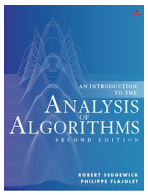

- Web Resources
- FAQ
- Data
- Code
- Errata
- Cheatsheet
- References
- Online Course
- Lecture Slides
- Programming Assignments

Search

Algorithms and Data Structures Cheatsheet

We summarize the performance characteristics of classic algorithms and data structures for sorting, priority queues, symbol tables, and graph processing.

Sorting.

The table below summarizes the number of compares for a variety of sorting algorithms, as implemented in this textbook. It includes leading constants but ignores lower-order terms.

ALGORITHM	CODE	IN PLACE	STABLE	BEST	AVERAGE	WORST	REMARKS
selection sort	Selection.java	✓		$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	n exchanges; quadratic in best case
insertion sort	Insertion.java	✓	✓	n	$\frac{1}{4} n^2$	$\frac{1}{2} n^2$	use for small or partially-sorted arrays
bubble sort	Bubble.java	✓	✓	n	$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	rarely useful; use insertion sort instead
shellsort	Shell.java	✓		$n \log_3 n$	unknown	$c\, n^{3/2}$	tight code; subquadratic
mergesort	Merge.java		✓	$\frac{1}{2} n \lg n$	$n \lg n$	$n \lg n$	$n \log n$ guarantee; stable
quicksort	Quick.java	✓		$n \lg n$	$2 n \ln n$	$\frac{1}{2} n^2$	$n \log n$ probabilistic guarantee; fastest in practice

heapsort	Heap.java	✓		n †	$2\,n \lg n$	$2\,n \lg n$	$n \log n$ guarantee; in place

† $n \lg n$ if all keys are distinct

Priority queues.

The table below summarizes the order of growth of the running time of operations for a variety of priority queues, as implemented in this textbook. It ignores leading constants and lower-order terms. Except as noted, all running times are worst-case running times.

DATA STRUCTURE	CODE	INSERT	DEL-MIN	MIN	DEC-KEY	DELETE	MERGE
array	BruteIndexMinPQ.java	1	n	n	1	1	n
binary heap	IndexMinPQ.java	$\log n$	$\log n$	1	$\log n$	$\log n$	n
***d*-way heap**	IndexMultiwayMinPQ.java	$\log_d n$	$d \log_d n$	1	$\log_d n$	$d \log_d n$	n
binomial heap	IndexBinomialMinPQ.java	1	$\log n$	1	$\log n$	$\log n$	$\log n$
Fibonacci heap	IndexFibonacciMinPQ.java	1	$\log n$ †	1	1 †	$\log n$ †	$\log n$

† amortized guarantee

Symbol tables.

The table below summarizes the order of growth of the running time of operations for a variety of symbol tables, as implemented in this textbook. It ignores leading constants and lower-order terms.

		worst case			average case		
DATA STRUCTURE	CODE	SEARCH	INSERT	DELETE	SEARCH	INSERT	DELETE
sequential search (in an unordered array)	SequentialSearchST.java	n	n	n	n	n	n
binary search (in a sorted array)	BinarySearchST.java	$\log n$	n	n	$\log n$	n	n
binary search tree (unbalanced)	BST.java	n	n	n	$\log n$	$\log n$	sqrt(n)
red-black BST (left-leaning)	RedBlackBST.java	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$
	SeparateChainingHashST.java	n	n	n	1 †	1 †	1 †

hash table (separate-chaining)							
hash table (linear-probing)	LinearProbingHashST.java	n	n	n	1 †	1 †	1 †

† uniform hashing assumption

Graph processing.

The table below summarizes the order of growth of the worst-case running time and memory usage (beyond the memory for the graph itself) for a variety of graph-processing problems, as implemented in this textbook. It ignores leading constants and lower-order terms. All running times are worst-case running times.

PROBLEM	ALGORITHM	CODE	TIME	SPACE
path	DFS	DepthFirstPaths.java	$E + V$	V
cycle	DFS	Cycle.java	$E + V$	V
directed cycle	DFS	DirectedCycle.java	$E + V$	V
topological sort	DFS	Topological.java	$E + V$	V
bipartiteness / odd cycle	DFS	Bipartite.java	$E + V$	V
connected components	DFS	CC.java	$E + V$	V
strong components	Kosaraju–Sharir	KosarajuSharirSCC.java	$E + V$	V
Eulerian cycle	DFS	EulerianCycle.java	$E + V$	$E + V$
directed Eulerian cycle	DFS	DirectedEulerianCycle.java	$E + V$	V
transitive closure	DFS	TransitiveClosure.java	$V (E + V)$	V^2
minimum spanning tree	Kruskal	KruskalMST.java	$E \log E$	$E + V$
minimum spanning tree	Prim	PrimMST.java	$E \log V$	V
minimum spanning tree	Boruvka	BoruvkaMST.java	$E \log V$	V
shortest paths (unit weights)	BFS	BreadthFirstPaths.java	$E + V$	V
shortest paths (nonnegative weights)	Dijkstra	DijkstraSP.java	$E \log V$	V
shortest paths (negative weights)	Bellman–Ford	BellmanFordSP.java	$V (V + E)$	V
all-pairs shortest paths	Floyd–Warshall	FloydWarshall.java	V^3	V^2
maxflow–mincut	Ford–Fulkerson	FordFulkerson.java	$E V (E + V)$	V
bipartite matching	Hopcroft–Karp	HopcroftKarp.java	$V^{1/2} (E + V)$	V
assignment problem	successive shortest paths	AssignmentProblem.java	$n^3 \log n$	n^2

