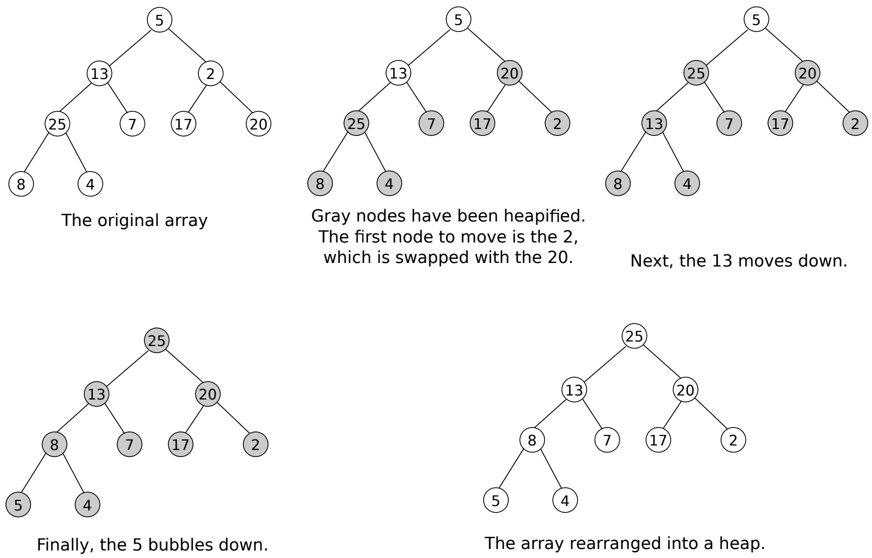
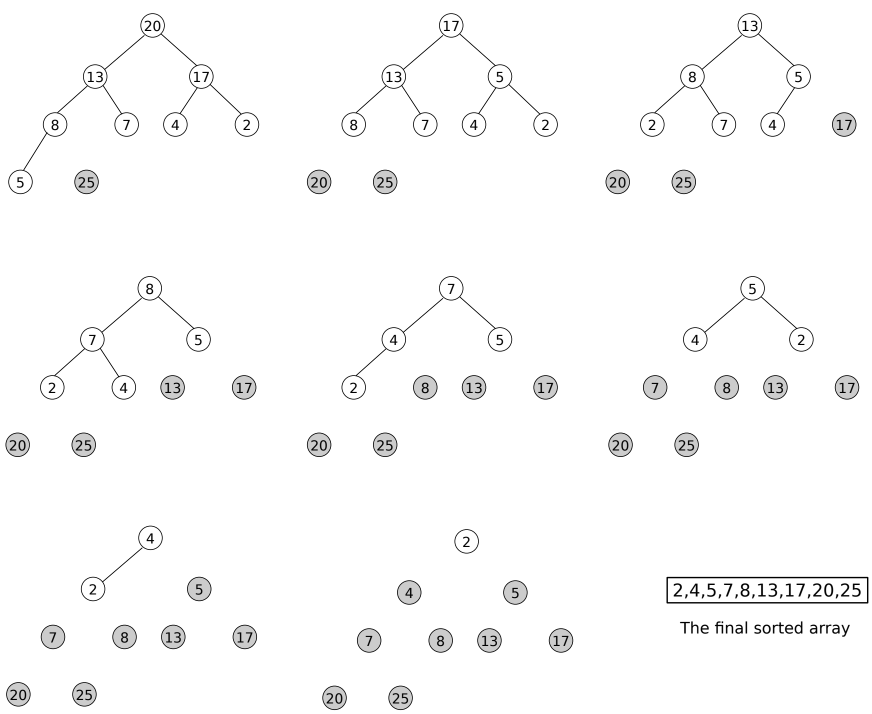
1. 6.4-1. The illustration that is given as a model only shows the second phase of heapsort, after the heap is already built, so it is not required to show the steps in building the heap. However, here are the steps. The gray nodes are the ones that are already heapified.



