

1. Data Structure and Algorithms 2

Priority Queue and Heap

목차

- ▶ Priority Queue
- ▶ Implementation & performance of Priority Queue
- ▶ Balanced Tree
- ▶ Binary Heap for Priority Queue
- ▶ Reference Structure of Binary
- ▶ Insert Operation of Binary Heap
- ▶ Delete Operation of Binary Heap
- ▶ Complexity of Priority Queue and Heap Sort

1. Priority Queue

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

```
graph TD; 3((3)) --> 2((2)); 3 --> 5((5)); 2 --> 0((0)); 5 --> 4((4)); 5 --> 7((7));
```

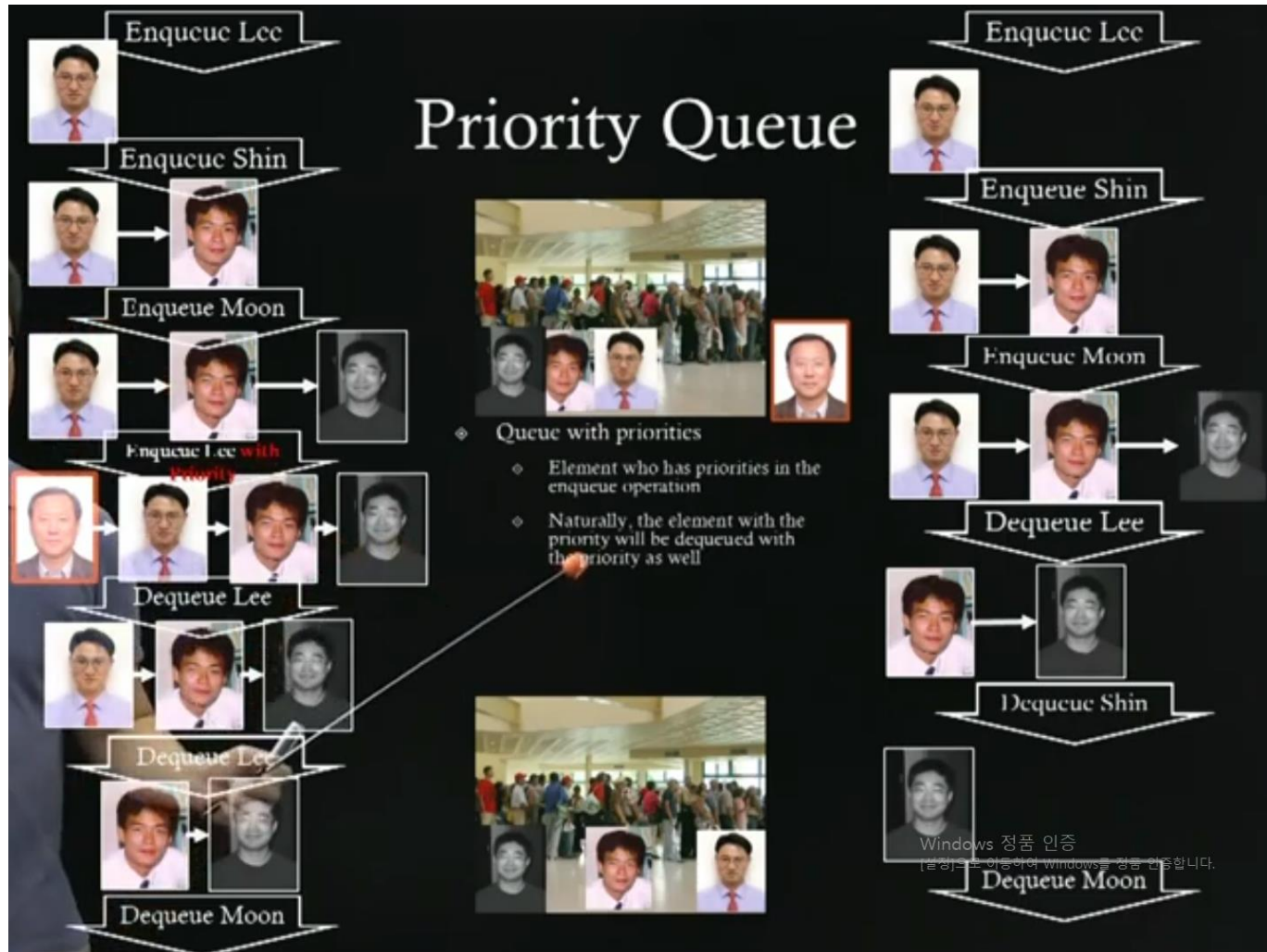
BST in Worst Case
Insert 0, 2, 3, 4, 5, 7

```
graph TD; 0((0)) --> 2((2)); 2 --> 3((3)); 3 --> 4((4)); 4 --> 5((5)); 5 --> 7((7));
```