Last modified on February 21, 2017.

Selection.java

Below is the syntax highlighted version of Selection.java from §2.1 Elementary Sorts.

```
/******************************************************************************
 *  Compilation:  javac Selection.java
 *  Execution:    java  Selection < input.txt
 *  Dependencies: StdOut.java StdIn.java
 *  Data files:   http://algs4.cs.princeton.edu/21elementary/tiny.txt
 *                http://algs4.cs.princeton.edu/21elementary/words3.txt
 *
 *  Sorts a sequence of strings from standard input using selection sort.
 *
 *  % more tiny.txt
 *  S O R T E X A M P L E
 *
 *  % java Selection < tiny.txt
 *  A E E L M O P R S T X                 [ one string per line ]
 *
 *  % more words3.txt
 *  bed bug dad yes zoo ... all bad yet
 *
 *  % java Selection < words3.txt
 *  all bad bed bug dad ... yes yet zoo    [ one string per line ]
 *
 ******************************************************************************/

import java.util.Comparator;

/**
 *  The {@code Selection} class provides static methods for sorting an
 *  array using selection sort.
 *  <p>
 *  For additional documentation, see <a href="http://algs4.cs.princeton.edu/21elementary">Section 2.1</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class Selection {

    // This class should not be instantiated.
    private Selection() { }

    /**
     * Rearranges the array in ascending order, using the natural order.
     * @param a the array to be sorted
     */
    public static void sort(Comparable[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            int min = i;
            for (int j = i+1; j < n; j++) {
                if (less(a[j], a[min])) min = j;
            }
            exch(a, i, min);
            assert isSorted(a, 0, i);
        }
        assert isSorted(a);
    }

    /**
     * Rearranges the array in ascending order, using a comparator.
     * @param a the array
     * @param comparator the comparator specifying the order
     */
    public static void sort(Object[] a, Comparator comparator) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            int min = i;
            for (int j = i+1; j < n; j++) {
                if (less(comparator, a[j], a[min])) min = j;
            }
```

```
            exch(a, i, min);
            assert isSorted(a, comparator, 0, i);
        }
        assert isSorted(a, comparator);
    }


   /***************************************************************************
    *  Helper sorting functions.
    ***************************************************************************/

    // is v < w ?
    private static boolean less(Comparable v, Comparable w) {
        return v.compareTo(w) < 0;
    }

    // is v < w ?
    private static boolean less(Comparator comparator, Object v, Object w) {
        return comparator.compare(v, w) < 0;
    }


    // exchange a[i] and a[j]
    private static void exch(Object[] a, int i, int j) {
        Object swap = a[i];
        a[i] = a[j];
        a[j] = swap;
    }


   /***************************************************************************
    *  Check if array is sorted - useful for debugging.
    ***************************************************************************/

    // is the array a[] sorted?
    private static boolean isSorted(Comparable[] a) {
        return isSorted(a, 0, a.length - 1);
    }

    // is the array sorted from a[lo] to a[hi]
    private static boolean isSorted(Comparable[] a, int lo, int hi) {
        for (int i = lo + 1; i <= hi; i++)
            if (less(a[i], a[i-1])) return false;
        return true;
    }

    // is the array a[] sorted?
    private static boolean isSorted(Object[] a, Comparator comparator) {
        return isSorted(a, comparator, 0, a.length - 1);
    }

    // is the array sorted from a[lo] to a[hi]
    private static boolean isSorted(Object[] a, Comparator comparator, int lo, int hi) {
        for (int i = lo + 1; i <= hi; i++)
            if (less(comparator, a[i], a[i-1])) return false;
        return true;
    }



    // print array to standard output
    private static void show(Comparable[] a) {
        for (int i = 0; i < a.length; i++) {
            StdOut.println(a[i]);
        }
    }

    /**
     * Reads in a sequence of strings from standard input; selection sorts them;
     * and prints them to standard output in ascending order.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        String[] a = StdIn.readAllStrings();
        Selection.sort(a);
        show(a);
    }
}
```

Last updated: Sat Mar 25 05:27:14 EDT 2017.

