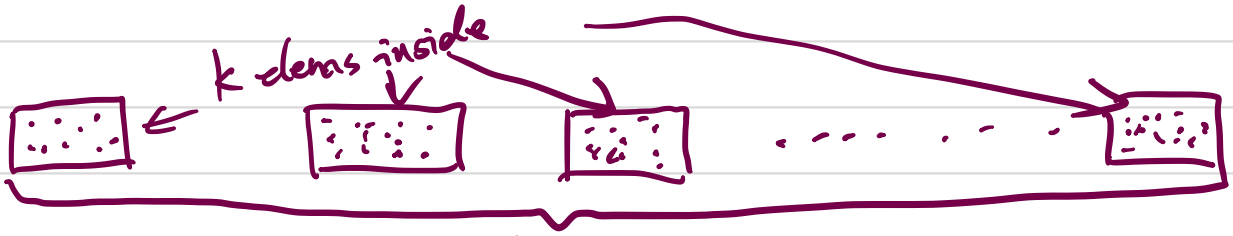
(1)

(a)



all add up to n

so there is $\frac{n}{k}$ group

In one group, inside is <u>not sorted</u>, so the possible permutation we have, is $k!$

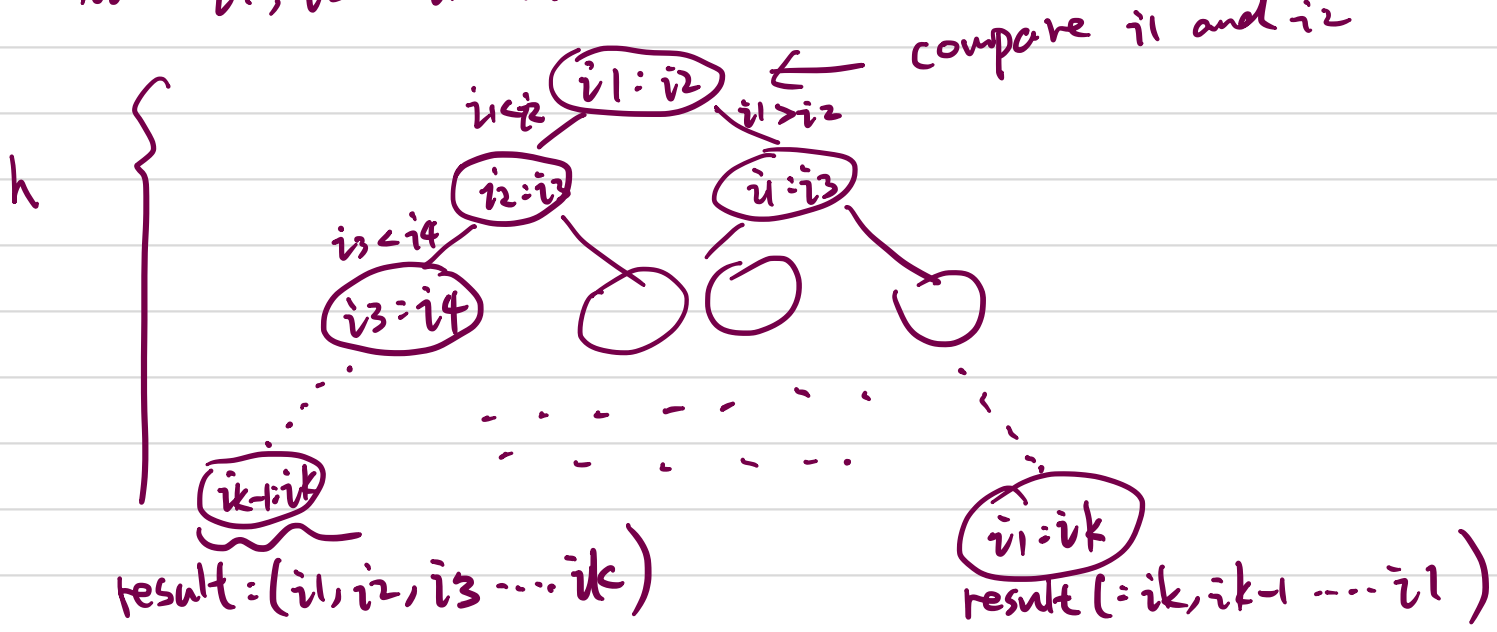
When one group is done, the permutation between two group is simply $k! \cdot k!$, because we are gaurenteed that every elems in one group is larger or less than another, so simply multiply their permutation is sufficient, no need to do permutation among all elems in two groups again. Because there are $\frac{n}{k}$ group, so all the possible permutations are $\underbrace{k! \cdot k! \cdot k! \cdots}_{\frac{n}{k} \text{ times}}$

the result then is $(k!)^{\frac{n}{k}}$

(b) Worst case: Algorithm's input maximize algorithm's run time

if sorting by comparison in every group of k, the process will like:

