**Problem Set # 2 (Basic Datastructures and Heaps)**

Topics covered:

* Basic data-structures
* Heap data-structures
* Using heaps and arrays to realize interesting functionality.

**Problem 1 (Least-k Elements Datastructure)**

We saw how min-heaps can efficiently allow us to query the least element in a heap (array). We would like to modify minheaps in this exercise to design a data structure to maintain the **least k** elements for a given 𝑘≥1 with

𝑘=1

being the minheap data-structure.

Our design is to hold two arrays:

* (a) a sorted array A of 𝑘 elements that forms our least k elements; and
* (b) a minheap H with the remaining 𝑛−𝑘 elements.

Our data structure will itself be a pair of arrays (A,H) with the following property:

* H must be a minheap
* A must be sorted of size 𝑘.
* Every element of A must be smaller than every element of H.

The key operations to implement in this assignment include:

* insert a new element into the data-structure
* delete an existing element from the data-structure.

We will first ask you to design the data structure and them implement it.

**(A) Design Insertion Algorithm**

Suppose we wish to insert a new element with key 𝑗 into this data structure. Describe the pseudocode. Your pseudocode must deal with two cases: when the inserted element 𝑗 would be one of the least k elements i.e, it belongs to the array A; or when the inserted element belongs to the heap H. How would you distinguish between the two cases?

* You can assume that heap operations such as insert(H, key) and delete(H, index) are defined.
* Assume that the heap is indexed as  H[1],...,H[n -k] with H[0] being unused.
* Assume 𝑛>𝑘, i.e, there are already more than 𝑘 elements in the data structure.

What is the complexity of the insertion operation in the worst case in terms of 𝑘,𝑛.

**Unfortunately, we cannot grade your answer. We hope you will use this to design your datastructure on paper before attempting to code it up**

YOUR ANSWER HERE function insert(j): if A is empty or j < A[k-1]:

# Case 1: j belongs to A

if size(A) == k:

# Remove largest element from A (A[k-1])

removed\_element = A[k-1]

delete\_from\_A(k-1)

# Insert j into the sorted array A

insert\_into\_A(j)

# Insert the removed element from A into the heap H

insert\_into\_heap(removed\_element)

else:

# Case 2: j belongs to H

insert\_into\_heap(j)

**(B) Design Deletion Algorithm**

Suppose we wish to delete an index 𝑗 from the top-k array 𝐴. Design an algorithm to perform this deletion. Assume that the heap is not empty, in which case you can assume that the deletion fails.

* You can assume that heap operations such as insert(H, key) and delete(H, index) are defined.
* Assume that the heap is indexed as  H[1],...,H[n -k] with H[0] being unused.
* Assume 𝑛>𝑘, i.e, there are already more than 𝑘 elements in the data structure.

What is the complexity of the insertion operation in the worst case in terms of 𝑘,𝑛.

**Unfortunately, we cannot grade your answer. We hope you will use this to design your datastructure on paper before attempting to code it up**

YOUR ANSWER HERE function delete\_from\_A(j): if A is empty: return failure # Can't delete from empty array

if j < 0 or j >= k:

return failure # Invalid index for deletion

# Remove the element at index j

element\_to\_remove = A[j]

delete\_from\_A(j)

# If H is not empty, replace the deleted element with the smallest element from H

if size(H) > 0:

smallest\_from\_H = delete\_min\_from\_H() # Delete the smallest element from heap H

insert\_into\_A(smallest\_from\_H) # Insert this element into the sorted array A

return element\_to\_remove

**(C) Program your solution by completing the code below**

Note that although your algorithm design above assume that your are inserting and deleting from cases where 𝑛≥𝑘, the data structure implementation below must handle 𝑛<𝑘 as well. We have provided implementations for that portion to help you out.

*# First let us complete a minheap data structure.*

*# Please complete missing parts below.*

​

**class** MinHeap:

**def** \_\_init\_\_(self):

self.H **=** [**None**]

**def** size(self):

**return** len(self.H)**-**1

**def** \_\_repr\_\_(self):

**return** str(self.H[1:])

**def** satisfies\_assertions(self):

**for** i **in** range(2, len(self.H)):

**assert** self.H[i] **>=** self.H[i**//**2], f'Min heap property fails at position {i**//**2}, parent elt: {self.H[i**//**2]}, child elt: {self.H[i]}'