Insertion.java

Below is the syntax highlighted version of Insertion.java from §2.1 Elementary Sorts.

```
/******************************************************************************
 *  Compilation:  javac Insertion.java
 *  Execution:    java Insertion < input.txt
 *  Dependencies: StdOut.java StdIn.java
 *  Data files:   http://algs4.cs.princeton.edu/21elementary/tiny.txt
 *                http://algs4.cs.princeton.edu/21elementary/words3.txt
 *
 *  Sorts a sequence of strings from standard input using insertion sort.
 *
 *  % more tiny.txt
 *  S O R T E X A M P L E
 *
 *  % java Insertion < tiny.txt
 *  A E E L M O P R S T X                 [ one string per line ]
 *
 *  % more words3.txt
 *  bed bug dad yes zoo ... all bad yet
 *
 *  % java Insertion < words3.txt
 *  all bad bed bug dad ... yes yet zoo   [ one string per line ]
 *
 ******************************************************************************/

import java.util.Comparator;

/**
 *  The {@code Insertion} class provides static methods for sorting an
 *  array using insertion sort.
 *  <p>
 *  This implementation makes ~ 1/2 n^2 compares and exchanges in
 *  the worst case, so it is not suitable for sorting large arbitrary arrays.
 *  More precisely, the number of exchanges is exactly equal to the number
 *  of inversions. So, for example, it sorts a partially-sorted array
 *  in linear time.
 *  <p>
 *  The sorting algorithm is stable and uses O(1) extra memory.
 *  <p>
 *  See <a href="http://algs4.cs.princeton.edu/21elementary/InsertionPedantic.java.html">InsertionPedantic.java</a>
 *  for a version that eliminates the compiler warning.
 *  <p>
 *  For additional documentation, see <a href="http://algs4.cs.princeton.edu/21elementary">Section 2.1</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class Insertion {

    // This class should not be instantiated.
    private Insertion() { }

    /**
     * Rearranges the array in ascending order, using the natural order.
     * @param a the array to be sorted
     */
    public static void sort(Comparable[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            for (int j = i; j > 0 && less(a[j], a[j-1]); j--) {
                exch(a, j, j-1);
            }
            assert isSorted(a, 0, i);
        }
        assert isSorted(a);
    }

    /**
     * Rearranges the subarray a[lo..hi) in ascending order, using the natural order.
     * @param a the array to be sorted
     * @param lo left endpoint (inclusive)
     * @param hi right endpoint (exclusive)
     */
    public static void sort(Comparable[] a, int lo, int hi) {
```

```
        for (int i = lo; i < hi; i++) {
            for (int j = i; j > lo && less(a[j], a[j-1]); j--) {
                exch(a, j, j-1);
            }
        }
        assert isSorted(a, lo, hi);
    }

    /**
     * Rearranges the array in ascending order, using a comparator.
     * @param a the array
     * @param comparator the comparator specifying the order
     */
    public static void sort(Object[] a, Comparator comparator) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            for (int j = i; j > 0 && less(a[j], a[j-1], comparator); j--) {
                exch(a, j, j-1);
            }
            assert isSorted(a, 0, i, comparator);
        }
        assert isSorted(a, comparator);
    }

    /**
     * Rearranges the subarray a[lo..hi) in ascending order, using a comparator.
     * @param a the array
     * @param lo left endpoint (inclusive)
     * @param hi right endpoint (exclusive)
     * @param comparator the comparator specifying the order
     */
    public static void sort(Object[] a, int lo, int hi, Comparator comparator) {
        for (int i = lo; i < hi; i++) {
            for (int j = i; j > lo && less(a[j], a[j-1], comparator); j--) {
                exch(a, j, j-1);
            }
        }
        assert isSorted(a, lo, hi, comparator);
    }


    // return a permutation that gives the elements in a[] in ascending order
    // do not change the original array a[]
    /**
     * Returns a permutation that gives the elements in the array in ascending order.
     * @param a the array
     * @return a permutation {@code p[]} such that {@code a[p[0]]}, {@code a[p[1]]},
     *    ..., {@code a[p[n-1]]} are in ascending order
     */
    public static int[] indexSort(Comparable[] a) {
        int n = a.length;
        int[] index = new int[n];
        for (int i = 0; i < n; i++)
            index[i] = i;

        for (int i = 0; i < n; i++)
            for (int j = i; j > 0 && less(a[index[j]], a[index[j-1]]); j--)
                exch(index, j, j-1);

        return index;
    }

/***************************************************************************
     *  Helper sorting functions.
     ***************************************************************************/

    // is v < w ?
    private static boolean less(Comparable v, Comparable w) {
        return v.compareTo(w) < 0;
    }

    // is v < w ?
    private static boolean less(Object v, Object w, Comparator comparator) {
        return comparator.compare(v, w) < 0;
    }

    // exchange a[i] and a[j]
    private static void exch(Object[] a, int i, int j) {
        Object swap = a[i];
        a[i] = a[j];
        a[j] = swap;
    }
```

```
    // exchange a[i] and a[j]  (for indirect sort)
    private static void exch(int[] a, int i, int j) {
        int swap = a[i];
        a[i] = a[j];
        a[j] = swap;
    }

   /***************************************************************************
    *  Check if array is sorted - useful for debugging.
    ***************************************************************************/
    private static boolean isSorted(Comparable[] a) {
        return isSorted(a, 0, a.length);
    }

    // is the array a[lo..hi) sorted
    private static boolean isSorted(Comparable[] a, int lo, int hi) {
        for (int i = lo+1; i < hi; i++)
            if (less(a[i], a[i-1])) return false;
        return true;
    }

    private static boolean isSorted(Object[] a, Comparator comparator) {
        return isSorted(a, 0, a.length, comparator);
    }

    // is the array a[lo..hi) sorted
    private static boolean isSorted(Object[] a, int lo, int hi, Comparator comparator) {
        for (int i = lo+1; i < hi; i++)
            if (less(a[i], a[i-1], comparator)) return false;
        return true;
    }

   // print array to standard output
    private static void show(Comparable[] a) {
        for (int i = 0; i < a.length; i++) {
            StdOut.println(a[i]);
        }
    }

    /**
     * Reads in a sequence of strings from standard input; insertion sorts them;
     * and prints them to standard output in ascending order.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        String[] a = StdIn.readAllStrings();
        Insertion.sort(a);
        show(a);
    }
}
```


