In general, I use two heaps to implement the median-heap data structure.

One is a min-heap to store the element on the right of the median, say, right_heap.

The other is a max-heap to store the element on the left of the median, and the median itself, say left_heap.

If there are odd elements, size(left_heap) == size(right_heap) +1, else even elements, size(left_heap) == size(right_heap).

The median is always the peek element of the left_heap.

Firstly, the median-heap is initialized with empty left_heap and right_heap:

```
class MedianHeap:

    def __init__(self):
        self.left_heap = []
        self.right_heap = []
```

For Build(S), I will apply 3 helper functions:

1. find_median(arr): given an unsorted array, it can output the lower median of the array in O(n) time by a median of median algorithm.
2. make_min_heap(arr): this function works like heapq.heapify in Python, it turns an array into a min-heap in O(n) time.
3. make_max_heap(arr): this function will make an array into a max-heap in O(n) time.

The implementations of make_min_heap and make_max_heap are similar.

Let the last level, say the h^{th} level has m arbitrary elements, then we add m/2 arbitrary elements to the $(h - 1)^{th}$ level, and make adjustments so that each $(h-1)^{th}$ node with two children is a heap(max or min). For each adjustment, only at most 1 operation is needed, to switch the upper node down or not depending on comparison. For example, if a min-heap is needed but the upper node is larger than lower node, just switch the upper node with the smaller lower node and a min-heap is made. Thus, for the $(h - 1)^{th}$ level, at most h/2 * 1 operations are needed to build a heap.

For the $(h - 2)^{th}$ level, # operations = $m/2^2 * 2$, ..., for the $(h - i)^{th}$ level, # operations = $m/2^i * i$.

$\#$ total operations $= \frac{m}{2} \times 1 + \frac{m}{2^2} \times 2 + \cdots + \frac{m}{2^h} \times h$,

$h = \log n$,

$\frac{m}{2^h} = 1$. $m = 2^h = n$.

$$S = m\left(\frac{1}{2} + \frac{1}{2^2} \times 2 + \cdots + \frac{1}{2^h} \times h\right)$$

$$\frac{1}{2}S = m\left(\frac{1}{2^2} + \cdots + \frac{1}{2^h} \times (h-1) + \frac{1}{2^{h+1}} \times h\right)$$

$$S - \frac{1}{2}S = m\left(\frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^h} - \frac{h}{2^{h+1}}\right)$$

$$= m\left(\frac{\frac{1}{2}\left(1 - \frac{1}{2^h}\right)}{1 - \frac{1}{2}} - \frac{h}{2^{h+1}}\right)$$

$$= m\left(1 - \frac{1}{2^h} - \frac{h}{2^{h+1}}\right).$$

$$S = 2m\left(1 - \frac{1}{2^h} - \frac{h}{2^{h+1}}\right).$$

$m = n$, $h = \log n$, $2^h = n$.

$$\therefore S = 2n\left(1 - \frac{1}{n} - \frac{\log n}{2n}\right) = O(n).$$

With the clarification above, the Build(S) function is like this:

1. Find the median by find_median in O(n) time.
2. Get the half_size of the array by dividing in O(1) time.
3. Loop the elements in the array, compared to the median and add to left_heap or right_heap depending on 2 conditions in O(n) time:

If size of left_heap > half_size + 1, elements must be added to the right_heap;

If size of right_heap > half_size, elements must be added to the left_heap;

If both sizes are suited, elements less than or equal to the median are added to the left_heap, else to the right_heap.

4. Make the left_heap a max-heap and the right_heap a min-heap in O(n) time.

```
def build(self, arr):
    # step 1: find the median of the given unsorted array, O(n) time.
    median = find_median(arr)
    # step 2: get the half_size of the array, O(1) time.
    half_size = len(arr) // 2
    # step 3: loop the arr,
    # if the element is less than or equal to the median, add it to self.left_heap
    # if the element is larger than median, add it to self.right_heap
    # if the size of the right_heap is already larger than half_size, add elements to the left_heap
    # if the size of the left_heap is larger than half_size + 1, add elements to the right_heap
    # O(n) time
    for element in arr:
        if element <= median:
            if len(self.left_heap) > half_size + 1:
                self.right_heap.append(element)
            else:
                self.left_heap.append(element)
        else:
            if len(self.right_heap) > half_size:
                self.left_heap.append(element)
            else:
                self.right_heap.append(element)
    # step 4: make self.left a max-heap, and make self.right a min-heap
    # O(n) time
    make_min_heap(self.right_heap)
    make_max_heap(self.left_heap)
```

The 4 steps in total take O(n) time.

