THE BINARY HEAP

WHILE INSERTING OR REMOVING AN ELEMENT INTO THE HEAP HOW DO WE KNOW WHICH IS THE RIGHT POSITION FOR THE ELEMENT TO OCCUPY?

WE PLACE A SINGLE ELEMENT IN THE WRONG POSITION

THEN WE TRY AND FIND THE RIGHT POSITION FOR THE ELEMENT

THIS PROCESS IS CALLED:

HEAPIFY

THE BINARY HEAP

HEAPIFY

SIFT DOWN

AN ELEMENT IS IN THE WRONG POSITION WITH RESPECT TO OTHER ELEMENTS BELOW IT IN THE HEAP

IT HAS TO BE MOVED DOWNWARDS IN THE HEAP TOWARDS THE LEAF NODES TO FIND IT'S RIGHT POSITION

SIFT UP

AN ELEMENT IS IN THE WRONG POSITION WITH RESPECT TO OTHER ELEMENTS ABOVE IT IN THE HEAP

IT HAS TO BE MOVED UPWARDS IN THE HEAP TOWARDS THE ROOT NODE TO FIND IT'S RIGHT POSITION

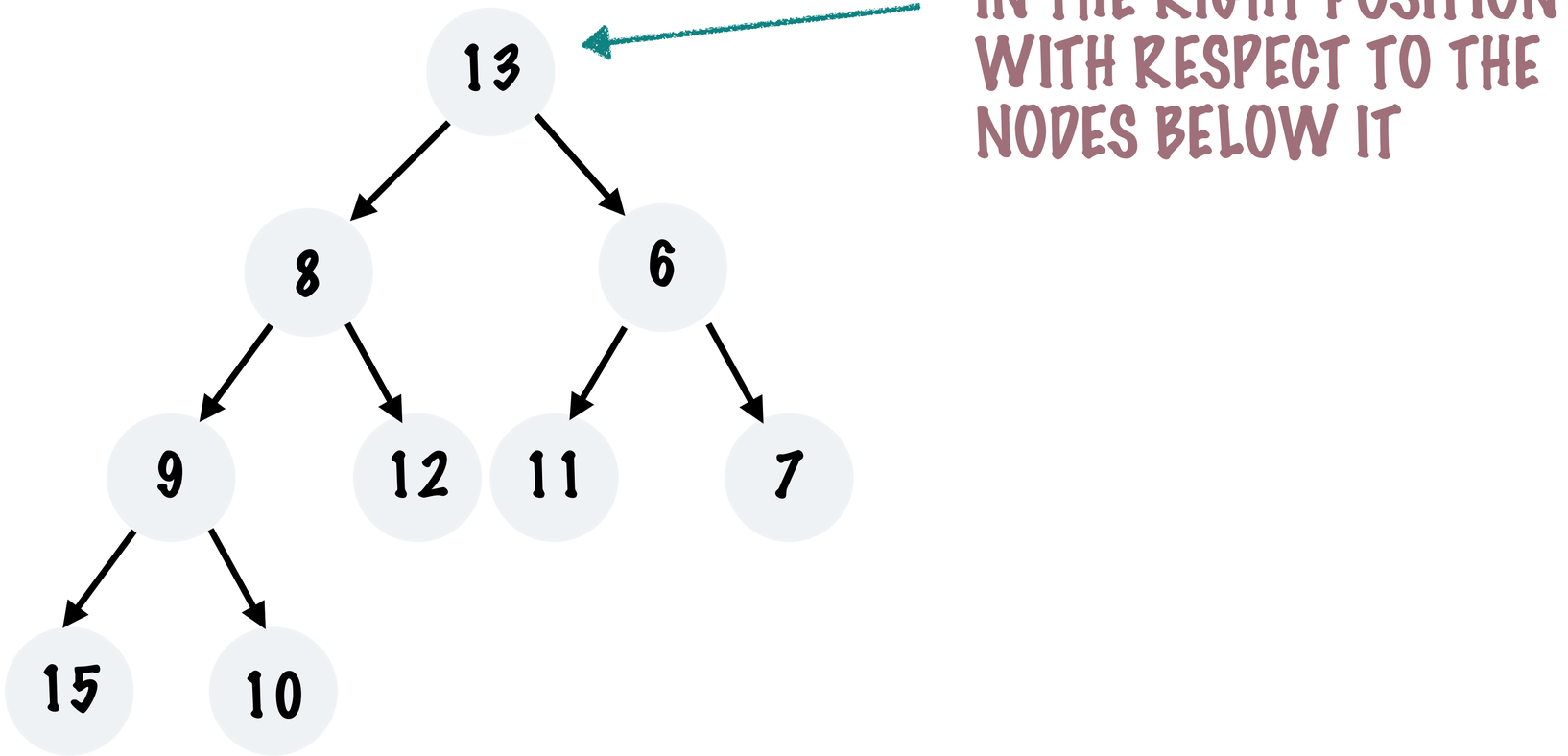
THE BINARY HEAP

HEAPIFY

SIFT DOWN

SIFT UP

THIS IS A MINIMUM HEAP



THE VALUE 13 IS NOT IN THE RIGHT POSITION WITH RESPECT TO THE NODES BELOW IT

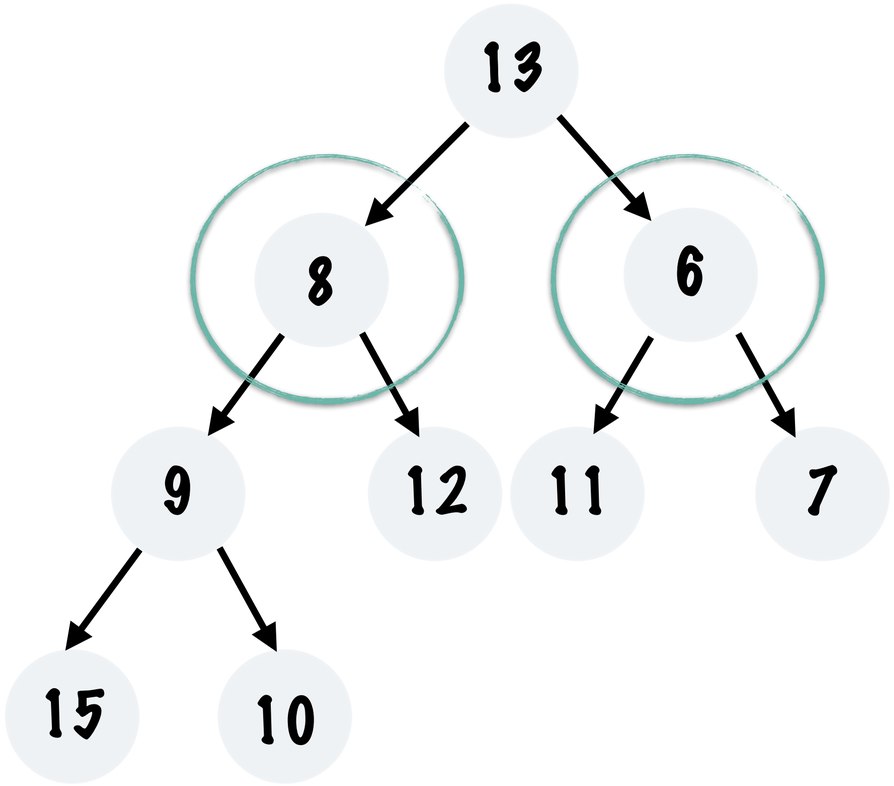
THE BINARY HEAP

HEAPIFY

SIFT UP