Last updated: Sat Apr 1 05:15:15 EDT 2017.

Bubble.java

Below is the syntax highlighted version of Bubble.java from §2.1 Elementary Sorts.

```
/******************************************************************************
 *  Compilation:  javac Bubble.java
 *  Execution:    java  Bubble N
 *  Dependencies: StdOut.java
 *
 *  Read strings from standard input and bubblesort them.
 *
 ******************************************************************************/

/**
 *  The {@code Bubble} class provides static methods for sorting an
 *  array using bubble sort.
 *  <p>
 *  This implementation makes ~ 1/2 n^2 compares and exchanges in
 *  the worst case, so it is not suitable for sorting large arbitrary arrays.
 *  Bubble sort is seldom useful because it is substantially slower than
 *  insertion sort on most inputs. The one class of inputs where bubble sort
 *  might be faster than insertion sort is arrays for which only
 *  a few passes of bubble sort are needed. This includes sorted arrays,
 *  but it does not include most partially-sorted arrays; for example,
 *  bubble sort takes quadratic time to sort arrays of the form
 *  [n, 1, 2, 3, 4, ..., n-1], whereas insertion sort takes linear time on
 *  such inputs.
 *  <p>
 *  The sorting algorithm is stable and uses O(1) extra memory.
 *  <p>
 *  For additional documentation,
 *  see <a href="http://algs4.cs.princeton.edu/21elementary">Section 2.1</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class Bubble {

   // This class should not be instantiated.
    private Bubble() { }

    /**
     * Rearranges the array in ascending order, using the natural order.
     * @param a the array to be sorted
     */
    public static <Key extends Comparable<Key>> void sort(Key[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            int exchanges = 0;
            for (int j = n-1; j > i; j--) {
                if (less(a[j], a[j-1])) {
                    exch(a, j, j-1);
                    exchanges++;
```

```
                }
            }
            if (exchanges == 0) break;
        }
    }

    // is v < w ?
    private static <Key extends Comparable<Key>> boolean less(Key v, Key w) {
        return v.compareTo(w) < 0;
    }

    // exchange a[i] and a[j]
    private static <Key extends Comparable<Key>> void exch(Key[] a, int i, int j) {
        Key swap = a[i];
        a[i] = a[j];
        a[j] = swap;
    }

   // print array to standard output
    private static void show(Comparable[] a) {
        for (int i = 0; i < a.length; i++) {
            StdOut.println(a[i]);
        }
    }

    /**
     * Reads in a sequence of strings from standard input; bubble sorts them;
     * and prints them to standard output in ascending order.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        String[] a = StdIn.readAllStrings();
        Bubble.sort(a);
        show(a);
    }
}
```


Last updated: Tue Aug 30 10:09:18 EDT 2016.

