue(node, priority)
    - current = head
    - While current.next().getPriority() > priority:
      - current = current.next()
      - Index = current.getIndex()
    - Insert
      - At index
      - PriorityQueueNode(value, priority)
- Unsorted implementation?
  - Have to change the Dequeue method

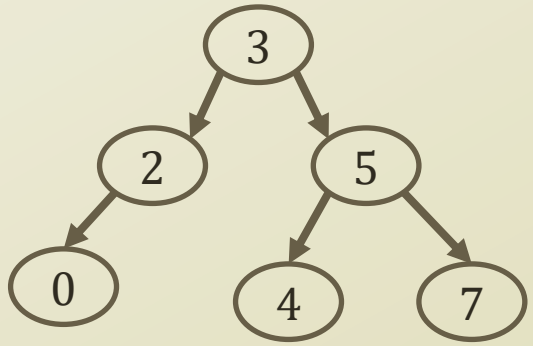
```
tae eog lee
hayong shin
taesik lee
il-chul moon
```

```python
from src.edu.kaist.seslab.ie362.week3.SinglyLinkedList import SinglyLinkedList

class PriorityNode:
    priority = -1
    value = ''
    def __init__(self, value, priority):
        self.priority = priority
        self.value = value
    def getValue(self):
        return self.value
    def getPriority(self):
        return self.priority

class PriorityQueue:

    list = ''

    def __init__(self):
        self.list = SinglyLinkedList()

    def enqueueWithPriority(self, value, priority):
        idxInsert = 0
        for itr in range(self.list.getSize()):
            node = self.list.get(itr)
            if node.getValue() == '':
                idxInsert = itr
                break
            if node.getValue().getPriority() < priority:
                idxInsert = itr
                break
            else:
                idxInsert = itr + 1
        self.list.insertAt( PriorityNode(value,priority), idxInsert )

    def dequeueWithPriority(self):
        return self.list.removeAt(0).getValue()

pq = PriorityQueue()
pq.enqueueWithPriority('il-chul moon', 1)
pq.enqueueWithPriority('taesik lee', 2)
pq.enqueueWithPriority('hayong shin', 3)
pq.enqueueWithPriority('tae eog lee', 99)

print(pq.dequeueWithPriority())
print(pq.dequeueWithPriority())
print(pq.dequeueWithPriority())
print(pq.dequeueWithPriority())
```

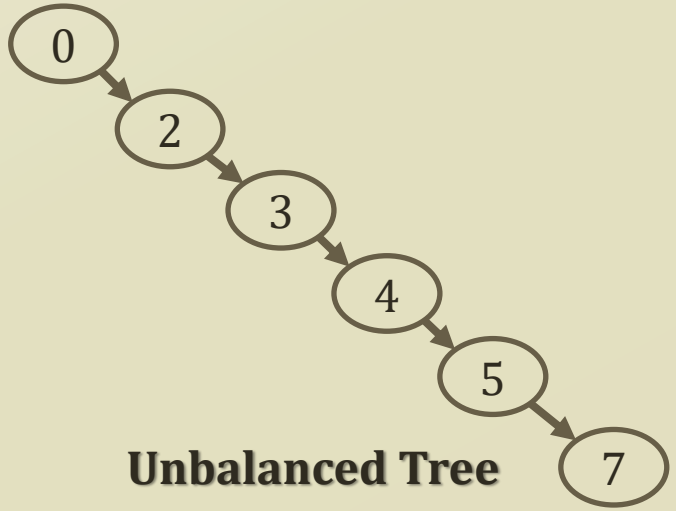
Performances of priority queue implementations

	Enqueue = Insert	Dequeue = Delete Highest Priority	FindMax = Find highest Priority
Unsorted Implemen-tation	O(1)	O(n)	O(n)
Sorted Implemen-tation	O(n)	O(1)	O(1)
Tree-based Implemen-tation	O(log n)	O(log n)	O(1)