**def** min\_element(self):

**return** self.H[1]

​

*## bubble\_up function at index*

*## WARNING: this function has been cut and paste for the next problem as well*

**def** bubble\_up(self, index):

**assert** index **>=** 1

**if** index **==** 1:

**return**

parent\_index **=** index **//** 2

**if** self.H[parent\_index] **<** self.H[index]:

**return**

**else**:

self.H[parent\_index], self.H[index] **=** self.H[index], self.H[parent\_index]

self.bubble\_up(parent\_index)

*## bubble\_down function at index*

*## WARNING: this function has been cut and paste for the next problem as well*

**def** bubble\_down(self, index):

**assert** index **>=** 1 **and** index **<** len(self.H)

lchild\_index **=** 2 **\*** index

rchild\_index **=** 2 **\*** index **+** 1

*# set up the value of left child to the element at that index if valid, or else make it +Infinity*

lchild\_value **=** self.H[lchild\_index] **if** lchild\_index **<** len(self.H) **else** float('inf')

*# set up the value of right child to the element at that index if valid, or else make it +Infinity*

rchild\_value **=** self.H[rchild\_index] **if** rchild\_index **<** len(self.H) **else** float('inf')

*# If the value at the index is lessthan or equal to the minimum of two children, then nothing else to do*

**if** self.H[index] **<=** min(lchild\_value, rchild\_value):

**return**

*# Otherwise, find the index and value of the smaller of the two children.*

*# A useful python trick is to compare*

min\_child\_value, min\_child\_index **=** min ((lchild\_value, lchild\_index), (rchild\_value, rchild\_index))

*# Swap the current index with the least of its two children*

self.H[index], self.H[min\_child\_index] **=** self.H[min\_child\_index], self.H[index]

*# Bubble down on the minimum child index*

self.bubble\_down(min\_child\_index)

*# Function: heap\_insert*

*# Insert elt into heap*

*# Use bubble\_up/bubble\_down function*

**def** insert(self, elt):

self.H.append(elt)

**assert** self.H[self.size()] **==** elt

self.bubble\_up(self.size())

*# Function: heap\_delete\_min*

*# delete the smallest element in the heap. Use bubble\_up/bubble\_down*

**def** delete\_min(self):

**if** self.size() **>** 1:

self.H[1] **=** self.H.pop(**-**1)

self.bubble\_down(1)

**else**:

self.H.pop(**-**1)

h **=** MinHeap()

print('Inserting: 5, 2, 4, -1 and 7 in that order.')

h.insert(5)

print(f'\t Heap = {h}')

**assert**(h.min\_element() **==** 5)

h.insert(2)

print(f'\t Heap = {h}')

**assert**(h.min\_element() **==** 2)

h.insert(4)

print(f'\t Heap = {h}')

**assert**(h.min\_element() **==** 2)

h.insert(**-**1)

print(f'\t Heap = {h}')

**assert**(h.min\_element() **==** **-**1)

h.insert(7)

print(f'\t Heap = {h}')

**assert**(h.min\_element() **==** **-**1)

h.satisfies\_assertions()

​

print('Deleting minimum element')

h.delete\_min()

print(f'\t Heap = {h}')

**assert**(h.min\_element() **==** 2)

h.delete\_min()

print(f'\t Heap = {h}')

**assert**(h.min\_element() **==** 4)

h.delete\_min()

print(f'\t Heap = {h}')

**assert**(h.min\_element() **==** 5)

h.delete\_min()

print(f'\t Heap = {h}')

**assert**(h.min\_element() **==** 7)

*# Test delete\_max on heap of size 1, should result in empty heap.*

h.delete\_min()

print(f'\t Heap = {h}')

print('All tests passed: 10 points!')

​

**class** TopKHeap:

*# The constructor of the class to initialize an empty data structure*

**def** \_\_init\_\_(self, k):

self.k **=** k

self.A **=** []

self.H **=** MinHeap()

**def** size(self):

**return** len(self.A) **+** (self.H.size())

**def** get\_jth\_element(self, j):

**assert** 0 **<=** j **<** self.k**-**1

**assert** j **<** self.size()

**return** self.A[j]

**def** satisfies\_assertions(self):

*# is self.A sorted*

**for** i **in** range(len(self.A) **-**1 ):

**assert** self.A[i] **<=** self.A[i**+**1], f'Array A fails to be sorted at position {i}, {self.A[i], self.A[i**+**1]}'

*# is self.H a heap (check min-heap property)*

self.H.satisfies\_assertions()

