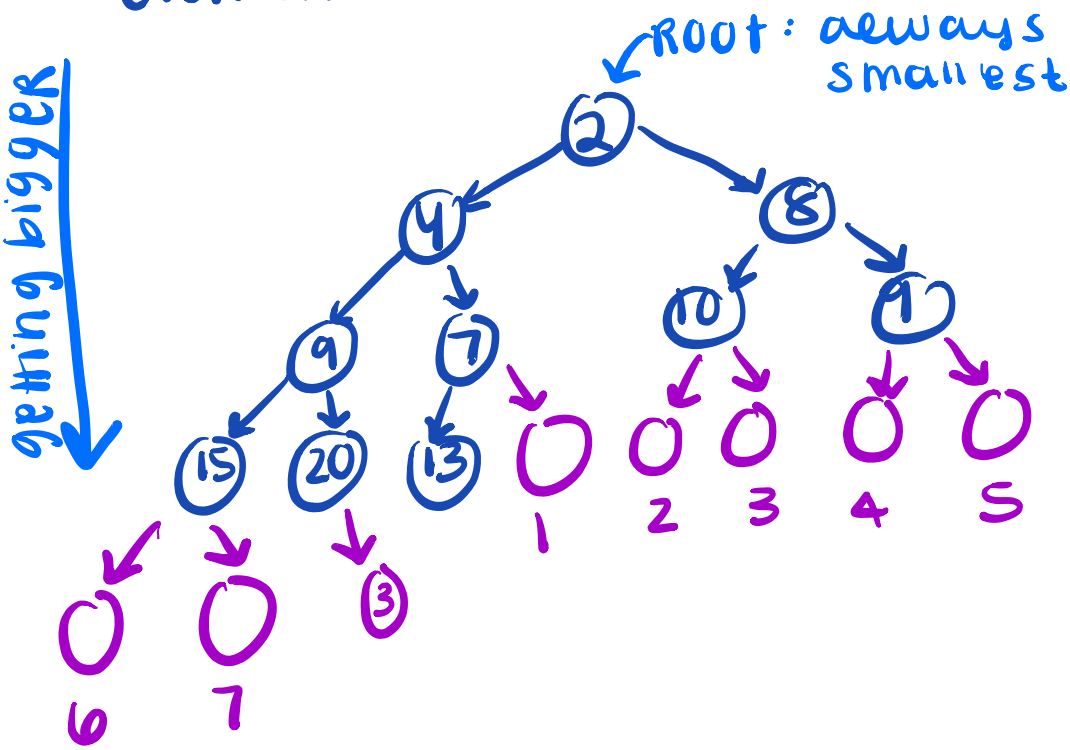
HEAPS

basically reverse of MIN

MIN

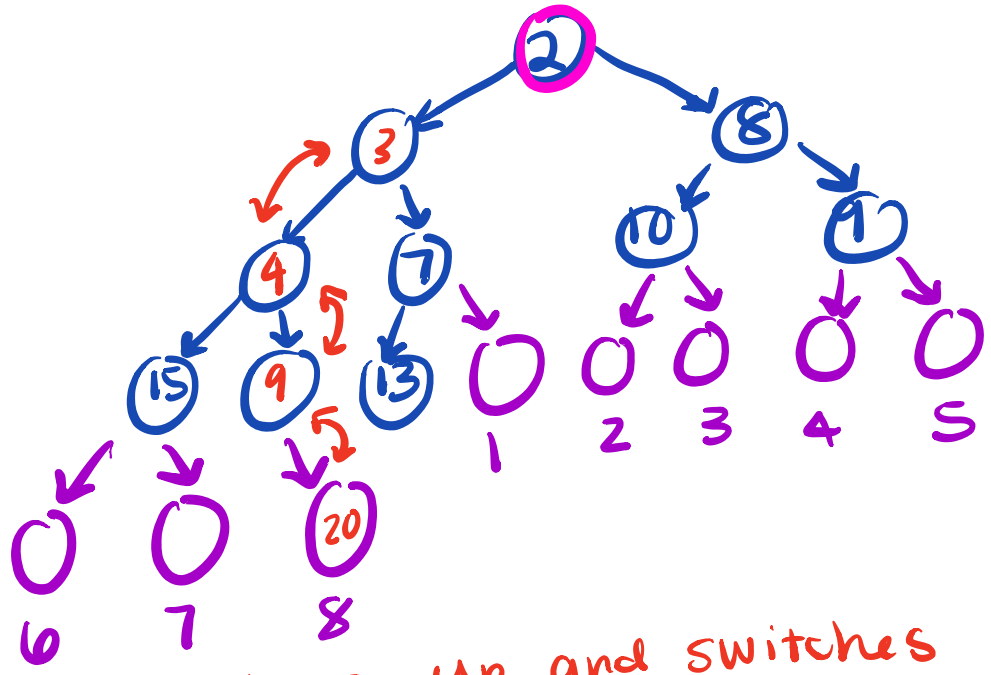
MAX

MIN: root node would be the very smallest elements because all the elements are all smaller than their children, looking down the tree (down the heap) the elements get bigger and bigger.



