# DAA HW 2

## Problem 2

1. 
2. For a-c, use the Master Theorem to derive asymptotic bounds for T(n) or argue why the Master Theorem does not apply. You don't have to solve the recurrence if the MT does not apply. If not explicitly stated, please assume that small instances need constant time c. Justify your answers, i.e., give the values of a, b, n^log_b(a), for case 3 of the Master Theorem also show that the regularity condition is satisfied. (3 points each)

   (a) $T(n) \;=\; 4T(n/2) \,+\, n^{3}sqrt(n)$
   (b) T(n) = 2T(n/2)+n/lg(n).
   (c) T(n) = 3T(n/4)+nlg(n).
   (d) (3 points) Assume T(n) = (n+1)T(n-1)/n+c(2n-1)/n for n>1 and T(1) = 0. Show T(n) = c(n+1)∑ 2$i$−1 $i$($i$+1) $n$ $i$=2 by forward or backward iteration

Ans:

a. For $T(n) \;=\; 4T(n/2) \,+\, n^{3}sqrt(n)$,
   a = 4,
   b = 2,
   $n^{lg_b a}$ = $n^{lg_2 4}$
   = $n^{2}$ ……………… 1
   f(n) = $n^{3}sqrt(n)$
   = $n^{3+(1/2)}$
   = $n^{7/2}$ ……………… 2
   from 1 and 2,
   f(n) ∈ Ω($n^{lg_b a\,+\,\varepsilon}$) as $n^{7/2} \;>\; n^{2}$
   This is an example of case 3 of the Master theorem.
   Regularity condition:
   Regularity condition is satisfied if there is c < 1 and $n_0$ >= 0 such that af(n/b) <= cf(n) for all n >= $n_0$
   Proof -
   By substituting values of a and b in regularity condition,
   We get,
   4f(n/2) = 4$*$ $(n/2)^{7/2}$
   = $2^{2}{*}n^{7/2}/2^{7/2}$
   = $2^{2}/2^{7/2}{*}n^{7/2}$
   = $2^{(2\,-7/2)}{*}n^{7/2}$

4f(n/2) = $2^{-3/2} * n^{7/2}$ = $2^{-3/2}$f(n)

$2^{-3/2} * n^{7/2} \;<=\; cf(n)$

$2^{-3/2} * n^{7/2} \;<=\; c(n^{7/2})$

$2^{-3/2}(n^{7/2}) \;<=\; c(n^{7/2})$

where $c \;=\; 2^{-3/2}$

Therefore for all values of n>=$n_0$,

$c \;>=\; 2^{-3/2}$

Which satisfies the condition,

1 > c >= $1/2^{3/2}$

Hence, T(n) ∈ Θ(f(n))

T(n) ∈ Θ($n^{7/2}$)

a = 4, b = 2, $n^{lg_b a}$ = $n^2$, ε = 1.5

b. For $T(n) \;=\; 2T(n/2) \;+\; n/lg(n)$,

a = 2,

b = 2,

watershed function = $n^{lg_b a}$ = $n^{lg_2 2}$ = $n^1$ …………….. 1

driving function = f(n) = $n/lg(n)$ ……………. 2

$n/lg(n) \;=\; o(n)$

from 1 and 2,

Case I -

For case 1 of Master theorem to apply, the watershed function should grow not only asymptotically faster but also polynomially faster than the driving function.

Here, the driving function is $n/lg(n) \;=\; o(n)$, which means that it grows asymptotically more slowly than the watershed function *n*. But *n*/lg *n* grows only *logarithmically* slower than *n*, not *polynomially* slower

Also, $lg(n) \;=\; o(n^{\varepsilon})$ for any constant ε > 0, which means that $1/lg(n) \;=\; \omega(n^{-\varepsilon})$ and $n/lg(n) \;=\; \omega(n^{1-\varepsilon}) \;=\; \omega(n^{lg_b a-\varepsilon})$ . Thus no constant ε > 0 exists such that $n/lg(n) \;=\; O(n^{lg_b a-\varepsilon})$, which is required for case 1 to apply.

Case II -

For case 2 to apply, $n/lg(n) \;=\; \Theta(n^{lg_b a} lg^k n)$ where k >= 0 should be true. But here k = -1. Hence we cannot apply case 2.

c. For $T(n) \;=\; 3T(n/4) \;+\; nlg(n)$,

a = 3,

b = 4,

$n^{lg_b a}$ = $n^{lg_4 3}$ $where\ lg_4 3 \approx 0.793$ …………….. 1

f(n) = $nlg(n)$

= $n^1(lg(n))^1$ ……………… 2

from 1 and 2,
k = 1,
and $lg_b a = lg_4 3$,
$lg_b a < k$

f(n) ∈ $\Omega(n^{lg_b a + \varepsilon})$ as $n^1 > n^{lg_4 3}$
This is an example of case 3 of the Master theorem.
Regularity condition:

Regularity condition is satisfied if there is c < 1 and $n_0$ >= 0

such that af(n/b) <= cf(n) for all n >= $n_0$

Proof -

By substituting values of a and b in regularity condition,
We get,

3f(n/4) = 3$*$ $(n/4)\ lg(n/4)$

$3(n/4)\ lg(n/4) \ <= \ cf(n)$
$(3/4)nlg(n)/lg(4) \ <= \ cnlg(n)$
$(3/8)nlg(n) \ <= \ c(nlg(n))$

where $c = 3/8$
Therefore for all values of n>=$n_0$,

$c >= 3/8$
Which satisfies the condition,
1 > c >= $3/8$

Hence, T(n) ∈ Θ(f(n))
T(n) ∈ Θ$(nlg(n))$

a = 3, b = 4, $n^{lg_b a}$ = $n^{lg_4 3}$ $\quad where\ lg_4 3 \approx 0.793$ , ε = 1-0.793 = 0.207

d. Assume T(n) = (n+1)T(n-1)/n+c(2n-1)/n for n>1 and T(1) = 0. Show

$T(n) = c(n+1)\sum_{i=2}^{n} \frac{2i-1}{i(i+1)}$ by forward or backward iteration