Shell.java

Below is the syntax highlighted version of Shell.java from §2.1 Elementary Sorts.

```
/******************************************************************************
 *  Compilation:  javac Shell.java
 *  Execution:    java Shell < input.txt
 *  Dependencies: StdOut.java StdIn.java
 *  Data files:   http://algs4.cs.princeton.edu/21elementary/tiny.txt
 *                http://algs4.cs.princeton.edu/21elementary/words3.txt
 *
 *  Sorts a sequence of strings from standard input using shellsort.
 *
 *  Uses increment sequence proposed by Sedgewick and Incerpi.
 *  The nth element of the sequence is the smallest integer >= 2.5^n
 *  that is relatively prime to all previous terms in the sequence.
 *  For example, incs[4] is 41 because 2.5^4 = 39.0625 and 41 is
 *  the next integer that is relatively prime to 3, 7, and 16.
 *
 *  % more tiny.txt
 *  S O R T E X A M P L E
 *
 *  % java Shell < tiny.txt
 *  A E E L M O P R S T X                 [ one string per line ]
 *
 *  % more words3.txt
 *  bed bug dad yes zoo ... all bad yet
 *
 *  % java Shell < words3.txt
 *  all bad bed bug dad ... yes yet zoo    [ one string per line ]
 *
 *
 ******************************************************************************/

/**
 *  The {@code Shell} class provides static methods for sorting an
 *  array using Shellsort with Knuth's increment sequence (1, 4, 13, 40, ...).
 *  <p>
 *  For additional documentation, see <a href="http://algs4.cs.princeton.edu/21elementary">Section 2.1</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class Shell {

    // This class should not be instantiated.
    private Shell() { }

    /**
     * Rearranges the array in ascending order, using the natural order.
     * @param a the array to be sorted
     */
    public static void sort(Comparable[] a) {
        int n = a.length;

        // 3x+1 increment sequence:  1, 4, 13, 40, 121, 364, 1093, ...
        int h = 1;
        while (h < n/3) h = 3*h + 1;

        while (h >= 1) {
            // h-sort the array
            for (int i = h; i < n; i++) {
                for (int j = i; j >= h && less(a[j], a[j-h]); j -= h) {
                    exch(a, j, j-h);
                }
            }
            assert isHsorted(a, h);
            h /= 3;
        }
        assert isSorted(a);
    }
```

```
   /***************************************************************************
    *  Helper sorting functions.
    ***************************************************************************/

    // is v < w ?
    private static boolean less(Comparable v, Comparable w) {
        return v.compareTo(w) < 0;
    }

    // exchange a[i] and a[j]
    private static void exch(Object[] a, int i, int j) {
        Object swap = a[i];
        a[i] = a[j];
        a[j] = swap;
    }


   /***************************************************************************
    *  Check if array is sorted - useful for debugging.
    ***************************************************************************/
    private static boolean isSorted(Comparable[] a) {
        for (int i = 1; i < a.length; i++)
            if (less(a[i], a[i-1])) return false;
        return true;
    }

    // is the array h-sorted?
    private static boolean isHsorted(Comparable[] a, int h) {
        for (int i = h; i < a.length; i++)
            if (less(a[i], a[i-h])) return false;
        return true;
    }

    // print array to standard output
    private static void show(Comparable[] a) {
        for (int i = 0; i < a.length; i++) {
            StdOut.println(a[i]);
        }
    }

    /**
     * Reads in a sequence of strings from standard input; Shellsorts them;
     * and prints them to standard output in ascending order.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        String[] a = StdIn.readAllStrings();
        Shell.sort(a);
        show(a);
    }

}
```


Last updated: Sat Mar 25 05:27:14 EDT 2017.