INSERTION: next element always goes to the next empty spot looking from top → bottom, left → right (1, 2, 3, ... 7)

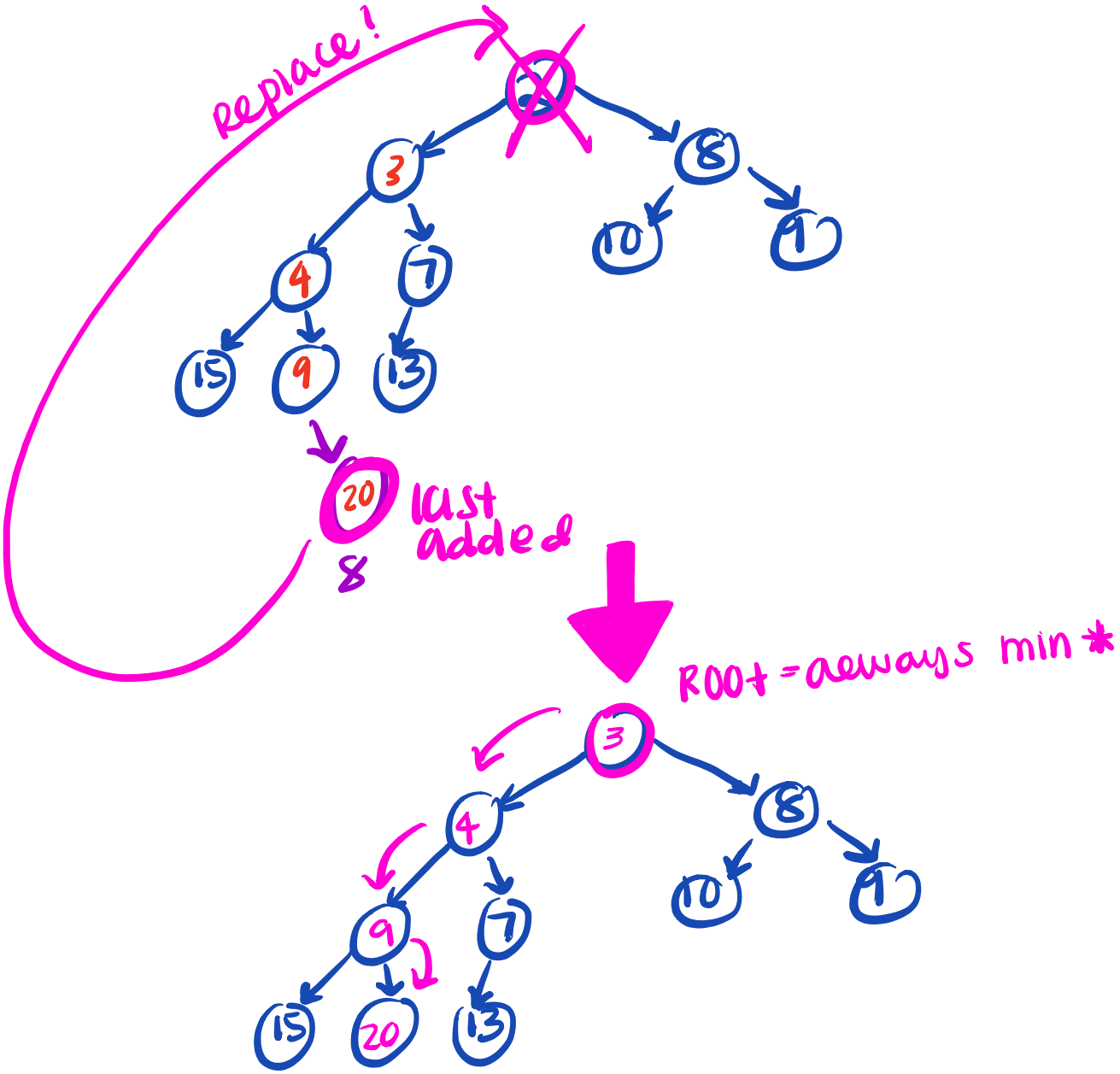
but if that isn't where that # should be on the MIN Heap, then we can bubble it up to its CORRECT position



