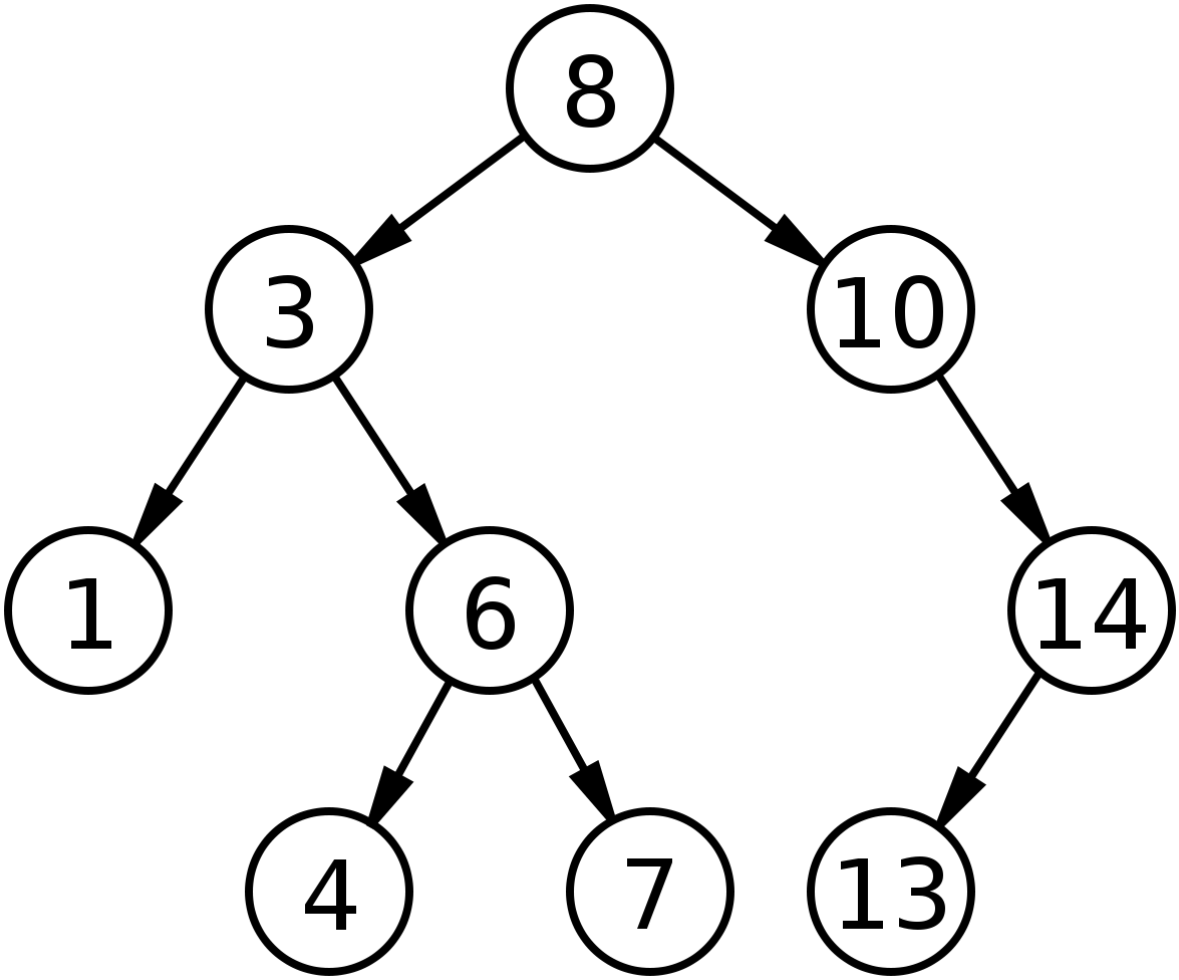
2주차: 이진 검색 트리

💡 트리는 현실의 계층적 구조를 나타내는 것 외에도, 자료들을 일정한 순서에 따라 정렬한 상태로 저장할 수도 있습니다. 이런 특성을 이용해서 원소의 추가, 삭제, 존재 역부 확인 등의 연산을 빠르게 수행할 수 있습니다. 오늘은 가장 많이 사용되는 이진 검색 트리 (Binary Search Tree)에 대해서 알아보겠습니다.

이진 검색 트리 파헤치기!