*# is every element of self.A less than or equal to each element of self.H*

**for** i **in** range(len(self.A)):

**assert** self.A[i] **<=** self.H.min\_element(), f'Array element A[{i}] = {self.A[i]} is larger than min heap element {self.H.min\_element()}'

*# Function : insert\_into\_A*

*# This is a helper function that inserts an element `elt` into `self.A`.*

*# whenever size is < k,*

*# append elt to the end of the array A*

*# Move the element that you just added at the very end of*

*# array A out into its proper place so that the array A is sorted.*

*# return the "displaced last element" jHat (None if no element was displaced)*

**def** insert\_into\_A(self, elt):

k **=** self.k

print("k = ", k)

*# assert(self.size() < k)*

self.A.append(elt)

j **=** len(self.A)**-**1

**while** (j **>=** 1 **and** self.A[j] **<** self.A[j**-**1]):

*# Swap A[j] and A[j-1]*

(self.A[j], self.A[j**-**1]) **=** (self.A[j**-**1], self.A[j])

j **=** j **-**1

**return**

*# bug here. it should be self.k (or add k = self.k)*

*# bug assert(self.size() < k) ????*

*# Function: insert -- insert an element into the data structure.*

*# Code to handle when self.size < self.k is already provided*

**def** insert(self, elt):

size **=** self.size()

*# If we have fewer than k elements, handle that in a special manner*

**if** size **<=** self.k:

self.insert\_into\_A(elt)

**return**

self.insert\_into\_A(elt)

self.H.insert(self.A.pop(**-**1))

*# Function: Delete top k -- delete an element from the array A*

*# In particular delete the j^{th} element where j = 0 means the least element.*

*# j must be in range 0 to self.k-1*

**def** delete\_top\_k(self, j):

k **=** self.k

**assert** self.size() **>** k *# we need not handle the case when size is less than or equal to k*

**assert** j **>=** 0

**assert** j **<** self.k

self.A.pop(j)

self.A.append(self.H.min\_element())

self.H.delete\_min()

h **=** TopKHeap(5)

*# Force the array A*

h.A **=** [**-**10, **-**9, **-**8, **-**4, 0]

*# Force the heap to this heap*

[h.H.insert(elt) **for** elt **in** [1, 4, 5, 6, 15, 22, 31, 7]]

​

print('Initial data structure: ')

print('\t A = ', h.A)

print('\t H = ', h.H)

​

*# Insert an element -2*

print('Test 1: Inserting element -2')

h.insert(**-**2)

print('\t A = ', h.A)

print('\t H = ', h.H)

*# After insertion h.A should be [-10, -9, -8, -4, -2]*

*# After insertion h.H should be [None, 0, 1, 5, 4, 15, 22, 31, 7, 6]*

**assert** h.A **==** [**-**10,**-**9,**-**8,**-**4,**-**2]

**assert** h.H.min\_element() **==** 0 , 'Minimum element of the heap is no longer 0'

h.satisfies\_assertions()

​

print('Test2: Inserting element -11')

h.insert(**-**11)

print('\t A = ', h.A)

print('\t H = ', h.H)

**assert** h.A **==** [**-**11, **-**10, **-**9, **-**8, **-**4]

**assert** h.H.min\_element() **==** **-**2

h.satisfies\_assertions()

​

print('Test 3 delete\_top\_k(3)')

h.delete\_top\_k(3)

print('\t A = ', h.A)

print('\t H = ', h.H)

h.satisfies\_assertions()

**assert** h.A **==** [**-**11,**-**10,**-**9,**-**4,**-**2]

**assert** h.H.min\_element() **==** 0

h.satisfies\_assertions()

​

print('Test 4 delete\_top\_k(4)')

h.delete\_top\_k(4)

print('\t A = ', h.A)

print('\t H = ', h.H)

**assert** h.A **==** [**-**11, **-**10, **-**9, **-**4, 0]

h.satisfies\_assertions()

​

print('Test 5 delete\_top\_k(0)')

h.delete\_top\_k(0)

print('\t A = ', h.A)

print('\t H = ', h.H)

**assert** h.A **==** [**-**10, **-**9, **-**4, 0, 1]

h.satisfies\_assertions()

​

print('Test 6 delete\_top\_k(1)')

h.delete\_top\_k(1)

print('\t A = ', h.A)

print('\t H = ', h.H)

**assert** h.A **==** [**-**10, **-**4, 0, 1, 4]

h.satisfies\_assertions()

print('All tests passed - 15 points!')

