Design Problem: A hospital emergency room needs to manage information about the patients who need to be seen. Patients are treated in order of urgency (most urgent first). As each person comes in, a record is created with their *name*, *urgency level* (a number, in which higher numbers have higher priority), and a *brief description of their injury*.

Think about the kinds of operations that we have studied on sequences of data this semester:

- add
- remove
- get by position
- contains

WANT TO DO THESE QUICKLY

GET QUICKLY, BY URGENCY

MAYBE LESS IMPORTANT

Which of these seem most important when managing the order in which to treat patients? Which operations need to be the most efficient?

Priority queue (PQ): 3 fundamental operations

1. **Insert a new item**
2. **Remove the maximum priority item**
3. **Get maximum priority item, without removing (peek)**

=> These are the operations we want to prioritize

How do we build such a data structure from scratch? Consider the data structures that we have studied so far: which seem to be suitable here?

- Linked List
- Array List
- Tree/Binary Search Tree
- Hashmap/Dictionary
- HashSet/set
- Class of our own design (e.g., graph)

NEXT PAGE ...

Write the type of your proposed data structure for managing hospital triage data (in terms of types)

Priority queue (PQ): needs 3 operations

1. **Insert a new item**
2. **Remove max-priority item**
3. **Get max-priority item (without removing) (peek)**

How do we build one from scratch?

Some data structure: arrays, lists, trees, hash maps/dictionary, hash sets/set, graphs

How might you use these? Which ones might be best?

HashSets/set

=> Not really a way to specify priority

HashMap/dict

1) Key = priority, Value = item
2) Key = item, Value = priority

→ We *could* implement a heap using one of these options, but we would need to search the whole map O(N)

Linked list
Array list

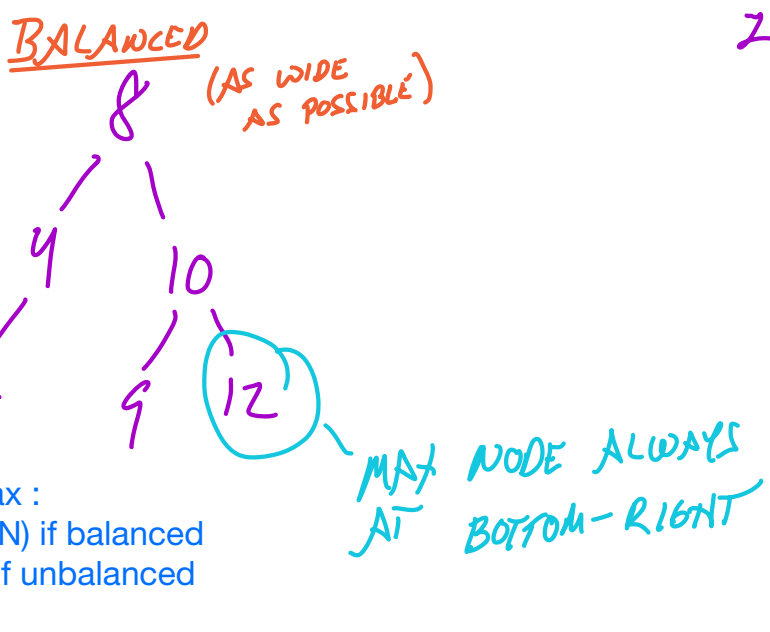
keep a sorted list
- get-max: O(1) (pick first element)
- remove-max: O(1) for linked list, for array list would need to shift elements to keep sorted order => O(N)
- insert: O(N) to find position in sorted list