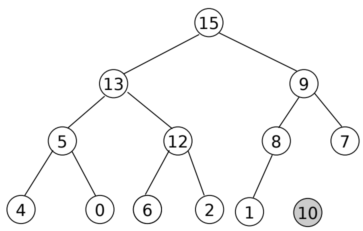
And here is the illustration that shows nodes being removed one-by-one from the heap, using *removeMax*():



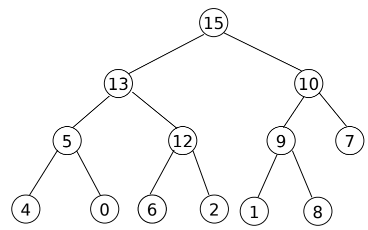
6.4-5. Initialization: Before the loop starts, $i = 0$, and the first subarray mentioned in the loop invariant is empty (an so is a heap containing the smallest 0 elements of A), and the second subarray is the entire array, which therefor contains the $(n - 0)$ largest elements of A. So the loop invariant is true at the start. Maintenance: An iteration of the loop removes the largest element from the first subarray and adds it to front of the second subarray. Since that element was in the first subarray, it is (according to the loop invariant) smaller than (or equal to) all the elements of the second subarray. So that subarray remains sorted, and since the element that was moved was the largest one in the heap, all of the elements in the heap are still less than or equal to all the elements in the second subarray. Termination: In the end, the first subarray is empty, the second contains every element, and the heap invariant says that those elements are in sorted order. So, heapsort has successfully sorted the array.

6.4-3. The running time will still be $\Theta(n * \log(n))$, in both cases. That is both the best case and the worst case running time for Heapsort, as long as all the elements in the array are different. (The heap can be built in $\Theta(n)$ time, but then calls to *removeMax*() always move a small element to the top of the heap, and that element has to bubble down the heap. On the other hand heapsort applied to an array in which all elements are the same only takes time $\Theta(n)$, since no element ever has to bubble up or down.)

6.5-2. The 10 is added at the bottom of the heap and bubbles up. Here is an illustration:



The original heap, with the new node that has to be added, shown in gray.



The 10 is attached to the tree, then it bubbles up, changing places with the 8 and the 9.

2. a) In one approach, the array is divided into three sections. `A[0..i-1]` contains only 1's, `A[i..j-1]` contains only 2's, and `A[j..n-1]` contains elements that have not yet been checked. In each iteration of the loop, we examine one new element, `A[j]`. If that element is a 2, nothing has to move. But if `A[j]` is a 1, it has to be moved into the first section of the array; we can do that by swapping `A[j]` with `A[i]`:

```
i = j = 0; // All elements start out in the third section
while (j < n) {
  if (A[j] == 1) {
    temp = A[j];
    A[j] = A[i];
    A[i] = temp;
    i++;
  }
  j++;
}
```

b) We now need 4 sections in the array: `A[0..i-1]` contains only ones, `A[i..j-1]` contains only 2's, `A[j..k-1]` contains elements that have not yet been checked, and `A[k..n-1]` contains only 3's. One iteration of the loop moves `A[j]` into appropriate section of the array (dending on whether it's a 1, a 2, or a 3). Note that in each iteration of the loop, either j is incremented or k is decremented, but not both:

```
i = j = 0; // All elements start out in the third section
k = n;
while (j < k) { // Done when 3rd section has zero elements
  if (A[j] == 1) { // Move it to first section
    temp = A[j];
    A[j] = A[i];
    A[i] = temp;
    i++;
    j++;
  }
  else if (A[j] == 2) {
    j++;
  }
  else if (A[j] == 3) { // Move it to fourth section
    k--;
    temp = A[j];
    A[j] = A[k];
    A[k] = temp;
  }
}
```

3. Answers will depend on the details of the programs. As an example, here are the times from my program for an array of one million items, with the just-in-time compiler disabled:

```
Array size is 1000000
    Time to sort with Arrays.sort:  607,875,959 nanoseconds.
    Time to sort with heapsort:  1,873,054,614 nanoseconds.
    Time to sort with funkyHeapsort:  1,565,402,095 nanoseconds.
    Time to sort with quicksort 772,393,976 nanoseconds.
```