Only true under the assumption that the tree is balanced…



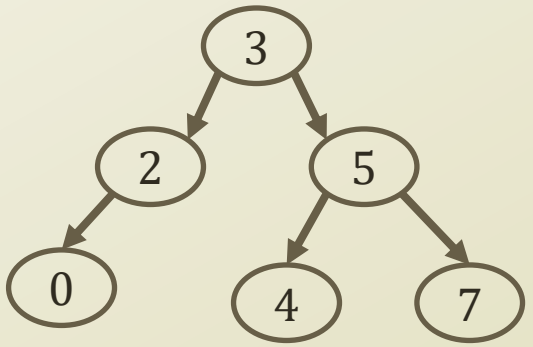
Balanced Tree



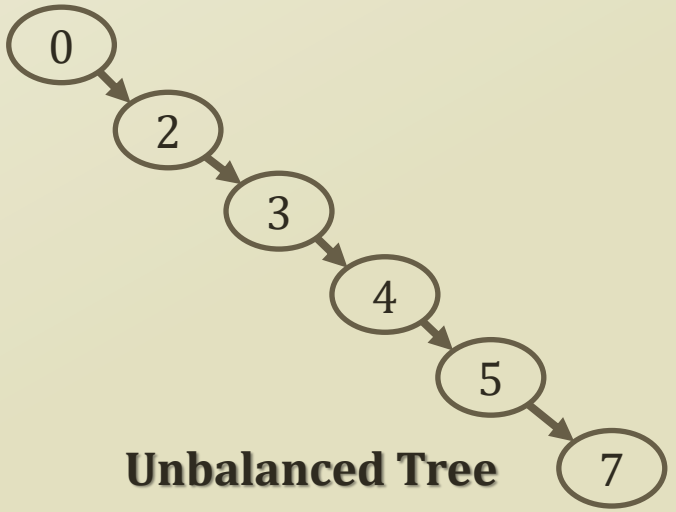
Unbalanced Tree

Balanced tree?

- Balanced tree
  - If its size is n,
    - $n \leq 2^{h+1} - 1$
    - 6 nodes in a tree of height 2
      - Correct: $6 \leq 2^{2+1} - 1 = 2^3 - 1 = 7$
    - 6 nodes in a tree of height 5
      - Correct: $6 \leq 2^{5+1} - 1 = 2^6 - 1 = 63$
  - What-if…..
    - $2^h - 1 < n \leq 2^{h+1} - 1$
    - 6 nodes in a tree of height 2
      - $2^2 - 1 < 6 \leq 2^{2+1} - 1$
      - Correct: $3 < 6 \leq 7$
    - 6 nodes in a tree of height 5
      - $2^5 - 1 < 6 \leq 2^{5+1} - 1$
      - Incorrect: $31 < 6 \leq 63$
- Complete tree → balanced tree
  - Yes
- Balanced tree → complete tree
  - No



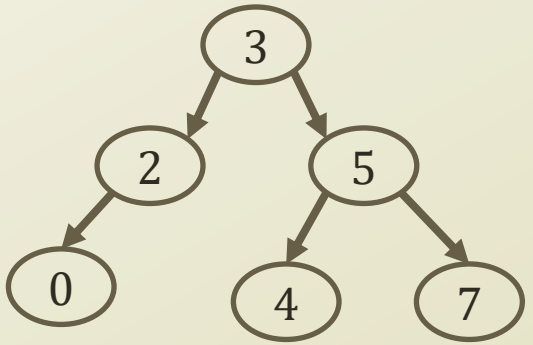
Balanced Tree



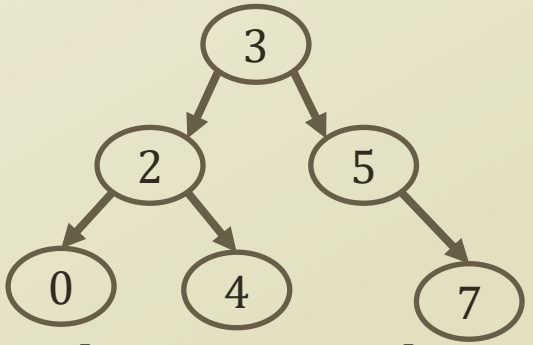
Unbalanced Tree

Binary heap for priority queue

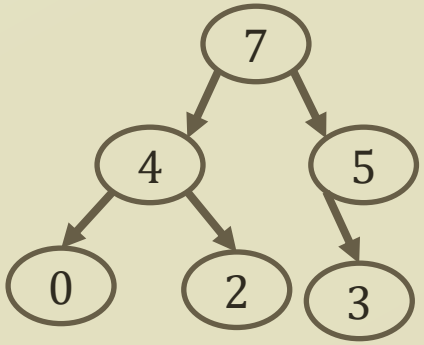
- Priority queue implementation
  - Linked list based implementation
    - Sorted implementation
    - Unsorted implementation
  - Tree based implementation
    - Tree should be balanced to justify the reason of using trees
- Binary heap is a binary tree with two properties
  - The shape property
    - The tree is a complete tree
  - The heap property
    - Each node is greater than or equal to each of its children
    - Max-heap since we defined a higher priority has a higher value



Is this tree a max-heap?

