```java
/**
 * The Heap ADT.  This is a max heap.
 *
 * As a student at Union College, I am part of a community that values
intellectual effort, curiosity and discovery. I understand that in order to
truly claim my educational and academic achievements, I am obligated to act
with academic integrity. Therefore, I affirm that I will carry out my
academic endeavors with full academic honesty, and I rely on my fellow
students to do the same.
 *
 * @author Blair Hagen
 * @version 6-2-16
 */
public class Heap
{
	private int[] itemArray;	//binary tree of ints in array form
	private int nodes;			//number of nodes in tree

	/**
	 * Builds a heap from an array of ints.
	 * The original array is not changed.
	 *
	 * @param items
	 *            an array of ints, which will be
	 *            interpreted as a binary tree.
	 */
	public Heap(int[] items)
	{
		itemArray = new int[items.length];
		nodes = items.length;

		for (int i = 0; i < items.length; i++) {
			itemArray[i] = items[i];
		}

		buildAHeap();
	}

	/**
	 * @return number of nodes in the heap.
	 */
	public int size()
	{
		return nodes;
	}

	/**
	 * Constructs a heap from the given binary tree (given as an array).
	 * Heapifies each internal node in reverse level-by-level order.
	 */
	public void buildAHeap()
	{
		int indexOfLastNode = nodes-1;
		for (int i = (indexOfLastNode-1)/2; i >= 0; i--) {
			heapify(i);
		}
	}
```

```
	/** string representation of a heap that looks (a little) like a tree
	 * @return string with one int on 1st line, two ints on 2nd line, four
ints on 3rd line, etc.
	 */
	public String toString()
	{
		String result = "\n";
		int lastNodeOnLevel = 0;
		int indexOfLastNode = nodes-1;

		for (int i = 0; i < indexOfLastNode; i++)
		{
			result += itemArray[i];
			if (i == lastNodeOnLevel) {
				result += "\n";
				lastNodeOnLevel = lastNodeOnLevel * 2 + 2;
			} else {
				result += " ";
			}
		}
		result += itemArray[indexOfLastNode];

		return result;
	}

	/**
	 * Turns a subtree into a heap, assuming that only the root of that
subtree
	 * violates the heap property.
	 *
	 * @param startingNode
	 *                the index of the node to start with.  This node
	 *                is the root of a subtree which must be turned into a
heap.
	 */
	public void heapify(int startingNode)
	{
		heapifyRec(startingNode);
	}

	/**
	 * Recursive method for turning subtree into a heap
	 *
	 * if head isnt leaf
	 *          if the largest children value is greater than the head
value
	 *                set head value equal to child value
	 *                set child value equal to head value
	 *                heapify children
	 *
	 * @param startingNode
	 * @param itemArray
	 * @return
	 */
	private void heapifyRec(int head)
	{
```

```
            if (!isLeaf(head))
            {
                  if (getVal(getSwapNode(head)) > getVal(head))
                  {
                        int headVal = getVal(head);
                        int swapIndex = getSwapNode(head);
                        int childVal = getVal(swapIndex);


                        itemArray[head] = childVal;
                        itemArray[swapIndex] = headVal;
                        heapify(swapIndex);
                  }
            }
      }

      /**
       * Removes the root from the heap, returning it.  The resulting array
is
       * then turned back into a heap.