SIFT DOWN

6 < 8 THIS MEANS THAT 6 IS THE CANDIDATE FOR SWAP



SWAP 6 AND 13

6 IS THE MINIMUM ELEMENT IN THIS HEAP

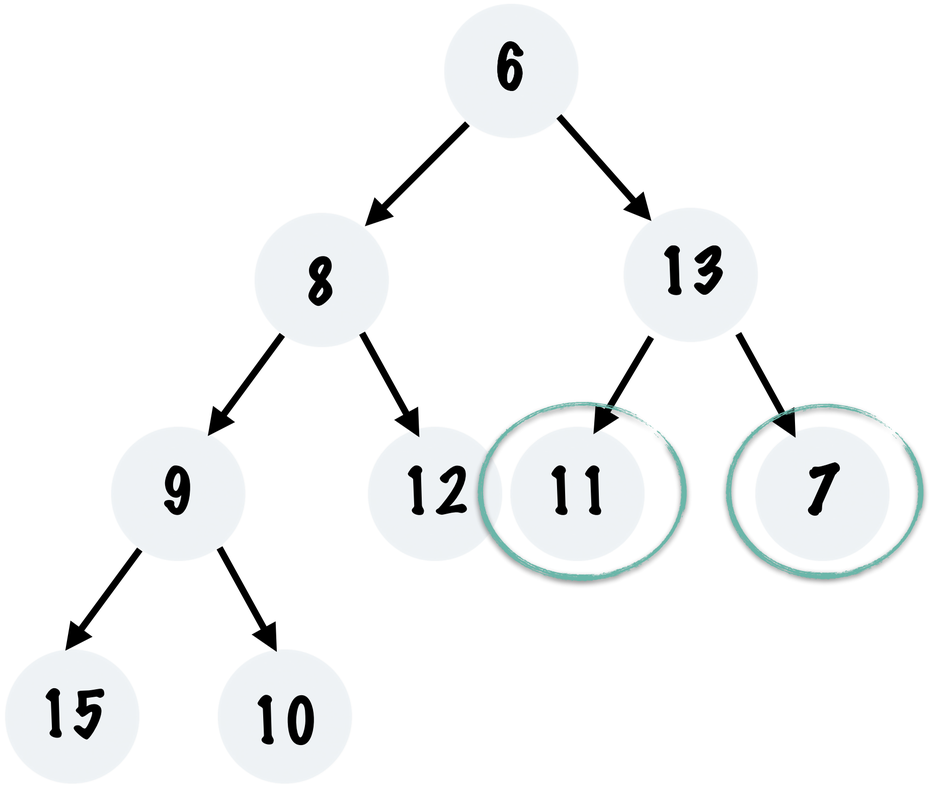
THE BINARY HEAP

HEAPIFY

SIFT UP

SIFT DOWN

7 < 11 THIS MEANS THAT 7 IS THE CANDIDATE FOR SWAP



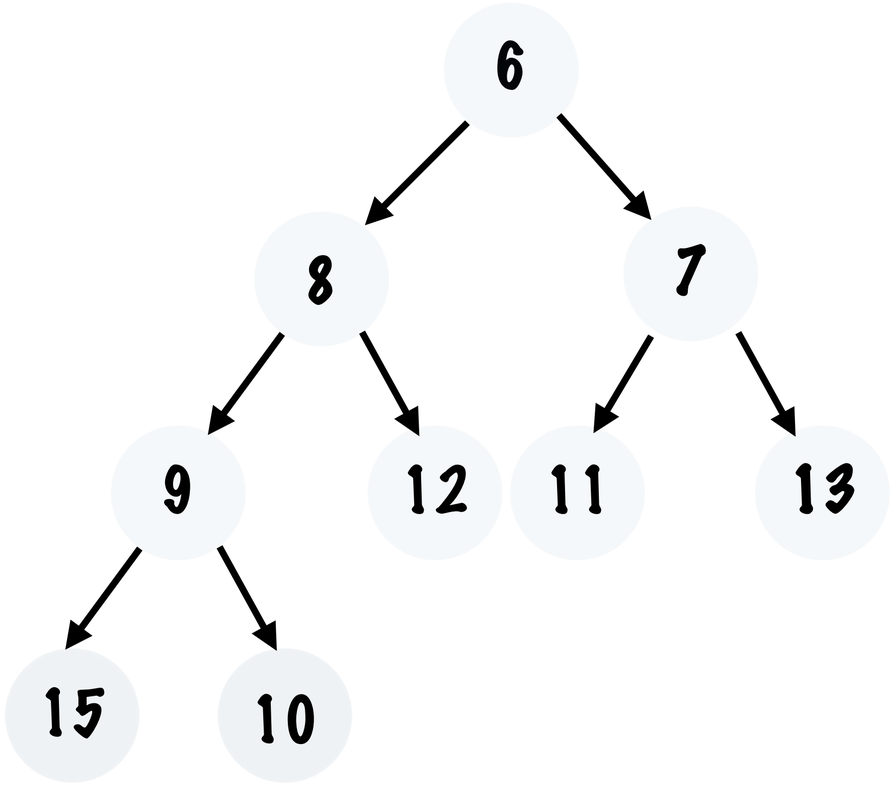
SWAP 7 AND 13

THE BINARY HEAP

HEAPIFY

SIFT UP

SIFT DOWN



13 IS NOW IN THE CORRECT POSITION - HEAPIFY COMPLETE!

THE BINARY HEAP

HEAPIFY

NOW LET'S SEE SOME CODE...