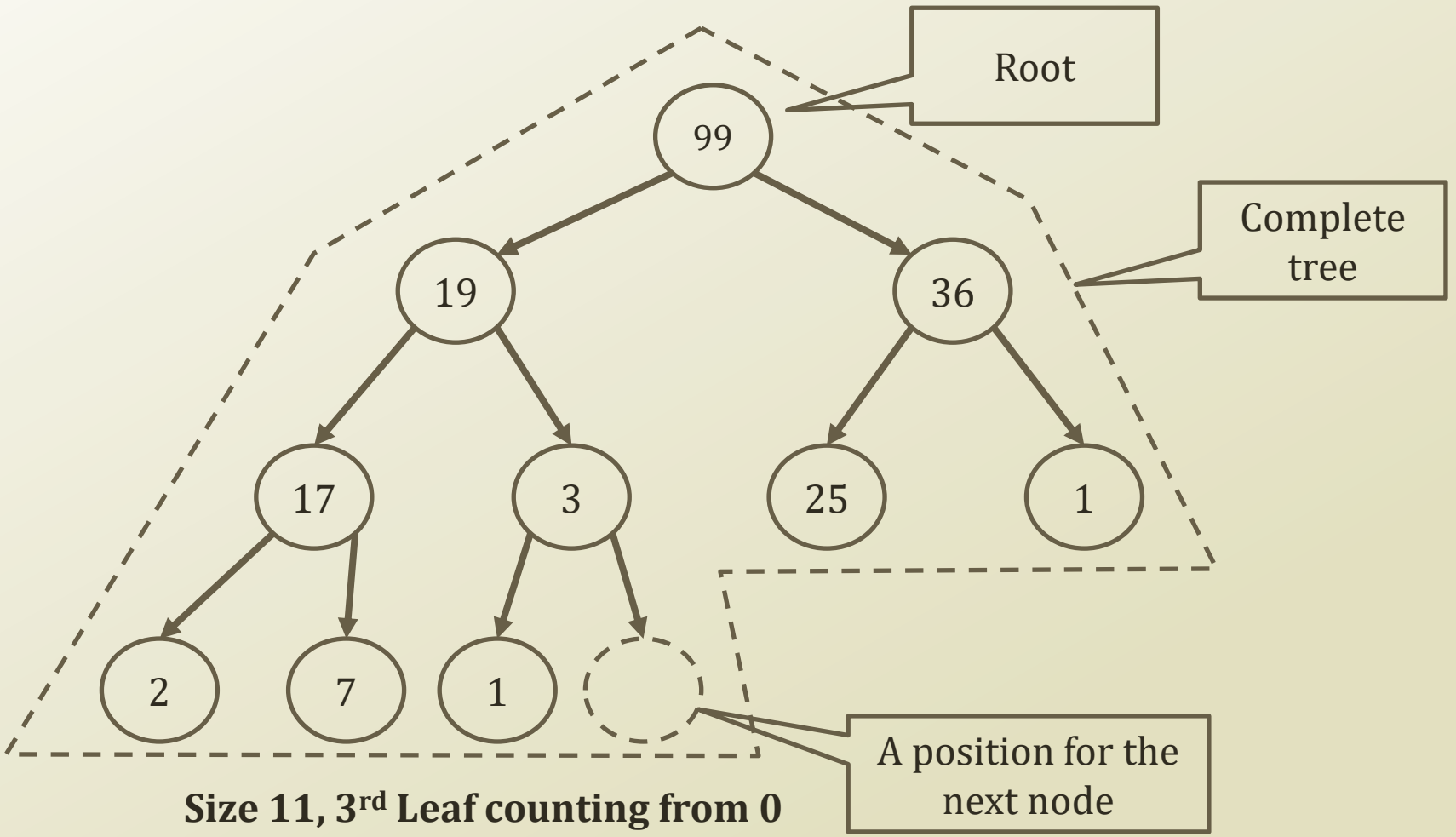


Is this tree a max-heap?



Is this tree a max-heap?