so 3 bubbles up and switches positions with 20 and the 3 in under 9 so 9 and 3 switch positions, and then 3 is under 4 so 3 and 4 switch positions so 3 is under the root 2 and that is correct so now 3 has correctly been inserted.

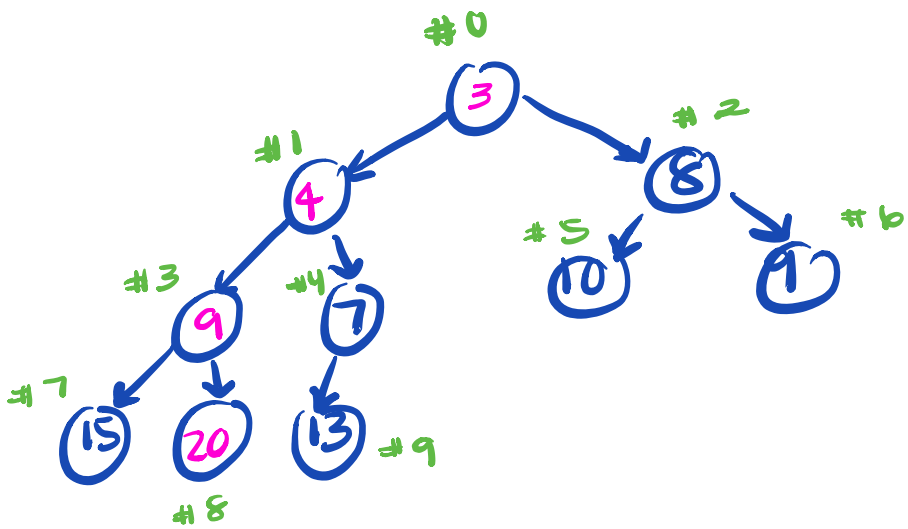
Removing: the minimum element?

you empty out the ROOT, then replace the node # with the last element added. then that element might not be at the correct spot, so we take the ROOT element and bubble it down to the next spot. you compare the ROOT with the left and Right children, the swap it with the smaller of the two.



20 above 3 and 8, swap 20-3, 20 above 4 & 7
swap 20-4, 20 above is 9, swap 20-9

Implementation: NO gaps in heap, and we can use an ARRAY to store these values because they all are in a specific spot. This makes them very compact, and a simple equation can map FROM an index to its left child, its right child OR to its parent



0	3
1	4
2	8
3	9
4	7
5	10
6	9
7	15
8	20
9	13

Parent = (index - 2) / 2

↑

index

↙ left = index * 2 + 1

↘ Right = index * 2 + 2

→ so we can get to the left and Right child without having this overhead Node class.