💡 아래의 기능들은 대부분의 이진 검색 트리 표준라이브러리에서 지원하고 있습니다. 따라서 단순한 동작 원리 정도만 알아둬도 문제가 없습니다.

▼ **[정의]** 각 노드가 왼쪽과 오른쪽, 최대 두 개의 자식 노드만을 가질 수 있습니다.

- 예를 들어 노드 ③ 에는 [1,6] 배열이 아닌 ①을 가리키는 포인터 “Left”와 ⑥을 가리키는 포인터 “Right”가 저장되어 있습니다.

▼ **[순회]** 중상 순회를 이용하면 크기 순서로 정렬된 원소의 목록을 얻을 수 있습니다.

- 위 그림 중상 순회 예시 출력: ① ③ ④ ⑥ ⑦ ⑧ ⑩ ⑬ ⑭

▼ **[검색]** 찾고자 하는 값이 현재 노드를 기준으로 왼쪽인지, 오른쪽인지 확인하며 내려갑니다.

- 방법 자체는 수치해석 챕터에서 다뤘던 이진 탐색과 거의 동일합니다.
- 예를 들어 1부터 9까지의 정수가 들어가있는 리스트에서 5를 찾으려면 리스트의 시작부터 끝까지 순차적으로 검색하는 for 루프를 총 5번 돌려서 찾을 수 있겠죠.

- 하지만 동일한 자료를 이진 검색 트리로 구현하면 평균적으로 더 빠르게 찾을 수 있습니다.

▼ **[삽입]** 단순히 노드를 추가하는 식으로 새로운 원소를 삽입할 수 있습니다.

- 배열의 경우, 새 원소를 삽입하기 위해서는 삽입 위치를 찾고, 그 이후에 있는 원소들을 모두 한 칸씩 뒤로 옮겨야 합니다.
- 이진 검색 트리는 선형적인 구조의 제약이 없기 때문에 새 원소가 들어갈 위치를 찾고, 해당 위치에 노드를 추가해주기만 하면 됩니다.
- 예를 들어 위 트리에서 ⑮를 추가하고 싶다면, ⑭의 오른쪽에 새로운 자식 노드를 생성해주면 됩니다.

▼ **[삭제]** 서브트리를 합치고 대체해서 노드를 삭제할 수 있습니다.

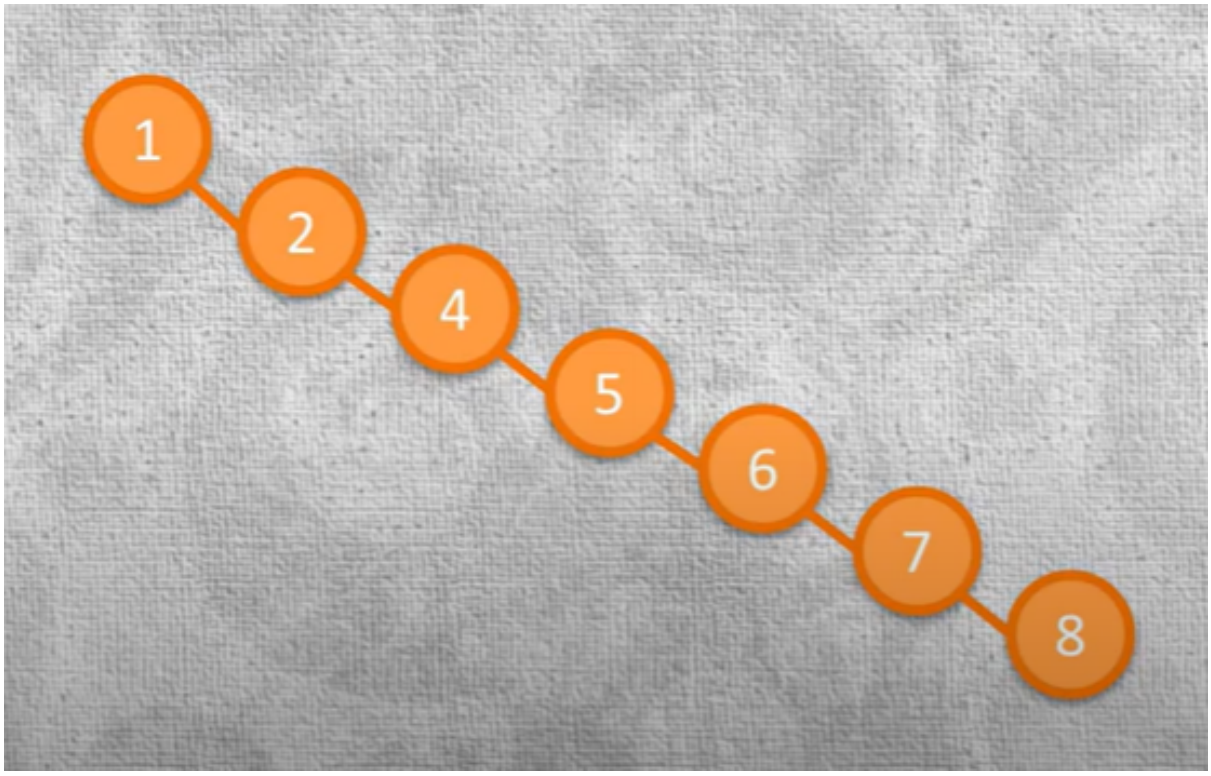
- 예를 들어 노드 ③을 삭제하고 싶다고 해봅시다.
- ③의 자식 노드는 ①과 ④ ← ⑥ → ⑦ 이 있고 합치기 연산을 이용해서 새로운 서브트리를 만들겠습니다. ① ← ④ ← ⑥ → ⑦
- 새롭게 만든 서브트리의 가장 상위에 있는 노드는 ⑥입니다.
- ⑥을 기존에 ③이 있던 위치로 옮겨주면 삭제 과정이 끝납니다.
- 삽입과 마찬가지로 위 과정을 배열 기준으로 생각해보면, 삭제할 원소의 위치를 찾아주고, 그 이후에 있는 원소들을 모두 한 칸씩 앞으로 당겨줘야 하는 귀찮은 과정이 있습니다.