MINIMUM HEAP

```
public class MinHeap<T extends Comparable> extends Heap<T> {


    public MinHeap(Class<T> clazz) {
        super(clazz);
    }

    public MinHeap(Class<T> clazz, int size) {
        super(clazz, size);
    }
```

EXTEND THE HEAP BASE CLASS TO CREATE A MINIMUM HEAP

SET UP THE CONSTRUCTORS

SIFT DOWN

```
@Override
protected void siftDown(int index) {
    int leftIndex = getLeftChildIndex(index);
    int rightIndex = getRightChildIndex(index);

    // Find the minimum of the left and right child elements.
    int smallerIndex = -1;
    if (leftIndex != -1 && rightIndex != -1) {
        smallerIndex = getElementAtIndex(leftIndex).compareTo(getElementAtIndex(rightIndex)) < 0
                ? leftIndex : rightIndex;
    } else if (leftIndex != -1) {
        smallerIndex = leftIndex;
    } else if (rightIndex != -1) {
        smallerIndex = rightIndex;
    }

    // If the left and right child do not exist stop sifting down.
    if (smallerIndex == -1) {
        return;
    }

    // Compare the smaller child with the current index to see if a swap
    // and further sift down is needed.
    if (getElementAtIndex(smallerIndex).compareTo(getElementAtIndex(index)) < 0) {
        swap(smallerIndex, index);
        siftDown(smallerIndex);
    }
}
```

THE ELEMENT AT THIS INDEX HAS TO BE SIFTED DOWN TO THE RIGHT POSITION

GET THE LEFT AND RIGHT CHILD INDICES TO COMPARE VALUES

STORE THE INDEX AT WHICH WE FIND THE MINIMUM VALUE IN SMALLER INDEX

IF THE NODE HAS BOTH LEFT AND RIGHT CHILDREN FIND THE SMALLER VALUE BETWEEN THEM

IF THE NODE HAS ONLY A LEFT OR ONLY A RIGHT CHILD THAT CHILD CAN BE CONSIDERED THE ONE WHICH HAS THE SMALLER VALUE

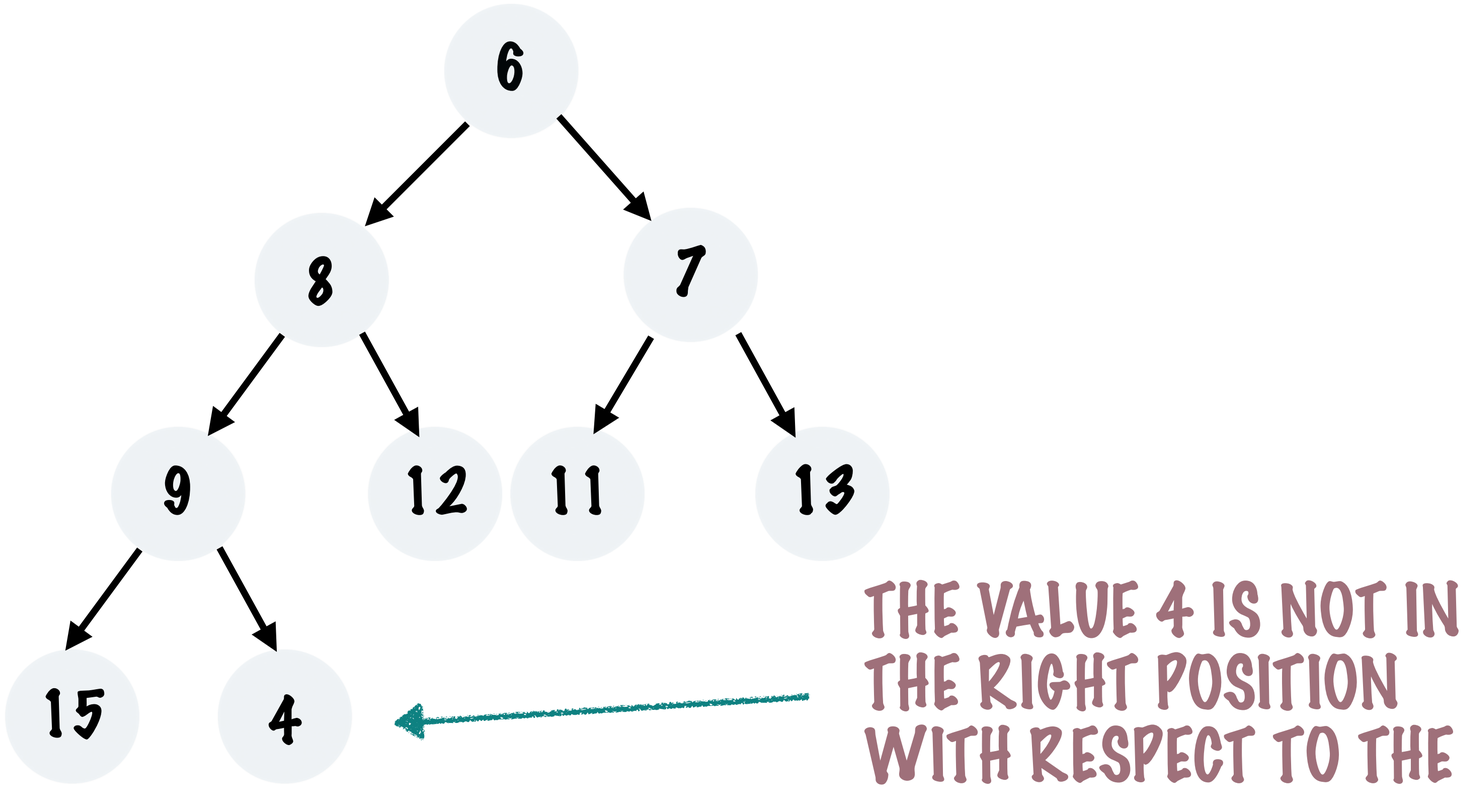
COMPARE THE SMALLER OF THE CHILDREN WITH THE PARENT INDEX AND SWAP IF NEEDED, SIFT THE PARENT DOWN FURTHER

