알고리즘 실습(3주차)

201000287 일어일문학과 유다훈

1. 과제 설명 및 해결 방법

1. Max heap

부모노드보다 크지 않은 자식노드들로 구성된 max heap을 생성한다.

6개의 함수의 성능테스트를 한다.

Max-heapify

Build-Max-heap

Heap-Extract-Max

Max-Heapsort

Max-Heap-Insert

Max-Heap-Increase-key

C++의 clock()함수를 이용하여 시간측정 및 그래프 확인

2. Min heap

부모노드보다 큰 값을 가진 자식노드들로 구성된 min heap을 생성한다.

6개의 함수의 성능테스트를 한다.

Min-heapify

Build-Min -heap

Heap-Extract-Min

Min-Heapsort

Min-Heap-Insert

Min-Heap-decrease-key

C++의 clock()함수를 이용하여 시간측정 및 그래프 확인

3. 이론 과제

이론 수업의 자료 p.22에 나와있는 max-heapify에 대한 Recursion 수식을 설명하시오.

연습문제 6-2.5

2. 주요 부분 코드 설명(알고리즘 부분)

1. Max heap

- Max-Heapify

```
void max_heapify(int array[], int i, int arrayLength){

    int left = Left(i);
    int right = Right(i);
    int largest;

    if (left < arrayLength && array[left] > array[i]) {
        largest = left;
    } else {
        largest = i;
    }
    if (right < arrayLength && array[right] > array[largest])
        largest = right;
    if (largest != i) {
        Swap(array, i, largest);

        max_heapify(array, largest, arrayLength);
    }


}
```

  - 부모노드의 왼쪽자식과 오른쪽자식의 위치를 확인한다.
  - 부모노드와 왼쪽노드를 비교하여 큰 값을 확인하고, 그 값을 다시 오른쪽 노드와 확인하여 어느 세 개의 노드 중 어느 값이 제일 큰지 확인하여 자리를 바꾸는 작업을 한다.
  - 위 작업을 재귀적으로 반복하여 큰 값을 부모노드로 보내는 작업을 한다.

- Build-max-heap

```
void build_max_heap(int array[], int arrayLength) {
    for (int i  = floor(arrayLength/2); i >= 0; i-- ){
        max_heapify(array, i, arrayLength);
    }
}
```

  - 특정 위치를 지정하지 않고, 현재 정렬되어 있지 않은 데이터 전체를 max-heap으로 만든다.

- Heap-Extract-Max

```
int heap_extract_max(int array[], int arrayLength) {
    build_max_heap(array, arrayLength);
    int maxValue = array[0];
    Swap(array, 0, arrayLength-1);
    arrayLength = arrayLength-1;
    max_heapify(array, 1, arrayLength);
    return maxValue;
}
```

- Max heap에서 가장 큰 값을 제외한 나머지 값들을 max_heapify해준다.
- 가장 큰 값을 리턴해준다.

- Max-Heap-Sort

```
void max_heapsort(int array[], int arrayLength) {
    build_max_heap(array, arrayLength);

    for(int i= arrayLength-1; i > 0; i--) {
        Swap(array, 0, i);
        arrayLength = arrayLength-1;
        max_heapify(array, 0, arrayLength);
    }
}
```

- 내부 데이터들을 max heap을 만든 다음, 자리를 루트노드와 마지막 노드와 바꿔가며 오름차순으로 정렬한다.

- Max-Heap-Insert

```
void max_heap_insert(int array[], int key, int arrayLength) {
    array = (int *)realloc(array, (arrayLength * sizeof(int)));
    array[arrayLength-1] = -1111111;
    max_heap_increase_key(array, arrayLength-1, key);
}
```

- 임의의 값을 하나 데이터 배열에 추가해준다.
- max_heap_insert를 호출하는 main()함수에서, 해당 함수가 호출되기 직전에 배열의 길이 정보를 가진 변수 arrayLength의 값을 1개 늘려주고, 함수 내부에서 배열을 realloc을 통해 메모리를 늘려주어 배열의 사이즈를 늘려준다.
- 배열의 맨 마지막 값에 매우 작은 값을 넣어주고, 해당 위치에 키 값을 삽입한다.