​

---------------------------------------------------------------------------

NameError Traceback (most recent call last)

<ipython-input-6-4dc7c72648d8> in <module>

----> 1 h = TopKHeap(5)

**2** # Force the array A

**3** h.A = [-10, -9, -8, -4, 0]

**4** # Force the heap to this heap

**5** [h.H.insert(elt) for elt in [1, 4, 5, 6, 15, 22, 31, 7]]

<ipython-input-5-64d5143cc7f6> in \_\_init\_\_(self, k)

**3** self.k = k

**4** self.A = [] # List to store all elements

----> 5 self.H = MinHeap() # Min-heap to store top-k elements

**6**

**7** def insert(self, elt):

NameError: name 'MinHeap' is not defined

**Problem 2: Heap data structure to mantain/extract median (instead of minimum/maximum key)**

We have seen how min-heaps can efficiently extract the smallest element efficiently and maintain the least element as we insert/delete elements. Similarly, max-heaps can maintain the largest element. In this exercise, we combine both to maintain the "median" element.

The median is the middle element of a list of numbers.

* If the list has size 𝑛 where 𝑛 is odd, the median is the (𝑛−1)/2𝑡ℎ element where 0𝑡ℎ is least and (𝑛−1)𝑡ℎ is the maximum.
* If 𝑛 is even, then we designate the median the average of the (𝑛/2−1)𝑡ℎ and (𝑛/2)𝑡ℎ elements.

**Example**

* List is [−1,5,4,2,3] has size 5, the median is the 2𝑛𝑑 element (remember again least element is designated as 0𝑡ℎ) which is 3.
* List is [−1,3,2,1] has size 4. The median element is the average of  1𝑠𝑡 element (1) and 2𝑛𝑑 element (2) which is  1.5.

**Maintaining median using two heaps.**

The data will be maintained as the union of the elements in two heaps 𝐻min and 𝐻max, wherein 𝐻min is a min-heap and 𝐻max is a max-heap. We will maintain the following invariant:

* The max element of  𝐻max will be less than or equal to the min element of  𝐻min.
* The sizes of 𝐻𝑚𝑎𝑥 and 𝐻𝑚𝑖𝑛 are equal (if number of elements in the data structure is even) or 𝐻𝑚𝑎𝑥 may have one less element than 𝐻𝑚𝑖𝑛 (if the number of elements in the data structure is odd).

**(A) Design algorithm for insertion.**

Suppose, we have the current data split between 𝐻𝑚𝑎𝑥 and 𝐻𝑚𝑖𝑛 and we wish to insert an element 𝑒 into the data structure, describe the algorithm you will use to insert. Your algorithm must decide which of the two heaps will 𝑒 be inserted into and how to maintain the size balance condition.

Describe the algorithm below and the overall complexity of an insert operation. This part will not be graded.

YOUR ANSWER HERE function insert(e): if Hmax is empty or e <= max(Hmax): insert\_into\_Hmax(e) else: insert\_into\_Hmin(e)

# Balance the heaps

if size(Hmax) > size(Hmin):

move\_max\_from\_Hmax\_to\_Hmin()

elif size(Hmin) > size(Hmax):

move\_min\_from\_Hmin\_to\_Hmax()

**(B) Design algorithm for finding the median.**

Implement an algorithm for finding the median given the heaps 𝐻min and 𝐻max. What is its complexity?

YOUR ANSWER HERE function get\_median(): if size(Hmin) == size(Hmax):

# Even number of elements, median is the average of the two middle elements

median = (max(Hmax) + min(Hmin)) / 2

else:

# Odd number of elements, median is the root of Hmax

median = max(Hmax)

return median

**(C) Implement the algorithm**

Complete the implementation for maxheap data structure. First complete the implementation of MaxHeap. You can cut and paste relevant parts from previous problems although we do not really recommend doing that. A better solution would have been to write a single implementation that could have served as min/max heap based on a flag.

**class** MaxHeap:

**def** \_\_init\_\_(self):

self.H **=** [**None**]

**def** size(self):

**return** len(self.H)**-**1

**def** \_\_repr\_\_(self):

**return** str(self.H[1:])

**def** satisfies\_assertions(self):

**for** i **in** range(2, len(self.H)):

**assert** self.H[i] **<=** self.H[i**//**2], f'Maxheap property fails at position {i**//**2}, parent elt: {self.H[i**//**2]}, child elt: {self.H[i]}'

