```
##########################################
#
#                データ構造
#
##########################################

class UnionFind:
    def __init__(self,N): # 頂点数 N
        self.table = [i for i in range(N)]    # 親 table[x] == x で根
        self.rank  = [1 for i in range(N)]    # 木の長さ
        self.size  = [1 for i in range(N)]    # 集合のサイズ

    def Find(self,x):    #xの根を返す
        if self.table[x] == x:
            return x
        else:
            self.table[x] = self.Find(self.table[x]) #親の更新
            self.size[x] = self.size[self.table[x]]
            return table[x]

    def Unite(self,x,y,w): #xとyをdiff(x,y)=W で繋げる
        w = w - self.weight(y) + self.weight(x)
        x,y = self.Find(x), self.Find(y)
        sx,sy = self.Size(x), self.Size(y)
        if x == y: return
        if self.rank[x] > self.rank[y]:
            self.table[y] = x
            self.size[x] = sx + sy
        else:
            self.table[x] = y
            self.size[y] = sx + sy
            if self.rank[x] == self.rank[y]:
                self.rank[y] += 1

    def Check(self,x,y):
        return self.Find(x) == self.Find(y)

    def Size(self,x):
        return self.size[self.Find(x)]



class BinaryIndexedTree():
    def __init__(self,N):
        self.N = N
        self.bit = [0]*(self.N+1)

    def add(self,a,w):
        x = a
        while x <= self.N:
            self.bit[x] += w
            x += x & -x

    def sum(self,a):
        tmp = 0
        x = a
        while x > 0:
            tmp += self.bit[x]
            x -= x & -x
        return tmp



class SegmentTree:
    def __init__(self,N,d):
        self.NN = 1
        while self.NN < N:
            self.NN *= 2
        self.SegTree = [d]*(self.NN*2-1)

    def update(self,i,x): #iの値をxに更新
        i += self.NN - 1
```

```
        self.SegTree[i] = x
        while i>0:
            i = (i-1)//2
            self.SegTree[i] = self.process(self.SegTree[i*2+1],self.SegTree[i*2+2])

    def query(self,a,b,k=0,l=0,r=None): #[A,B)の値，呼ぶときはquery(a,b)
        if r == None: r = self.NN
        if r <= a or b <= l: #完全に含まない
            return INF
        elif a <= l and r <= b : #完全に含む
            return self.SegTree[k]
        else: #交差する
            vl = self.query(a,b,k*2+1,l,(l+r)//2)
            vr = self.query(a,b,k*2+2,(l+r)//2,r)
            return(self.process(vl,vr))

    def process(self,x,y): #x,yが子の時，親に返る値
        return min(x,y)



class PotentialUnionFind:
    def __init__(self,N): # 頂点数 N
        self.table = [i for i in range(N)]    # 親 table[x] == x で根
        self.rank  = [1 for i in range(N)]    # 木の長さ
        self.size  = [1 for i in range(N)]    # 集合のサイズ
        self.diffweight = [0 for i in range(N)]

    def Find(self,x):    #xの根を返す
        if self.table[x] == x:
            return x
        else:
            root = self.Find(self.table[x]) #親の更新
            self.size[x] = self.size[self.table[x]]
            self.diffweight[x] += self.diffweight[self.table[x]]
            self.table[x] = root
            return root

    def Unite(self,x,y,w): #xとyをDiff(x,y)=W で繋げる
        w = w - self.Weight(y) + self.Weight(x)
        x,y = self.Find(x), self.Find(y)
        sx,sy = self.Size(x), self.Size(y)
        if x == y: return
        if self.rank[x] > self.rank[y]:
            self.table[y] = x
            self.size[x] = sx + sy
            self.diffweight[y] = w
        else:
            self.table[x] = y
            self.size[y] = sx + sy
            self.diffweight[x] = -w
            if self.rank[x] == self.rank[y]:
                self.rank[y] += 1

    def Check(self,x,y):
        return self.Find(x) == self.Find(y)

    def Size(self,x):
        return self.size[self.Find(x)]

    def Weight(self,x): # 重さ(根からの距離)
        self.Find(x)
        return self.diffweight[x]

    def Diff(self,x,y): # 繋がってる二点間距離
        return self.Weight(y) - self.Weight(x)
```

```python

# 優先度付きキュー
class HeapQueue:
    def __init__(self,flag): # flag == true 最小値pop / False 最大値pop
        if flag: self.inv = 1
        else: self.inv = -1
        self.hq = []
        heapq.heapify(self.hq)

    def push(self,x):
        heapq.heappush(self.hq,self.inv*x)

    def pop(self):
        return self.inv * heapq.heappop(self.hq)


# 動的中央値管理
class DynamicMedian:
    def __init__(self):
        self.Hq = HeapQueue(True) #Lqより多く
        self.Hv = None
        self.Lv = None
        self.Lq = HeapQueue(False)
        self.N = 0

    def pop(self,x):
        if self.N == 0:
            self.Hv = x
        elif self.N == 1:
            if x <= self.Hv:
                self.Lv = x
            else:
                self.Lv,self.Hv = self.Hv,x
        elif self.N%2 == 0: # Hq と Lq が同数
            if self.Hv <= x:
                self.Hq.push(x)
            elif self.Lv < x < self.Hv:
                self.Hq.push(self.Hv)
                self.Hv = x
            elif x <= self.Lv:
                self.Hq.push(self.Hv)
                self.Hv = self.Lv
                self.Lv = self.Lq.pop()
                self.Lq.push(x)
        else: # Hq が一つ多い
            if self.Hv <= x:
                self.Lq.push(self.Lv)
                self.Lv = self.Hv
                self.Hv = self.Hq.pop()
                self.Hq.push(x)
            elif self.Lv < x < self.Hv:
                self.Lq.push(self.Lv)
                self.Lv = x
            elif x <= self.Lv:
                self.Lv.push(x)
        self.N += 1

    def median(self):
        if self.N%2 == 0:
            return (self.Hv+self.Lv)/2
        else:
            return self.Hv

```