Given - $T(n) = \frac{(n+1)T(n-1)}{n} + \frac{c(2n-1)}{n}$
This can be written as
$T(n) = (n+1)(\frac{T(n-1)}{n} + \frac{c(2n-1)}{n(n+1)})$ by taking (n+1) common …………………1
By using equation 1 for forward iteration,
$T(1) = 0$
$T(2) = (2+1)(\frac{T(2-1)}{2} + \frac{c(2.2-1)}{2(2+1)})$

$= (2+1)(\frac{T(1)}{2} + \frac{c(2.2-1)}{2(2+1)})$

$$= (2+1)(\frac{0}{2} + \frac{c(2.2-1)}{2(2+1)})$$

$$= (2+1)c\frac{(2.2-1)}{2(2+1)}$$

$$T(2) = c\frac{(2.2-1)}{2}$$

$$T(3) = (3+1)(\frac{T(3-1)}{3} + \frac{c(2.3-1)}{3(3+1)})$$

$$= (3+1)(\frac{T(2)}{3} + \frac{c(2.3-1)}{3(3+1)})$$

$$= (3+1)(\frac{c\frac{(2.2-1)}{2}}{3} + \frac{c(2.3-1)}{3(3+1)})$$

$$= (3+1)(c\frac{(2.2-1)}{2(2+1)} + \frac{c(2.3-1)}{3(3+1)})$$ by replacing 3 -> 2+1

$$T(3) = (3+1)c(\frac{(2.2-1)}{2(2+1)} + \frac{(2.3-1)}{3(3+1)})$$

$$T(4) = (4+1)(\frac{T(4-1)}{4} + \frac{c(2.4-1)}{4(4+1)})$$

$$= (4+1)(\frac{T(3)}{4} + \frac{c(2.4-1)}{4(4+1)})$$

$$= (4+1)(\frac{(3+1)(c\frac{(2.2-1)}{2(2+1)} + \frac{c(2.3-1)}{3(3+1)})}{4} + \frac{c(2.4-1)}{4(4+1)})$$

$$= (4+1)(c\frac{(2.2-1)}{2(2+1)} + \frac{c(2.3-1)}{3(3+1)} + \frac{c(2.4-1)}{4(4+1)})$$ dividing (3+1)/4 = 1

$$T(4) = (4+1)c(\frac{(2.2-1)}{2(2+1)} + \frac{(2.3-1)}{3(3+1)} + \frac{(2.4-1)}{4(4+1)})$$

$$T(5) = (5+1)(\frac{T(5-1)}{5} + \frac{c(2.5-1)}{5(5+1)})$$

$$= (5+1)(\frac{T(4)}{5} + \frac{c(2.5-1)}{5(5+1)})$$

$$= (5+1)(\frac{(4+1)(c\frac{(2.2-1)}{2(2+1)} + \frac{c(2.3-1)}{3(3+1)} + \frac{c(2.4-1)}{4(4+1)})}{5} + \frac{c(2.5-1)}{5(5+1)})$$

$$= (5+1)(c\frac{(2.2-1)}{2(2+1)} + c\frac{(2.3-1)}{3(3+1)} + c\frac{(2.4-1)}{4(4+1)} + c\frac{(2.5-1)}{5(5+1)})$$ dividing (4+1)/5 = 1

$$T(5) = (5+1)c(\frac{(2.2-1)}{2(2+1)} + \frac{(2.3-1)}{3(3+1)} + \frac{(2.4-1)}{4(4+1)})$$

.

.

.

$$T(k) = (k+1)c(\frac{(2.2-1)}{2(2+1)} + \frac{(2.3-1)}{3(3+1)} + \frac{(2.4-1)}{4(4+1)} + \ldots + \frac{(2.k-1)}{k(k+1)})$$ for k <= n

Hence,

$$T(n) = (n+1)c(\frac{(2.2-1)}{2(2+1)} + \frac{(2.3-1)}{3(3+1)} + \frac{(2.4-1)}{4(4+1)} + \ldots + \frac{(2.n-1)}{n(n+1)})$$

$$T(n) = (n+1)c\sum_{i=2}^{n}\frac{(2.i-1)}{i(i+1)}$$ is proved.

# Problem 3

3. Recursive function calls often use precious stack space and might lead to stack overflow errors. Tail-recursion elimination is one method to avoid this problem by replacing certain recursive calls with an iterative control structure. Learn how this technique can be applied to QUICKSORT. (10 points) Solve Problem 7-5 Stack depth for quicksort of our textbook

```
TAIL-RECURSIVE-QUICKSORT(A, p, r)
    while p < r // Partition and sort left subarray.
        q = PARTITION(A, p, r)
        TAIL-RECURSIVE-QUICKSORT(A, p, q - 1)
        p = q + 1
```

