

ProblemSet2__Solutions

March 27, 2025

1 Problem Set # 2 (Basic Datastructures and Heaps)

Topics covered: - Basic data-structures - Heap data-structures - Using heaps and arrays to realize interesting functionality.

1.1 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 $k \geq 1$ 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 then implement it.

1.1.1 (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 $H[1], \dots, H[n - k]$ 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

1.1.2 (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 $H[1], \dots, H[n-k]$ 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

1.2 (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 $n \geq k$, 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 less than 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):
    # your code here

# Function: heap_delete_min
# delete the smallest element in the heap. Use bubble_up/bubble_down
def delete_min(self):
    # your code here

```

```

[ ]: 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):
        print("k = ", self.k)
        assert(self.size() < self.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

# 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
    # Code up your algorithm here.
    # your code here

# Function: Delete top k -- delete an element from the array A
# In particular delete the jth 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
    # your code here

```

```

[ ]: 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!')

```

1.3 Problem 2: Heap data structure to maintain/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 $(n - 1)/2^{th}$ element where 0^{th} 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 2^{nd} element (remember again least element is designated as 0^{th}) which is 3.
- List is $[-1, 3, 2, 1]$ has size 4. The median element is the average of 1^{st} element (1) and 2^{nd} element (2) which is 1.5.

1.4 Maintaining median using two heaps.

The data will be maintained as the union of the elements in two heaps H_{min} and H_{max} , wherein H_{min} is a min-heap and H_{max} is a max-heap. We will maintain the following invariant: - The max element of H_{max} will be less than or equal to the min element of H_{min} . - The sizes of H_{max} and H_{min} are equal (if number of elements in the data structure is even) or H_{max} may have one less element than H_{min} (if the number of elements in the data structure is odd).

1.5 (A) Design algorithm for insertion.

Suppose, we have the current data split between H_{max} and H_{min} 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

1.6 (B) Design algorithm for finding the median.

Implement an algorithm for finding the median given the heaps H_{min} and H_{max} . What is its complexity?

YOUR ANSWER HERE

1.7 (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):
        # your code here

    def bubble_down(self, index):
        # your code here

    # Function: insert
    # Insert elt into minheap
    # Use bubble_up/bubble_down function
    def insert(self, elt):
        # your code here

    # Function: delete_max
    # delete the largest element in the heap. Use bubble_up/bubble_down
    def delete_max(self):
        # your code here
```

```
[ ]: 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!')
```

```
[ ]: class MedianMaintainingHeap:
    def __init__(self):
        self.hmin = MinHeap()
        self.hmax = MaxHeap()

    def satisfies_assertions(self):
        if self.hmin.size() == 0:
            assert self.hmax.size() == 0
```

```

        return
    if self.hmax.size() == 0:
        assert self.hmin.size() == 1
        return
    # 1. min heap min element >= max heap max element
    assert self.hmax.max_element() <= self.hmin.min_element(), f'Failed:␣
↪Max element of max heap = {self.hmax.max_element()} > min element of min␣
↪heap {self.hmin.min_element()}'
    # 2. size of max heap must be equal or one less than min heap.
    s_min = self.hmin.size()
    s_max = self.hmax.size()
    assert (s_min == s_max or s_max == s_min - 1), f'Heap sizes are␣
↪unbalanced. Min heap size = {s_min} and Maxheap size = {s_max}'

    def __repr__(self):
        return 'Maxheap:' + str(self.hmax) + ' Minheap:' + str(self.hmin)

    def get_median(self):
        if self.hmin.size() == 0:
            assert self.hmax.size() == 0, 'Sizes are not balanced'
            assert False, 'Cannot ask for median from empty heaps'
        if self.hmax.size() == 0:
            assert self.hmin.size() == 1, 'Sizes are not balanced'
            return self.hmin.min_element()
        # your code here

    # function: balance_heap_sizes
    # ensure that the size of hmax == size of hmin or size of hmax + 1 == size␣
↪of hmin
    # If the condition above does not hold, move the max element from max heap␣
↪into the min heap or
    # vice versa as needed to balance the sizes.
    # This function could be called from insert/delete_median methods
    def balance_heap_sizes(self):
        # your code here

    def insert(self, elt):
        # Handle the case when either heap is empty
        if self.hmin.size() == 0:
            # min heap is empty -- directly insert into min heap
            self.hmin.insert(elt)
            return
        if self.hmax.size() == 0:
            # max heap is empty -- this better happen only if min heap has size␣
↪1.

```

```

        assert self.hmin.size() == 1
        if elt > self.hmin.min_element():
            # Element needs to go into the min heap
            current_min = self.hmin.min_element()
            self.hmin.delete_min()
            self.hmin.insert(elt)
            self.hmax.insert(current_min)
            # done!
        else:
            # Element goes into the max heap -- just insert it there.
            self.hmax.insert(elt)
        return
    # Now assume both heaps are non-empty
    # your code here

def delete_median(self):
    self.hmax.delete_max()
    self.balance_heap_sizes()

```

```

[ ]: 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')

```

1.8 Solutions to Manually Graded Portions

1.8.1 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. - $j \geq A[k - 1]$: 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 $\Theta(k + \log(n))$ for each insertion operation.

1.8.2 Problem 1B

- First, in order to delete the index j from array, move elements from $j+1 \dots k-1$ 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 = $\Theta(k + \log(n))$ in the worst case.

1.8.3 Problem 2 A

Let a be the largest element in H_{\max} and b be the least element in H_{\min} . - If $elt < a$, then we insert the new element into H_{\max} . - If $elt \geq a$, then we insert the new element into H_{\min} .

If the size of H_{\max} and H_{\min} differ by 2, then - If H_{\max} is larger then, extract the largest element from H_{\max} and insert into H_{\min} . - If H_{\min} is larger then, extract the least element from H_{\min} and insert into H_{\max} .

The overall complexity is $\Theta(\log(n))$.

1.8.4 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 element of the min-heap.

Overall complexity is $\Theta(1)$.

1.9 That's all folks