**Min-Max-Median Priority Queue**

이름: **이세형**

학번: **2020-312145**

1. **Code Style:**

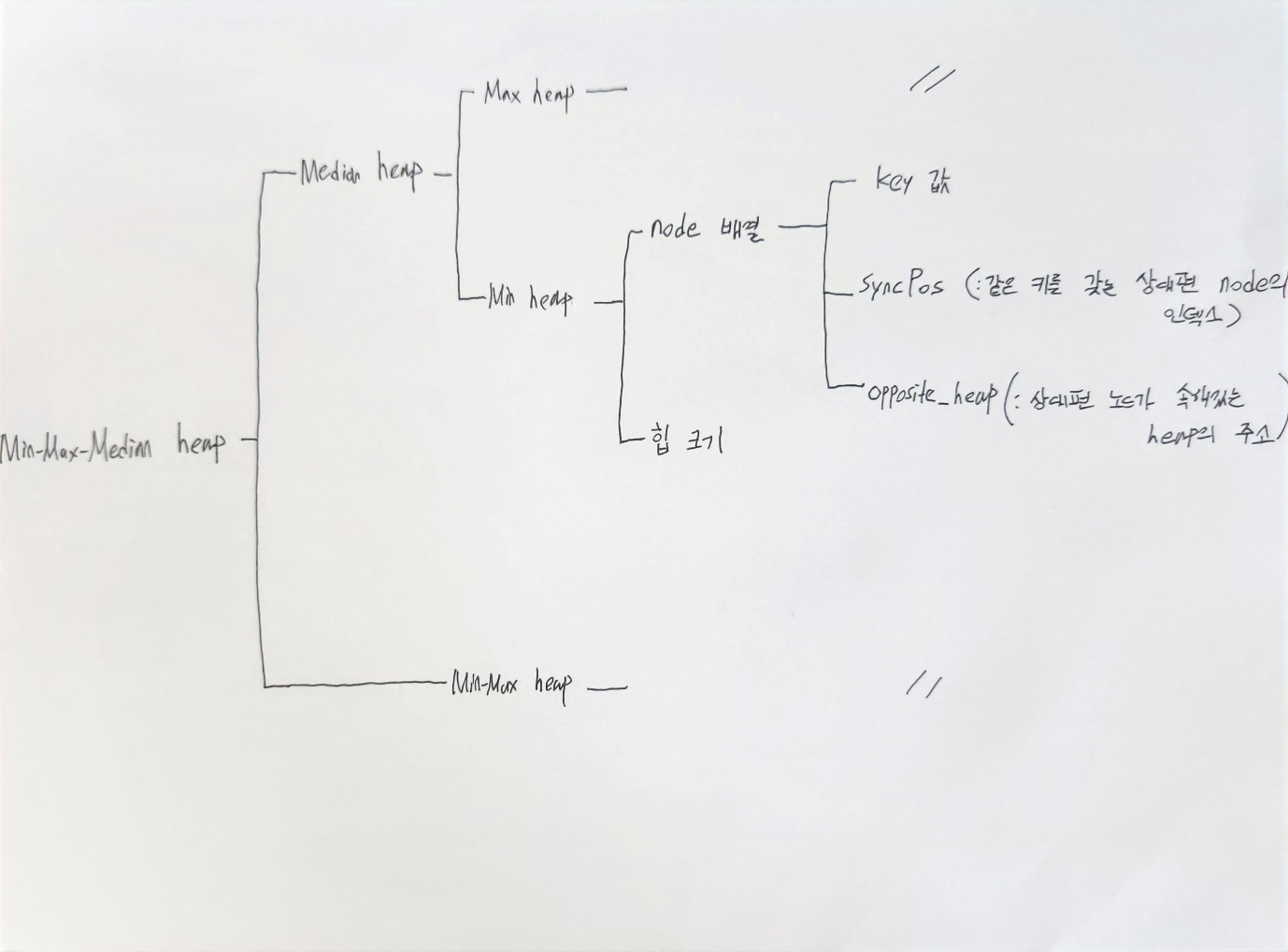
자료형, 매크로 상수와 같은 경우 대문자로 표기하였고 (ex. HEAP, NODE, MAX\_MODES) 변수명과 함수명의 경우 소문자와 snake case로 표기하였다. (ex. max\_heap, nodes, size)

함수의 경우 앞 단어에는 함수의 기능을 담아서 동사형으로 표기하였고 뒤의 단어의 경우 연산의 대상을 표기하였다. (ex. insert\_MEDIAN\_HEAP, insert\_MIN\_MAX\_HEAP)

소스코드의 핵심이 되는 줄들에는 주석을 달아 코드의 기능을 설명하였다.