- Max-Heap-Increase-Key

```
void max_heap_increase_key(int array[], int i, int key) {
    if (key > array[i]){
        array[i] = key;
        while( i>0 && array[Parent(i)] < array[i]) {
            Swap(array, i, Parent(i));
            i = Parent(i);
        }
    }
}
```

  - 전달받은 위치의 값과 삽입할 키값을 비교하여 키 값이 더 크다면, 전달 받은 위치의 값의 자리에 키 값을 넣어주어 교체한다.
  - 이후 부모노드와 자식노드의 값을 비교하여, 자식노드의 값이 크다면 자식노드와 부모노드의 위치를 바꾸어준다.

2. Min Heap

- Min-Heapify

```
void min_heapify(int array[], int i, int arrayLength){

    int left = Left(i);
    int right = Right(i);
    int smallest;

    if (left < arrayLength && array[left] <= array[i]) {
        smallest = left;
    } else {
        smallest = i;
    }
    if (right < arrayLength && array[right] <= array[smallest])
        smallest = right;
    if (smallest != i) {
        Swap(array, i, smallest);

        min_heapify(array, smallest, arrayLength);
    }
```

  - 부모노드의 왼쪽자식과 오른쪽자식의 위치를 확인한다.
  - 부모노드와 왼쪽노드를 비교하여 작은 값을 확인하고, 그 값을 다시 오른쪽 노드와 확인하여 어느 세 개의 노드 중 어느 값이 제일 작 지 확인하여 자리를 바꾸는 작업을 한다.
  - 위 작업을 재귀적으로 반복하여 작은 값을 부모 노드로 보내는 작업을 한다.

- Build-Min-Heap

```
void build_min_heap(int array[], int arrayLength) {
    for (int i  = floor(arrayLength/2); i >= 0; i-- ){
        min_heapify(array, i, arrayLength);
    }
}
```

  - 특정 위치를 지정하지 않고, 현재 정렬되어 있지 않은 데이터 전체를 min-heap으로 만든다.

- Heap-Extract-Min

```
int heap_extract_min(int array[], int arrayLength) {
    build_min_heap(array, arrayLength);
    int minValue = array[0];
    Swap(array, 0, arrayLength-1);
    arrayLength = arrayLength-1;
    min_heapify(array, 1, arrayLength);
    return minValue;
}
```

  - 최소값을 출력하는 함수
  - 최소값을 제외한 나머지값들에 대해 다시 min-heapify를 한다.

- Min-Heapsort

```
void min_heapsort(int array[], int arrayLength) {
    build_min_heap(array, arrayLength);

    for(int i= arrayLength-1; i >= 0; i--) {
        Swap(array, 0, i);
        arrayLength = arrayLength-1;
        min_heapify(array, 0, arrayLength-1);
    }
}
```

  - 자료들을 전부 min-heap으로 만든 다음, 맨 마지막 노드와 루트노드의 교환을 반복하여 내림차순의 자료로 만든다.

- Min-Heap-Insert

```
void min_heap_insert(int array[], int key, int arrayLength) {
    array = (int*)realloc(array, (arrayLength) * sizeof(int));
    array[arrayLength-1] = -11111111;
    min_heap_decrease_key(array, arrayLength-1, key);
}
```

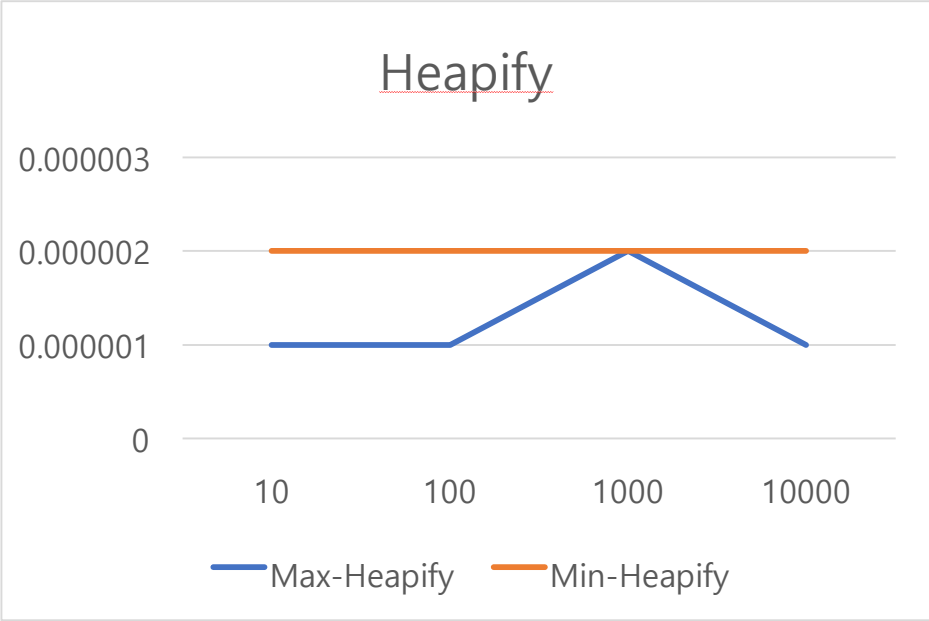
- 임의의 값을 하나 데이터 배열에 추가해준다.
- max_heap_insert를 호출하는 main()함수에서, 해당 함수가 호출되기 직전에 배열의 길이 정보를 가진 변수 arrayLength의 값을 1개 늘려주고, 함수 내부에서 배열을 realloc을 통해 메모리를 늘려주어 배열의 사이즈를 늘려준다.
- 배열의 맨 마지막 값에 매우 작은 값을 넣어주고, 해당 위치에 키 값을 삽입한다.