Merge.java

Below is the syntax highlighted version of Merge.java from §2.2 Mergesort.

```
/******************************************************************************
 *  Compilation:  javac Merge.java
 *  Execution:    java Merge < input.txt
 *  Dependencies: StdOut.java StdIn.java
 *  Data files:   http://algs4.cs.princeton.edu/22mergesort/tiny.txt
 *                http://algs4.cs.princeton.edu/22mergesort/words3.txt
 *
 *  Sorts a sequence of strings from standard input using mergesort.
 *
 *  % more tiny.txt
 *  S O R T E X A M P L E
 *
 *  % java Merge < tiny.txt
 *  A E E L M O P R S T X                 [ one string per line ]
 *
 *  % more words3.txt
 *  bed bug dad yes zoo ... all bad yet
 *
 *  % java Merge < words3.txt
 *  all bad bed bug dad ... yes yet zoo    [ one string per line ]
 *
 ******************************************************************************/

/**
 *  The {@code Merge} class provides static methods for sorting an
 *  array using mergesort.
 *  <p>
 *  For additional documentation, see <a href="http://algs4.cs.princeton.edu/22mergesort">Section 2.2</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *  For an optimized version, see {@link MergeX}.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class Merge {

    // This class should not be instantiated.
    private Merge() { }

    // stably merge a[lo .. mid] with a[mid+1 ..hi] using aux[lo .. hi]
    private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
        // precondition: a[lo .. mid] and a[mid+1 .. hi] are sorted subarrays
        assert isSorted(a, lo, mid);
        assert isSorted(a, mid+1, hi);

        // copy to aux[]
        for (int k = lo; k <= hi; k++) {
            aux[k] = a[k];
        }

        // merge back to a[]
        int i = lo, j = mid+1;
        for (int k = lo; k <= hi; k++) {
            if      (i > mid)              a[k] = aux[j++];
            else if (j > hi)               a[k] = aux[i++];
            else if (less(aux[j], aux[i])) a[k] = aux[j++];
            else                           a[k] = aux[i++];
        }

        // postcondition: a[lo .. hi] is sorted
        assert isSorted(a, lo, hi);
    }

    // mergesort a[lo..hi] using auxiliary array aux[lo..hi]
    private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
        if (hi <= lo) return;
        int mid = lo + (hi - lo) / 2;
        sort(a, aux, lo, mid);
```

```
        sort(a, aux, mid + 1, hi);
        merge(a, aux, lo, mid, hi);
    }

    /**
     * Rearranges the array in ascending order, using the natural order.
     * @param a the array to be sorted
     */
    public static void sort(Comparable[] a) {
        Comparable[] aux = new Comparable[a.length];
        sort(a, aux, 0, a.length-1);
        assert isSorted(a);
    }


   /***************************************************************************
    *  Helper sorting function.
    ***************************************************************************/

    // is v < w ?
    private static boolean less(Comparable v, Comparable w) {
        return v.compareTo(w) < 0;
    }

   /***************************************************************************
    *  Check if array is sorted - useful for debugging.
    ***************************************************************************/
    private static boolean isSorted(Comparable[] a) {
        return isSorted(a, 0, a.length - 1);
    }

    private static boolean isSorted(Comparable[] a, int lo, int hi) {
        for (int i = lo + 1; i <= hi; i++)
            if (less(a[i], a[i-1])) return false;
        return true;
    }


   /***************************************************************************
    *  Index mergesort.
    ***************************************************************************/
    // stably merge a[lo .. mid] with a[mid+1 .. hi] using aux[lo .. hi]
    private static void merge(Comparable[] a, int[] index, int[] aux, int lo, int mid, int hi) {

        // copy to aux[]
        for (int k = lo; k <= hi; k++) {
            aux[k] = index[k];
        }

        // merge back to a[]
        int i = lo, j = mid+1;
        for (int k = lo; k <= hi; k++) {
            if      (i > mid)                    index[k] = aux[j++];
            else if (j > hi)                     index[k] = aux[i++];
            else if (less(a[aux[j]], a[aux[i]])) index[k] = aux[j++];
            else                                 index[k] = aux[i++];
        }
    }

    /**
     * Returns a permutation that gives the elements in the array in ascending order.
     * @param a the array
     * @return a permutation {@code p[]} such that {@code a[p[0]]}, {@code a[p[1]]},
     *    ..., {@code a[p[N-1]]} are in ascending order
     */
    public static int[] indexSort(Comparable[] a) {
        int n = a.length;
        int[] index = new int[n];
        for (int i = 0; i < n; i++)
            index[i] = i;

        int[] aux = new int[n];
        sort(a, index, aux, 0, n-1);
        return index;
    }

    // mergesort a[lo..hi] using auxiliary array aux[lo..hi]
    private static void sort(Comparable[] a, int[] index, int[] aux, int lo, int hi) {
```

```
        if (hi <= lo) return;
        int mid = lo + (hi - lo) / 2;
        sort(a, index, aux, lo, mid);
        sort(a, index, aux, mid + 1, hi);
        merge(a, index, aux, lo, mid, hi);
    }

    // print array to standard output
    private static void show(Comparable[] a) {
        for (int i = 0; i < a.length; i++) {
            StdOut.println(a[i]);
        }
    }

    /**
     * Reads in a sequence of strings from standard input; mergesorts them;
     * and prints them to standard output in ascending order.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        String[] a = StdIn.readAllStrings();
        Merge.sort(a);
        show(a);
    }
}
```


Last updated: Tue Aug 30 10:09:18 EDT 2016.