**def** max\_element(self):

**return** self.H[1]

**def** bubble\_up(self, index):

*# recycle bubble\_up in min-heap, modify to Funcitonal programming style*

**assert** index **>=** 1

**if** index **==** 1:

**return** self.H

parent\_index **=** index **//** 2

**if** self.H[parent\_index] **>** self.H[index]:

**return** self.H

**else**:

self.H[parent\_index], self.H[index] **=** self.H[index], self.H[parent\_index]

**return** self.bubble\_up(parent\_index)

**def** bubble\_down(self, index):

*# recycle bubble\_down in min-heap*

**assert** index **>=** 1 **and** index **<** len(self.H)

lchild\_index **=** 2 **\*** index

rchild\_index **=** 2 **\*** index **+** 1

*# set up the value of left child to the element at that index if valid, or else make it +Infinity*

lchild\_value **=** self.H[lchild\_index] **if** lchild\_index **<** len(self.H) **else** **-**float('inf')

*# set up the value of right child to the element at that index if valid, or else make it +Infinity*

rchild\_value **=** self.H[rchild\_index] **if** rchild\_index **<** len(self.H) **else** **-**float('inf')

*# If the value at the index is lessthan or equal to the minimum of two children, then nothing else to do*

**if** self.H[index] **>=** max(lchild\_value, rchild\_value):

**return** self.H

*# Otherwise, find the index and value of the smaller of the two children.*

*# A useful python trick is to compare*

min\_child\_value, min\_child\_index **=** max ((lchild\_value, lchild\_index), (rchild\_value, rchild\_index))

*# Swap the current index with the least of its two children*

self.H[index], self.H[min\_child\_index] **=** self.H[min\_child\_index], self.H[index]

*# Bubble down on the minimum child index*

**return** self.bubble\_down(min\_child\_index)

*# Function: insert*

*# Insert elt into minheap*

*# Use bubble\_up/bubble\_down function*

**def** insert(self, elt):

*#(recycle)*

self.H.append(elt)

**assert** self.H[self.size()] **==** elt

*# use bubble\_up*

self.bubble\_up(self.size())

*# Function: delete\_max*

*# delete the largest element in the heap. Use bubble\_up/bubble\_down*

**def** delete\_max(self):

*#(recycle)*

**if** self.size() **>** 1:

self.H[1] **=** self.H.pop(**-**1)

self.bubble\_down(1)

**else**:

self.H.pop(**-**1)

h **=** MaxHeap()

print('Inserting: 5, 2, 4, -1 and 7 in that order.')

h.insert(5)

print(f'\t Heap = {h}')

**assert**(h.max\_element() **==** 5)

h.insert(2)

print(f'\t Heap = {h}')

**assert**(h.max\_element() **==** 5)

h.insert(4)

print(f'\t Heap = {h}')

**assert**(h.max\_element() **==** 5)

h.insert(**-**1)

print(f'\t Heap = {h}')

**assert**(h.max\_element() **==** 5)

h.insert(7)

print(f'\t Heap = {h}')

**assert**(h.max\_element() **==** 7)

h.satisfies\_assertions()

​

print('Deleting maximum element')

h.delete\_max()

print(f'\t Heap = {h}')

**assert**(h.max\_element() **==** 5)

h.delete\_max()

print(f'\t Heap = {h}')

**assert**(h.max\_element() **==** 4)

h.delete\_max()

print(f'\t Heap = {h}')

**assert**(h.max\_element() **==** 2)

h.delete\_max()

print(f'\t Heap = {h}')

**assert**(h.max\_element() **==** **-**1)

*# Test delete\_max on heap of size 1, should result in empty heap.*

h.delete\_max()

print(f'\t Heap = {h}')

print('All tests passed: 5 points!')

​

**import** heapq

​

**class** MedianMaintainingHeap:

**def** \_\_init\_\_(self):

self.max\_heap **=** [] *# Max-heap (inverted values for min-heap behavior)*

self.min\_heap **=** [] *# Min-heap*

​

**def** insert(self, elt):

*# Insert into one of the heaps*

**if** len(self.min\_heap) **==** 0 **or** elt **>=** self.min\_heap[0]:

heapq.heappush(self.min\_heap, elt) *# Insert into min-heap*

**else**:

heapq.heappush(self.max\_heap, **-**elt) *# Insert into max-heap (invert value)*

​

*# Balance the heaps if necessary*

**if** len(self.min\_heap) **>** len(self.max\_heap) **+** 1:

heapq.heappush(self.max\_heap, **-**heapq.heappop(self.min\_heap))

**elif** len(self.max\_heap) **>** len(self.min\_heap):

heapq.heappush(self.min\_heap, **-**heapq.heappop(self.max\_heap))