▼ **[문제]** 이진 검색 트리의 표준 라이브러리에서 지원하지 않는 기능도 있습니다.

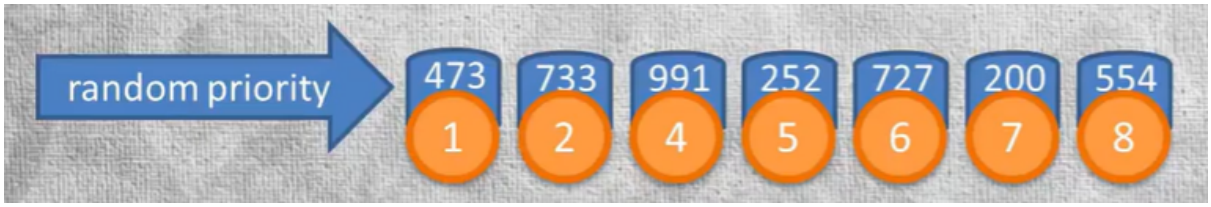
- X보다 작은 원소의 수 찾기
- K번째 원소 찾기
- 균형 잡힌 트리 구축하기

TREAP을 알아보자!

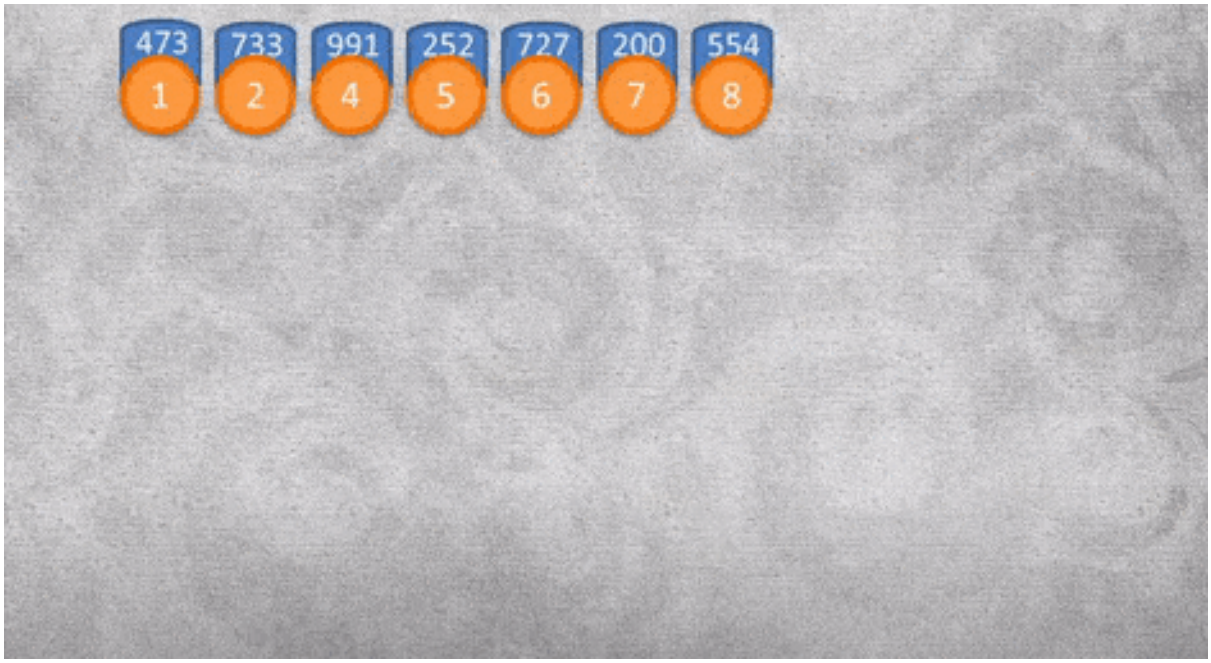
- 위에서 언급드린 이진 검색 트리의 [문제]를 해결하기 위한 해결방법은 다양합니다. X보다 작은 원소의 수 찾기나 K번째 원소 찾기는 간단한 메소드 구현으로 해결할 수 있지만, 문제는 균형 잡힌 트리에 있습니다. 