My program actually runs two sets of experiments. The first gives the just-in-time compiler a chance to work (assuming that it's enabled). The second gives the results for the sort methods as optimized by the just-in-time compiler. Here are the results from the second set of tests in a program with the JIT compiler enabled. The JIT compiler has sped up the code by a factor of six or more:

```
Array size is 1000000
    Time to sort with Arrays.sort:  92,822,641 nanoseconds.
    Time to sort with heapsort:  194,145,195 nanoseconds.
    Time to sort with funkyHeapsort:  222,883,725 nanoseconds.
    Time to sort with quicksort 113,776,180 nanoseconds.
```

4. My programs are attached

```java
import java.util.Arrays;


/**
 * A program to measure sorting times for various sorting methogs.
 */
public class SortTimes {


    /**
     * Run the sorting methods on arrays of size 1000, 10000, 100000, and 1000000.
     * The experiments are run twice.  The first set gives the just-in-time compiler
     * a chance to work on the code (if the just-in-time compiler is enabled); the
     * second set of experiments will then give more accurate results.
     */
    public static void main(String[] args) {
        for (int size = 1000; size <= 1000000; size *= 10)
            runExperiment(size);
        System.out.println();
        System.out.println("-------------------------------------------------------");
        for (int size = 1000; size <= 1000000; size *= 10)
            runExperiment(size);
    }


    /**
     * Apply four sorting algorithms to the same random array, and report the times.
     */
    private static void runExperiment(int arraySize) {
        double[] A = new double[arraySize];
        for (int i = 0; i < arraySize; i++)
            A[i] = Math.random();
        double[] B = A.clone();
        double[] C = A.clone();
        double[] D = A.clone();
        long time;
        System.out.println();
        System.out.println("Array size is " + arraySize);
        time = builtInSort(A);
        System.out.printf("   Time to sort with Arrays.sort:   %,d nanoseconds.%n", time);
        time = heapsort(B);
        System.out.printf("   Time to sort with heapsort:      %,d nanoseconds.%n", time);
        time = funkyHeapSort(C);
        System.out.printf("   Time to sort with funkyHeapsort: %,d nanoseconds.%n", time);
        time = quicksort(D);
        System.out.printf("   Time to sort with quicksort      %,d nanoseconds.%n", time);

        // for testing, make sure my sorts give the same result as Arrays.sort...

//      System.out.println();
//      boolean ok;
//      ok = true;
//      for (int i = 0; i < arraySize; i++) {
//          if (B[i] != A[i]) {
//              System.out.println("heapsort failed at index " + i);
//              ok = false;
//              break;
//          }
//      }
//      if (ok)
//          System.out.println("heapsort worked");
//
//      ok = true;
//      for (int i = 0; i < arraySize; i++) {
//          if (C[i] != A[i]) {
//              System.out.println("funkyHeapSort failed at index " + i);
//              ok = false;
//              break;
//          }
//      }
//      if (ok)
//          System.out.println("funkyHeapsort worked");
//
//      ok = true;
//      for (int i = 0; i < arraySize; i++) {
//          if (D[i] != A[i]) {
//              System.out.println("quicksort failed at index " + i);
//              ok = false;
```

```
//              break;
//          }
//      }
//      if (ok)
//          System.out.println("quicksort worked");
    }


    /**
     * Sorts the array using Array.sort().
     * @param A the array to be sorted.
     * @return The number of nanoseconds that it took to do the sort.
     */
    private static long builtInSort(double[] A) {
        long start, end;
        start = System.nanoTime();
        Arrays.sort(A);
        end = System.nanoTime();
        return (end-start);
    }


    /**
     * Sorts the array using the heapsort algorithm.
     * @param A the array to be sorted.
     * @return The number of nanoseconds that it took to do the sort.
     */
    private static long heapsort(double[] A) {
        long start, end;
        start = System.nanoTime();
        for (int root = (A.length-1)/2; root >= 0; root--) {
            heapify(A,root,A.length);
        }
        for (int i = A.length-1; i > 0; i--) {
            double temp = A[i];
            A[i] = A[0];
            A[0] = temp;
            heapify(A,0,i);
        }
        end = System.nanoTime();
        return (end-start);
    }

    /* subroutine for use in heapsort */
    private static void heapify(double[] heap, int root, int size) {
        while (true) {
            if (2*root+1 >= size)
                break;
            int big = (2*root+2);
            if (big >= size || heap[2*root+1] > heap[big])
                big = (2*root+1);
            if (heap[big] < heap[root])
                break;
            double temp = heap[big];
            heap[big] = heap[root];
            heap[root] = temp;
            root = big;
        }
    }


    /**
     * Sorts the array using a MaxHeap.
     * @param A the array to be sorted.
     * @return The number of nanoseconds that it took to do the sort.
     */
    private static long funkyHeapSort(double[] A) {
        long start,end;
        start = System.nanoTime();
        MaxHeap heap = new MaxHeap();
        for (int i = 0; i < A.length; i++)
            heap.insert(A[i]);
        for (int j = A.length-1; j>= 0; j--)
            A[j] = heap.removeMax();
        end = System.nanoTime();
        return end-start;
    }