- Min-Heap-Decrease-Key

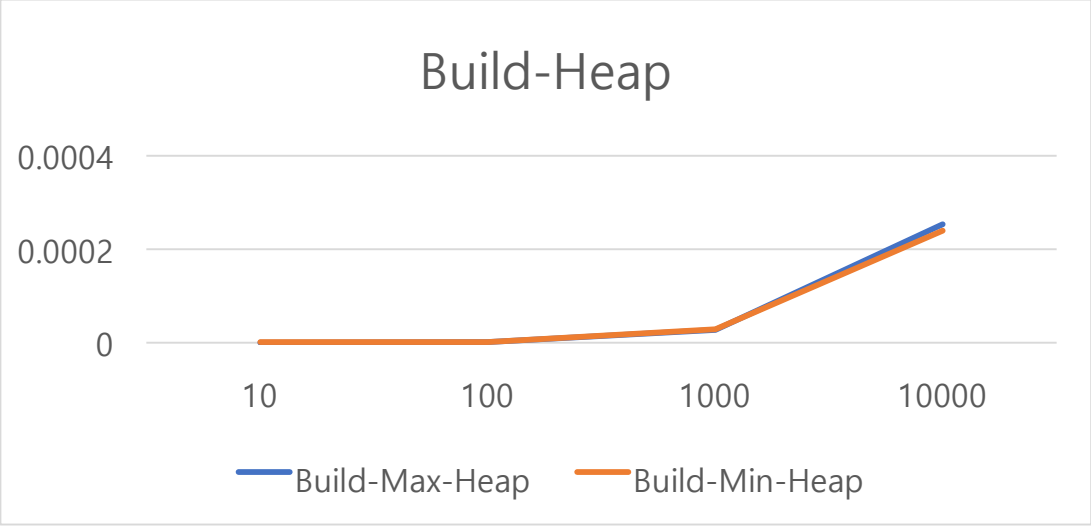
```
void min_heap_decrease_key(int array[], int i, int key) {
    if (key >= array[i]){
        array[i] = key;
        while( i > 0 && array[Parent(i)] >= array[i]) {
            Swap(array, i, Parent(i));
            i = Parent(i);
        }
    }
}
```

- 전달받은 위치의 값과 삽입할 키 값을 비교하여 키 값이 더 작다면, 전달 받은 위치의 값의 자리에 키 값을 넣어주어 교체한다.
- 이후 부모 노드와 자식 노드의 값을 비교하여, 자식 노드의 값이 작다면 자식노드와 부모노드의 위치를 교체하여 min-heap구조에 맞게 바꾸어 나간다..