​

**def** get\_median(self):

*# The median is the root of the min-heap if it has more elements*

**if** len(self.min\_heap) **>** len(self.max\_heap):

**return** float(self.min\_heap[0])

*# Otherwise, it is the average of the roots of the two heaps*

**return** (self.min\_heap[0] **-** self.max\_heap[0]) **/** 2.0

​

**def** satisfies\_assertions(self):

*# Optional: Custom assertions to check the heaps' balance and properties*

**pass**

​

**def** \_\_repr\_\_(self):

**return** f'MedianMaintainingHeap(min\_heap={self.min\_heap}, max\_heap={self.max\_heap})'

​

m **=** MedianMaintainingHeap()

print('Inserting 1, 5, 2, 4, 18, -4, 7, 9')

​

m.insert(1)

print(m)

print(m.get\_median())

m.satisfies\_assertions()

**assert** m.get\_median() **==** 1, f'expected median = 1, your code returned {m.get\_median()}'

​

m.insert(5)

print(m)

print(m.get\_median())

m.satisfies\_assertions()

**assert** m.get\_median() **==** 3, f'expected median = 3.0, your code returned {m.get\_median()}'

​

m.insert(2)

print(m)

print(m.get\_median())

m.satisfies\_assertions()

​

**assert** m.get\_median() **==** 2, f'expected median = 2, your code returned {m.get\_median()}'

m.insert(4)

print(m)

print(m.get\_median())

m.satisfies\_assertions()

**assert** m.get\_median() **==** 3, f'expected median = 3, your code returned {m.get\_median()}'

​

m.insert(18)

print(m)

print(m.get\_median())

m.satisfies\_assertions()

**assert** m.get\_median() **==** 4, f'expected median = 4, your code returned {m.get\_median()}'

​

m.insert(**-**4)

print(m)

print(m.get\_median())

m.satisfies\_assertions()

**assert** m.get\_median() **==** 3, f'expected median = 3, your code returned {m.get\_median()}'

​

m.insert(7)

print(m)

print(m.get\_median())

m.satisfies\_assertions()

**assert** m.get\_median() **==** 4, f'expected median = 4, your code returned {m.get\_median()}'

​

m.insert(9)

print(m)

print(m.get\_median())

m.satisfies\_assertions()

**assert** m.get\_median()**==** 4.5, f'expected median = 4.5, your code returned {m.get\_median()}'

​

print('All tests passed: 15 points')

​

**Solutions to Manually Graded Portions**

**Problem 1 A**

In order to insert a new element j, we will first distinguish between two cases:

* 𝑗<𝐴[𝑘−1] : In this case 𝑗 belongs to the array 𝐴.
  + First, let 𝑗′=𝐴[𝑘−1].
  + Replace 𝐴[𝑘−1] by 𝑗.
  + Perform an insertion to move 𝑗 into its correct place in the sorted array 𝐴.
  + Insert 𝑗′ into the heap using heap insert.
* 𝑗≥𝐴[𝑘−1]: In this case, 𝑗 belongs to the heap 𝐻.
  + Insert 𝑗 into the heap using heap-insert.

In terms of 𝑘,𝑛, the worst case complexity is Θ(𝑘+log(𝑛)) for each insertion operation.

**Problem 1B**

* First, in order to delete the index j from array, move elements from j+1 .. k-1 left one position.
* Insert the minimum heap element at position 𝑘−1 of the array A.
* Delete the element at index 1 of the heap.

Overall complexity = Θ(𝑘+log(𝑛)) in the worst case.

**Problem 2 A**

Let 𝑎 be the largest element in 𝐻max and 𝑏 be the least element in 𝐻min.

* If 𝑒𝑙𝑡<𝑎, then we insert the new element into 𝐻max.
* If 𝑒𝑙𝑡>=𝑎, then we insert the new element into 𝐻min.

If the size of 𝐻max and 𝐻min differ by 2, then

* If 𝐻max is larger then, extract the largest element from 𝐻max andd insert into 𝐻min.
* If 𝐻min is larger then, extract the least element from 𝐻min andd insert into 𝐻max.

The overall complexity is Θ(log(𝑛)).

**Problem 2 B**

If sizes of heaps are the same, then median is the average of maximum element of the max heap and minimum element of the minheap.

Otherwise, the median is simply the minimum elemment of the min-heap.

Overall complexity is Θ(1).

**That's all folks**

Thanks for stopping by ! Sulay ☺