THE BINARY HEAP

HEAPIFY

SIFT DOWN

SIFT UP



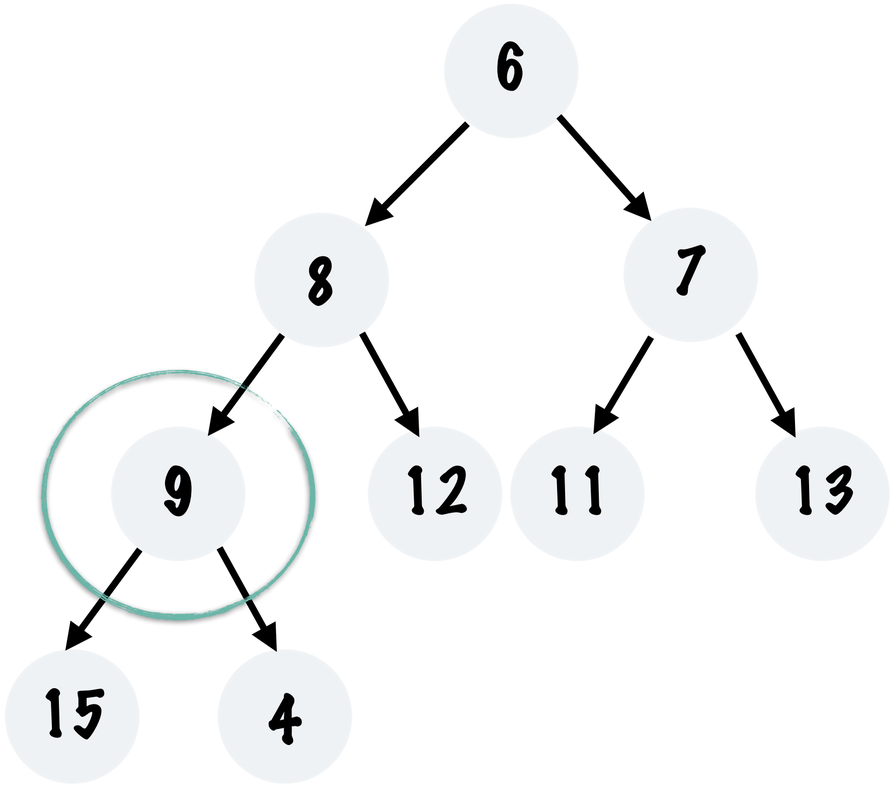
THE BINARY HEAP

HEAPIFY

SIFT DOWN

SIFT UP

4 < 9 THIS MEANS THAT
4 SHOULD BE SWAPPED
WITH ITS PARENT



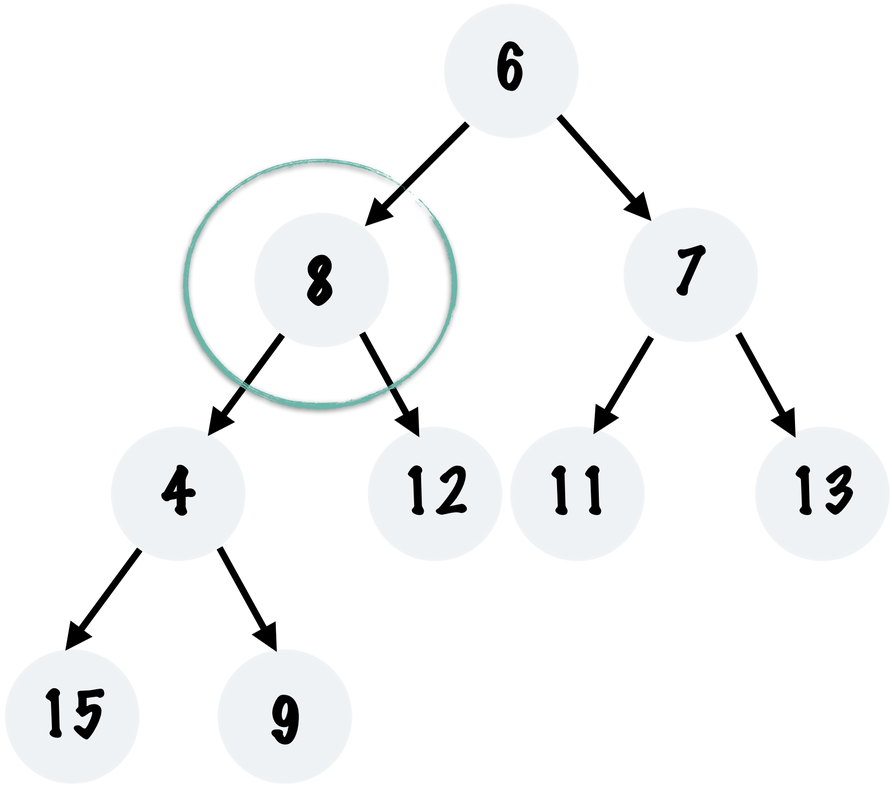
THE BINARY HEAP

HEAPIFY

SIFT DOWN

SIFT UP

4 < 8 THIS MEANS THAT
4 SHOULD BE SWAPPED
WITH ITS PARENT



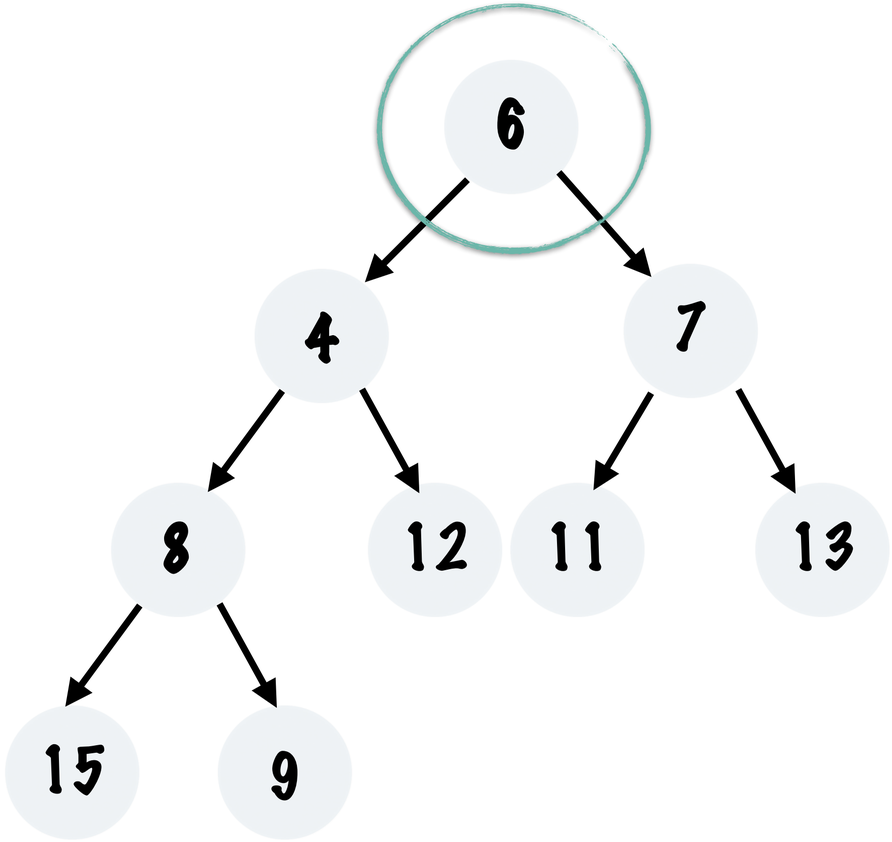
THE BINARY HEAP

HEAPIFY

SIFT DOWN

SIFT UP

