```java
package Homework.HW_13;

/**
 * @author Mark Burrell
 * @version 1.0
 * @since 2018-12-06
 */
public class Heap {
    private int[] data;
    private int numberOfValues;

    /**
     * Constructor sets the maximum size of the heap size value is not saved
     * since it is data.length
     *
     * @param size Maximum capacity of the heap
     */
    public Heap(int size) {
        data = new int[size];
        numberOfValues = 0;
    }

    /**
     * Given the index of a node, return the index of its left child
     *
     * @param index Index of node
     * @return      Index of node's left child
     */
    private int indexOfLeftChild(int index) {
        // TODO: Replace with correctly calculated index
        return 2 * index + 1;
    }

    /**
     * Given the index of a node, return the index of its right child
     *
     * @param index Index of node
     * @return      Index of node's right child
     */
    private int indexOfRightChild(int index) {
        // TODO: Replace with correctly calculated index
        return 2*index + 2;
    }

    /**
     * Given the index of a node, return the index of its parent
     *
     * @param index Index of node
     * @return      Index of node's parent
     */
    private int indexOfParent(int index) {
        // TODO: Replace with correctly calculated index
        return (index-1) / 2;
    }
```

```java
/**
 * This method adds a new value to the heap, maintaining heap property
 *
 * @param value
 */
public boolean insert(int value) {
    if (numberOfValues < data.length) {
        // TODO: Write the 3 missing lines
        // Put the new value at the end of the array
        // Trickle up from newly filled location
        // Increment the number of values in the heap
         data[numberOfValues] = value;
         trickleUp(numberOfValues++);
        return true;
    } else
        return false;
}

/**
 * This method removes the highest value from the heap and returns it
 *
 * @return Highest value from heap that is now removed
 */
public int remove() {
    // Grab the last value, put it at root, trickle it down
    if (numberOfValues > 0) {
        // TODO: Write the 5 (4 plus a fix) missing lines
        // Keep a copy of the current root value
        // Decrement number of values
        // Move last value into root position
        // Trickle the new root value down
        // Return the saved root value
         int root = data[0];
         data[0] = data[--numberOfValues];
         trickleDown(0);
        return root;
    } else
        return 0;
}

/**
 * This method moves the value at index up to restore the heap property
 *
 * @param index Starting index for trickling up
 */
private void trickleUp(int index) {
    int bottomKey = data[index];
    int parent = indexOfParent(index);
    // As long as values in chain and bottomKey is bigger
    while (index > 0 && data[parent] < bottomKey) {
        // Move value down and try higher up
        data[index] = data[parent];
        index = parent;
        parent = indexOfParent(index);
    }
```

```
    // Done moving things down, bottomKey goes here now
    data[index] = bottomKey;
}

/**
 * This method moves value at index down to restore the heap property
 *
 * @param index Starting index for trickling down
 */
private void trickleDown(int index) {
    // See if value at index should move down
    int topKey = data[index];
    int largerChild;
    // While still children to consider
    while (index < (numberOfValues / 2)) {
        // Get the indexes of the two children
        int left = indexOfLeftChild(index);
        int right = indexOfRightChild(index);
        // Choose the child with the larger value
        if (right < numberOfValues && data[right] > data[left]) {
            largerChild = right;
        } else
            largerChild = left;
        if (topKey < data[largerChild]) {
            // Shift larger child data up and continue down
            data[index] = data[largerChild];
            index = largerChild;
        } else
            // Next down is smaller, so stop here
            break;
    }
    data[index] = topKey;
}

/**
 * This method prints the array indexes and values on one line
 */
public void display() {
    for (int i = 0; i < numberOfValues; i++)
        System.out.print(" " + i + ":" + data[i]);
    System.out.println();
}

/**
 * This method is the basic add-at-end version of an array insert
 *
 * @param value The value to be inserted
 * @return      True if there was space to insert value
 */
public boolean rawInsert(int value) {
    if (numberOfValues < data.length) {
        // TODO: Write the 2 missing lines
         data[numberOfValues++] = value;
        return true;
    } else
```

```
            return false;
    }

    /**
     * This method takes an unordered array and turns it into a valid heap
     * Since it only needs to work on half of the values it is quicker
     * to fill the array and then heapify than to use heap insert
     */
    public void heapify() {
        // Loop backwards over first half of array
        // No need to make a heap of leaf nodes on the bottom
        for (int i = (numberOfValues / 2) -1; i >=0; i--) {
             trickleDown(i);
        }
            // Left and right sub heaps below each value
            // are already built by the time you get to its index
            // Trickle value at i down to build its sub heap
            /* CODE LINE FOR LOOP BODY GOES HERE */
    }

    /*
     * The sort is just like selection sort (of max) only faster
     */
    public void sort() {
        // Save the current number of values (remove below will decrement it)
        int temp = numberOfValues;
        // Loop from the end backwards towards 0 (but don't include 0)
        for (int i = numberOfValues; i >= 0; i--) {
            // Put the high value removed in the next vacated end position
             data[numberOfValues] = remove();
        }
            /* CODE LINE FOR LOOP BODY GOES HERE */
        // Restore the original number of values
        numberOfValues = temp;
    }

}
```

```
Current heap array:
 0:93 1:69 2:92 3:30 4:21 5:81 6:82 7:0 8:1 9:2 10:9 11:30 12:53 13:0 14:22
Values removed in descending order:
 93 92 82 81 69 53 30 30 22 21 9 2 1 0 0
End of main part.

Unordered array:
 0:32 1:25 2:13 3:90 4:79 5:15 6:62 7:8 8:88 9:7 10:36 11:94 12:4 13:41 14:37
Heapified array:
 0:94 1:90 2:62 3:88 4:79 5:15 6:41 7:8 8:25 9:7 10:36 11:13 12:4 13:32 14:37
```

```
Sorted array:
 0:0 1:4 2:7 3:8 4:13 5:15 6:25 7:32 8:36 9:37 10:41 11:62 12:79 13:88 14:90
```