*Note: i1, i2 ··· ik means item #



if every leaves represent a sorting permutations the worst case is the longest root-leaf path (h),

if the algorithm sufficient, the height is

$\log_2$ # leaves

from part A, we know that distinct permutati we can have for each group is $k!$. However if an algorithm compares same element more than once (repeat permutation, e.g. bubble sort), the leave nodes of one group's sorting process will always end up $\geq k!$

so, $k! \leq$ # leaves in one group

$(k!)^{\frac{n}{k}} <$ # leaves in all items

Combine to the worst case analysis before, So the lower bound of worst case is

$$\log_2 (k!)^{\frac{n}{k}} = \frac{n}{k} \cdot \log_2 k!$$

(C) For the whole input, we get the worst case lower bound from part b, which is $\frac{n}{k} \cdot \log_2 k!$

Big-O: for $k \geq 1$, $k! \leq k^n$

so, $\frac{n}{k} \cdot \log_2 k! \leq \frac{n}{k} \cdot \log_2 k^k$

$$\Rightarrow \frac{n}{k} \cdot \log_2 k! \leq k \cdot \frac{n}{k} \cdot \log_2 k$$

$$\Rightarrow \frac{n}{k} \cdot \log_2 k! \leq n \log_2 k$$

$$\frac{n}{k} \cdot \log_2 k! = O(n \log_2 k)$$

Big Ω:

$$\frac{n}{k} \log_2 k! = \frac{n}{k} \log_2 (k \cdot k-1 \cdot k-2 \cdots 3 \cdot 2 \cdot 1)$$

$$= \frac{n}{k} \log_2 (k \cdot 1 \cdot (k-1) \cdot 2 \cdot (k-2) \cdot 3 \cdots (k-\tfrac{k}{2}) \cdots (k-\tfrac{k}{2}))$$

underestimate all to k ⟶

$$k \quad > k \quad > k \quad \cdots\cdots \quad k$$

$$\geq \frac{n}{k} \log_2 (k^{k/2}) \Rightarrow \text{(assume } k \text{ even)}$$

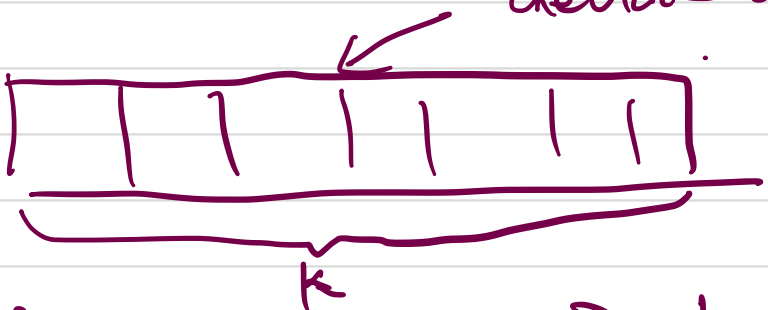
$$\Rightarrow \frac{n}{k} \log_2 k! \geq \frac{n}{2} \log_2 k = \frac{1}{2} \cdot n \log_2 k$$

$$\Rightarrow \frac{n}{k} \log_2 k! = \Omega(n \log_2 k)$$

so the θ notation is : $\log_2 k!)^{\frac{n}{k}} = \theta(n \log k)$

(d) In part C, we get that the worse case of a non-repeating-comparison algorithm is $\theta(n \log k)$, I will choose merge sort, which divide problem into sub problem and merge without redundent comparison to match that result.