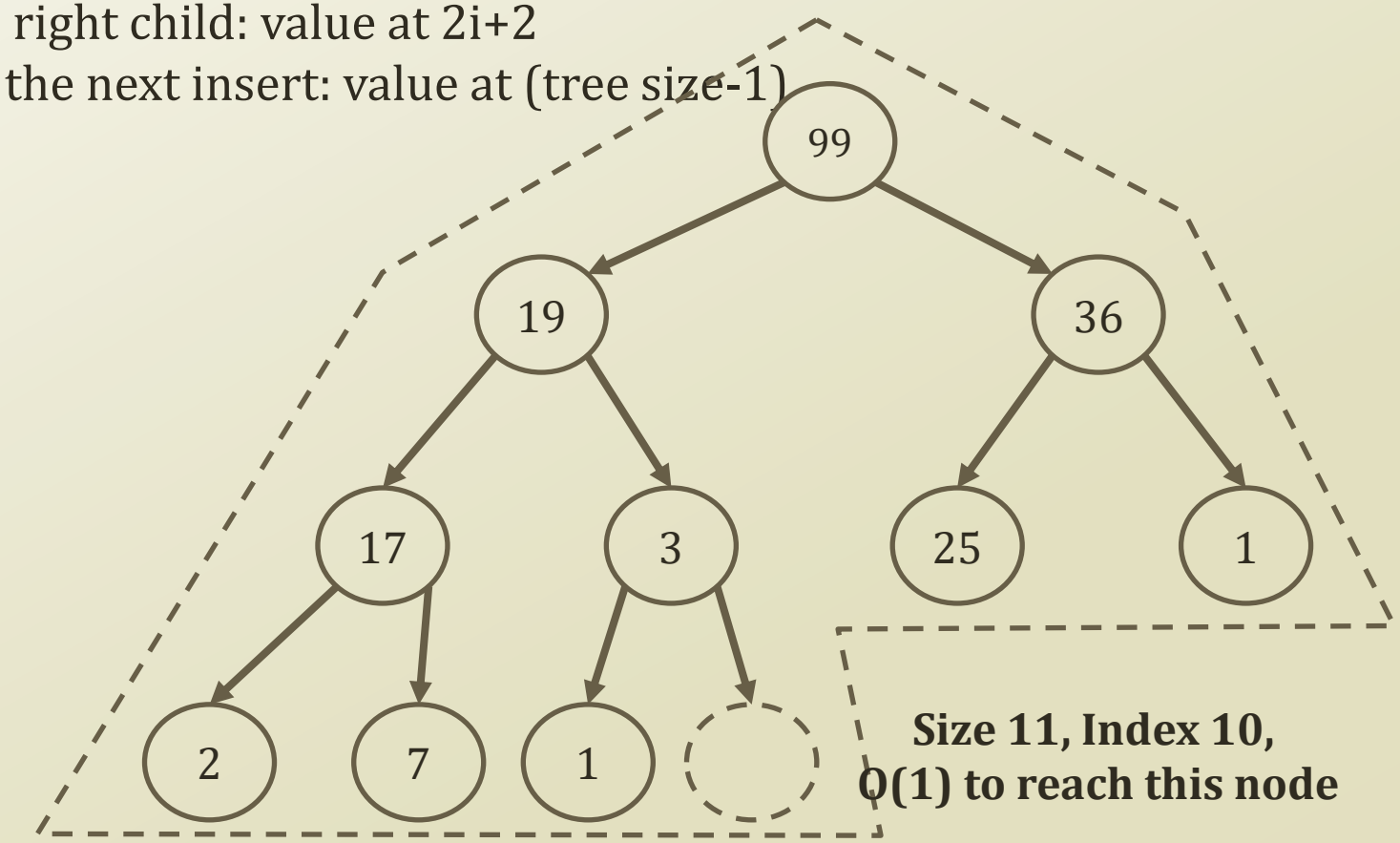
Structure of binary heap using reference



Size 11, 3rd Leaf counting from 0
(011: Left, Right, Right)
O(log n) to reach this node

Structure of binary heap using array

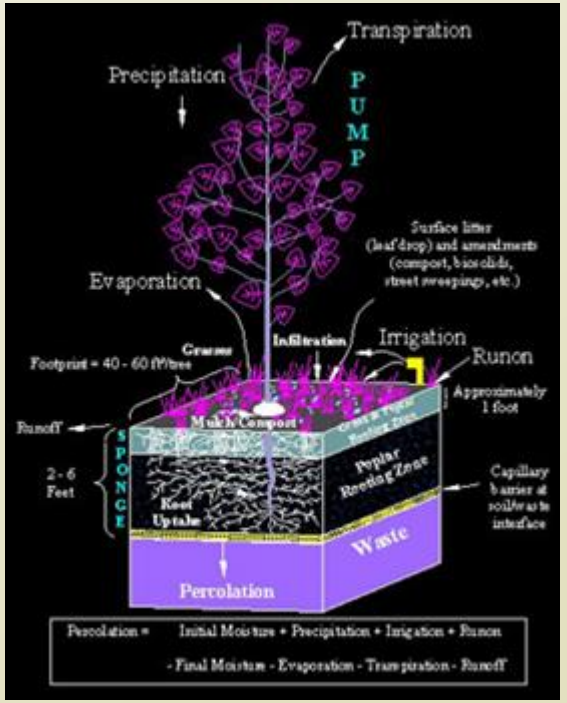
- Root: value at idx 0
- i^{th} node's parent: value at $\left\lfloor \frac{i-1}{2} \right\rfloor$
- i^{th} node's left child: value at 2i+1
- i^{th} node's right child: value at 2i+2
- Node for the next insert: value at (tree size-1)

