```java
package edu.neu.coe.info6205.sort.linearithmic;

import edu.neu.coe.info6205.sort.Helper;
import edu.neu.coe.info6205.sort.InstrumentedHelper;
import edu.neu.coe.info6205.util.Benchmark;
import edu.neu.coe.info6205.util.Benchmark_Timer;
import edu.neu.coe.info6205.util.Config;

import java.util.ArrayList;
import java.util.List;

public class QuickSort_DualPivot<X extends Comparable<X>> extends QuickSort<X> {

    public static final String DESCRIPTION = "QuickSort dual pivot";

    public QuickSort_DualPivot(String description, int N, Config config) {
        super(description, N, config);
        setPartitioner(createPartitioner());
    }

    /**
     * Constructor for QuickSort_3way
     *
     * @param helper an explicit instance of Helper to be used.
     */
    public QuickSort_DualPivot(Helper<X> helper) {
        super(helper);
        setPartitioner(createPartitioner());
    }

    /**
     * Constructor for QuickSort_3way
     *
     * @param N      the number elements we expect to sort.
     * @param config the configuration.
     */
    public QuickSort_DualPivot(int N, Config config) {
        this(DESCRIPTION, N, config);
    }

    @Override
    public Partitioner<X> createPartitioner() {
        return new Partitioner_DualPivot(getHelper());
    }

    public class Partitioner_DualPivot implements Partitioner<X> {

        public Partitioner_DualPivot(Helper<X> helper) {
            this.helper = helper;
        }

        /**
         * Method to partition the given partition into smaller partitions.
         *
         * @param partition the partition to divide up.
         * @return an array of partitions, whose length depends on the sorting method being used.
         */
        public List<Partition<X>> partition(Partition<X> partition) {
            final X[] xs = partition.xs;
            final int lo = partition.from;
            final int hi = partition.to - 1;
            helper.swapConditional(xs, lo, hi);
            int lt = lo + 1;
            int gt = hi - 1;
            int i = lt;
            X v1 = xs[lo];
            X v2 = xs[hi];
            // NOTE: we are trying to avoid
```

```java
checking on instrumented for every time in the inner loop for performance reasons (probably a silly idea).
            // NOTE: if we were using Scala, it would be easy to set up a comparer function and a swapper function. With java, it's possible but much messier.
            if (helper.instrumented()) {
                helper.incrementHits(2); // XXX these account for v1 and v2.
                while (i <= gt) {
                    if (helper.compare(xs, i, v1) < 0) helper.swap(xs, lt++, i++);
                    else if (helper.compare(xs, i, v2) > 0) helper.swap(xs, i, gt--);
                    else i++;
                }
                helper.swap(xs, lo, --lt);
                helper.swap(xs, hi, ++gt);
            } else {
                while (i <= gt) {
                    X x = xs[i];
                    if (x.compareTo(v1) < 0) swap(xs, lt++, i++);
                    else if (x.compareTo(v2) > 0) swap(xs, i, gt--);
                    else i++;
                }
                swap(xs, lo, --lt);
                swap(xs, hi, ++gt);
            }

            List<Partition<X>> partitions = new ArrayList<>();
            partitions.add(new Partition<>(xs, lo, lt));
            partitions.add(new Partition<>(xs, lt + 1, gt));
            partitions.add(new Partition<>(xs, gt + 1, hi + 1));
            return partitions;
```

```java
        }

        // CONSIDER invoke swap in BaseHelper.
        private void swap(X[] ys, int i, int j) {
            X temp = ys[i];
            ys[i] = ys[j];
            ys[j] = temp;
        }

        private final Helper<X> helper;
    }

    public static void main (String[] args) {
        int N = 1000;

        while(N<=64000) {

            InstrumentedHelper<Integer> instrumentedHelper = new InstrumentedHelper<>("QuickSort_DualPivot", Config.setupConfig("true", "0", "0", "", ""));
            QuickSort_DualPivot<Integer> s = new QuickSort_DualPivot<>(instrumentedHelper);

            int j = N;
            s.init(j);

            Integer[] temp = instrumentedHelper.random(Integer.class, r -> r.nextInt(j));

            Partitioner<Integer> partitioner = s.createPartitioner();
            List<Partition<Integer>> partitionList = partitioner.partition(new Partition<>(temp, 0, temp.length));
            Partition<Integer> p1 = partitionList.get(0);
            Partition<Integer> p2 = partitionList.get(1);
            Partition<Integer> p3 = partitionList.get(2);
```

```

                Benchmark<Boolean> benchmark1 = new Benchmark_Timer<>("Sorting", b -> s.sort(temp, 0, p1.to, 0));
                double b1 = benchmark1.run(true, 20);
                Benchmark<Boolean> benchmark2 = new Benchmark_Timer<>("Sorting", b -> s.sort(temp, p2.from, p2.to, 0));
                double b2 = benchmark2.run(true, 20);
                Benchmark<Boolean> benchmark3 = new Benchmark_Timer<>("Sorting", b -> s.sort(temp, p3.from, j, 0));
                double b3 = benchmark3.run(true, 20);

                long nCompares = instrumentedHelper.getCompares();
                int nSwaps = instrumentedHelper.getSwaps();
                int nHits = instrumentedHelper.getHits();

                double nTime = (b1 + b2 + b3);

                System.out.println("When array size is : " + j);
                System.out.println("Compares: " + nCompares);
                System.out.println("Swaps: " + nSwaps);
                System.out.println("Hits: " + nHits);
                System.out.println("Time: " + nTime);

                System.out.println("\nFor referencs:\t" + j + "\t" + nCompares + "\t" + nSwaps + "\t" + nHits + "\t" + nTime + "\n");

                N *= 2;
            }
        }

}
```