① Divide:

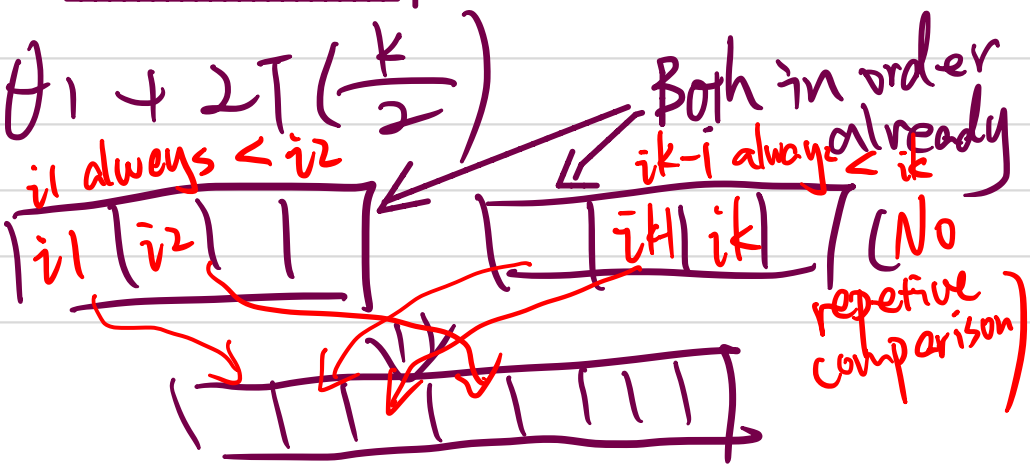


By index, $\theta 1$ time

② conquer,

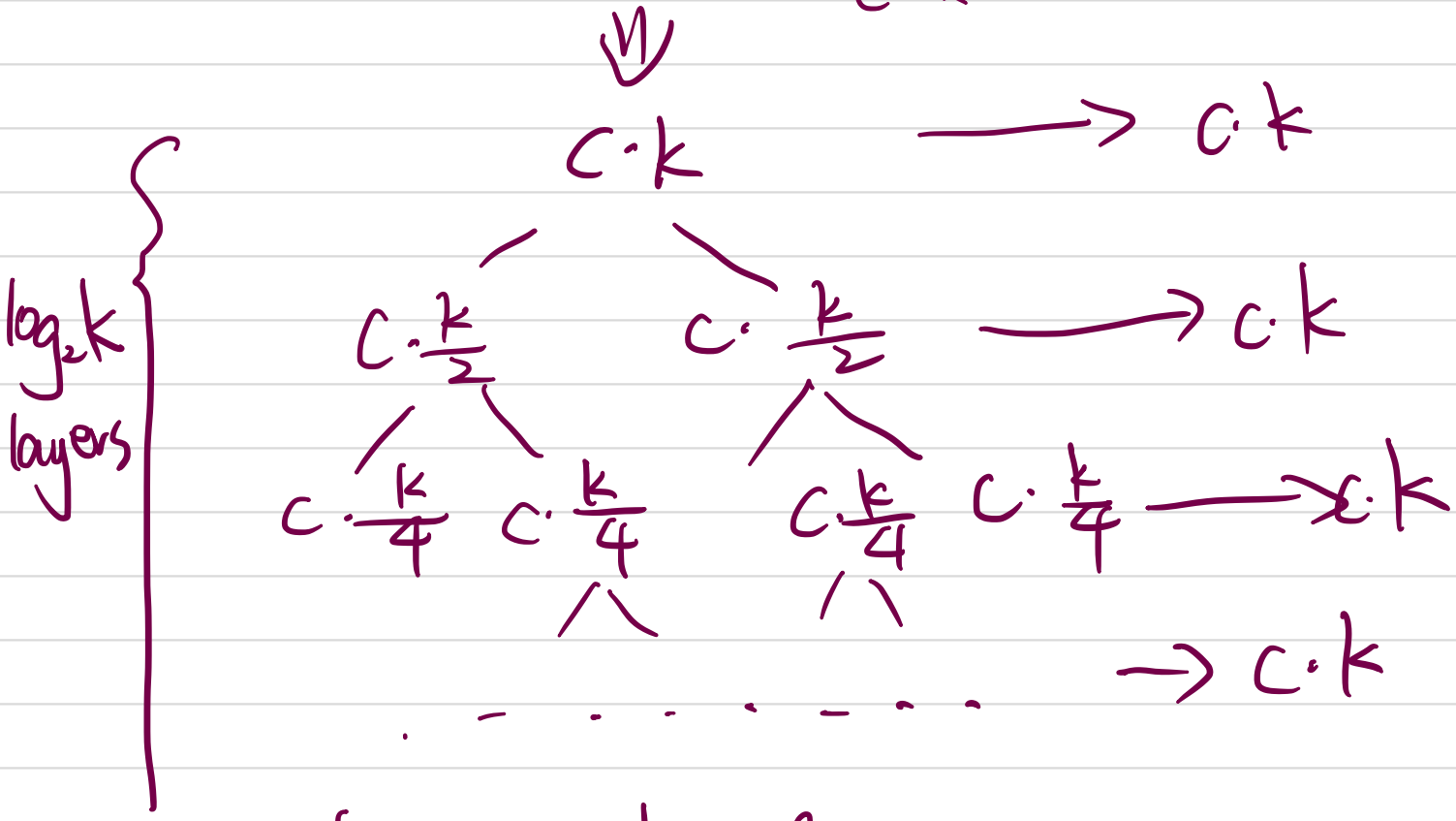
③ Merge

$\theta 1 + 2T(\frac{k}{2})$



Merge Sort complexity of one group

$$T(k) = 2T\left(\frac{k}{2}\right) + \underbrace{\theta(k)}_{c \cdot k}$$



leaves = k, $\theta(k)$

Total work = $\underbrace{c_1 k \log k}_{\text{layers}} + \underbrace{c_2 k}_{\text{leaves}}$

$\theta(n \log k)$

$= \theta(k \log k)$

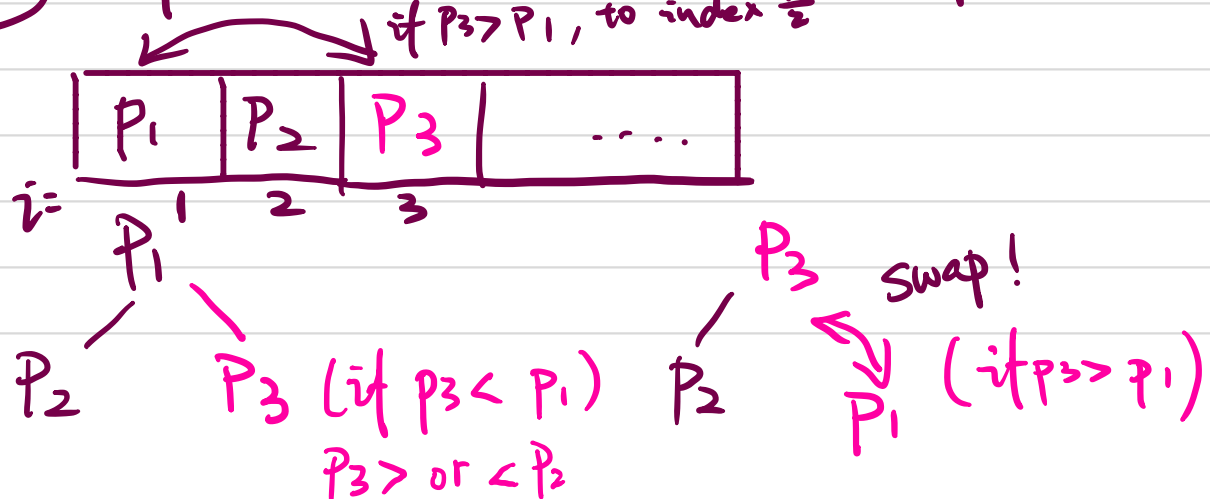
for $\frac{n}{k}$ groups, total work will be $\frac{n}{k} \cdot \theta(k \log k) =$