Quick.java

Below is the syntax highlighted version of Quick.java from §2.3 Quicksort.

```
/******************************************************************************
 *  Compilation:  javac Quick.java
 *  Execution:    java Quick < input.txt
 *  Dependencies: StdOut.java StdIn.java
 *  Data files:   http://algs4.cs.princeton.edu/23quicksort/tiny.txt
 *                http://algs4.cs.princeton.edu/23quicksort/words3.txt
 *
 *  Sorts a sequence of strings from standard input using quicksort.
 *
 *  % more tiny.txt
 *  S O R T E X A M P L E
 *
 *  % java Quick < tiny.txt
 *  A E E L M O P R S T X                 [ one string per line ]
 *
 *  % more words3.txt
 *  bed bug dad yes zoo ... all bad yet
 *
 *  % java Quick < words3.txt
 *  all bad bed bug dad ... yes yet zoo    [ one string per line ]
 *
 *
 *  Remark: For a type-safe version that uses static generics, see
 *
 *    http://algs4.cs.princeton.edu/23quicksort/QuickPedantic.java
 *
 ******************************************************************************/

/**
 *  The {@code Quick} class provides static methods for sorting an
 *  array and selecting the ith smallest element in an array using quicksort.
 *  <p>
 *  For additional documentation, see <a href="http://algs4.cs.princeton.edu/21elementary">Section 2.1</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class Quick {

    // This class should not be instantiated.
    private Quick() { }

    /**
     * Rearranges the array in ascending order, using the natural order.
     * @param a the array to be sorted
     */
    public static void sort(Comparable[] a) {
        StdRandom.shuffle(a);
        sort(a, 0, a.length - 1);
        assert isSorted(a);
    }

    // quicksort the subarray from a[lo] to a[hi]
    private static void sort(Comparable[] a, int lo, int hi) {
        if (hi <= lo) return;
        int j = partition(a, lo, hi);
        sort(a, lo, j-1);
        sort(a, j+1, hi);
        assert isSorted(a, lo, hi);
    }

    // partition the subarray a[lo..hi] so that a[lo..j-1] <= a[j] <= a[j+1..hi]
    // and return the index j.
    private static int partition(Comparable[] a, int lo, int hi) {
        int i = lo;
        int j = hi + 1;
        Comparable v = a[lo];
        while (true) {
```

```
        // find item on lo to swap
        while (less(a[++i], v))
            if (i == hi) break;

        // find item on hi to swap
        while (less(v, a[--j]))
            if (j == lo) break;      // redundant since a[lo] acts as sentinel

        // check if pointers cross
        if (i >= j) break;

        exch(a, i, j);
    }

    // put partitioning item v at a[j]
    exch(a, lo, j);

    // now, a[lo .. j-1] <= a[j] <= a[j+1 .. hi]
    return j;
}

/**
 * Rearranges the array so that {@code a[k]} contains the kth smallest key;
 * {@code a[0]} through {@code a[k-1]} are less than (or equal to) {@code a[k]}; and
 * {@code a[k+1]} through {@code a[n-1]} are greater than (or equal to) {@code a[k]}.
 *
 * @param  a the array
 * @param  k the rank of the key
 * @return the key of rank {@code k}
 * @throws IllegalArgumentException unless {@code 0 <= k < a.length}
 */
public static Comparable select(Comparable[] a, int k) {
    if (k < 0 || k >= a.length) {
        throw new IllegalArgumentException("index is not between 0 and " + a.length + ": " + k);
    }
    StdRandom.shuffle(a);
    int lo = 0, hi = a.length - 1;
    while (hi > lo) {
        int i = partition(a, lo, hi);
        if      (i > k) hi = i - 1;
        else if (i < k) lo = i + 1;
        else return a[i];
    }
    return a[lo];
}



/***************************************************************************
 *  Helper sorting functions.
 ***************************************************************************/

// is v < w ?
private static boolean less(Comparable v, Comparable w) {
    return v.compareTo(w) < 0;
}

// exchange a[i] and a[j]
private static void exch(Object[] a, int i, int j) {
    Object swap = a[i];
    a[i] = a[j];
    a[j] = swap;
}


/***************************************************************************
 *  Check if array is sorted - useful for debugging.
 ***************************************************************************/
private static boolean isSorted(Comparable[] a) {
    return isSorted(a, 0, a.length - 1);
}

private static boolean isSorted(Comparable[] a, int lo, int hi) {
    for (int i = lo + 1; i <= hi; i++)
        if (less(a[i], a[i-1])) return false;
    return true;
}
```

```
    // print array to standard output
    private static void show(Comparable[] a) {
        for (int i = 0; i < a.length; i++) {
            StdOut.println(a[i]);
        }
    }

    /**
     * Reads in a sequence of strings from standard input; quicksorts them;
     * and prints them to standard output in ascending order.
     * Shuffles the array and then prints the strings again to
     * standard output, but this time, using the select method.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        String[] a = StdIn.readAllStrings();
        Quick.sort(a);
        show(a);
        assert isSorted(a);

        // shuffle
        StdRandom.shuffle(a);

        // display results again using select
        StdOut.println();
        for (int i = 0; i < a.length; i++) {
            String ith = (String) Quick.select(a, i);
            StdOut.println(ith);
        }
    }

}
```


Last updated: Fri Jul 7 10:04:13 EDT 2017.