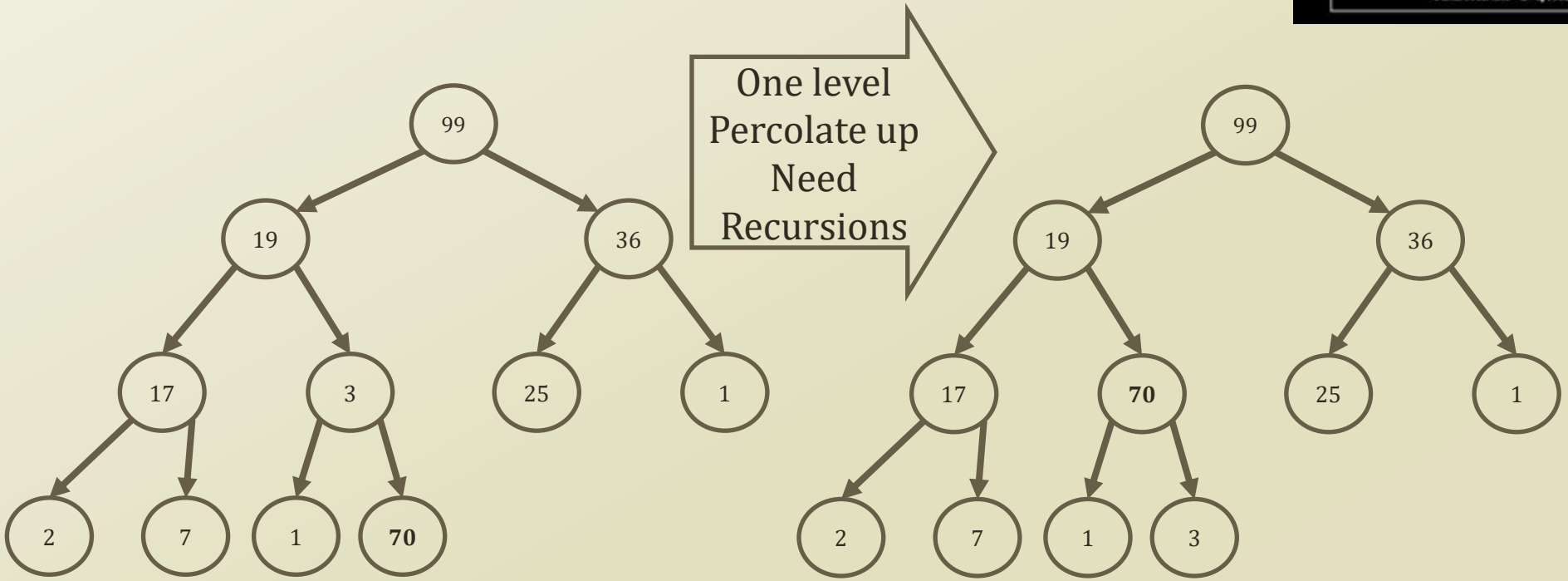


index	value
0	99
1	19
2	36
3	17
4	3
5	25
6	1
7	2
8	7
9	1
10	Next to Insert

Insert operation of binary heap

- Insert of binary heap, a.k.a. Percolate-up
  - Starting from a leaf
  - Approaching toward a root
  - How to?
    - Insert a value at the next node to insert
    - Compare the value to the value of the inserted node's parent
    - If the value is bigger than the parent's
      - The heap property is broken
      - Exchange the two values
      - Repeat this comparison at the parent's node