CODE: Implement a MIN HEAP with a class that WRAPS the items ARRAY...

```
Public class Min Int Heap{
    capacity = 10;
    size = 0;
    int[] items = new int[capacity];
```

ARRAY of all tree values.

```
    int getLeftChildIndex(int parentIndex)
        {return 2 * parentIndex + 1;}
    int getRightChildIndex(int parentIndex)
        {return 2 * parentIndex + 2;}
    int getParentIndex(int childIndex)
        {return (childIndex - 1) / 2;}

    bool hasLeftChild(int index)
        {return getLeftChildIndex(index) < size;}
    bool hasRightChild(int index)
        {return getRightChildIndex(index) < size;}
    bool hasParent(int index)
        {return getParentIndex(index) >= size;}

    int leftChild(int index)
        {return items[getLeftChildIndex(index)];}
    int RightChild(int index)
        {return items[getRightChildIndex(index)];}
    int parent(int index)
        {return items[getParentIndex(index)];}
```

```
void swap (int index1, int index2) {
    //swaps values at these two
      indicies
}

void ensureExtraCapacity () {
    if (size == capacity) {
        items = ARRAY.copyof (items, capacity*2);
        capacity *= 2;
    }
}
```

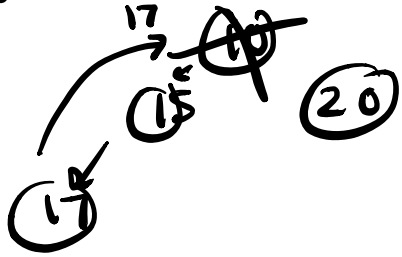
* this doubles the ARRAY capacity and copies the data into the bigger ARRAY.

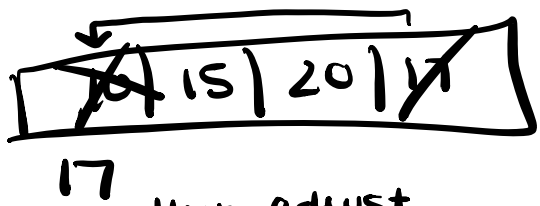
Basics on how ARRAYlists operates

* **peak method** → returns 1st element in the array (aka the ROOT) if not empty.

* **pull method** → extracts the min. element and removes it from the ARRAY if it is not empty. So take the last element in the ARRAY, move it to the element befor, shrink size.

then heapify down (method)





then adjust.

* **add method** → ensure extra capacity()

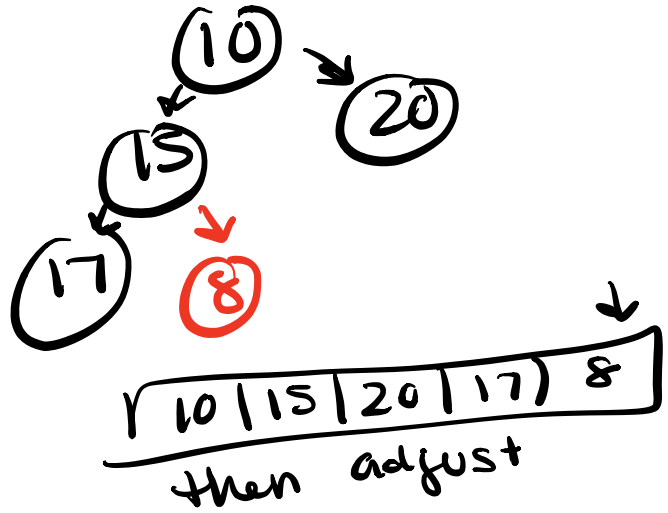
add element into last spot

items[size] = item;

increase size

size ++;

heapifyUp();



Heapify up → starts at element last added (index = size - 1;) the as long as there is a parent item, walk up (while (hasParent(index)) and while out of order (parent(index) > items[index])

then swap the parent with the index

swap (getParentIndex[index], index);

Index = getParentIndex[index]

heapify down → start at root (0) and as long as you have children and its out of order, replace.

very similar to heapify up

```
index = 0
while (hasLeftChild(index)){
    int smallerChildIndex = getLeftChildIndex(index);
    if (hasRightChild() && rightChild(index) < leftChild(index)){
        smallerChildIndex = getRightChildIndex(index);
    }
    if (items[index] < items[smallerChildIndex]){
        break; // all good! done!
    }
    else{
        swap(index, smallerChildIndex);
    }
    index = smallerChildIndex;
}
}
```

MIN HEAP:
- The root would be the smallest element and then the nodes would get larger and larger as you move down the tree with the largest node being the left child.