1. **자료구조들의 관계:**

Min-Max-Median 우선순위 큐를 구현하기 위해 min-max heap과 median heap을 각각 구현한 다음, 같은 key값을 갖는 노드끼리 연동시켜 두 자료구조를 합쳤다. median heap은 max heap과 min heap, 두 heap으로 구현되었다. 각 자료구조들 간의 관계를 도식화하면 다음과 같다.



같은 key값을 갖는 노드끼리 서로의 위치를 알기 위해 ‘syncPos’에 상대편 노드의 인덱스 위치를, ‘opposite\_heap’에 상대편 노드가 속해 있는 heap의 주소를 저장하게 했다. 이로써 Median heap에서 진행한 연산의 결과가 Min-Max heap에서도 반영이 된다.

1. **각 자료구조 구현 방법:**

**Median heap:** max heap과 min heap, 두 힙을 이용하여 구현되었다. Max heap에는 중앙값 이하의 값을 갖는 노드들이 있고, Min heap에는 중앙값 초과의 값을 갖는 노드들이 있다. Max heap의 노드의 개수는 Min heap의 노드의 개수와 같거나 하나 크도록 유지한다. 이로써 중앙값은 Max heap의 root node의 key값이 된다.

**<함수>**

삽입: **void insert\_MEDIAN\_HEAP(int item)**

* Item을 key값으로 하는 노드를 삽입한다. 삽입하려는 key 값이 중앙값보다 크다면 min\_heap에 삽입하고 중앙값 이하라면 max\_heap에 삽입한다. 그 후, 최대 힙의 노드 개수와 최소 힙의 노드 개수 간의 균형을 맞추기 위해 balance\_MEDIAN\_HEAP 함수를 호줄한다.

삭제: **NODE delete\_MAX\_HEAP(int index),** **NODE delete\_MIN\_HEAP(int index)**

* 최대 힙 또는 최소 힙에서 해당 index의 노드를 삭제 후 heapify 하고 삭제한 노드의 정보를 반환한다.

탐색: **int find\_median()**

* 중앙값을 반환한다. 만약 heap이 비어 있다면 -1을 반환한다.

Heapify-down: **void heapifty\_down\_MAX\_HEAP(int index), void heapifty\_down\_MIN\_HEAP(int index)**

* 최대 힙 또는 최소 힙에서 해당 index의 노드부터 시작하여 lower level의 노드들에 대하여 Heapify-down 연산을 진행한다. Heapify-down 연산은 일반 최대 힙 또는 최소 힙에서의 heapify-down 연산과 동일하다.

Heapify-up: **void heapifty\_up\_MAX\_HEAP(int index), void heapifty\_up\_MAX\_HEAP(int index)**

* 최대 힙 또는 최소 힙에서 해당 index의 노드부터 시작하여 upper level의 노드들에 대하여 heapify-up 연산을 진행한다. Heapify-up 연산은 일반 최대 힙 또는 최소 힙에서의 heapify-up 연산과 동일하다.

노드 개수의 차 유지: **void balance\_MEDIAN\_HEAP()**

* 최대 힙의 노드의 개수와 최소 힙의 노드의 개수 간의 균형이 유지되도록 한다. 예를 들어, 만약 최대 힙의 노드의 개수가 최소 힙의 노드 개수보다 2 크다면 최대 힙의 root 노드를 pop하여 최소 힙에 삽입한다.

**Min-Max heap:** 차례로 min level, max level이 교차하면서 다음 두 조건을 만족하도록 하였다.

* 조건1: min level에 있는 노드는 해당 노드를 root 노드로 하는 subtree에서 가장 작은 값이다.
* 조건2: max level에 있는 노드는 해당 노드를 root 노드로 하는 subtree에서 가장 큰 값이다.

따라서, Min-Max heap에서 최솟값은 root 노드의 key 값이고, 최댓값은 root 노드의 child node들의 key 값 중 큰 값이다.

**<함수>**

삽입: **void insert\_MIN\_MAX\_HEAP(int item)**

* 주어진 item을 key값으로 하는 노드를 맨 마지막 위치에 삽입한 후, heapify한다. 먼저 삽입된 위치가 max level인지 min level인지 판단한다. max level의 경우, min level에 있는 부모 노드의 key값과 비교하여 작다면 부모 노드와 swap 연산을 한다. 그 후 해당 노드부터 시작하여 heapify-up 연산을 진행한다. min level의 경우에도 같은 방법으로 진행한다.

삭제: **NODE delete\_MIN\_MAX\_HEAP(int index)**

* 해당 index의 노드를 삭제 및 heapify를 하고 삭제된 노드의 정보를 반환한다. 먼저 해당 index의 노드를 맨 마지막 노드와 swap한 후, index 위치부터 heapify-dwon 연산을 진행한다. 이로써 index 위치의 lower level에 있는 노드들에 대하여 heapify가 완료된다.

그 후, index 위치에 대하여 heapify-up 연산을 진행하여 upper-level에 있는 노드들에 대하여도 heapify가 완료된다. 이로써 전체 heap에 대하여 heapify가 된다.