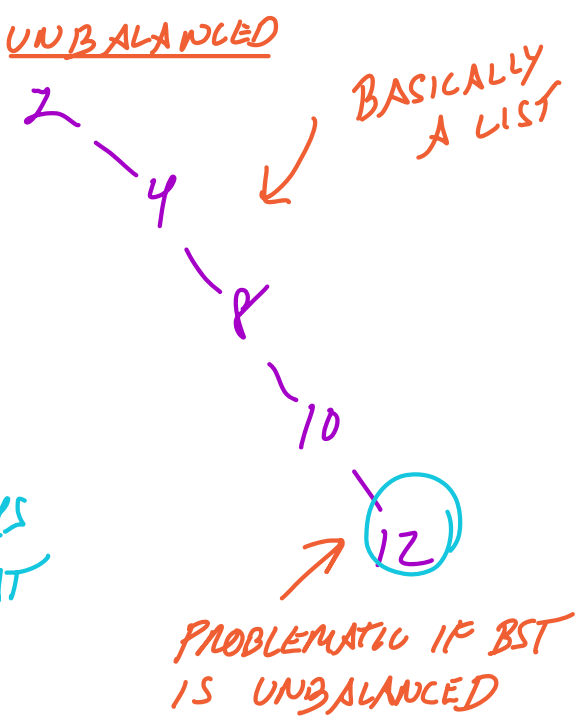
Trees

We know one way to do an ordered representation with trees…

=> BST (Binary search tree): for any node, every smaller node is on the left, any larger node is on the right

What if we used this as a priority queue?



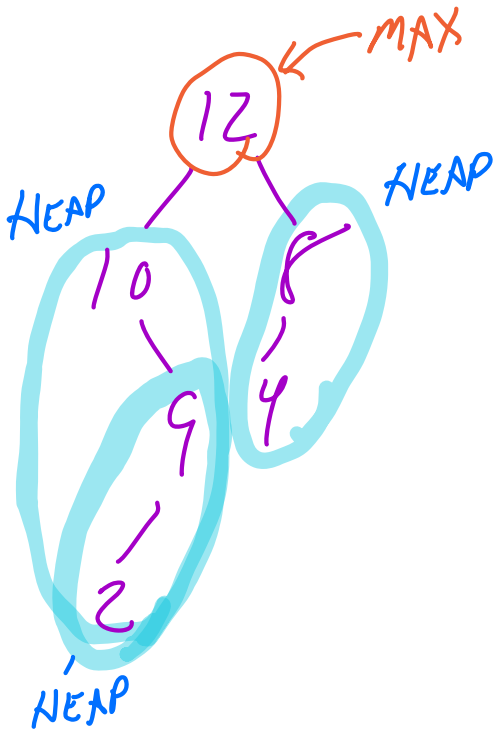


insert, get-max :
- O(logN) if balanced
- O(N) if unbalanced

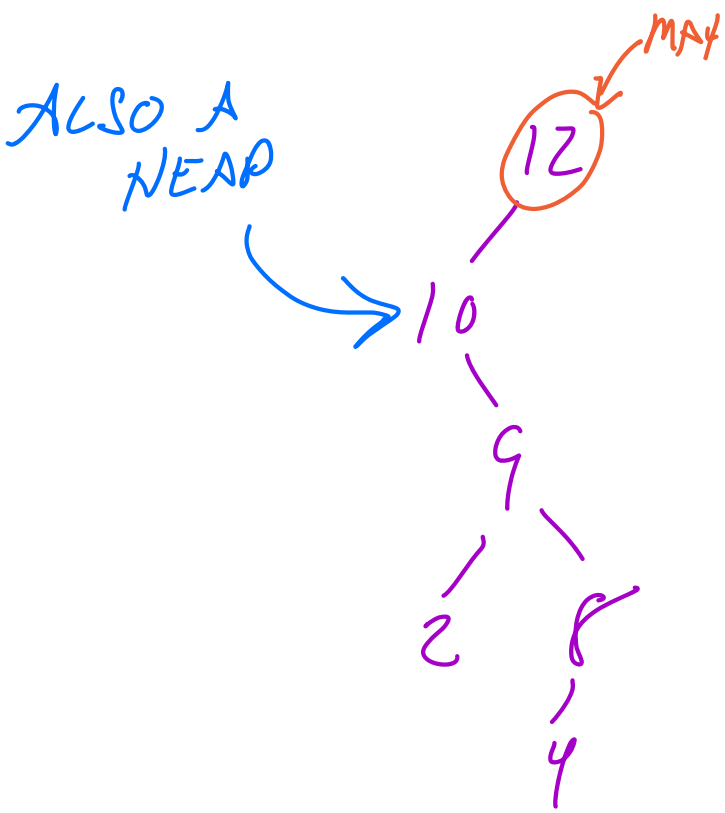
What if we relax the rules a bit?
=> For a priority queue, we don't need a total order like a BST.
What if we just keep the max item at the top??

