HEAP SORT

HEAPIFY

NOW LET'S SEE SOME CODE...

GET LEFT CHILD INDEX

```
private static int getLeftChildIndex(int index, int endIndex) {
    int leftChildIndex = 2 * index + 1;
    if (leftChildIndex > endIndex) {
        return -1;
    }

    return leftChildIndex;
}
```

CALCULATE THE LEFT CHILD INDEX USING THE FORMULA

CHECK TO SEE IF A LEFT CHILD OF THIS NODE IS PRESENT. IF IT'S LESS THAN COUNT (THE NUMBER OF NODES) THEN IT'S A VALID LEFT CHILD

RETURN -1 IF A VALID LEFT CHILD WAS NOT FOUND

GET RIGHT CHILD INDEX

```java
private static int getRightChildIndex(int index, int endIndex) {
    int rightChildIndex = 2 * index + 2;
    if (rightChildIndex > endIndex) {
        return -1;
    }

    return rightChildIndex;
}
```

CALCULATE THE RIGHT CHILD INDEX USING THE FORMULA

CHECK TO SEE IF A RIGHT CHILD OF THIS NODE IS PRESENT. IF IT'S LESS THAN COUNT (THE NUMBER OF NODES) THEN IT'S A VALID RIGHT CHILD

RETURN -1 IF A VALID RIGHT CHILD WAS NOT FOUND

GET PARENT INDEX

```java
private static int getParentIndex(int index, int endIndex) {
    if (index < 0 || index > endIndex) {
        return -1;
    }

    return (index - 1) / 2;
}
```

CHECK THAT THE INDEX IS NOT OUT OF RANGE

USE THE FORMULA TO GET THE PARENT INDEX

SWAP

```java
private static void swap(int index1, int index2) {
    int tempValue = array[index1];
    array[index1] = array[index2];
    array[index2] = tempValue;
}
```

HOLD THE VALUE OF INDEX1 IN A TEMPORARY VARIABLE

SWAP THE VALUES AT THE TWO SPECIFIED INDICES

PERCOLATE DOWN

```java
private static void percolateDown(int index, int endIndex) {
    int leftChildIndex = getLeftChildIndex(index, endIndex);
    int rightChildIndex = getRightChildIndex(index, endIndex);

    if (leftChildIndex != -1 && array[leftChildIndex] > array[index]) {
        swap(leftChildIndex, index);
        percolateDown(leftChildIndex, endIndex);
    }
    if (rightChildIndex != -1 && array[rightChildIndex] > array[index]) {
        swap(rightChildIndex, index);
        percolateDown(rightChildIndex, endIndex);
    }
}
```

GET THE LEFT AND RIGHT CHILD INDICES

FOR LEFT CHILD AND RIGHT CHILD IN RANGE CHECK IF THE MAX HEAP PROPERTY IS SATISFIED

IF NOT SWAP ELEMENTS AND PERCOLATE DOWN FURTHER TO ENSURE ALL CHILD NODES SATISFY THE MAX HEAP PROPERTY

NOTE THAT THE CURRENT INDEX SHOULD HOLD THE MAX OF THE LEFT AND RIGHT CHILD - BOTH COMPARISONS NEED TO BE DONE

HEAPIFY

```
public static void heapify(int endIndex) {
    int index = getParentIndex(endIndex, endIndex);
    while (index >= 0) {
        percolateDown(index, endIndex);
        index--;
    }
}
```