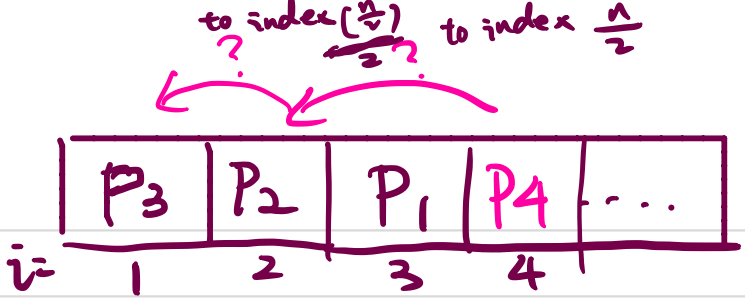
2.

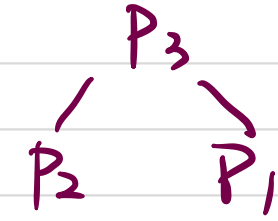
(a) I will build a max heap to handle data

So, when my first patient comes in and got the assigned injury num value, I will put him/her on the top of the heap (only one patient) (P1), when second comes in and the first one hasn't be called, there are two situations

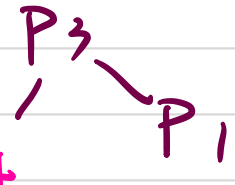
P2 is more severe or less. if more, switch P1, P2, P2 now on the top of heap (P2) (P2>P1) (P1), or less severe, just put P2 next to (P1) (P1>P2) (P2). Same logic, the third, forth .... n patient comes in, it follows