Heap (binary max heap): a binary tree (*NOT a BST*) with two constraints:
- max item is at the root
- left and right subtrees are also heaps

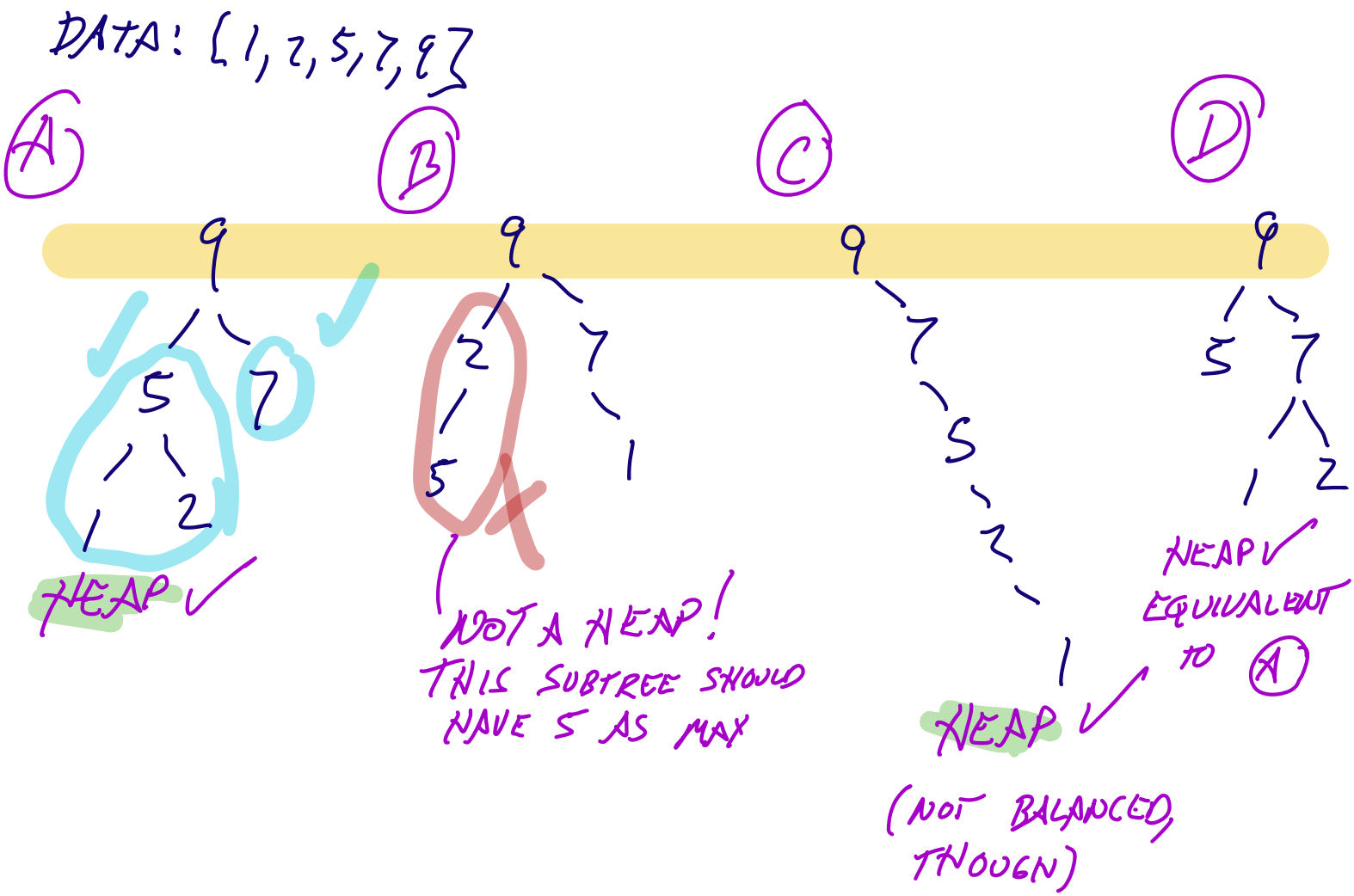
DATA: 2, 4, 6, 9, 10, 12



Note: can have different valid representations for the same heap
(may be more or less-balanced… more on this later)