- Binary search tree는 height가 낮아야 성능이 높다.
- BST가 worst case라면 linked list와 같다.
- Worst case를 없애주는게 더 나은 structure가 될 수 있다.
- 이를 위해 **balanced tree**를 만들 수 있는 것을 제안을 많이 하는데 다양한 형태의 **balanced tree**가 있다.

1. Priority Queue

Priority Queue



- Enqueue한 순서대로 Dequeue한다.
- 공항의 VIP는 Enqueue한 순서에 상관없이 제일 먼저 Dequeue하게 된다.
- 예를들어, 생산에서 빨리, 먼저 생산해야하는것들이 있다면 Priority Queue 공식을 이용 할 수 있다.
- Priority Queue는 elements들이 무순위가 아니라 순위가 정해져 있다.
- 순서가 있는 상태에서 Priority를 매겨서 Queue를 관리 하는 것이다.

1. Priority Queue

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



- 이전의 Queue는 하나의 element를 enqueue
- Priority enqueue는 enqueue를 하긴 하는데 priority를 같이 enqueue
- Higher value : higher priority
- Lower value : lower priority
- 모든 elements들의 priority가 0이면 enqueue한 순서대로 dequeue
- Priority가 높으면 먼저 dequeue
- Enqueue(element, key) 이전에는 key가 없었음

2. Implementation & performance of Priority 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

④ Lazy approach == Unsorted implementation

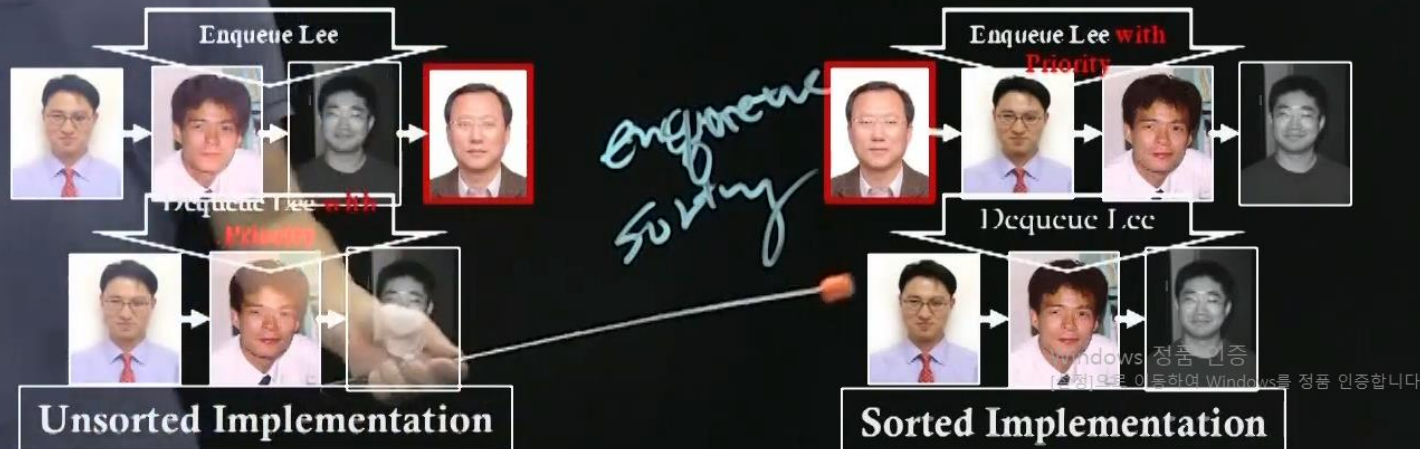
◇ When there is an enqueue event, just insert the element and the priority value at the end of the queue