그래서 이를 비교적 한정된 시간 안에 간단하게 해결할 수 있는 treap 자료 구조를 소개합니다. treap은 tree와 heap의 합성어로, 그 이유는 아래에서 보실 수 있습니다.



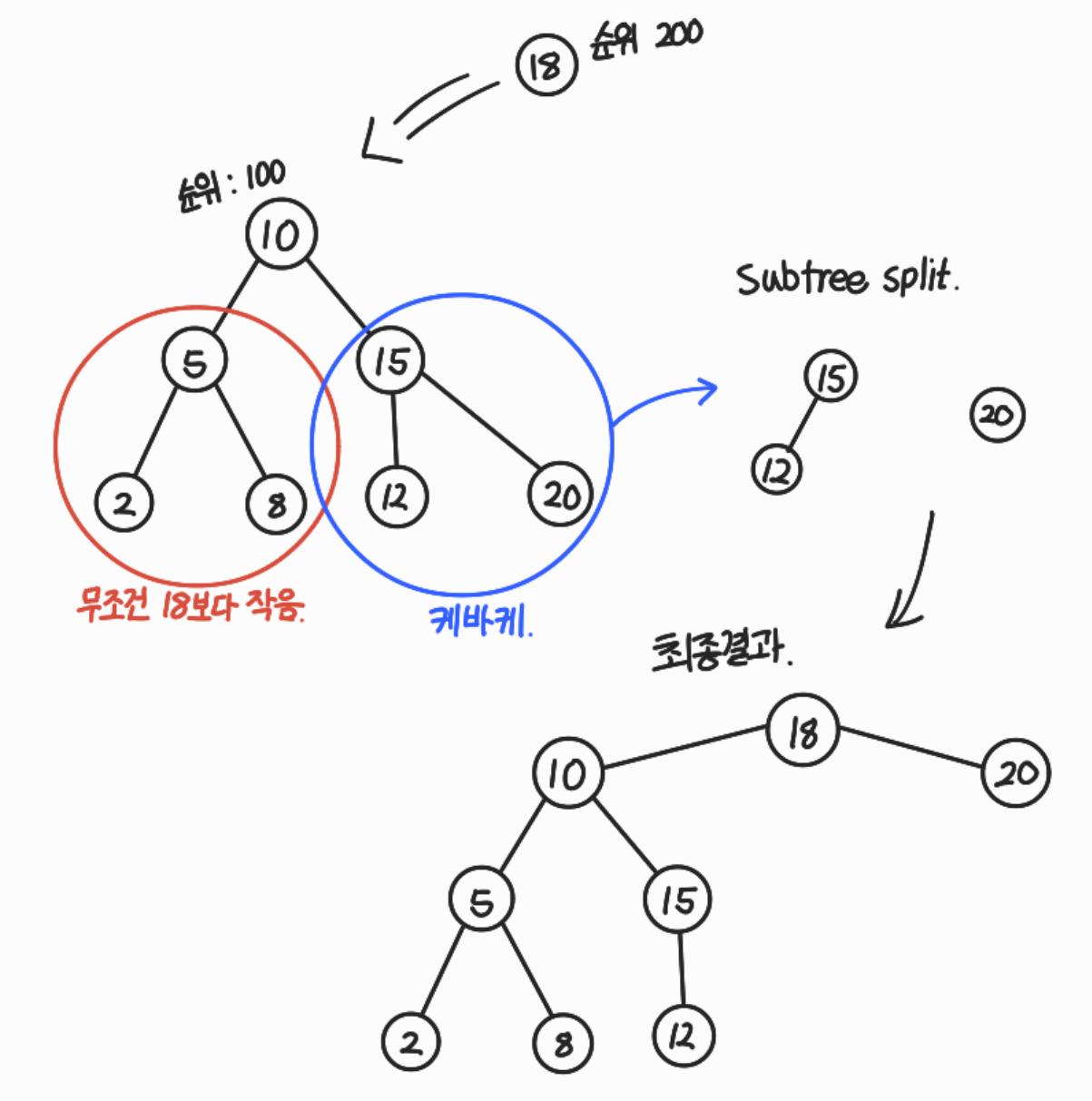
- 예를 들어 이진 탐색 트리에 1부터 8까지의 노드가 순서대로 입력되었다고 생각해봅시다. 현재의 트리는 1차원 배열만큼이나 비효율적입니다.
- 여기서 각자의 노드에 들어가 있는 정수값을 “key” 라고 부르겠습니다.



