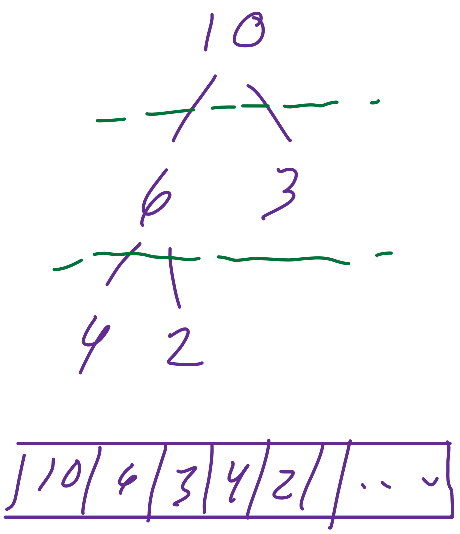
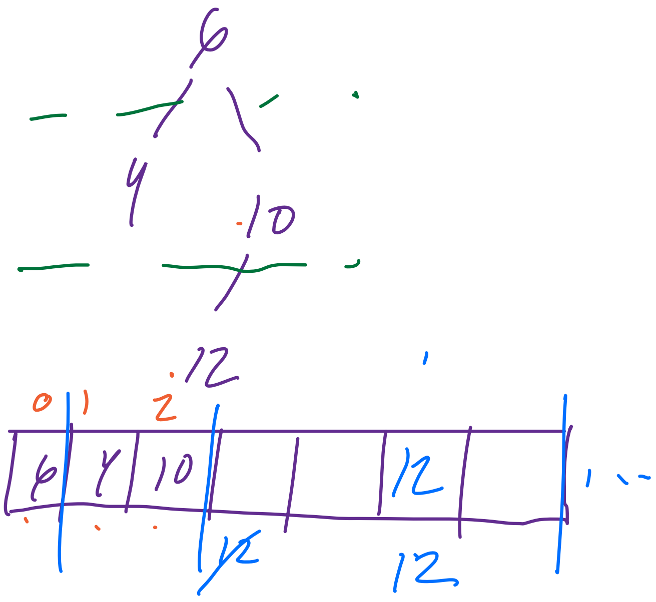
WHAT ARE THE ARRAYS FOR THESE TREES?

Recap: Embedding trees/heaps in Arrays

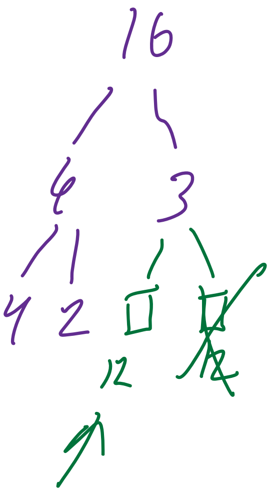




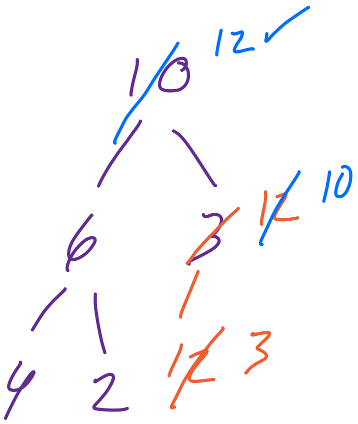
```
For some node at index i:
 - left(i)   is at (i * 2) + 1
 - right(i)  is at (i * 2) + 2
 - parent(i) is at floor((i - 1)/2)
```

Why not keep stuff condensed? Need to respect parent/child formulas to match the structure of the tree
=> Without spaces, parent, left/right formulas will break

How to insert?
In the code version: always add new elements to the next available space
 => Keeps balanced
 => Always know how to find next slot (just the end of the array)