◇ When there is a dequeue event, remove the element with the highest priority by searching the queue from the beginning to the end

④ 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



- Queue는 기본적으로 linked list위에 쌓아 만듦
- Elements 뿐만 아니라 Priority도 함께 저장
- 두가지 접근 방법:
- Lazy approach == Unsorted implementation
- Early-bird approach == Sorted implementation
- Lazy approach :
 - dequeue와 함께 sorting이 일어남,
 - 기본적으로 저장될때는 sorting이 안되어 있음
 - Dequeue할 때 sorting해서 제일 priority가 높은 element를 찾아서 dequeue하는 것
- Enqueue event -> just insert
- Dequeue event -> searching priority and remove
- Early-bird approach :
 - Enqueue 할때 sorting 하는것
 - Dequeue는 그대로 나감
 - Enqueue event -> sorting priority and insert
 - Dequeue event -> remove

2. Implementation & performance of Priority Queue

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

```
from 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()
```

Windows 정품 인증 7
[설정]으로 이동하여 Windows를 정품 인증합니다.

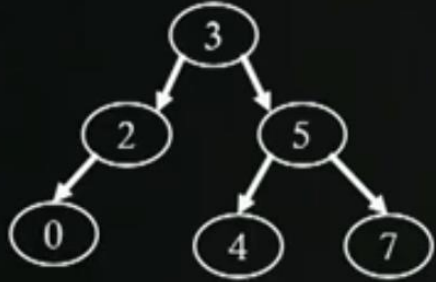
- Sorted implementation
- Enqueue(node, priority) priority가 생김
- Linked list를 for loop를 통해서 반복
- List의 각 node를 찾고 node의 value를 찾음
- 값이 없으면 insert
- value의 priority와 현재의 priority를 비교
- 현재 Priority가 크면 바로 insert
- 마지막 줄 code 위치에 바로 insert

2. Implementation & performance of Priority Queue

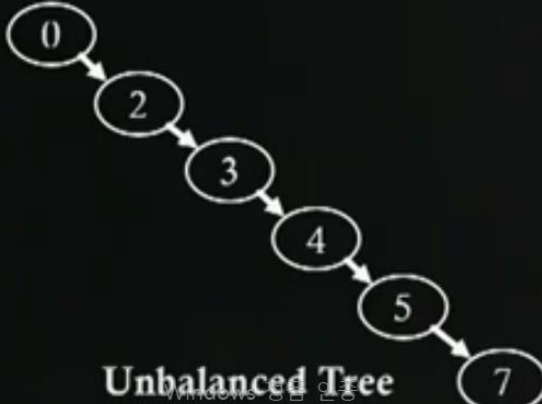
Performances of priority queue implementations

	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	$O(\log n)$	$O(\log n)$	$O(\log n)$

Only true under the assumption that the tree is balanced...



Balanced Tree



Unbalanced Tree

- Linked list뿐만 아니라 BST도 구현 가능
- Enqueue :
 - Unsorted : 뒤에 붙여서 $O(1)$
 - Sorted : 처음에 비교를 해야하기 때문에 $O(n)$
- Tree : log가 depth로 따라가서 $O(\log n)$, $O(1)$ 보다는 좀 더 걸리지만 $O(n)$ 보다는 훨씬 덜 걸린다.
- Dequeue :
 - Unsorted : searching 해야 해서 $O(n)$
 - Sorted : 앞에서 작업을 해놔서 $O(1)$
- Tree : $O(\log n)$
- FindMax는 같고 $O(1)$ 만드는 것이 좋음
- Unbalancing Tree : $O(n)$, $O(n)$, $O(n)$

3. Balanced Tree

Balanced tree?