Ans:

a. Argue that TAIL-RECURSIVE-QUICKSORT (A,1,A.length) correctly sorts the array
We can prove that TAIL-RECURSIVE-QUICKSORT (A,1,A.length) correctly sorts the array using induction.
Base case - if size of array is 1 then p < r condition becomes false and the array is returned as it is. And it is a sorted array as it contains only 1 element.
Let 1 <= k <= n-1, for an array of size k,
p = starting index
q = pivot index
r = ending index
In tail-recursion quicksort, we use a normal recursive call for elements from p to q-1 hence by induction hypothesis, it will correctly sort the left subarray as it does in traditional quicksort.
For elements from index q+1 to r, instead of normal recursive call, we are using an iterative approach which is replacing recursive and directly checking for condition p < r. In this call, p = q+1 and r = r.
It is following everything as a recursive call of the right hand side subarray from the pivot q. The only difference is that we are directly entering into the code execution of quicksort(A, q+1, r) rather than doing a recursive call first and then entering in code execution.
Hence the right side subarray will follow the exact same code execution as traditional quicksort. The only difference is that it will skip the recursive call and directly enter into code execution.
The above is true for an array of size k, 1 <= k <= n-1.
So by induction hypothesis, it correctly sorts A[q+1…n] as desired.

b. Describe a scenario in which TRE-QUICKSORT's stack depth is $\Theta(n)$ on an *n*-element input array.
The stack depth will be $\Theta(n)$ when all the elements in an array are sorted.

Let's assume we are considering the pivot element as the last element in the array.
So on the left side the subarray would have n-1 elements and on the right side the subarray would have 0 elements.
According to the TAIL-RECURSIVE-QUICKSORT there will be n-1 recursive calls to the left part before the while-condition p < r is violated. Hence stack depth will become Θ(*n*) in an n-element input array.
Same will be the case if we want to sort in descending order and the array has already been sorted. Considering the pivot element the last one, again there will be n − 1 elements in the left part which will result in n-1 recursive calls to TAIL-RECURSIVE-QUICKSORT. Thus the stack will have a depth of Θ(*n*) in case where the input array is in sorted format.

c. Modify TRE-QUICKSORT so that the worst-case stack depth is Θ(lg *n*). Maintain the *O*(*n* lg *n*) expected running time of the algorithm.

```
MODIFIED-TAIL-RECURSIVE_QUICKSORT(A, p, r)
      while p < r
            q = Partition(A, p, r)
            if q < floor((r-p)/2)
                  MODIFIED-TAIL-RECURSIVE_QUICKSORT(A, p, q-1)
                  p = q+1
            else
                  MODIFIED-TAIL-RECURSIVE_QUICKSORT(A, q+1, r)
                  r = q-1
            end if
      end while
```

In TAIL-RECURSIVE-QUICKSORT, when the array is already sorted, the stack depth was Θ(*n*). To minimize the stack depth from Θ(*n*) to Θ(*lg(n)*) we can check the position of the pivot element if pivot element's position is the rightmost position then left side subarray will be of size n-1 and rightside subarray will be of size 0. In this case there will be 2 recursive calls - 1st for current call and 2nd for smallest subarray call.
In MODIFIED-TAIL-RECURSIVE-QUICKSORT we choose the subarray of smaller size between left and right subarray to perform a recursive call. Hence in case of unequal subarray sizes, the subarray with small size will get a recursive call.
Worst case of MODIFIED-TAIL-RECURSIVE-QUICKSORT in terms of stack depth is when pivot element is the middle element in every recursive call, the max number of recursive calls are Θ(*lg(n)*) as everytime the elements included in the recursive call will be almost half of the array size. And height of the recursion tree is Θ(*lg(n)*).

Hence above function is the modified function which gives the worst-case stack depth Θ(lg *n*) and maintain the *O*(*n* lg *n*) expected running time of the algorithm.

## Problem 4

4. Consider a situation where your data is almost sorted—for example, you are receiving time-stamped stock quotes, and earlier quotes may arrive after later quotes because of differences in server loads and network traffic routes. Focus only on the time stamps. Assume that each time stamp is an integer, all time stamps are different, and for any two time stamps, the earlier time stamp corresponds to a smaller integer than the later time stamp. The time stamps arrive in a stream that is too large to be kept in memory. The time stamps in the stream are not in their correct order, but you know that every time stamp (integer) in the stream is at most a hundred positions away from its correctly sorted position. Design an algorithm that outputs the time stamps in the correct order and uses only a constant amount of storage, i.e., the memory used should be independent of the number of time stamps processed. Solve the problem using a heap.

Ans:

Let's assume that the constant amount of storage used is 101 heap structure space. This is calculated as, since the timestamps are at most 100 positions away, we will assume a size of heap as 100+1 = 101

1) Textual description of the algorithm -
   Data structure used - Min heap
   As we need to sort the timestamps in increasing order so the min heap will be the most suitable because the smallest element is at the root.
   Min heap size - 101
   As the timestamp can be at max 100 positions away from its correct position, we need an array which can store 100+1 elements at a time.
   Methods used -
   a) **Add_next_time_stamp** - This will add a next time stamp in the min heap structure at the root node and it will heapify down the root node to maintain the min heap structure.
   b) **Heapify_down** - It will heapify the subtree from the given node.
   c) **Remove_time_stamp** - This will remove the smallest time stamp(root) from the min heap structure.
   d) **Build_min_heap** - This will be used only for first 101 time stamps to build a min heap structure as a single time stamp can be at max 100 positions away from its correct position in sorted list. Hence we need to add the first 101 elements in the min heap structure because the 1 st correct position time stamp can be at 101st position so in order to sort it correctly we need to build a min heap first.
   e) **Find_smallest_time_stamp** - This will return the smallest timestamp present in the min heap structure.

f) **Empty_min_heap** - This is used to empty the min heap in case of input stream stops. When the input stream stops we need to give the remaining 101 timestamps out.