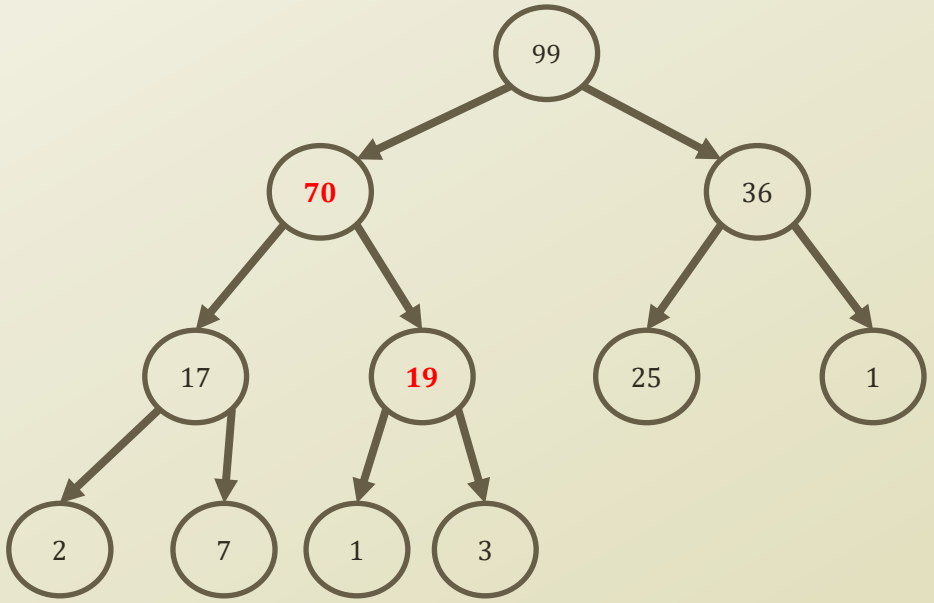


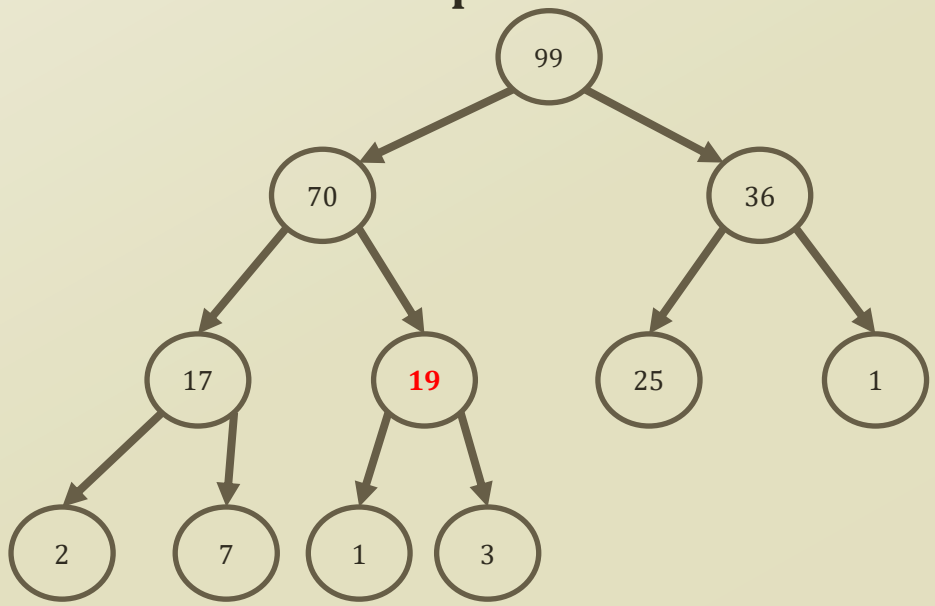


Implementation of insert of binary heap

```
def enqueueWithPriority(self, value, priority):
    self.arrPriority[self.size] = priority
    self.arrValue[self.size] = value
    self.size = self.size + 1
    self.percolateUp(self.size-1)

def percolateUp(self, idxPercolate):
    if idxPercolate == 0:
        return
    parent = int( (idxPercolate-1) / 2 )
    if self.arrPriority[parent] < self.arrPriority[idxPercolate]:
        self.arrPriority[parent], self.arrPriority[idxPercolate] = self.arrPriority[idxPercolate], self.arrPriority[parent]
        self.arrValue[parent], self.arrValue[idxPercolate] = self.arrValue[idxPercolate], self.arrValue[parent]
        self.percolateUp(parent)
```

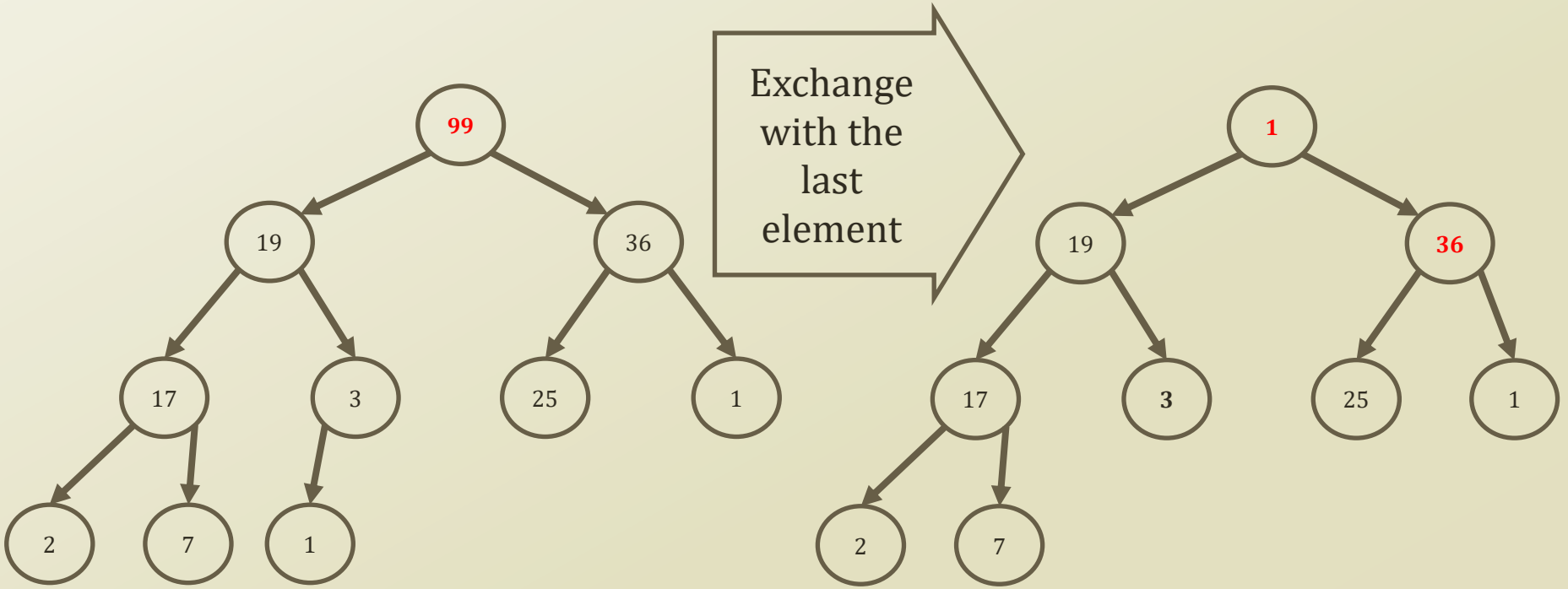
No percolation!