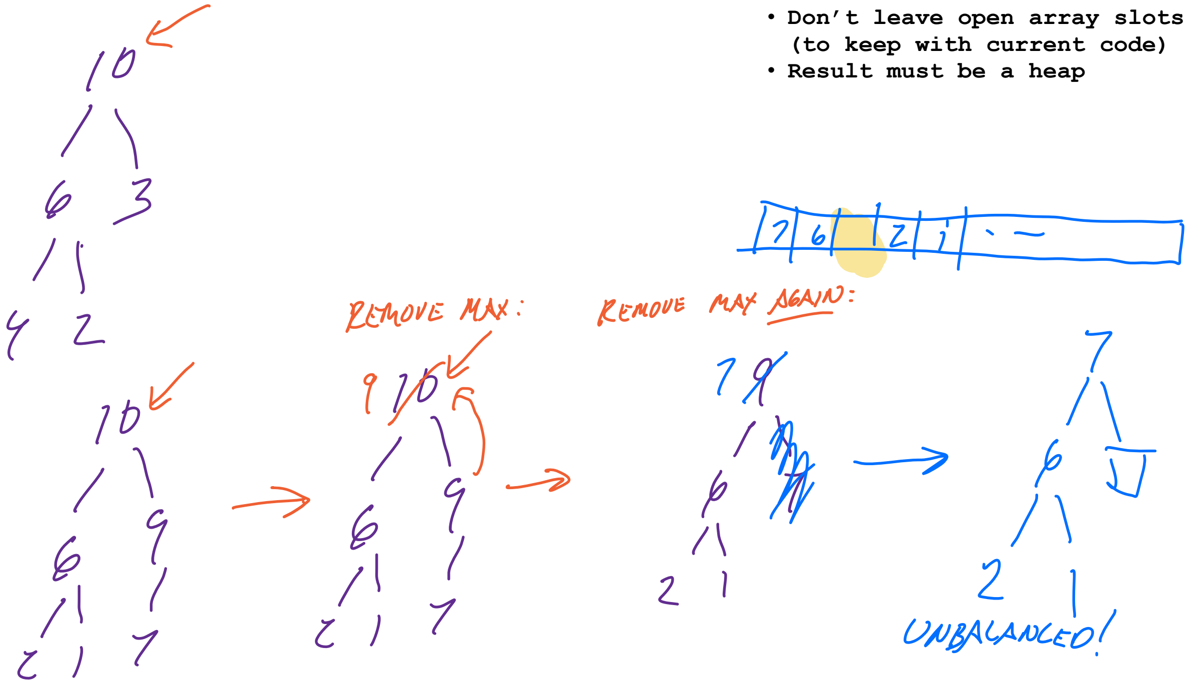


Ex. INSERT 12



How would we implement delete_max?
Let's try something similar to insert, where we swap nodes up after removing an element:

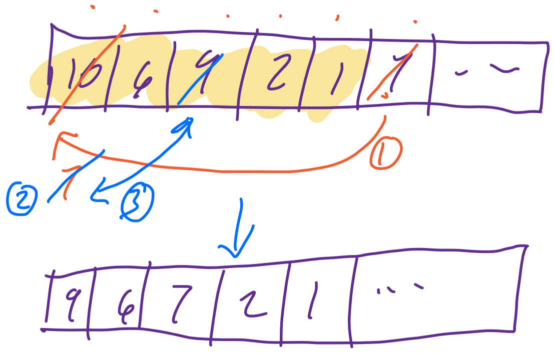
Requirements
- Maintain balance
- Don't leave open array slots (to keep with current code)
- Result must be a heap



Problem: If use the same remove + swap from here, can end up with unbalanced tree
=> Need to do delete without creating gaps
=> If we can just swap values within existing cells in use, can avoid gaps
=> Leverages array structure!

We didn't implement remove_max, but here's the intuition:
- If we're removing an element, the resulting array must be one smaller than before
- Removing the max element creates a hole => swap the last element (ie, the one that would get "abandoned" if we were to shrink the array) to the top (in the hole left by the max)
- Swap this new top element <u>down</u> until the result is a heap

This idea leverages the same principles as insert, but involves swapping in a different direction. Similar to how we knew where to add a new element when inserting into the heap, we leverage the array to know where to find a new element to start swapping down (ie, last one)



① REMOVE 10
② SWAP LAST (7) TO FRONT
③ SWAP 7 DOWN UNTIL RESULT IS HEAP

RECAP: HOW TO THINK ABOUT ARRAYS

Essential: have items in predictable, and computable, locations in memory
=> "where is the i'th element"