4 < 6 THIS MEANS THAT
4 SHOULD BE SWAPPED
WITH ITS PARENT



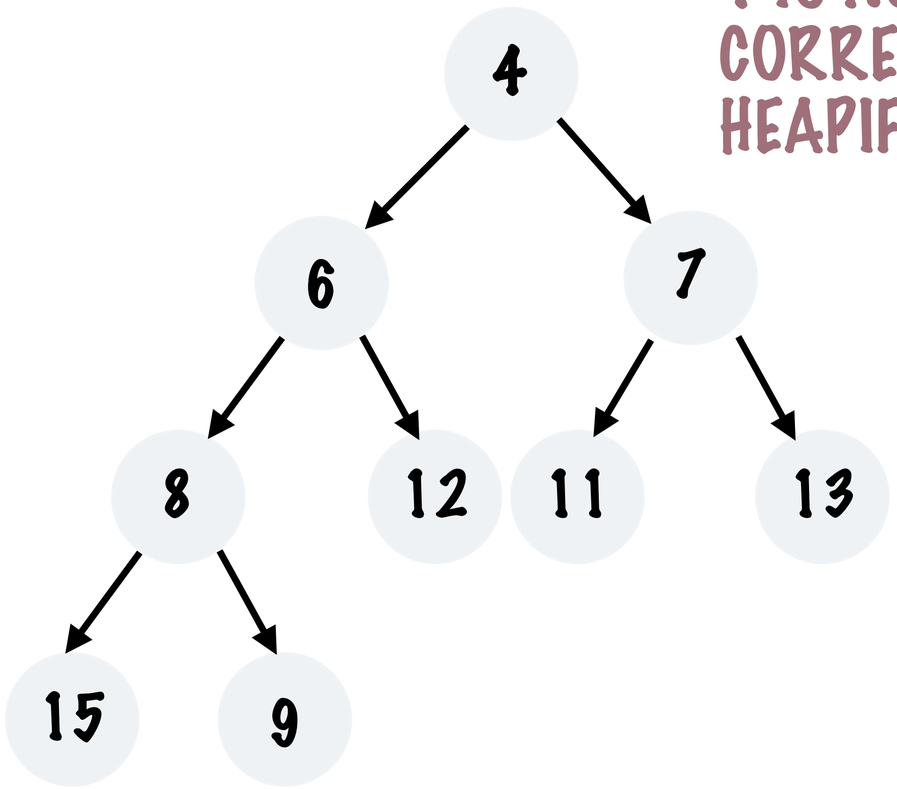
THE BINARY HEAP

HEAPIFY

SIFT DOWN

SIFT UP

4 IS NOW IN THE CORRECT POSITION - HEAPIFY COMPLETE!



THE BINARY HEAP

HEAPIFY

NOW LET'S SEE SOME CODE...

SIFT UP

```
@Override
protected void siftUp(int index)  {
    int parentIndex = getParentIndex(index);

    if (parentIndex != -1 &&
            getElementAtIndex(index).compareTo(getElementAtIndex(parentIndex)) < 0) {
        swap(parentIndex, index);

        siftUp(parentIndex);
    }
}
```

THE ELEMENT AT THIS INDEX HAS TO BE SIFTED UP TO THE RIGHT POSITION

FIND THE PARENT ELEMENT OF THE CURRENT NODE AND COMPARE VALUES

IF THE PARENT IS SMALLER THAN THE CURRENT NODE VALUE THEN PERFORM THE SWAP

CONTINUE TO SIFT UP - AS LONG AS ELEMENTS ARE SWAPPED THE RIGHT POSITION FOR THE ORIGINAL ELEMENT HAS NOT BEEN FOUND