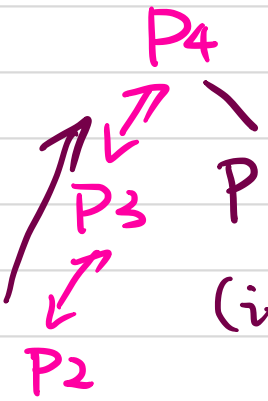




P4 (if it smaller than P2)



(if P4 > P2 but < P3)



(if P4 > P2 > P3)

As a result, I will handle the data by creating a list with max-heap concept, every new patient got to compare their severe level with $\frac{\text{new person index}}{2}$ (index start with zero), if less, stay in their spot. If larger, then swith spot with person in the index $\frac{\text{new person index}}{2}$, and the new person use its own index, keep comparing to $\frac{\text{new person index}}{2}$ (new) until the severe number is less in the comparison or it reaches the first spot (the new one most severe!)

However, because we need a track system (search) on every patient. So if we get people's name, we put the name in to a balanced binary search tree according to alphabetical order. Inside the node, stores a pointer to the corresponding severity in the max-heap.

Both insert new input can have the best case $\theta(1)$ (both max-heap and bST), worse case $\theta \log_2 n$ (both max-heap and bst)

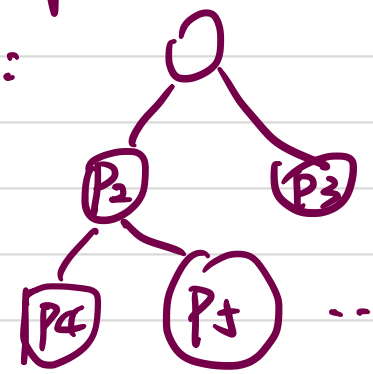
(b) Because of the property of max-heap built in part A, the most severe patient will always be able to switch to the first. The first one of the waiting list is always the largest. So when a doctor calls, simply pop the first patient in the waiting list, and the time is constant, $\Theta(1)$.

However, before let the first person going to the doctor, need to get name, delete it from the search system, which takes at most $(\log_2 n)$, then let the person go to the doctor.

So the total time come to $\log_2 n + 1$

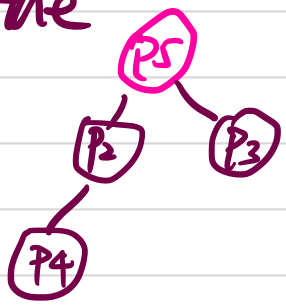
$= O(\log_2 n)$

(c) When a person is sent to the doctor, the first item of the list is emptied looks like :