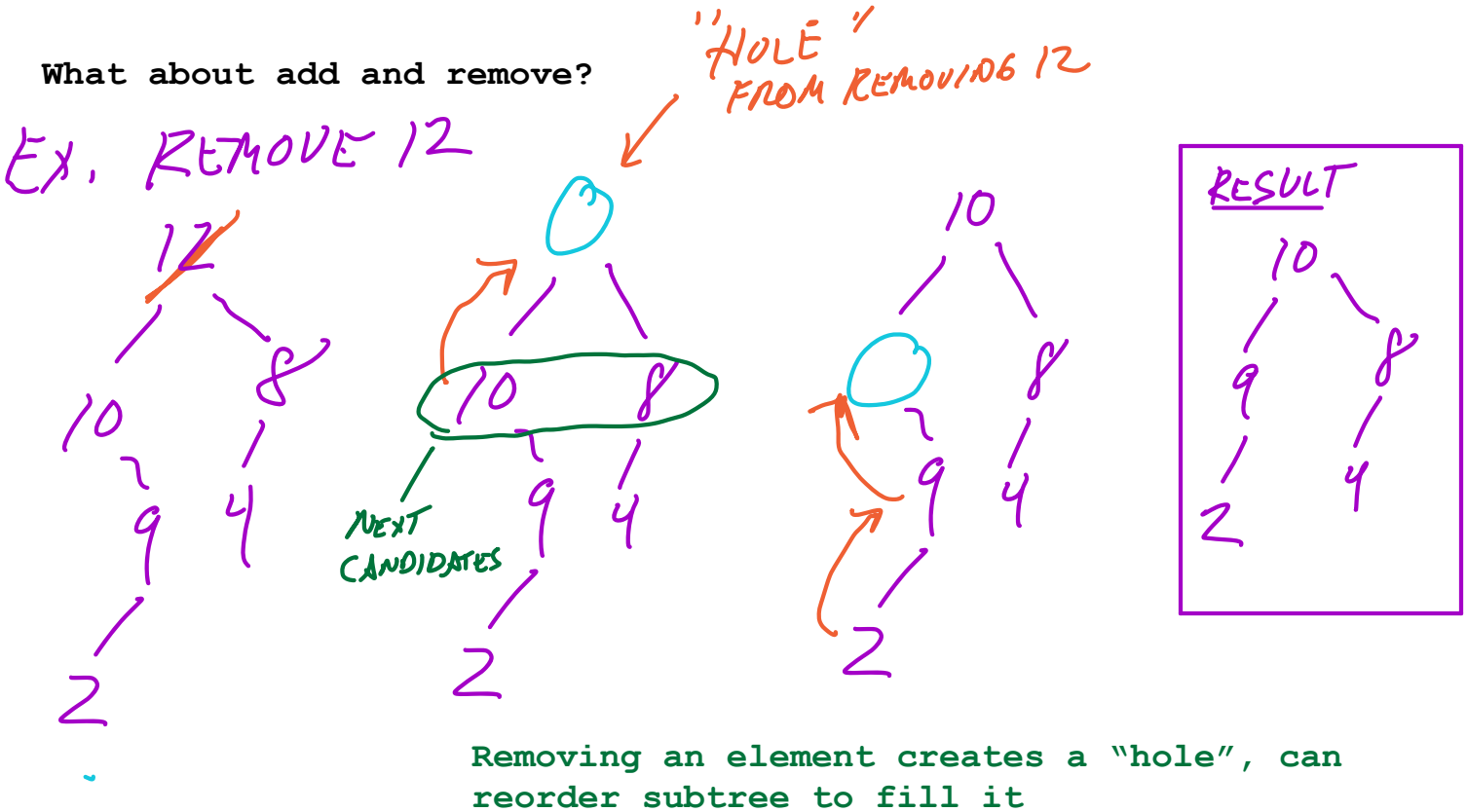
Example: which of these are heaps?