HEAPIFY THE ENTIRE ARRAY

START WITH THE PARENT INDEX OF THE LAST ELEMENT IN THE ARRAY

PERCOLATE THE ELEMENTS DOWN THE HEAP TO THE RIGHT LOCATIONS

HEAP SORT

SORT

HEAP SORT

SORT

THIS IS A MAXIMUM HEAP CONSTRUCTED IN-PLACE USING HEAPIFY

0	1	2	3	4	5	6	7	8	9	10
56	17	12	5	14	9	4	2	1	10	6

MOVE THE LARGEST ELEMENT TO THE CORRECT POSITION IN THE SORTED ARRAY

HEAP SORT

SORT

LEAVE THE LAST ELEMENT OUT OF THE HEAP - IT'S PART OF THE SORTED ARRAY AND NO LONGER PART OF THE HEAP

0	1	2	3	4	5	6	7	8	9	10
6	17	12	5	14	9	4	2	1	10	56

NOW THE ELEMENT 6 AT INDEX 0 DOES NOT SATISFY THE HEAP PROPERTY

PERCOLATE DOWN THE ELEMENT TO IT'S CORRECT POSITION

HEAP SORT

SORT

0	1	2	3	4	5	6	7	8	9	10
17	6	12	5	14	9	4	2	1	10	56

NOW THE ELEMENT 6 AT INDEX 1 DOES NOT SATISFY THE HEAP PROPERTY

PERCOLATE DOWN THE ELEMENT TO IT'S CORRECT POSITION

HEAP SORT

SORT

0	1	2	3	4	5	6	7	8	9	10
17	14	12	5	6	9	4	2	1	10	56

NOW THE ELEMENT 6 AT INDEX 4 DOES NOT SATISFY THE HEAP PROPERTY

HEAP SORT

SORT

0	1	2	3	4	5	6	7	8	9	10
17	14	12	5	10	9	4	2	1	6	56

WE'VE NOW GOT A VALID
HEAP ONCE AGAIN!

HEAP SORT

SORT

THE LARGEST ELEMENT IS ONCE AGAIN AT THE VERY FIRST POSITION AT INDEX 0

0	1	2	3	4	5	6	7	8	9	10
17	14	12	5	10	9	4	2	1	6	56

MOVE THE LARGEST ELEMENT TO ITS CORRECT LOCATION IN THE SORTED ARRAY

REMEMBER THE ELEMENT 56 AT INDEX 10 IS ALREADY SORTED

HEAP SORT

SORT

0	1	2	3	4	5	6	7	8	9	10
6	14	12	5	10	9	4	2	1	17	56

ONCE AGAIN HEAPIFY ELEMENT 6 AT INDEX 0

THE LAST 2 ELEMENTS ARE IN THE CORRECT SORTED POSITION, THEY ARE NO LONGER PART OF THE HEAP

REMOVING FROM THE HEAP AND MOVING TO THE END OF THE ARRAY CONTINUES TILL THE ENTIRE ARRAY IS SORTED!

HEAP SORT

SORT

0	1	2	3	4	5	6	7	8	9	10
1	2	4	5	6	9	10	12	14	17	56

A SORTED ARRAY!

HEAP SORT

SORT

NOW LET'S SEE SOME CODE...

HEAPSORT

```
public static void heapsort() {
    heapify(array.length - 1);

    int endIndex = array.length - 1;
    while (endIndex > 0) {
        swap(0, endIndex);
        endIndex--;
        percolateDown(0, endIndex);
    }
}
```

HEAPIFY THE ENTIRE UNSORTED ARRAY SO IT'S NOW A HEAP

START WITH THE VERY LAST INDEX, AND PLACE THE LARGEST ELEMENT IN THE LAST POSITION

REDUCE THE END INDEX INDICATING THAT THE HEAP NO LONGER INCLUDES THE ELEMENTS WHICH ARE IN THE CORRECTLY SORTED POSITION

HEAP SORT

SORT

HEAP SORT USES OPERATIONS ON A HEAP TO GET A SORTED LIST

INSERTION INTO A HEAP IS DONE N TIMES TO GET ALL THE ELEMENTS IN HEAP FORM

REMOVAL OF THE MAXIMUM ELEMENT IS DONE N TIMES, FOLLOWED BY HEAPIFY

INSERTION AND REMOVAL HAVE LOG N TIME COMPLEXITY SO DOING IT FOR N ELEMENTS MEANS:

THE AVERAGE CASE COMPLEXITY OF HEAP SORT IS O(NLOGN).

HEAP SORT

SORT

HEAP SORT IS NOT ADAPTIVE

IT'S NOT A STABLE SORT

IT DOES NOT NEED ADDITIONAL SPACE
- SPACE COMPLEXITY IS O(1)