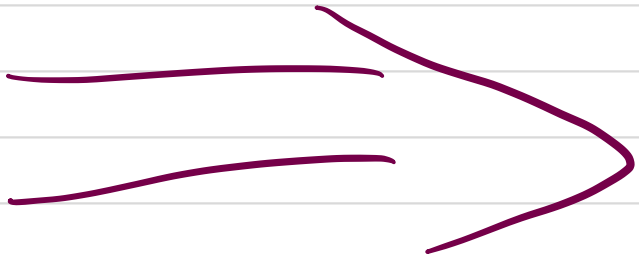


... etc ,

then I will reorganized by put the least severe patient to the top, like

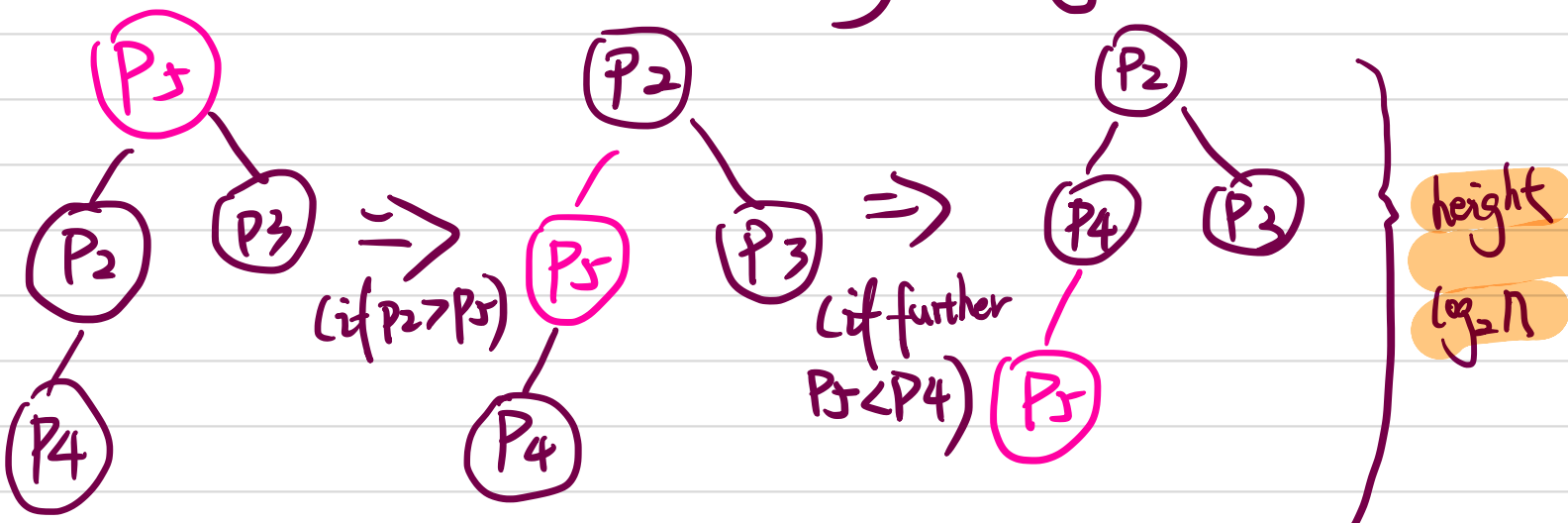


but not valid!



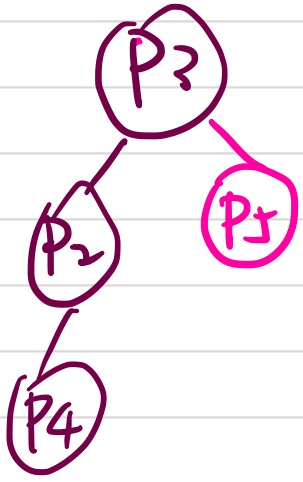
continue next page

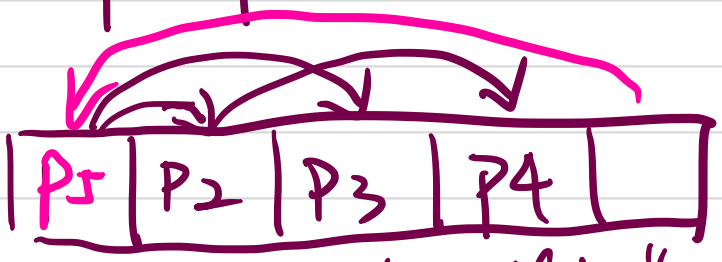
Then, to make the waiting list system valid,



or

when (P2 < P5) but P5 < P3





Starts from the "fake" max, keeps it compares to its children (2n, 2n+1), switch if it is smaller than any one of the children, switch, and keep switching to new 2n / 2n+1 until it is larger than both children or a spot has no children yet.

2n: left child

2n+1: right child

So the total time to reorganize is :

$\Theta(1)$ + $\log_2 n$

↑ move the last to first empty spot

↑ the worst case time it needs for the last item from top to tumbling down (longest root-leave path)

In total $O(\log_2 n)$

(d) New arrival of patient :

First, add the new patient's name to the search system (balanced bst) according to alphabetical order. Insert takes $\Theta(\log_2 n)$.

Then, add the new patient (V: 1/3 severe number) to the end of the max-heap. Continuely compares to its parent, if larger, switch with parent, and (index = newIndex / 2) continue switching with parent until it is smaller than parent or reach the top of the heap. In worse case, it takes $\log_2 n$ to reach the correct spot. So the total time for reorganize for a new patient is:

$$\log_2 n + \log_2 n$$

$\log_2 n$ ↓ insert in balanced bst

$\log_2 n$ ↓ heapify the new patient

$$= 2\log_2 n = O(\log_2 n)$$

(e) Someone gets worse:

if someone gets worse, their severe level gets larger. To find the severe number that needs to modify in the max-heap, first need to search the name of patient in the balanced-binary-tree system, which at most takes $\log_2 n$, then find the severe number it links to in the max-heap.