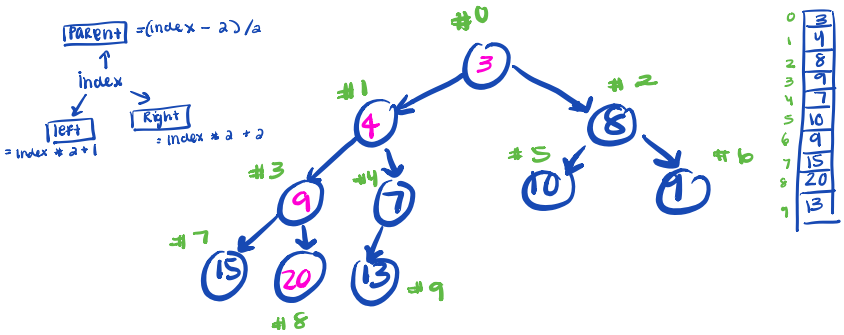
Insertion: Inserted element always goes to the next empty spot looking from Top -> Bottom and then Left -> Right, and then to put it in the right position you need to bubble up by compares the current node with parent node.

Removing: You empty the root and then replace the root with the last element then bubble it down by comparing it to the children of that parent.

Implementation: No gaps in the heap, and we can use an array to store these values bc they are in a specific spot. This makes them very compact, and a simple equation can map from an index to its left child, its right child or its parent.

```
class MinIntHeap{
      capacity = 10;
      size = 0;
      int[] items = new int[capacity];

      int getleftchildindex(int parentindex) { return 2*parentindex + 1 }
      int getrightchildindex(int parentindex) { return 2*parentindex + 2 }
      int getparentindex(int childindex) { return ( childindex -1 ) / 2 }

      bool hasleftchild(int index) { return getleftchildindex(index) < size}
      bool hasrightchild(int index) { return getrightchildindex(index) < size}
      bool hasparent(int index) { return getparentindex(index) >= size }

      int leftchild(int index) { return items[getleftchild(index)]}
      int rightchild(int index) { return items[getrightindex(index)]}
      int parent(int index) { return items[getparentindex(index)]}

      void swap(int index1, int index2){
      // swap values at these two indicies
      }

      void ensureExtraCredit(){
            if (size == capacity){
                  items = array.copyof(items, capacity * 2);
                  capacity *= 2;
            }
      }

      int peak() method -> returns 1st element in the array (root)
      if it is not empty

      int pull() method -> extracts the min. element and removes it
      from the array if it is not empty. So take the last element
      in the array, move it to the element before it, shrink the size,
      then call the heapify_down() method

      void add() method -> ensure extra capacity, add element into
      the last spot items[size] = item; then increase the size, then
      call the heapify_up() method.

      void heapify_up() -> start at the last added element (index = size -1)
      then as long as there is a parent item, walk up (while (hasParent(index))
      and while out of order ( && parent(index) > items[index]) then swap
      the parent with the index swap(getParentindex[index], index) then
      index = getParentindex[index]

      heapify_down() -> start at the root (0) and as long as you have
      children and it is out of order, replace.

      index = 0;

      while (hasleftchild(index)){
            int smallerchildindex = getleftchildindex[index];
            if (hasrightchild) && rightchild[index] < leftchild[index];
                  smallerchildindex = getrightchild[index];
            }
            if (items[index] < items[smallerchildindex]){
                  break; // all good, done.
            }
            else{
                  swap(index, smallerchildindex);
            }
            index = smallerchildindex;
            }
      }
}
```

MAX HEAP:
- The root would be the largest element and then the nodes would get smaller and smaller as you move down the tree with the smallest node being the left child.



heaps have to balanced.