Checking in on our priority queue goals: what can we infer about the runtime of using a heap for a PQ?

```
Priority queue (PQ):  needs 3 operations
 1. Insert a new item          => ???
 2. Remove max-priority item   => ???
 3. Get max-priority item (without removing)
    => O(1) => can just look at top of heap!
```

What about add and remove?

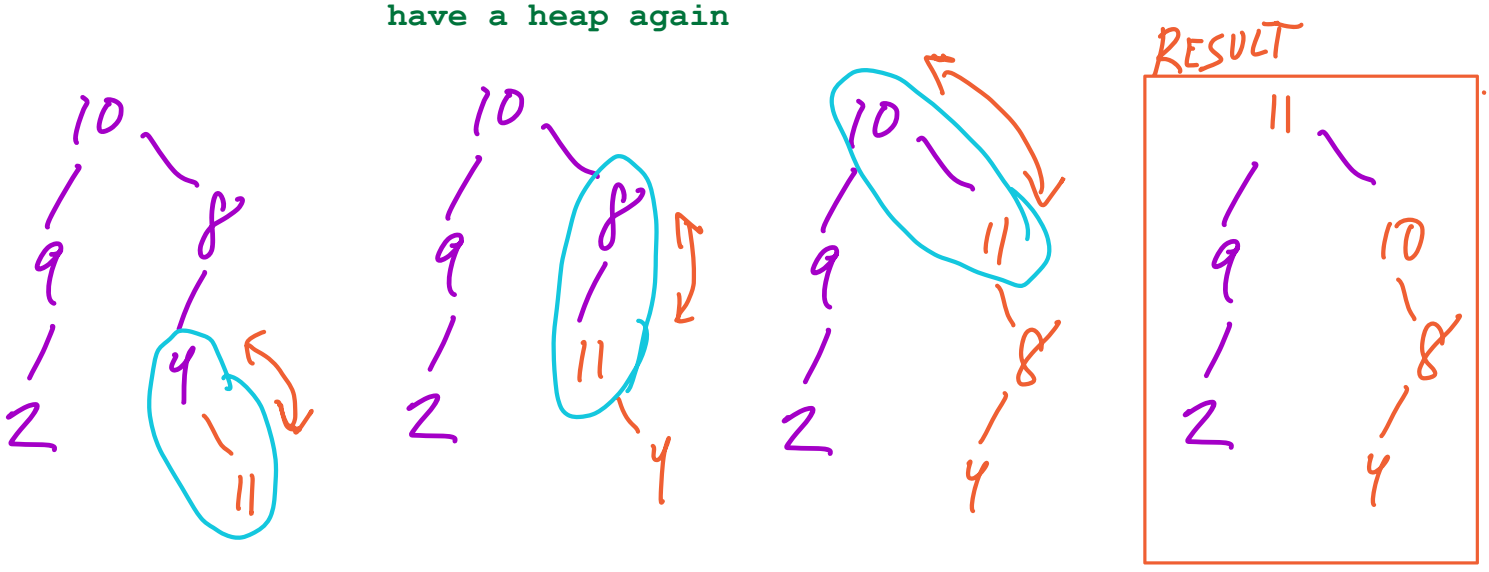


Removing an element creates a "hole", can reorder subtree to fill it

To reorder, we only need to consider one "branch" of the heap => If heap is balanced, this takes O(logN)

WHAT IF WE WANT TO INSERT 11?

Strategy: add to bottom, reorder until we have a heap again



Again, we only need to reorder one "branch" of the heap => O(logN)

So to summarize...

Priority queue (PQ): needs 3 operations

if heap is balanced:

1. Insert a new item => O(logN)
2. Remove max-priority item => O(logN)
3. Get max-priority item (without removing)
   => O(1)

If heap is unbalanced, the insert/delete steps are harder:

- insert: O(N)
- remove_max: O(N)
- get_max: O(1)

Open questions (for next time):

- How to find an empty spot to insert?
- How to keep the heap balanced to ensure logN runtime?
- What does "balanced" even mean, anyway?