Binary Heap

Binary Heap is a Binary Tree with following properties:

1) It's a complete tree

    All levels are completely filled

    the last level has all keys as left as possible.

2) A Binary Heap is either Min Heap or Max Heap

    In a Min Binary Heap, the key at root must be minimum among

    all keys present in Binary Heap.

    This is true for all nodes in Binary Tree.

min heap

```
         1(0)
        /    \
    3(1)      6(2)
    /  \      /
5(3)   9(4)  8(5)
```

Binary Heap can be stored in an array.

[1 3 6 5 9 8]

arr[0]	the root element
arr[(i-1)//2]	the parent node
arr[2*i+1]	the left child node
arr[2*i+2]	the right child node

The Level Order traversal method is used to get array representation.

max heap

```
        10(0)
        /  \
    5(1)  3(2)
    /   \
4(3)    1(4)
```

```
# heapify subtree rooted at index i, down-heap
# n is the size of heap
def heapify(arr, n, i):
    smallest = i    # Initialize smallest as root
    le = 2 * i + 1  # left  = 2*i + 1
    ri = 2 * i + 2  # right = 2*i + 2

    # See if left child exists and is smaller than root
    if le < n and arr[le] < arr[smallest]:
        smallest = le
    # See if right child exists and is smaller than root
    if ri < n and arr[ri] < arr[smallest]:
        smallest = ri

    # change root, if needed
    if smallest != i:
        # swap
        arr[i], arr[smallest] = arr[smallest], arr[i]

        # heapify the root at smallest
        heapify(arr, n, smallest)

# Build a mminheap, bottom-up heap construction
def makeHeap(arr):
    n = len(arr)
    for i in range(n//2 -1, -1, -1):
        heapify(arr, n, i)

# test
if __name__ == '__main__':
    arr = [3, 9, 5, 1, 6, 8]
    print('arr     =', arr)

    makeHeap(arr)
    print('minHeap =', arr)
```

Operations on Min Heap

---

1) getMin():

    It returns the root element of Min Heap.

    Time Complexity is O(1).

2) extractMin():

    Removes the minimum element from MinHeap.

    Time Complexity is O(logn) as this operation needs to

    maintain the heap property by calling heapify() after removing root.

3) decreaseKey():

    Decreases value of key. Time complexity is O(logn).

    If the decreased key value of a node

        is greater than the parent of the node,

        then we don't need to do anything.

    Otherwise, we need to traverse up

        to fix the violated heap property (up-heap).

4) increaseKey():

    Increases value of key. Time complexity is O(logn).

    If the increased key value of a node

        is less than the children of the node,

        then we don't need to do anything.

    Otherwise, we need to traverse down

        to fix the violated heap property (down-heap).

5) insert():

    Inserting a new key takes O(logn) time.

    We add a new key at the end of the tree.

    If new key is greater than its parent,

        then we don't need to do anything.

    Otherwise, we need to traverse up

        to fix the violated heap property (up-heap).

6) delete():

Deleting a key takes O(logn) time.

We replace the key to be deleted with

minimum infinite by calling decreaseKey().

then as the minus infinite is root,

we call extractMin() to remove the key.

```
class MinHeap():
    def __init__(self, arr):
        self.array = []          # tuple (key, value)
        self.pos = {}            # position of key in array
        self.size = len(arr)     # elements in min heap

        for i, item in enumerate(arr):
            self.array.append((item[0], item[1]))
            self.pos[item[0]] = i

        for i in range(self.size // 2, -1, -1):
            self.heapify(i)

    # display items of heap
    def display(self):
        print('array =', end=' ')
        for i in range(self.size):
            print(f'({self.array[i][0]} : {self.array[i][1]:2d})', end=' ')
        print()

    # is heap empty ?
    def isEmpty(self):
        return self.size == 0

    # i is the array index
    # modify array so that i roots a heap, down-heap
    def heapify(self, i):

        smallest = i
```

```
        le = 2 * i + 1
        ri = 2 * i + 2

        if le < self.size and self.array[le][1] < self.array[smallest][1]:
            smallest = le
        if ri < self.size and self.array[ri][1] < self.array[smallest][1]:
            smallest = ri

        if smallest != i:
            # update pos
            self.pos[self.array[smallest][0]] = i
            self.pos[self.array[i][0]] = smallest
            # swap
            self.array[smallest], self.array[i] = \
                self.array[i], self.array[smallest]

            self.heapify(smallest)

    # return the min element of the heap
    def getMin(self):
        if self.size == 0:
            return None

        return self.array[0]

    # return and remove the min element of the heap
    def extractMin(self):
        if self.size == 0:
            return None

        root = self.array[0]
        lastNode = self.array[self.size - 1]

        self.array[0] = lastNode

        # update pos
```

```
        self.pos[lastNode[0]] = 0
        del self.pos[root[0]]

        self.size -= 1
        self.heapify(0)

        return root

    # insert item (key, value)
    def insert(self, item):
        # insert an item at the end with big value
        if self.size < len(self.array):
            self.array[self.size] = (item[0], 10**80)
        else:
            self.array.append((item[0], 10**80))

        self.pos[item[0]] = self.size
        self.size += 1
        self.decreaseKey(item)

    # decreace value of item (key, value)
    def decreaseKey(self, item):
        i = self.pos[item[0]]
        val = item[1]

        # new value must be smaller than current
        if self.array[i][1] <= val:
            return

        self.array[i] = item

        # check if is smaller than parent
        p = (i - 1) // 2
        while i > 0 and self.array[i][1] < self.array[p][1]:
            # update pos
            self.pos[self.array[i][0]] = p
```

```
            self.pos[self.array[p][0]] = i
            # swap
            self.array[p], self.array[i] = self.array[i], self.array[p]

            i = p
            p = (i - 1) // 2

    # increace value of item (key, value)
    def increaseKey(self, item):
        i = self.pos[item[0]]
        val = item[1]

        # new value must be greater than current
        if self.array[i][1] >= val:
            return

        self.array[i] = item
        # check children
        self.heapify(i)



    # check if exists an item with key = v
    def isInMinHeap(self, v):
         pass

    # delete item
    # first reduce value of item to minus infinite and then calls extractMin()
    def deleteKey(self, item):
        self.decreaseKey((item[0], -1e100))
        self.extractMin()
```

```
# test
if __name__ == '__main__':
    arr = [('a', 50), ('b', 30), ('c', 10), ('d', 20), ('e', 40), ('f', 80)]

    h = MinHeap(arr)
    h.display()

    h.decreaseKey(('b', 2))
    h.display()
    h.increaseKey(('b', 30))
    h.display()

    h.extractMin()
    h.display()
    h.extractMin()
    h.display()

    h.insert(('x', 5))
    h.display()

    print('min=', h.getMin())

    h.deleteKey(('e', 0))
    h.display()
```