Description -

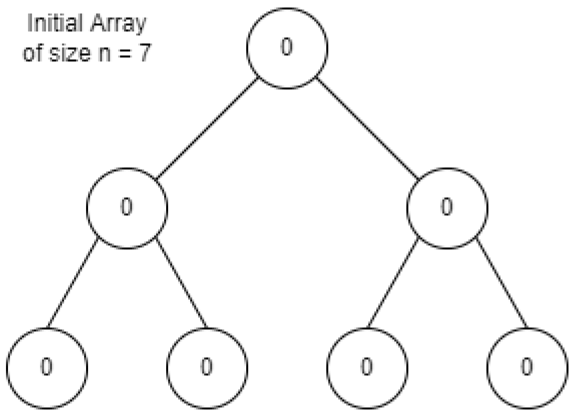
a) Add first 101 timestamps in the min heap structure
b) Before adding the next timestamp in the min heap structure, find the smallest timestamp and return it.
c) Remove the smallest timestamp from the min heap structure.
d) Add the next time stamp in the min heap structure.
e) Repeat step c and d till the timestamps stop coming.
f) Empty the min heap structure by removing all the timestamps.

2) Pseudocode for the algorithm -

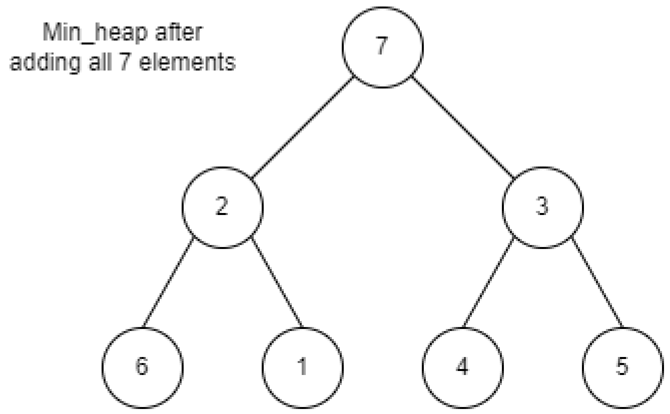
```
Add_next_time_stamp(time_stamp)
      Min_heap[0] = time_stamp
      Heapify_down(0, size_of_heap)

Heapify_down(i, n)
      left = i*2+1
      right = i*2+2
      parent = i
      if left < n and Min_heap[left] < Min_heap[i]
            i = left
      if right < n and Min_heap[right] < Min_heap[i]
            i = right
      if i != parent
            swap(Min_heap[i], Min_heap[parent])
            Heapify_down(i, n)

Remove_time_stamp(size_of_min_heap)
      if(size_of_min_heap > 0)
            swap(Min_heap[0], Min_heap[size_of_min_heap-1])

Build_min_heap(n)
      for i = n/2 -1 downto 0
            Heapify_down(i, n)

Find_smallest_time_stamp()
      return Min_heap[0]

Empty_min_heap()
      for i = n downto 1
            time_stamp = Find_smallest_time_stamp()
```

```
                  print time_stamp
                  Remove_time_stamp(i)
                  Heapify_down(0, i-1)

      Main_function()
            current_min_heap_size = 0
            min_heap_size = 101
            continue_data_stream = True
            Min_heap[101]                  // empty array of 101 size
            while continue_data_stream
                  input(time_stamp)
                  if current_min_heap_size < 101
                        Min_heap.add_time_stamp(time_stamp)
                        current_min_heap_size = current_min_heap_size+1
                  else if current_min_heap_size == 101
                        Build_min_heap()
                        min_time_stamp = Find_min_time_stamp()
                        print min_time_stamp
                        Add_next_time_stamp(time_stamp)
                        Heapify_down(0, min_heap_size-1)
                        current_min_heap_size = current_min_heap_size +1
                  else
                        min_time_stamp = Find_min_time_stamp()
                        print min_time_stamp
                        Add_next_time_stamp(time_stamp)
                        Heapify_down(0, min_heap_size-1)
                  //check if there is any next timestamp
                  input(continue_data_stream)
            end while
            if continue_data_stream == False
                  Empty_min_heap()
            end if
```

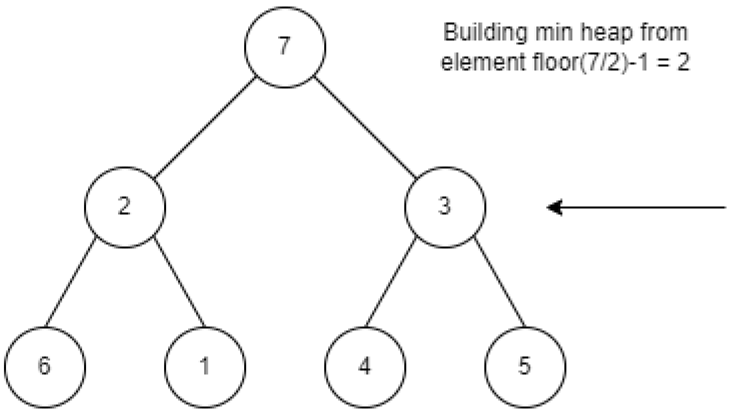
3) Working example -
   As the heap of 101 nodes is not feasible to show diagrammatically so using the small instance of above requirement.
   Let k = 6, means a timestamp can be 6 places away from its correct position
   So size of the heap = n = k+1 = 6+1 = 7
   Step 1 - Initializing array of size n
   Here n = 7



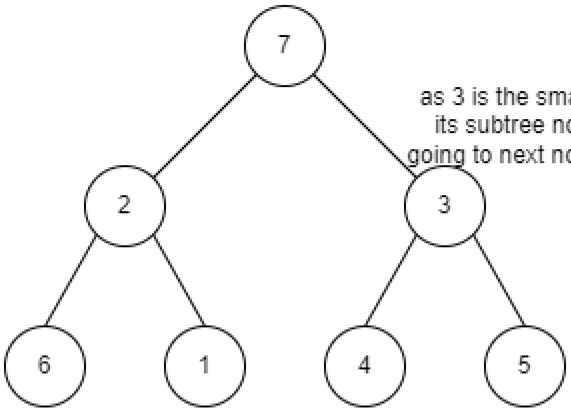
Step 2 - Adding 7 elements - [7,2,3,6,1,4,5]