전체화면을 종료하려면 [Esc] 을(를) 누르세요.

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

Wif Books 정품 인증
[설정]으로 이동하여 Windows를 정품

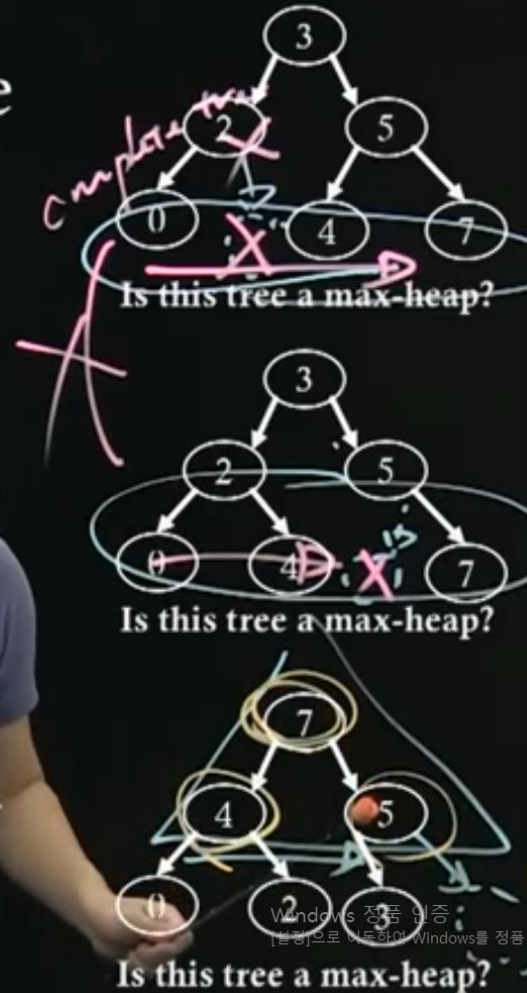
- Tree 공식 $n = 2^{H+1} - 1$
- Size = n, h = height
- 6 nodes in a tree of height 2 -> balanced tree
- 6 nodes in a tree of height 5 -> Unbalanced tree
- 두번째 공식을 만족하면 Balanced Tree
- 만족하지 못하면 Unbalanced Tree
-