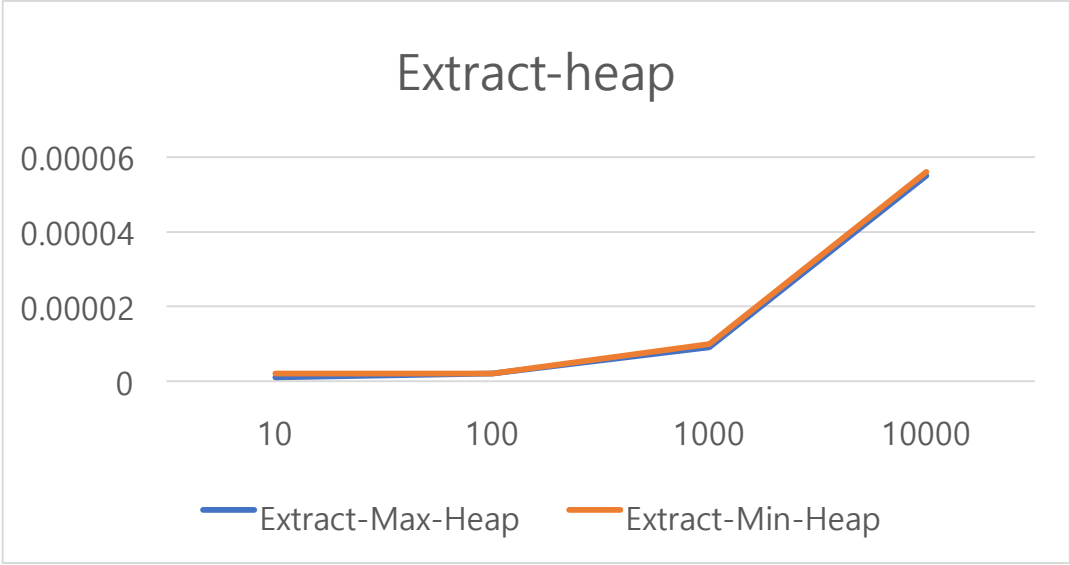
3. 결과(시간 복잡도 포함)



1개의 데이터만을 가지고 Heapify를 했기 때문에 시간 상의 큰 변화를 찾아 볼 수 없다.



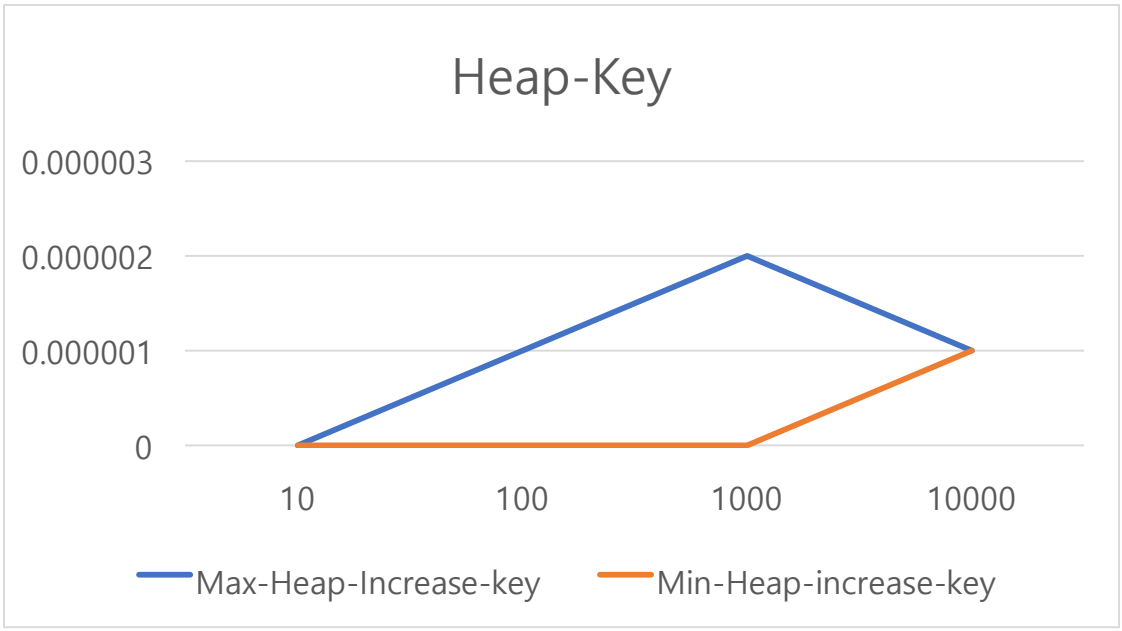
데이터 전체에 대해서 Build-Heap을 진행하였으므로 O(n log n)의 시간복잡도가 걸렸다.



특정 위치의 값을 뽑아 낸 후 나머지 값들에 대해 재정렬 하였으므로 O(n log n)만큼의 시간이 걸렸다.