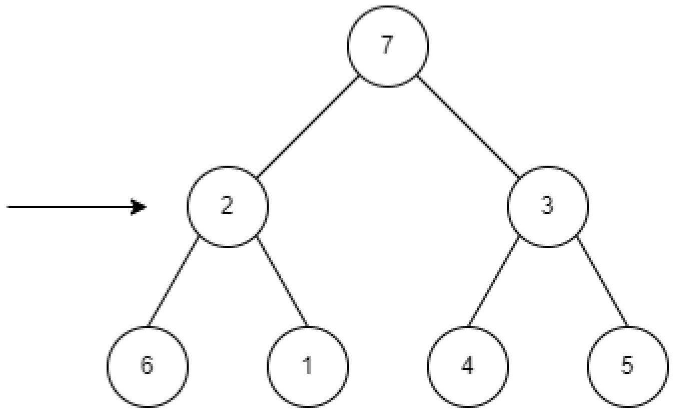
Step 3 - Build min heap

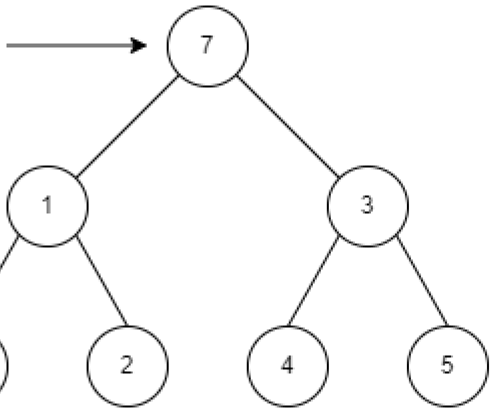


Building min heap from element floor(7/2)-1 = 2



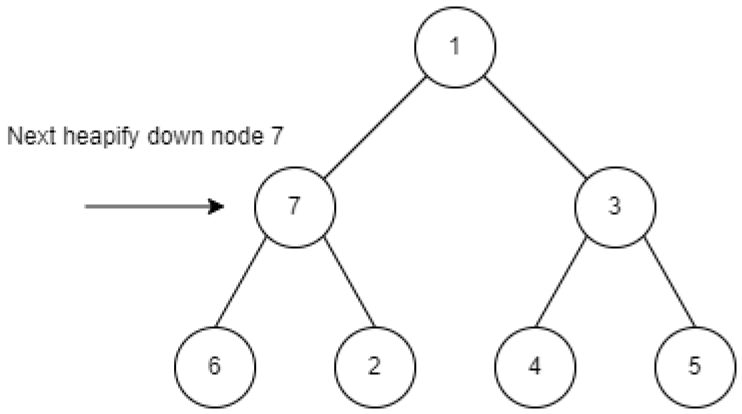
as 3 is the smallest among its subtree nodes hence going to next node for heapify

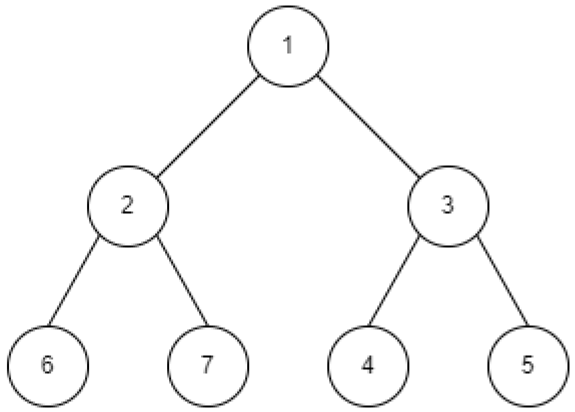




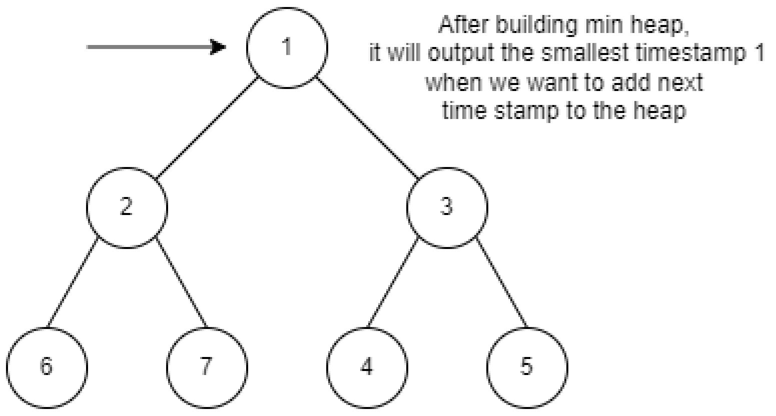
After heapify of element 2, going to the next node at root