       *
       * @return the largest value in the heap
       */
      public int deleteRoot()
      {
        int root = itemArray[0];

        itemArray[0] = itemArray[nodes - 1];
        nodes--;
        heapify(0);
        return root;

      }

      /**
       * Gets the correct node of the children of a node to swap
       * @param startingNode
       * @return index of node to swap
       */
      private int getSwapNode(int head)
      {
            if (hasRightChild(head) && getRightVal(head) > getLeftVal(head))
            {
                  return(head * 2 + 2);
            }
            else
            {
                  return(head * 2 + 1);
            }
      }

      /**
       * Gets current node
       * @param index of current node
       * @return value of current node
       */
      private int getVal(int index)
```

```
{
	return(itemArray[index]);
}

/**
 * Gets right child of node
 * Prerequisite: Must have a right child
 * @param index of subtree root
 * @return index of right child
 */
private int getRightVal(int index)
{
	return(itemArray[index * 2 + 2]);
}

/**
 * Checks if node has a right tree
 * @param index of subtree root
 * @return true if has node, false if not
 */
private boolean hasRightChild(int index)
{
	return(index*2+2<nodes);
}

/**
 * Gets left child of node
 * Prerequisite: Must have a left child
 * @param index of subtree root
 * @return index of right child
 */
private int getLeftVal(int index)
{
	return(itemArray[index * 2 + 1]);
}

/**
 * Checks if node has a left tree
 * @param index of subtree root
 * @return true if has node, false if not
 */
private boolean hasLeftChild(int index)
{
	return(index*2+1<nodes);
}

/**
 * Checks if a node is a leaf
 * @param index of node
 * @return true if it is a leaf, false if not
 */
private boolean isLeaf(int index)
{
	return !hasLeftChild(index);
}
```

```
}
```

```
/**
 * Tester for your Heap
 *
 * @author Blair Hagen and Chris Fernandes
 * @version 6/2/16
 *
 */
public class HeapTests {

      public static void main(String[] args)
    {
        Testing.startTests();
        Testing.setVerbose(true);

        shallowHeap();
        largeHeap();
        heapedHeap();

        sortUnique();
        sortedDuplicates();

        Testing.finishTests();
    }

      /**
       * given an array-based tree of integers, returns the contents of the
       * tree in the same sort-of-tree-like format that Heap's toString does
       *
       * @param someArray array-based tree of ints
       * @return contents of tree in level-by-level order with \n's to make
it tree-like
       */
      private static String treeify(int[] someArray) {
            String result = "\n";
            int lastNodeOnLevel = 0;

            for (int i = 0; i < someArray.length-1; i++)
            {
                  result += someArray[i];
                  if (i == lastNodeOnLevel) {
                        result += "\n";
                        lastNodeOnLevel = lastNodeOnLevel * 2 + 2;
                  } else {
                        result += " ";
                  }
            }
```

```
            result += someArray[someArray.length-1];
            return result;
      }

      private static void shallowHeap()
    {
      Testing.testSection("shallow heap test: subtree root swaps just
once,\n"
                  + "unbalanced tree, all internal nodes have two kids");

      int[] anArray = {11, 12, 5, 1, 23, 33, 9, 21, 14, 10, 4};
        Testing.printArray("before building heap:",anArray);

        Heap sample = new Heap(anArray);
        int[] answer = {33,23,11,21,12,5,9,1,14,10,4};

      Testing.assertEquals("after building heap",
                  treeify(answer), sample.toString());
    }

    private static void largeHeap()
    {
      Testing.testSection("large head test: subtree root swaps multiple
times,\n"
                  + "unbalanced tree, all internal nodes have two kids");

      int[] anArray = {12, 13, 18, 9, 70, 1, 56, 25, 45, 14, 20, 30, 6, 4};
      Testing.printArray("before building heap:", anArray);

      Heap sample = new Heap(anArray);
      int[] answer = {70, 45, 56, 25, 20, 30, 18, 12, 9, 14, 13, 1, 6, 4};

      Testing.assertEquals("after building heap", treeify(answer),
sample.toString());
    }

    private static void sortUnique()
    {
      Testing.testSection("sort test: random, no duplicates");

      int[] unsorted = {11, 12, 5, 1, 23, 33, 9, 21, 14, 10};
      Testing.printArray("before sorting",unsorted);
        int[] sorted = Sorter.priorityQueueSort(unsorted);
        int[] answer = {1, 5, 9, 10, 11, 12, 14, 21, 23, 33};

        Testing.assertEquals("after sorting", answer, sorted);
    }

    private static void heapedHeap()
    {
      Testing.testSection("already heapified heap");

      int[] unsorted = {70, 60, 50, 40, 30, 20, 10};
      Testing.printArray("before building heap:", unsorted);

      Heap sample = new Heap(unsorted);
      int[] answer = {70, 60, 50, 40, 30, 20, 10};
```

```
      Testing.assertEquals("after building heap", treeify(answer),
sample.toString());
    }

    private static void sortedDuplicates()
    {
      Testing.testSection("sorting a heap with duplicates sorted heap");

      int[] unsorted = {15, 15, 16, 18, 30, 15, 16, 30};
      Testing.printArray("before sorting", unsorted);
      int[] sorted = Sorter.priorityQueueSort(unsorted);
      int[] answer = {15, 15, 15, 16, 16, 18, 30, 30};

      Testing.assertEquals("after sorting", answer, sorted);
    }


}
```

1. The runtime would be O(n log(n)) because it uses the heapify method, and that method must touch every element in the array.

2. The runtime for sorting an unsorted linked list, the runtime would be O(n) because the entire linked list has to be searched
every time the method is called.