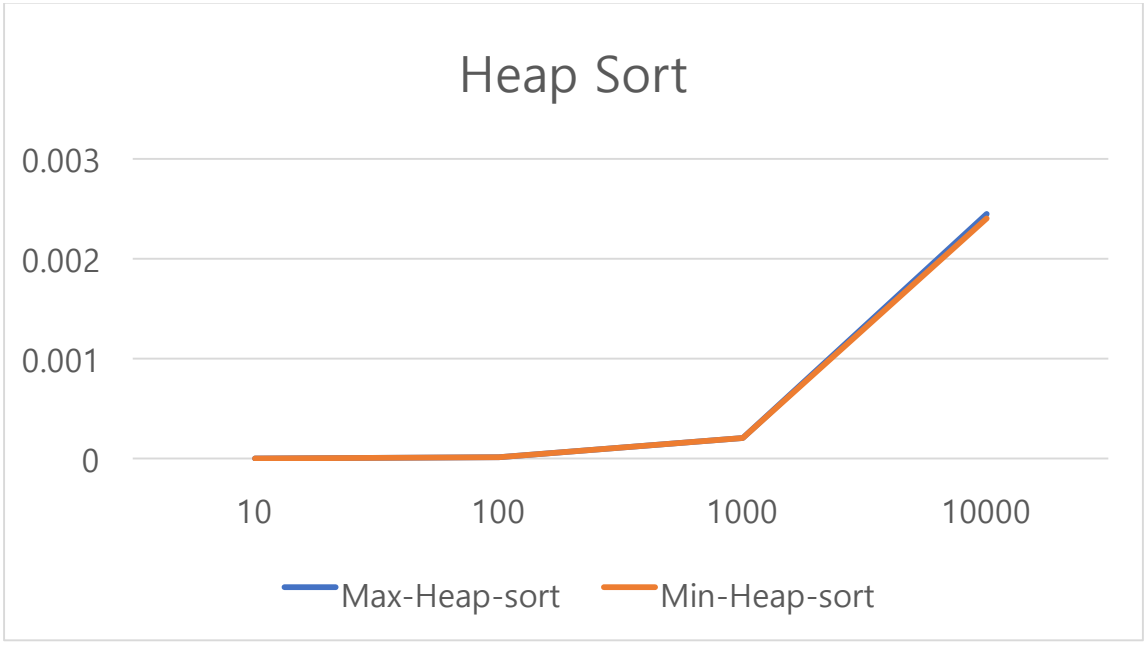
임의의 1개의 키 값을 삽입하였으나 데이터결과가 일정하지 않다.

가장 작은 값을 넣는 부분과 배열을 realloc해주는 부분에서 무언가의 간섭이 있는 것 같다.