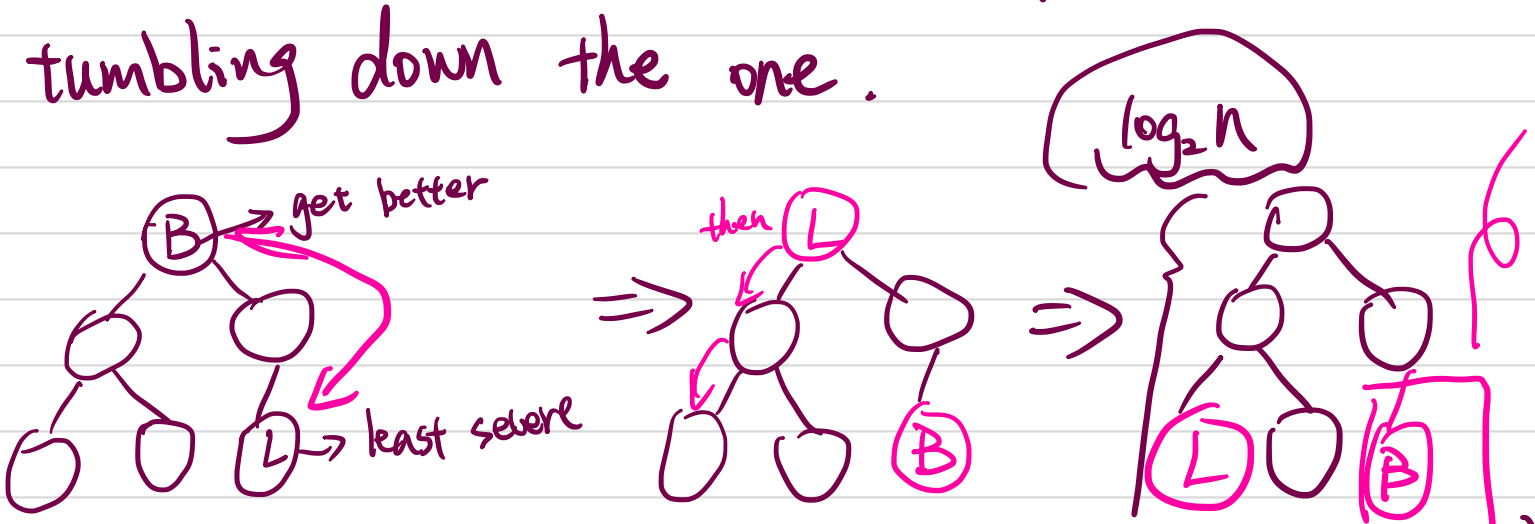
The worst case is the patient that is least severe before gets most severe, so it needs to be switched all the way up, which takes $\log_2 n$.

Combine search and heapify part, the worst time is $\log_2 n + \log_2 n$, so $O(\log_2 n)$

(f) Some one gets better

Same as the case in (e), first we need to find the corresponding severity by searching patient's name. Again, worst case $\log_2 n$ for searching.

After find the patient's severity, the worst case is the most severe patient become all better, so first swich that person with the least severe person. Then tumbling down the one.



Then, pop the last one (the get-better patient) out of the wait list

Time complexity:

$\log_2 n + \theta(1) + \log_2 n + \theta(1)$

- $\log_2 n$: search time
- $\theta(1)$: switch L and B
- $\log_2 n$: heapify time
- $\theta(1)$: pop time

in total: $2\log_2 n + 2\theta(1) = O(\log_2 n)$

(g) Because real number becomes number with one decimal at most.