```

```java

    /**
     * The Quicksort partitioning algorithm.
     * @param A array in which a subarray is to be partitioned
     * @param low the start index of the subarray
     * @param high the end index of the subarray
     * @return the index of the pivot element after the partitioning
     */
    private static int partition(double[] A, int low, int high) {
        double pivot = A[low];
        int j = low;
        for (int i = low+1; i <= high; i++) {
            if (A[i] < pivot) {
                A[j] = A[i];
                A[i] = A[j+1];
                j++;
            }
        }
        A[j] = pivot;
        return j;
    }

    /**
     * Apply quicksort to the entire array A, and return how long it took
     * to do the sort, in nanoseconds.  This is just the most basic recursive
     * version of quicksort.
     */
    private static long quicksort( double[] A ) {
        long start, end;
        start = System.nanoTime();
        quicksort(A,0,A.length-1);
        end = System.nanoTime();
        return (end-start);
    }

    /**
     * Apply quicksort recursively to a subarray in A.
     */
    private static void quicksort(double[] A, int low, int high) {
        if (high > low) {
            int mid = partition(A,low,high);
            quicksort(A,low,mid-1);
            quicksort(A,mid+1,high);
        }
    }


}
```

```java

/**
 * An object of this class represents a max-heap of doubles.
 * The heap has operations insert(x) and removeMax(), and it
 * can be used as a max-priority-queue.
 */
public class MaxHeap {

    private double[] heap = new double[8];
    private int size = 0;

    /**
     * Returns the number of items on the heap.
     */
    public int size() {
        return size;
    }

    /**
     * Test whether the heap is empty.
     */
    public boolean isEmpty() {
        return size == 0;
    }

    /**
     * Remove and return the largest element from the heap.
     */
    public double removeMax( ) {
        if (size == 0)
            throw new IllegalStateException("Attempt to delete from an empty heap.");
        double max = heap[0];
        heap[0] = heap[size-1];
        size--;
        heapify(heap,0,size);
        return max;
    }

    /* subroutine for use in removeMax() */
    private void heapify(double[] heap, int root, int size) {
        while (true) {
            if (2*root+1 >= size)
                break;
            int big = (2*root+2);
            if (big >= size || heap[2*root+1] > heap[big])
                big = (2*root+1);
            if (heap[big] < heap[root])
                break;
            double temp = heap[big];
            heap[big] = heap[root];
            heap[root] = temp;
            root = big;
        }
    }

    /**
     * Adds an item to the heap.
     * @param x the number to be added.
     */
    public void insert( double x ) {
        if (size == heap.length) {
            double[] temp = new double[heap.length*2];
            for (int i = 0; i < heap.length; i++)
                temp[i] = heap[i];
            heap = temp;
        }
        int i = size;
        while (i > 0 && heap[(i-1)/2] < x) {
            heap[i] = heap[(i-1)/2];
            i = (i-1)/2;
        }
        heap[i] = x;
        size++;
    }

//  private void isHeap( ) {  // This was used during testing.
//      for (int i = 1; i < size; i++) {
//          if (heap[i] > heap[(i-1)/2])
```

```
//              throw new IllegalStateException("not a heap ");
//          }
//     }

}
```