For Insert(x), I will apply 4 helper functions:

1. max_heap_insert(heap, x), this function takes in a max-heap and an element x, and insert x to the heap.

This function takes O(logn) time, because x is added to the end/right-most of the heap, and swapped up if it is larger than its parent to maintain a max-heap. As the height of heap is logn, the swap-up operation will take at most logn times, thus the time complexity of inserting is O(logn).

2. max_heap_pop(heap), this function takes in a max-heap, pop the root of the heap and maintain as a max-heap.

This function takes O(logn) time, because we firstly swap the root with the end/right-most of the heap, drop the end of the heap, i.e. the root value in O(1) time, and swap the new root value down if it is smaller than its child to maintain a max-heap. As the height of heap is logn, the swap-down operation will take at most logn times, thus the time complexity of popping is O(logn).

3. min_heap_insert(heap, x), this function takes in a min-heap and an element x, and insert x to the heap in O(logn) time. The implementation is similar to max_heap_insert.

4. min_heap_pop(heap), this function takes in a min-heap, pop the root of the heap and maintain as a min-heap in O(logn) time. The implementation is similar to max_heap_pop.

With the clarification above, the Insert(x) function is like this:

1. Get the current median in O(1) time by peek function.
2. Compare the x with the curr_median:

If x > curr_median, add x to the right_heap in O(logn) time by min_heap_insert function in O(logn) time, and balance the left_heap and the right_heap in O(logn) time.

For balance, if size of right_heap > size of left_heap, we pop out the minimum of right_heap by min_heap_pop function in O(logn) time and insert this element into the left_heap by max_heap_insert function in O(logn) time.

Else, add x to the left_heap by max_heap_insert function in O(logn) time, and balance the two heaps in O(logn) time.

For balance, if size of left_heap > size of right_heap + 1, we pop out the maximum of left_heap by max_heap_pop function in O(logn) time and insert this element into the right_heap by min_heap_insert function in O(logn) time.

Above steps take O(logn) time.

```python
def insert(self, x):
    # step 1: get the current median in O(1) time
    curr_median = self.peek()
    # step 2: compare x with current median, add it into left_heap or right_heap,
    # and balance the left_heap and right_heap in O(logn) time.
    if x > curr_median:
        min_heap_insert(self.right_heap, x)
        # balance
        if len(self.right_heap) > len(self.left_heap):
            element = min_heap_pop(self.right_heap)
            max_heap_insert(self.left_heap, element)
    else:  # x <= curr_median
        max_heap_insert(self.left_heap, x)
        # balance
        if len(self.left_heap) - len(self.right_heap) > 1:
            element = max_heap_pop(self.left_heap)
            min_heap_insert(self.right_heap, element)
```

For Peek(), the root of left_heap is just the median we want, we can return it in O(1) time.

```
def peek(self):
    '''
    return the root of the left_heap, i.e. the max-heap
    '''
    return self.left_heap[0]
```

For Extract():

1. The root of left_heap is the median, we pop it in O(logn) time by max_heap_pop function.

2. Balance the left_heap and right_heap if size of right_heap > size of left_heap by pop the minimum from right_heap in O(logn) time and insert it into left_heap in O(logn) time.

Such steps takes O(logn) time in total.

```
def extract(self):
    # step 1: pop the median, i.e. the root of the left_heap in O(logn) time.
    median = max_heap_pop(self.left_heap)
    # balance in O(logn) time.
    if len(self.right_heap) > len(self.left_heap):
        element = min_heap_pop(self.right_heap)
        max_heap_insert(self.left_heap, element)
    return median
```