4. Binary Heap for Priority Queue

Binary heap for priority queue

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

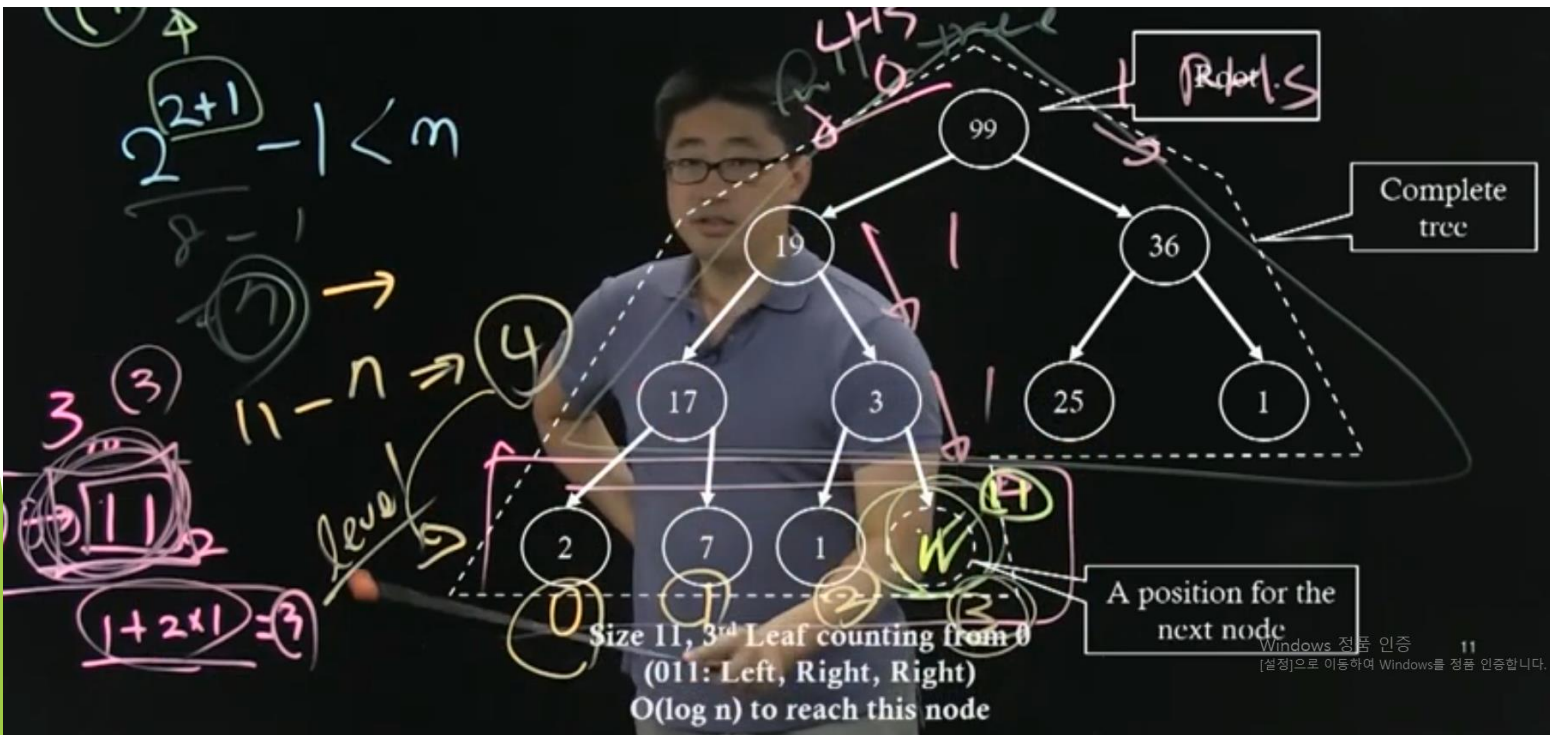


- Linked list:
- Sorted and Unsorted implementation
- Tree based:
- Should be balanced tree
- Binary tree:
- Shape property : complete tree여야 한다.
- Heap property :
- 첫번째는 complete tree가 아니기 때문에 No
- 두번째도 complete tree가 아니기 때문에 NO
- 세번째는 complete tree이고 parents value가 child value보다 크다

5. Reference Structure of Binary

Structure of binary heap using reference

- 위쪽은 full tree
- 과정은 complete tree
- 11번째에 값을 채워야 함
- 그 level의 3번째 node에 넣으면 됨
- 찾아가는 방법은 3을 2진수로 표현
- 11의 2진수가 되는데, $\rightarrow 00011$
- Root에서 0이면 LHS, 1이면 RHS
- 0=LSH, 1=RHS, 1=RHS



6. Insert Operation of Binary Heap

Insert operation of binary heap

of

- ◆ 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

children

One level Percolate up Need Recursions

정품 인증
[설정] 메뉴 이동하여 Windows를 정품 인증합니다.

- Percolate-up = 스며들다 = leap서 위로 올라간다
- Binary heap을 만족시키기 위해 approaching toward a root 한다.
- 1. 다음 노드에 value를 insert한다.
- heap property와 structure property때문에
- 하지만 heap의 조건은 깨짐. 고쳐줘야함
- 2. node's의 parents와 value를 비교
- value가 크면 exchange
- -> Percolate-up
- 3. 그 다음 Recursion한다.