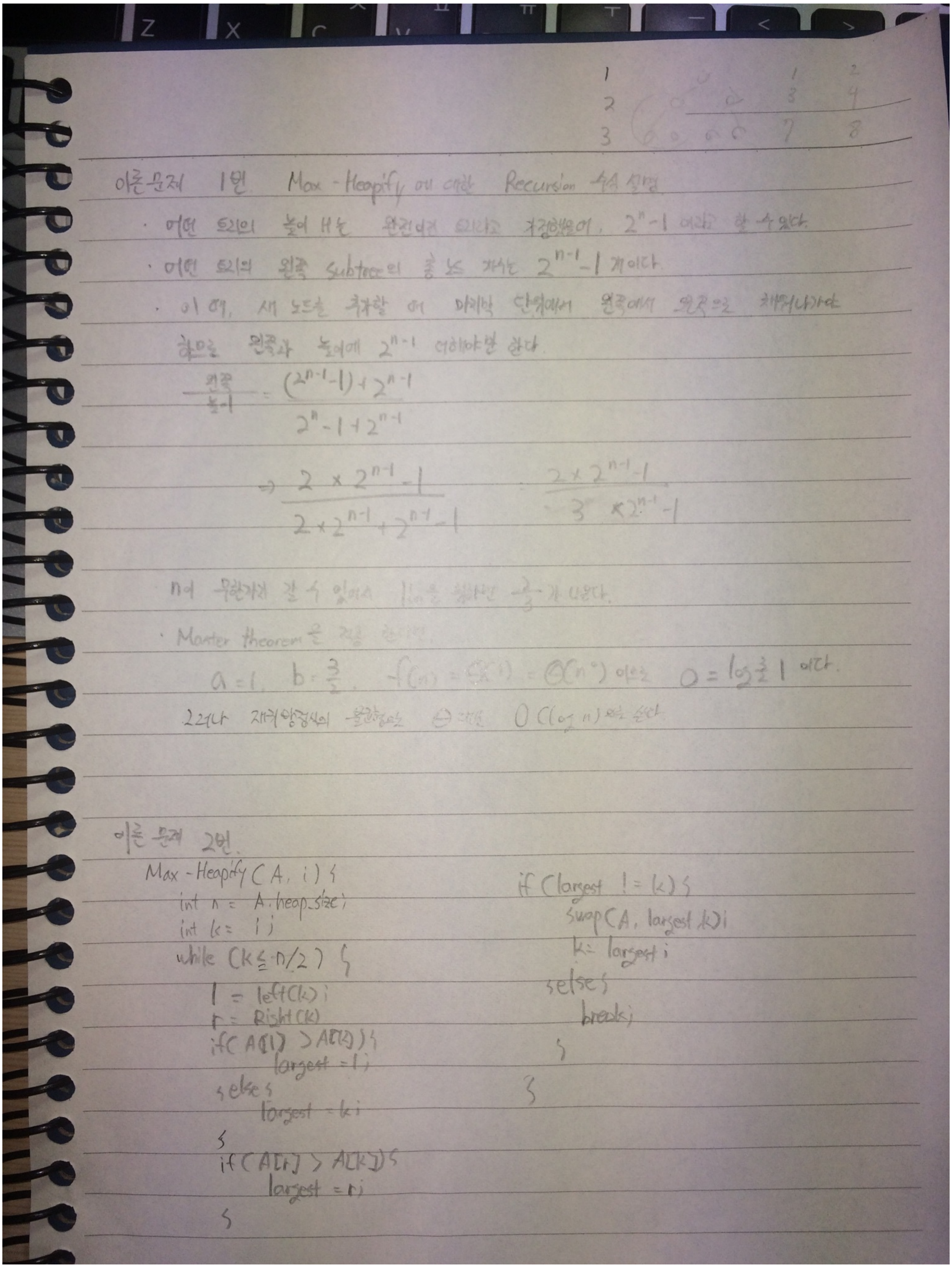
임의의 값 key를 삽입하여 원래 자리에 교체를 해주는 작업을 진행하였다.

부모 노드와 비교하여 스왑하는 시간에서 시간이 어느정도 들쭉날쭉 거렸으나 0.000001초 단위 이기 때문에 큰 신경은 쓰지 않아도 되는 수준이다.



전체적인 정렬 그래프는 O(n log n)의 시간복잡도 그래프를 나타내고 있다.

4. 이론 과제



이론문제 1번 Max-Heapify 에 대한 Recursion 수식 설명

- 어떤 트리의 높이 H는 완전이진 트리라고 가정했을 때, 2^n-1 이라고 할 수 있다.
- 어떤 트리의 왼쪽 subtree의 총 노드 개수는 $2^{n-1}-1$ 개이다.
- 이 때, 새 노드를 추가할 때 마지막 단계에서 왼쪽에서 오른쪽으로 채워나가야 하므로 왼쪽과 높이에 2^{n-1} 더해야만 한다.

$$\frac{\text{왼쪽}}{\text{높이}} = \frac{(2^{n-1}-1)+2^{n-1}}{2^n-1+2^{n-1}}$$

$$\Rightarrow \frac{2\times 2^{n-1}-1}{2\times 2^{n-1}+2^{n-1}-1} = \frac{2\times 2^{n-1}-1}{3\times 2^{n-1}-1}$$

n이 무한대로 갈 수 있어서 limit을 취하면 $\frac{2}{3}$가 나온다.

- Master theorem을 적용 한다면,

$a=1$, $b=\frac{3}{2}$, $f(n)=\Theta(1)=\Theta(n^0)$ 이므로 $0=\log_{\frac{3}{2}}1$ 이다.

그러나 재귀방정식의 풀이방법은 Θ 대신 $O(\log n)$ 써도 된다

이론 문제 2번.

```
Max-Heapify(A, i){
    int n = A.heap_size;
    int k = i;
    while (k ≤ n/2) {
        l = left(k);
        r = Right(k)
        if(A[l] > A[k]){
            largest = l;
        }else{
            largest = k;
        }
        if(A[r] > A[k]){
            largest = r;
        }
        if(largest != k){
            swap(A, largest, k);
            k = largest;
        }else{
            break;
        }
    }
}
```