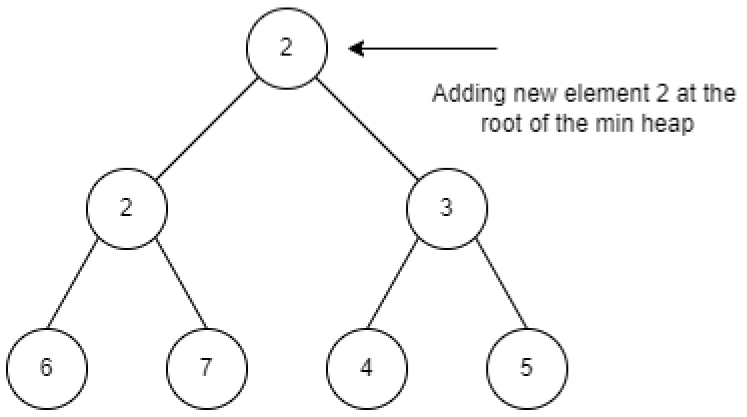
As 2>1 so swap

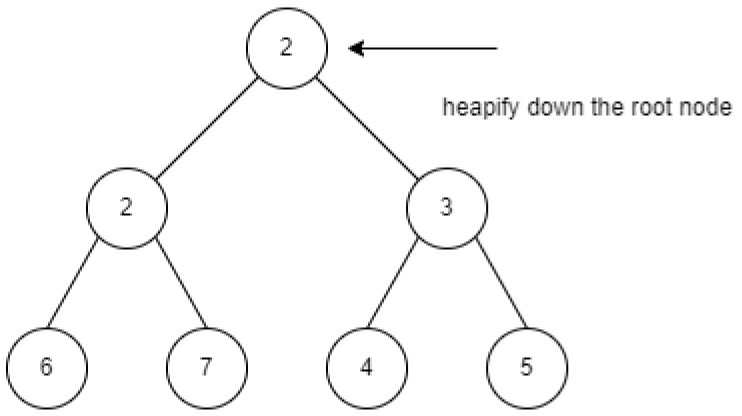




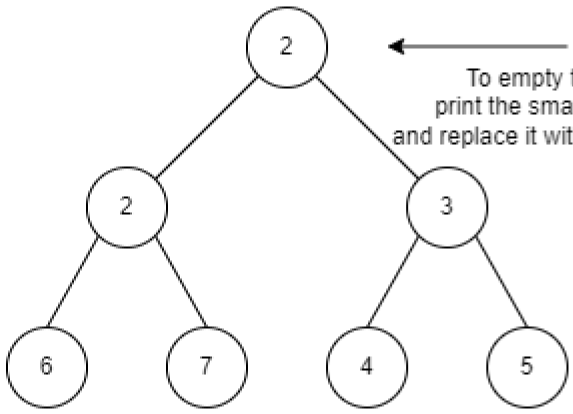
Step 4 - Output the smallest time stamp
Add new time stamp
Heapify the root node
and follow the same steps as in step 4



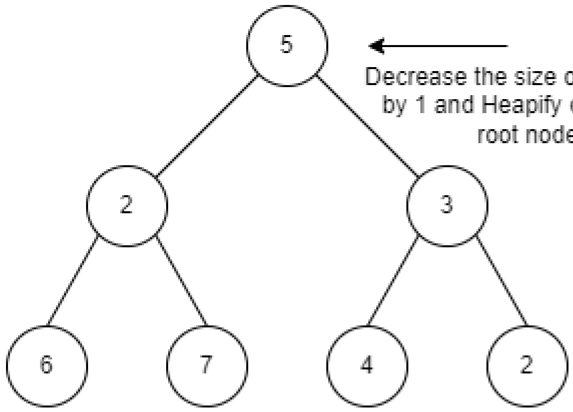




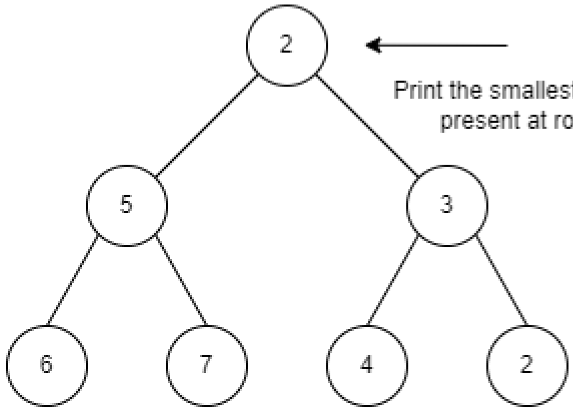
Step 5 - suppose the input stream has stopped so
To empty the mean heap start removing elements from the root node while maintaining min heap structure



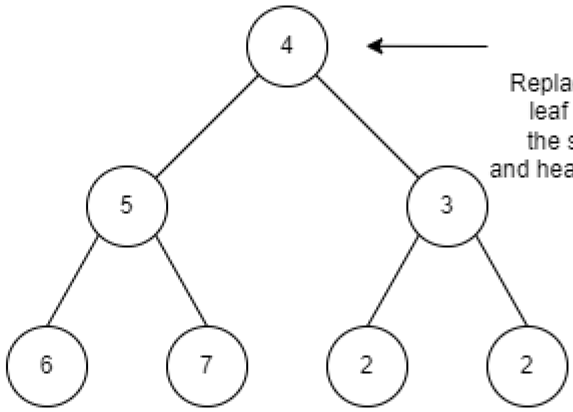
To empty the min heap
print the smallest timestamp
and replace it with the last leaf node



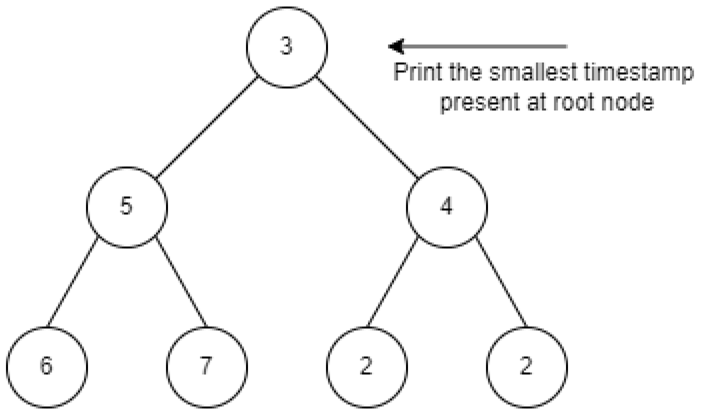
Decrease the size of the heap
by 1 and Heapify down the
root node