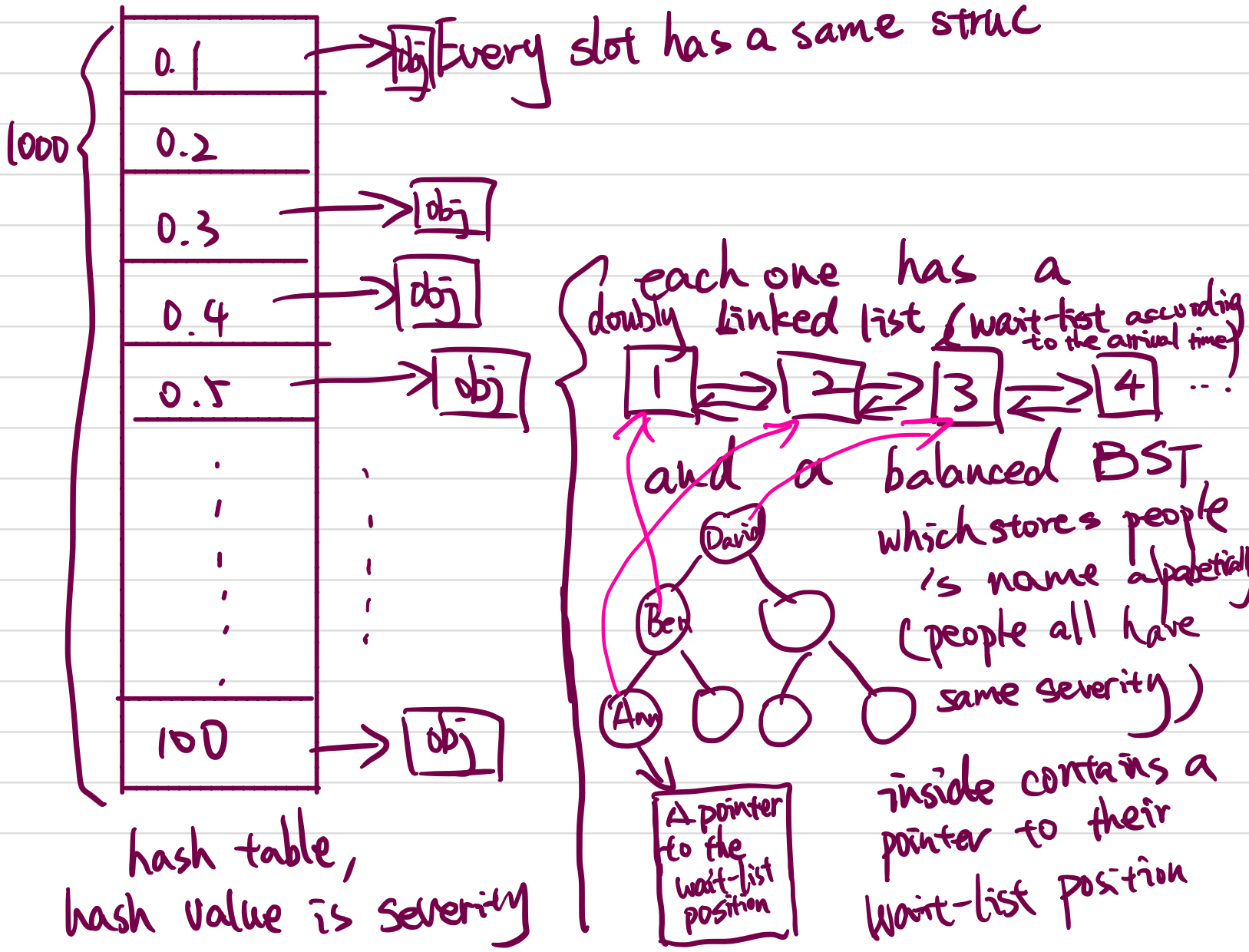
So from 0–100 has 1000 possible severity levels.

So the solution is:
use a hash table, each slot represent a severe level. But when a new patient comes, they get a number represent their order in the line too. Even two people have same severity level, they are recorded

balanced bst

to a (searching system), which uses their name or id as key, each node contains a pointer to their waiting list position in the same severity.

(I know it is comfusing, it is showed in picture below)



So when the next patient is sent to doctor, the worst case time complexity is

$O(1000) + O(1) + O(\log_2 n)$

- $O(1000)$: search the most severity value of hashtable
- $O(1)$: delete the first node of linked list
- $O(\log_2 n)$: search and delete the one going to the doctor (assuming the worst case all the patient has same injury level)

$= O(\log_2 n)$

after the patient went to see the doctor, there is no need to reorganize, so $O(1)$

A new patient arrives, it takes

$O(1) + O(1) + O(\log_2 n) = O(\log_2 n)$

- $O(1)$: find the severity of corresponding severe value
- $O(1)$: to link the new patient to the end of linked-list
- $O(\log_2 n)$: to insert the new patient, assume worst case that all patients has same severe level

Someone gets worse, it takes

$O(1) + O(\log_2 n) + O(1) + O(1) + O(1)$

- $O(1)$: the time to find the original severe level
- $O(\log_2 n)$: search through all people in this severe level, get its link in the waitlist, delete
- $O(1)$: get to the exact spot in linked list and delete the node
- $O(1)$: get to the new severe level in hash table
- $O(1)$: append the patient to the last of the waitlist linked list of that new severe level

$= O(\log_2 n)$

Someone gets better and leaves

$O(1) + O(\log_2 n) + O(1) = O(\log_2 n)$

- $O(1)$: the time to find the original severe level
- $O(\log_2 n)$: search through all people in this severe level, get its link in waitlist, delete
- $O(1)$: get to the exact spot in linked list and delete the node

Note: All $O(\log_2 n)$ is in the worst case that all people in same severe level, the average case would be much better in this approach.