- 이제 노드마다 랜덤 난수로 생성된 “priority”값, 즉 우선 순위를 부여합니다.
- key와 priority 값이 부여된 노드들로 treap을 만들건데, 이 과정에서 tree와 heap 자료구조의 조건을 준수해야 합니다.
  - 조건 1) 이진 검색 트리: 모든 노드에 대해 왼쪽 서브트리에 있는 노드들의 key들은 해당 노드의 key보다 작고, 오른쪽은 반대로 생각하시면 됩니다.
  - 조건 2) 힙: 모든 노드의 priority는 각자의 자식 노드보다 크거나 같습니다.
- 백문이불여일견. 아래의 예시를 한번 보겠습니다.



- 과정을 간단하게 생략해서 적어두겠습니다.
  1. ①이 가장 앞에 있으므로 루트 노드로 잡아줍니다.
  2. ②가 새로 입력되는데, 조건 2를 따라서 ①의 오른쪽 위치에 들어갑니다.
  3. 하지만 ②의 priority는 733으로, ①의 priority인 473보다 높습니다.
  4. 조건 1에 의하면 priority가 더 높은 ②가 ①의 부모 노드가 되어야 합니다.
  5. ②를 부모 노드로 지정해주면, 다시 조건 1)에 따라서 ①이 ②의 왼쪽 자식 노드가 됩니다.
  6. 똑같은 과정을 입력이 끝날 때까지 반복합니다.



- 노드의 순위값에 따라 루트를 바꿔야 하는 상황이 조금 더 복잡해질수도 있습니다. 그래서 직접 만든 예시를 하나 더 가져와봤습니다.

- 결론적으로 위에서 진행한 과정에 대해 설명드리자면, 기존에 입력 순서대로 주어졌던 자료에 랜덤으로 priority값들을 부여하고, priority순으로 재정렬을 해서 이진 검색 트리를 만드는 것과 똑같다고 생각하시면 됩니다.

- 결론은 위와 동일합니다.
- 결론적으로 treap을 활용하면 이진 검색 트리의 높이의 기대치가 O(N), 즉 n개의 데이터에 대해 n-1의 높이를 가지는 상황은 가급적 피할 수 있습니다. 수학적 증명이 궁금하시면 pg.711에서 찾아보실 수 있겠습니다.