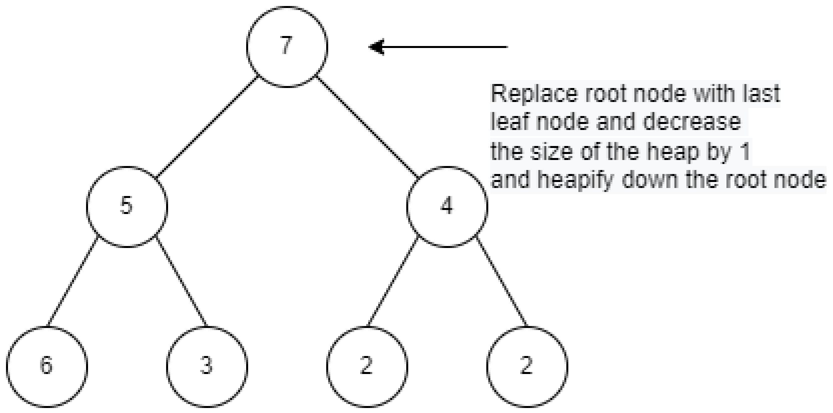


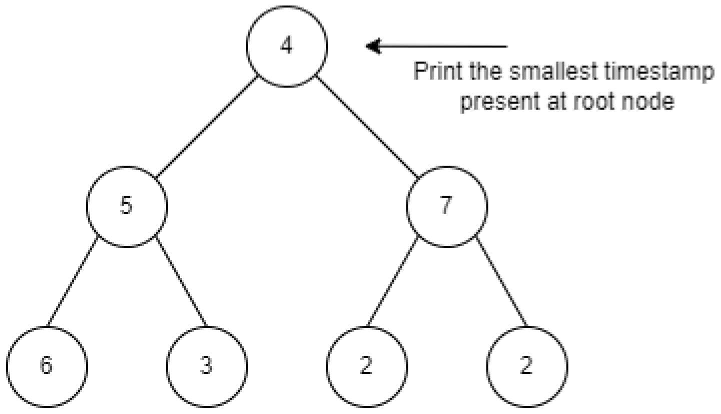
Print the smallest timestamp
present at root node

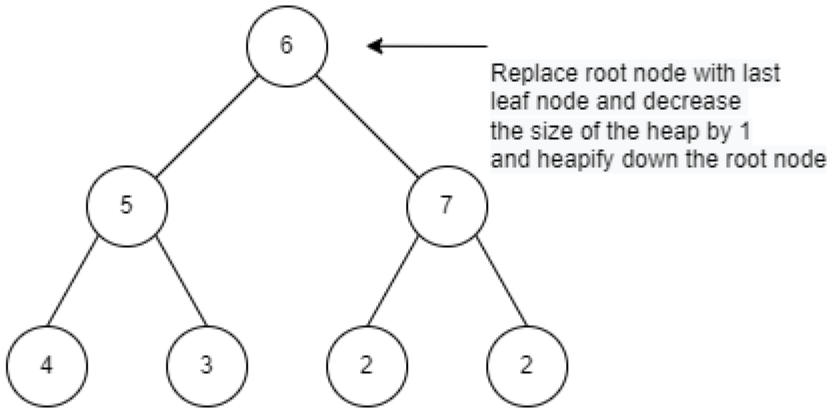


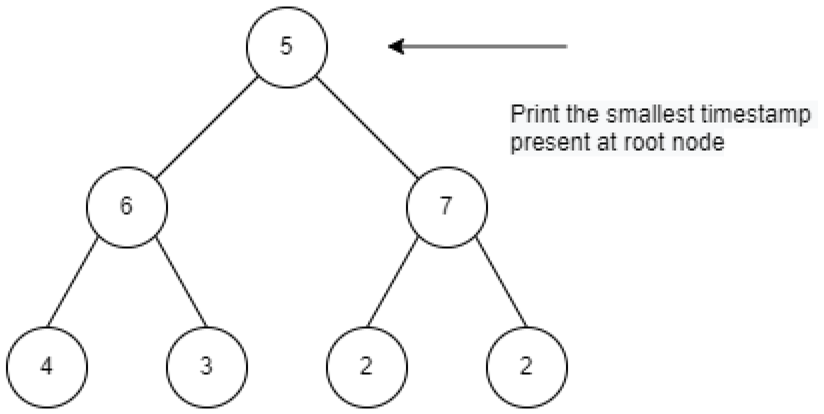
Replace root node with last
leaf node and decrease
the size of the heap by 1
and heapify down the root node