6. Insert Operation of Binary Heap

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)  
    recursion
```

Windows 정품 인증 14
[설정]으로 이동하여 Windows를 정품 인증합니다.

- 70과 19를 비교해서 child의 value가 크기때문에 다시한번 exchange
- 더 이상 자리가 바뀌지 않으면 전체가 heap property가 만족하게 된다.
- Enqueue는 value와 priority
- Value를 array차원에 넣고 그 다음에 percolateUp을 하게 된다.
- 저장된 값의 index가 들어오게 된다.
- Index가 0이 되면 root가 된다.
- Parents를 찾고 비교를 하게 된다.
- Value가 크면 exchange
- 다음 recursion 한다.

7. Delete Operation of Binary Heap

Delete operation of binary heap

on

◆ 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

Exchange with the last element

Windows 정품 인증 15
[설정]으로 이동하여 Windows를 정품 인증합니다.

- Delete는 Percolate-down을 한다.
- Root를 지운다는 가정 하에 시작
- 1을 지우는게 이상적, root로 올린다.
- Structure만족, heap은 만족하지 않음
- 1. root node의 값을 지우고 last node의 값을 가져온다.
- 2. value를 child values과 비교
- 3. value가 child values(두 값중 큰 값)작다면 exchange
- 4. 그 다음 recursion

7. Delete Operation of Binary Heap

Delete operation of binary heap

Implementation of
Delete operation of binary
heap

```
def deleteWithPriority(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.percorlateDown(0)
    return retValue

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

Handwritten notes:
- Last node value/priority → root copy
- Break

Diagram illustrating the delete operation:
The diagram shows two binary heap states. The left state shows the root node (36) being swapped with the last node (25). The right state shows the root node (36) being swapped with its left child (19). A box labeled "Percorlate the value from the root to a leaf" points to the right state.

Windows 정품 인증 16
[설정]으로 이동하여 Windows를 정품 인증합니다.

- 1과 25를 비교
- 큰 값과 exchange
- Root의 값을 없애고 index스 저장
- Percorlate-down함
- Last node value와 priority를 root로 copy
- percorlate가 leap node에 도달하게 되면 escape
- Left child와 priority, Right child와 priority 비교
- RHS가 없을 경우 항상 LHS를 선택하게끔 아주 낮은 값으로 지정
- Bigger child를 찾아내서 그 값과 parent값 비교
- Bigger child가 크면 exchange
- 마지막으로 recursion function call 한다.

8. Complexity of Priority Queue and Heap Sort

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	$O(N \log N)$	$O(\log N)$	$O(\log N)$	$O(1)$
	Array based (Naive build)	$O(N \log N)$	$O(\log N)$	$O(\log N)$	$O(1)$

- Build(여러 개) :
- Unsorted : 계속 쌓아 가기 때문에 $O(N)$
- Sorted : 항상 sort하기 때문에 $O(N)$
- Reference :
- Array :
- Enqueue = Insert(하나만) :
- Unsorted : sort안해서 $O(1)$
- Sorted : sort하기 때문에 $O(N)$
- Reference :
- Array :
- Dequeue = delete and FindMax = Find :
- Unsorted : sort하기 때문에 $O(N)$
- Sorted : sort를 이미 했기 때문에 $O(1)$
- Reference :
- Array :
- 중요한건 $O(N)$ 보다 Binary Heap의 $O(\log N)$ 훨씬더 작은 값이기 때문에 전반적으로 binary heap을 이용한 queue 관리가 더 효율적

8. Complexity of Priority Queue and Heap Sort

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 $O(\log N)$
 - ◆ For dequeueing all elements, it takes $O(N \log N)$
 - ◆ 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 = $O(N \log N)$
 - ◆ It is the same problem of building a binary heap
- ◆ Secondly, take out one element at a time = $O(N \log N)$
 - ◆ For itr in range(0, N):
 - ◆ Sorted[itr] = Heap.getHighestPriority()

- Priority queue :
- Priority가 높은것은 dequeue하는 것
- 다시 말해 queue에서 maximum value가 dequeue
- 이 과정이 sorting algorithm
- 이러한 sorting algorithm을 binary heap을 이용하여 구현
- How :
- 0에서부터 n까지의 값의 리스트를 가지고 있는다.(sorting 해야할 list)
- 1. heap에 insert = binary heap을 build하는것과 같다.
- 2. 하나하나 element를 take한다.
- 따라서, list를 heap에 넣고 다시 나오는 과정에서 자연스럽게 sort가 된다.