HASH MAP
HASH SET

Use predictable location to get from hash value to some specific index

=> Not all indices are used
=> Positions correspond to "array slots"

ARRAY LIST

Use predictable location for get(i)

=> Items in consecutive locations in memory

USUALLY THINK ABOUT SHIFTING ELEMENTS TO MAINTAIN THIS

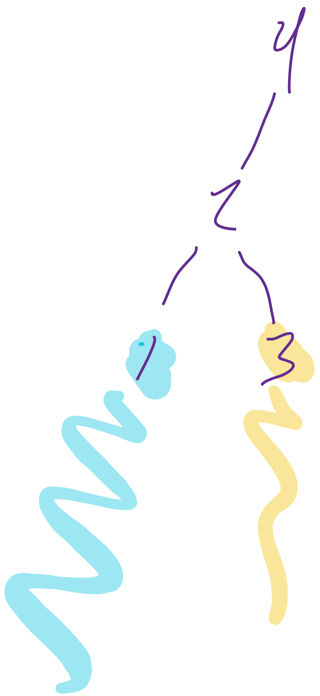
HEAP (TREE)

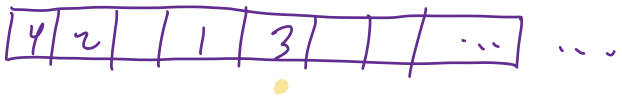
Use predictable location to navigate between parent/child nodes

=> Positions correspond to where we are in tree

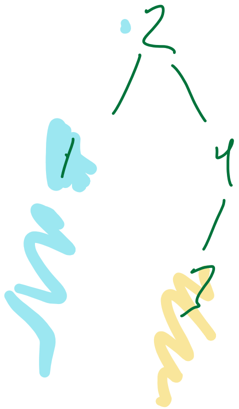
Important to think about
- Ways different data structures can be used
- How the underlying data structure (arrays, in this case) can matter for different applications

Do all binary trees belong in arrays? What about BSTs?





What's questions do you need to ask to decide if this is a good idea?



Here's a way this tree could be balanced. Try it: think about how the array structure would need to change if we made this change.

What if (1) and (3) had lots of child nodes? With an array, we'd need to do a lot of shifting nodes around to make this work! For cases like this, it makes more sense to use a node-based representation (ie, the traditional recursive data structure approach we've seen before), where we just need to alter the parent/child references.

Overall: think about how are going to use the data structure
= > What operations do you need to perform (at a high level)
=> For BSTs: need to keep the BST ordered and balanced

=> How does that translate into operations on the data structure
=> If we used an array, we would need to shift a lot of elements around to keep the BST balanced!

```
import math

"""implementation of a max heap"""
class Heap:
   def __init__(self):
       self.data = []
       self.size = 0

   def __str__(self):
       """string representation is the underlying list"""
       return str(self.data)

   def parent_index(self, of_index):
       """compute parent index of given index. Assumes of_index > 0"""
       return math.floor((of_index - 1) / 2)

   def swap(self, index1, index2):
       """swaps values in index1 and index2 within self.data"""
       tmp = self.data[index1]
       self.data[index1] = self.data[index2]
       self.data[index2] = tmp

   def insert(self, new_elt):
       """insert element into the heap"""
       self.data.append(new_elt)
       self.sift_up(self.size)
       self.size += 1

   def sift_up(self, from_index):
       """swap element in from_index up heap until it is in the right place"""
       if from_index > 0:
           parent = self.parent_index(from_index)
           if self.data[from_index] > self.data[parent]:
               self.swap(parent, from_index)
               self.sift_up(parent)

   def sift_up_while(self, from_index):
       """a while-loop based version of sift_up"""
       if from_index > 0:
           curr_index = from_index
           parent = self.parent_index(curr_index)
           while (curr_index > 0) and \
                 (self.data[curr_index] > self.data[parent]):
               self.swap(parent, curr_index)
               curr_index = parent
               parent = self.parent_index(curr_index)
               # Note: this version repeats last two lines, unlike the recursive one
```

Heaps Implementation with Arrays