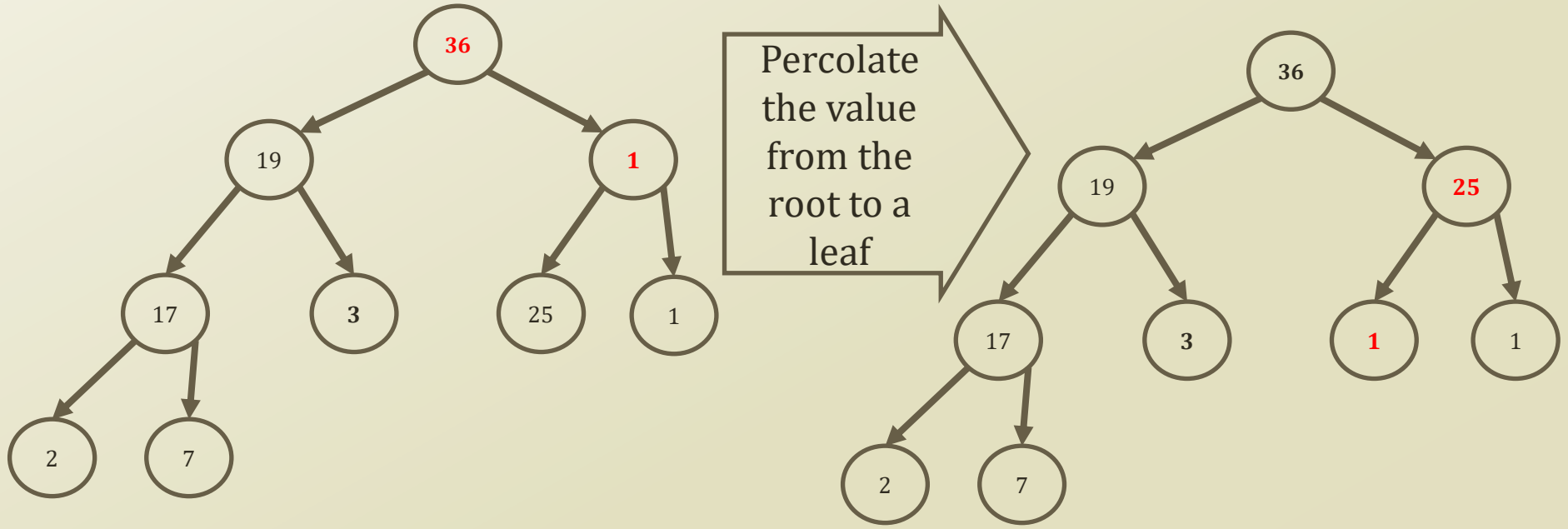
Delete operation of binary heap

- Delete of binary heap, a.k.a. Percolate-down or cascade-down
  - Starting from a root
  - Approaching toward a leaf
  - How to?
    - Delete the root node value by replacing the node with the last node
    - Compare the value to the value of the inserted node's children
    - If the children's value is bigger than the parent's, (pick a bigger child)
      - The heap property is broken
      - Exchange the root value and the bigger value from children
      - Repeat this comparison at the exchanged child's node



Implementation of delete of binary heap

```
def dequeueWithPriority(self):
    if self.size == 0:
        return ''
    retPriority = self.arrPriority[0]
    retValue = self.arrValue[0]
    self.arrPriority[0] = self.arrPriority[self.size - 1]
    self.arrValue[0] = self.arrValue[self.size - 1]
    self.size = self.size - 1
    self.percolateDown(0)
    return retValue

def percolateDown(self, idxPercolate):
    if 2*idxPercolate + 1 >= self.size:
        return
    else:
        leftChild = 2*idxPercolate+1
        leftPriority = self.arrPriority[leftChild]
    if 2*idxPercolate + 2 >= self.size:
        rightPriority = -99999
    else:
        rightChild = 2*idxPercolate+2
        rightPriority = self.arrPriority[rightChild]
    if leftPriority > rightPriority:
        biggerChild = leftChild
    else:
        biggerChild = rightChild
    if self.arrPriority[idxPercolate] < self.arrPriority[biggerChild]:
        self.arrPriority[idxPercolate], self.arrPriority[biggerChild] = self.arrPriority[biggerChild], self.arrPriority[idxPercolate]
        self.arrValue[idxPercolate], self.arrValue[biggerChild] = self.arrValue[biggerChild], self.arrValue[idxPercolate]
        self.percolateDown(biggerChild)
```