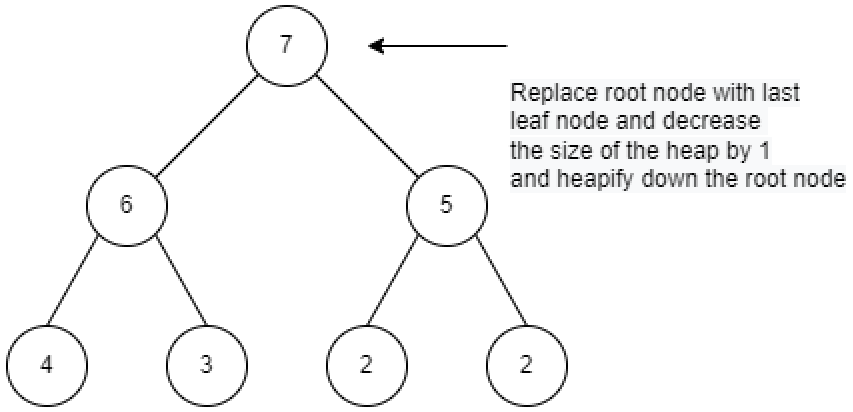

3
5
4
6
7
2
2
Print the smallest timestamp present at root node

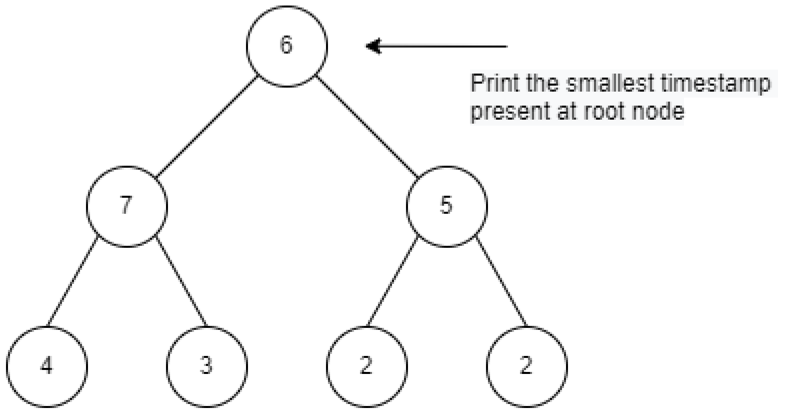

7
5
4
6
3
2
2
Replace root node with last leaf node and decrease the size of the heap by 1 and heapify down the root node

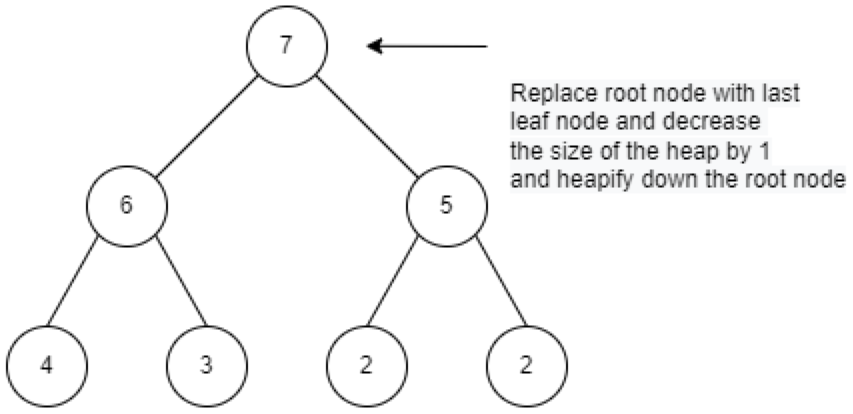

4
5
7
6
3
2
2
Print the smallest timestamp present at root node


6
5
7
4
3
2
2
Replace root node with last leaf node and decrease the size of the heap by 1 and heapify down the root node

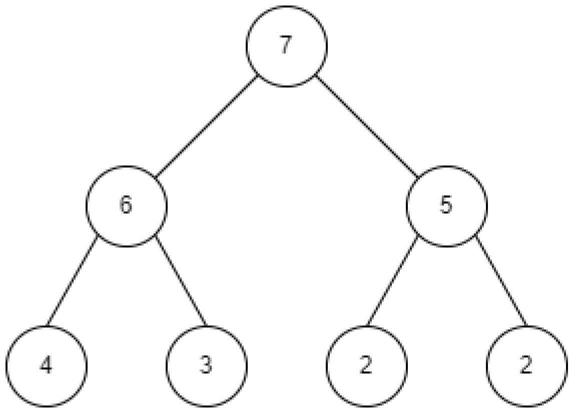

5
6
7
4
3
2
2
Print the smallest timestamp present at root node


7
6
5
4
3
2
2
Replace root node with last leaf node and decrease the size of the heap by 1 and heapify down the root node


6
7
5
4
3
2
2
Print the smallest timestamp present at root node


7
6
5
4
3
2
2
Replace root node with last leaf node and decrease the size of the heap by 1 and heapify down the root node

Final Empty Min heap



4) Proof of correctness of the algorithm -
**Precondition** - Initial data stream is of size 0
Min heap structure is of size 101 and contains all elements as 0

**Postcondition** - Min heap contains the smallest element at root
Output the data stream in sorted order.

**Initialization** -
Initially, the min heap of size 101 contains all elements as 0 and the root node is the smallest element as its value is 0.
No output stream is produced so empty data is sorted in order.

**Maintenance** -
At any iteration, the min heap structure size is 101 and min heap structure is maintained.It will give the smallest element at root.
As the smallest element from the root node is given as an output so data going out of the min heap is in sorted order.

**Termination** -
After termination, means after emptying the min heap, no output data is remaining hence returning every output in sorted order.
As the size of the min heap is reduced to 0 so no element is at the root node.

5) Time and space complexity analysis -
For heapify it takes lg(k) time.
Time complexity -
Suppose there are n timestamps,
Min heap size = k = 101
Time complexity to add k elements in the min heap - Θ(k)
Time complexity to build a min heap is - Θ(k*lg(k))
Time complexity to output minimum element - Θ(1)

Time complexity to add new element - Θ(lg(k))
So Time complexity to add almost n new element - Θ(n*lg(k))
Time complexity to empty the min heap - Θ(k(lg(k))

Let the total number of timestamps = n
Time complexity of giving output minimum element and removing it from the min heap - Θ(n * lg(k)
Total time complexity of the above algorithm
= Θ(k) + Θ(2k*lg(k)) + Θ(1) + Θ(n * lg(k))
= Θ(n * lg(k))
where lg(k)= lg(101) = constant
= Θ(n)

Space complexity -
As it requires only min heap structure of size 101.
Let k = 101
The total space complexity of above algorithm is Θ(k) = Θ(101) = constant