TREAP을 구현해보자!

```
import random

class TreapNode:
    def __init__(self, key, priority):
        # Treap 노드는 key(값)와 priority(우선순위)를 가지는 클래스입니다.
        self.key = key
        self.priority = priority
        self.left = None  # 왼쪽 자식 노드를 가리키는 포인터
        self.right = None  # 오른쪽 자식 노드를 가리키는 포인터

class Treap:
    def __init__(self):
        self.root = None  # Treap의 루트 노드를 가리키는 포인터

    def _rotate_left(self, node):
        # 주어진 노드를 기준으로 오른쪽으로 회전하는 함수입니다.
        right_child = node.right
        node.right = right_child.left
        right_child.left = node
        return right_child
```

```
    def _rotate_right(self, node):
        # 주어진 노드를 기준으로 왼쪽으로 회전하는 함수입니다.
        left_child = node.left
        node.left = left_child.right
        left_child.right = node
        return left_child

    def _insert(self, node, key, priority):
        # 새 노드를 삽입하는 내부 함수입니다.
        # 이 함수는 재귀적으로 호출되며, 새 노드를 적절한 위치에 삽입합니다.
        if not node:
            return TreapNode(key, priority)

        if key < node.key:
            node.left = self._insert(node.left, key, priority)
            if node.left.priority > node.priority:
                # 삽입 후에 왼쪽 자식 노드의 우선순위가 더 크다면, 오른쪽으로 회전합니다.
                node = self._rotate_right(node)
        else:
            node.right = self._insert(node.right, key, priority)
            if node.right.priority > node.priority:
                # 삽입 후에 오른쪽 자식 노드의 우선순위가 더 크다면, 왼쪽으로 회전합니다.
                node = self._rotate_left(node)

        return node

    def insert(self, key):
        # 외부에서 호출하는 삽입 함수입니다.
        priority = random.random()  # 우선순위를 랜덤으로 지정
        self.root = self._insert(self.root, key, priority)

    def _find(self, node, key):
        # 주어진 키 값을 가진 노드를 찾는 내부 함수입니다.
        if not node or node.key == key:
            return node

        if key < node.key:
            return self._find(node.left, key)
        else:
            return self._find(node.right, key)

    def find(self, key):
        # 외부에서 호출하는 특정 키를 찾는 함수입니다.
        return self._find(self.root, key)

    def _delete(self, node, key):
        # 주어진 키 값을 가진 노드를 삭제하는 내부 함수입니다.
        if not node:
            return node

        if key < node.key:
            node.left = self._delete(node.left, key)
        elif key > node.key:
            node.right = self._delete(node.right, key)
        else:
            if not node.left:
                return node.right
```

```
            elif not node.right:
                return node.left

            if node.left.priority < node.right.priority:
                # 왼쪽 자식 노드의 우선순위가 더 작다면, 오른쪽으로 회전합니다.
                node = self._rotate_left(node)
                node.left = self._delete(node.left, key)
            else:
                # 오른쪽 자식 노드의 우선순위가 더 작다면, 왼쪽으로 회전합니다.
                node = self._rotate_right(node)
                node.right = self._delete(node.right, key)

        return node

    def delete(self, key):
        # 외부에서 호출하는 삭제 함수입니다.
        self.root = self._delete(self.root, key)

    def _inorder_traversal(self, node, result):
        # 중위 순회를 통해 정렬된 결과를 반환하는 내부 함수입니다.
        if node:
            self._inorder_traversal(node.left, result)
            result.append(node.key)
            self._inorder_traversal(node.right, result)

    def inorder_traversal(self):
        # 외부에서 호출하는 중위 순회 함수입니다.
        result = []
        self._inorder_traversal(self.root, result)
        return result

# 예시로 사용해보기
if __name__ == "__main__":
    treap = Treap()
    keys = [5, 2, 8, 1, 3, 7, 9]

    for key in keys:
        treap.insert(key)

    print("Inserted keys:", keys)
    print("Inorder traversal:", treap.inorder_traversal())

    treap.delete(2)
    treap.delete(9)

    print("Inorder traversal after deletion:", treap.inorder_traversal())
```

```
Inserted keys: [5, 2, 8, 1, 3, 7, 9]
Inorder traversal: [1, 2, 3, 5, 7, 8, 9]
Inorder traversal after deletion: [1, 3, 5, 7, 8]
```

링크: https://www.notion.so/2-726540a2b08a469d93e67a9da146a8a7?pvs=4