Complexity of priority queue, *again*

		Build	**Enqueue = Insert**	**Dequeue = Delete Highest Priority**	**FindMax = Find highest Priority**
Unsorted Implementation		O(N)	O(1)	O(N)	O(N)
Sorted Implementation		$O(N^2)$	O(N)	O(1)	O(1)
Binary Heap	Reference based	O(NlogN)	O(logN)	O(logN)	O(1)
	Array based (Naive build)	O(NlogN)	O(logN)	O(logN)	O(1)

Heap sort

- Priority queue
  - Repeated, dequeue with the highest priority
  - = dequeue the maximum value
  - Well-utilizable for sorting
  - Particularly
    - Binary heap enables the dequeueing with O(logN)
    - For dequeueing all elements, it takes O(NlogN)
      - Same to the sorting all of the elements
- How to perform a sorting with a heap (= heap sort)
  - Given a list whose index ranges from 0 to N
  - Firstly, Consider it as an insert to the heap from an array = O(NlogN)
    - It is the same problem of building a binary heap
  - Secondly, take out one element at a time = O(NlogN)
    - For itr in range(0, N):
      - Sorted[itr] = Heap.getHighestPriority()

Further Reading

- Introductions to Algorithms, 2nd ed., by Cormen et al.
  - pp. 455-475

Implementation of binary heap using array (1)

```
class BinaryHeap:

    arrPriority = {}
    arrValue = {}
    size = 0

    def __init__(self):
        self.arrPriority = {}
        self.arrValue = {}
        self.size = 0

    def enqueueWithPriority(self, value, priority):
        self.arrPriority[self.size] = priority
        self.arrValue[self.size] = value
        self.size = self.size + 1
        self.percolateUp(self.size-1)

    def percolateUp(self, idxPercolate):
        if idxPercolate == 0:
            return
        parent = int( (idxPercolate-1) / 2 )
        if self.arrPriority[parent] < self.arrPriority[idxPercolate]:
            self.arrPriority[parent], self.arrPriority[idxPercolate] = self.arrPriority[idxPercolate], self.arrPriority[parent]
            self.arrValue[parent], self.arrValue[idxPercolate] = self.arrValue[idxPercolate], self.arrValue[parent]
            self.percolateUp(parent)

    def dequeueWithPriority(self):
        if self.size == 0:
            return ''
        retPriority = self.arrPriority[0]
        retValue = self.arrValue[0]
        self.arrPriority[0] = self.arrPriority[self.size - 1]
        self.arrValue[0] = self.arrValue[self.size - 1]
        self.size = self.size - 1
        self.percolateDown(0)
        return retValue
```

Implementation of binary heap using array (2)

```
    def percolateDown(self, idxPercolate):
        if 2*idxPercolate + 1 >= self.size:
            return
        else:
            leftChild = 2*idxPercolate+1
            leftPriority = self.arrPriority[leftChild]
        if 2*idxPercolate + 2 >= self.size:
            rightPriority = -99999
        else:
            rightChild = 2*idxPercolate+2
            rightPriority = self.arrPriority[rightChild]
        if leftPriority > rightPriority:
            biggerChild = leftChild
        else:
            biggerChild = rightChild
        if self.arrPriority[idxPercolate] < self.arrPriority[biggerChild]:
            self.arrPriority[idxPercolate], self.arrPriority[biggerChild] = self.arrPriority[biggerChild], self.arrPriority[idxPercolate]
            self.arrValue[idxPercolate], self.arrValue[biggerChild] = self.arrValue[biggerChild], self.arrValue[idxPercolate]
            self.percolateDown(biggerChild)

    def build(self, arrInputPriority, arrInputValue):
        for itr in range(len(arrInputPriority)):
            self.arrPriority[itr] = arrInputPriority[itr]
            self.arrValue[itr] = arrInputValue[itr]
        self.size = len(arrInputPriority)
        for itr in range(self.size-1,-1,-1):
            self.percolateDown(itr)


pq = BinaryHeap()
pq.enqueueWithPriority('il-chul mooon', 1)
pq.enqueueWithPriority('taesik lee', 2)
pq.enqueueWithPriority('hayong shin', 3)
pq.enqueueWithPriority('tae eog lee', 99)

print(pq.dequeueWithPriority())
print(pq.dequeueWithPriority())
print(pq.dequeueWithPriority())
print(pq.dequeueWithPriority())

pq2 = BinaryHeap()
pq2.build({0:1, 1:2, 2:3, 3:99}, {0:'il-chul moon',1:'taesik lee', 2:'hayong shin', 3:'tae eog lee'})

print(pq2.dequeueWithPriority())
print(pq2.dequeueWithPriority())
print(pq2.dequeueWithPriority())
print(pq2.dequeueWithPriority())
```