이 알고리즘으로 heapify가 유지되는 이유에 대한 증명은 아래와 같다.

탐색: **int find\_min(), int find\_max(), int foo()**

* 각각 최소인 key값, 최대인 key값, 최대인 key를 갖는 노드의 인덱스를 반환한다. 최솟값의 경우, min-max heap의 root 노드에 있고, 최댓값의 경우, root 노드의 child 노드 key값들 중 최대인 값이다. foo() 함수는 최대값을 삭제하는 delete\_max() 함수에서 사용된다.

Heapify-down: **void heapify\_down\_MIN\_MAX\_HEAP(int index)**

* 해당 index 노드의 lower level에 있는 노드들에 대하여 heapify-down을 한다.

먼저 index 위치가 min level인지, max level인지 판단한다. max level의 경우, child, grand child 노드들 중에서 최대인 key값을 갖는 노드의 인덱스(min\_index)를 찾아낸 후, min\_index 노드가 child 노드인지, grand child 노드인지 판단한다. ‘

child 노드인 경우, index 노드의 key값과 비교하여 child 노드의 key 값이 더 크다면 index 노드와 swap한다. 이 경우, 자동으로 lower level에 있는 노드들이 heap 성질을 만족하기에 더 이상 연산을 진행하지 않는다. 증명은 아래에 첨부하였다.

grand child 노드인 경우, index 노드의 key값과 비교하여 grand child 노드의 key값이 더 크다면 index 노드와 swap 연산을 진행한다. 그리고 부모 노드와 비교하여 부모 노드의 key 값이 더 크다면, 다시 swap 연산을 진행한다. 그 후, lower level에 대하여 heapify-down한다.

Index 위치가 min level인 경우에도 같은 방식으로 진행한다.

Heapify-up: **void heapify\_up\_MIN\_MAX\_HEAP(int index)**

* 해당 index 위치의 upper level에 대하여 heapify-up 연산을 진행한다.

먼저 해당 index 위치가 min level에 있는지, max level에 있는지 판단한다.

max level의 경우, 자신의 grand parent 노드의 key 값과 비교하여 자신의 key값이 더 크다면 swap 연산을 진행한다. grand parent 노드의 key값이 이하인 순간까지 반복한다.

min level의 경우에도 같은 방법으로 진행한다. 이에 대한 증명은 아래와 같다.

그 외: **int find\_max\_child(int index), int find\_min\_child(int index)**

* 각각 해당 index 노드의 grand child 노드들과 child 노드들 중에서 최대인 key 값을 갖는 노드의 인덱스와 최소인 key값을 갖는 노드의 인덱스를 반환한다. 이는 각각 index 노드를 root 노드로 하는 subtree에서 최대, 최소인 노드의 인덱스를 찾는 효과를 갖는다.

**Min-Max-Median Heap:** 위에서 설명한 Median heap과 Min-Max heap을 멤버변수로 포함한다. 같은 key 값을 갖는 노드들에 서로의 위치 정보를 저장하여 두 heap간에 연동을 하였다.

삽입: **void insert(int element)**

* ‘element’를 key값으로 갖는 노드를 삽입한다. insert\_MIN\_MAX\_HEAP(int item) 함수가 insert\_MEDIAN\_HEAP(int item) 함수의 기능을 포함하기 때문에 내부에서 insert\_MIN\_MAX\_HEAP(int item) 함수만 호출한다.

삭제: **int delete\_min(), int delete\_max(), int delete\_median()**

* delete\_min 함수와 delete\_max 함수의 경우, 먼저 Min-Max heap에서 해당 노드를 삭제한다. Median heap에서의 삭제하려는 노드의 위치를 알기 위해 중앙값과 비교한다. 중앙값보다 크다면 Median heap의 min heap에서 삭제하고 작다면 max heap에서 삭제한다. 그 후, Median heap의 성질을 유지하기 위해 balance\_MEDIAN\_HEAP 함수를 호줄한다.

delete\_median()의 경우, 중앙값은 항상 max heap의 root 노드에 있기에, max heap에서 root 노드를 삭제한 후, delete\_MIN\_MAX\_HEAP 함수를 이용하여 같은 key 값을 노드를 삭제한다. 그 후, Median heap의 성질을 유지하기 위해 balance\_MEDIAN\_HEAP 함수를 호출한다.

세 함수 모두 삭제된 노드의 key값을 반환하며, 만약 heap이 비어 있을 시, ‘-1’을 반환하여 아무런 동작을 하지 않는다.

탐색: **int find\_min(), int find\_max(), int find\_median()**

* 세 함수 모두 이미 위에서 설명하였다. 각각 최대, 최소, 중앙인 key값을 반환하고 삭제 연산은 진행하지 않는다. 만약 heap이 비어 있다면 ‘-1’을 반환하여 아무런 동작을 하지 않는다.

1. **Performance Analysis:**