Heap.java

Below is the syntax highlighted version of Heap.java from §2.4 Priority Queues.

```
/******************************************************************************
 *  Compilation:  javac Heap.java
 *  Execution:    java Heap < input.txt
 *  Dependencies: StdOut.java StdIn.java
 *  Data files:   http://algs4.cs.princeton.edu/24pq/tiny.txt
 *                http://algs4.cs.princeton.edu/24pq/words3.txt
 *
 *  Sorts a sequence of strings from standard input using heapsort.
 *
 *  % more tiny.txt
 *  S O R T E X A M P L E
 *
 *  % java Heap < tiny.txt
 *  A E E L M O P R S T X                 [ one string per line ]
 *
 *  % more words3.txt
 *  bed bug dad yes zoo ... all bad yet
 *
 *  % java Heap < words3.txt
 *  all bad bed bug dad ... yes yet zoo   [ one string per line ]
 *
 ******************************************************************************/

/**
 *  The {@code Heap} class provides a static methods for heapsorting
 *  an array.
 *  <p>
 *  For additional documentation, see <a href="http://algs4.cs.princeton.edu/24pq">Section 2.4</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class Heap {

    // This class should not be instantiated.
    private Heap() { }

    /**
     * Rearranges the array in ascending order, using the natural order.
     * @param pq the array to be sorted
     */
    public static void sort(Comparable[] pq) {
        int n = pq.length;
        for (int k = n/2; k >= 1; k--)
            sink(pq, k, n);
        while (n > 1) {
            exch(pq, 1, n--);
            sink(pq, 1, n);
        }
    }

   /***************************************************************************
    * Helper functions to restore the heap invariant.
    ***************************************************************************/

    private static void sink(Comparable[] pq, int k, int n) {
        while (2*k <= n) {
            int j = 2*k;
            if (j < n && less(pq, j, j+1)) j++;
            if (!less(pq, k, j)) break;
            exch(pq, k, j);
            k = j;
```

```
        }
    }

   /***************************************************************************
    * Helper functions for comparisons and swaps.
    * Indices are "off-by-one" to support 1-based indexing.
    ***************************************************************************/
    private static boolean less(Comparable[] pq, int i, int j) {
        return pq[i-1].compareTo(pq[j-1]) < 0;
    }

    private static void exch(Object[] pq, int i, int j) {
        Object swap = pq[i-1];
        pq[i-1] = pq[j-1];
        pq[j-1] = swap;
    }

    // print array to standard output
    private static void show(Comparable[] a) {
        for (int i = 0; i < a.length; i++) {
            StdOut.println(a[i]);
        }
    }

    /**
     * Reads in a sequence of strings from standard input; heapsorts them;
     * and prints them to standard output in ascending order.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        String[] a = StdIn.readAllStrings();
        Heap.sort(a);
        show(a);
    }
}
```


Last updated: Mon Feb 20 17:01:50 EST 2017.

BruteIndexMinPQ.java

Below is the syntax highlighted version of BruteIndexMinPQ.java from §2.4 Priority Queues.

```
/******************************************************************************
 *  Compilation:  javac BruteIndexMinPQ.java
 *  Execution:    java BruteIndexMinPQ
 *  Dependencies: none
 *
 *  Indexed PQ implementation using an array.
 *  The elements are integers between 0 and n-1.
 *
 *  Operation   Running time
 *  -------------------------
 *  Construct      n
 *  is empty       1
 *  size           1
 *  contains       1
 *  insert         1
 *  change key     1
 *  increase key   1
 *  decrease key   1
 *  delete min     n
 *  min            n
 *
 ******************************************************************************/

import java.util.NoSuchElementException;

/**
 *  The {@code BruteIndexMinPQ} class represents an indexed priority queue of generic keys.
 *  It supports the usual <em>insert</em> and <em>delete-the-minimum</em>
 *  operations, along with <em>delete</em> and <em>change-the-key</em>
 *  methods. In order to let the client refer to keys on the priority queue,
 *  an integer between {\code 0} and {@code maxN-1}
 *  is associated with each key—the client
 *  uses this integer to specify which key to delete or change.
 *  It also supports methods for peeking at the minimum key,
 *  testing if the priority queue is empty, and iterating through
 *  the keys.
 *  <p>
 *  This implementation uses an array of keys as the underlying data structure.
 *  The <em>is-empty</em>, <em>size</em>, <em>insert</em>,
 *  <em>delete</em>, <em>key-of</em>, <em>change-key</em>,
 *  <em>decrease-key</em>, and <em>increase-key</em> operations take constant time.
 *  The <em>min-key</em>, <em>min-index</em>, and <em>delete-the-minimum</em>
 *  operations take linear time.
 *  Construction takes time proportional to the specified capacity.
 *  <p>

 *  For additional documentation, see <a href="http://algs4.cs.princeton.edu/24pq">Section 2.4</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 *
 *  @param <Key> the generic type of key on this priority queue
 */
public class BruteIndexMinPQ<Key extends Comparable<Key>> {
    private int n;           // number of elements on PQ
    private Key[] keys;      // keys[i] = key of element i

    /**
     * Initializes an empty indexed priority queue with indices between {@code 0}
     * and {@code maxN - 1}.
     *
     * @param   maxN the keys on this priority queue are index from {@code 0}
     *         {@code maxN - 1}
     * @throws IllegalArgumentException if {@code maxN < 0}
     */
    public BruteIndexMinPQ(int maxN) {
        keys = (Key[]) new Comparable[maxN];
    }

    /**
     * Returns true if this priority queue is empty.
     *
     * @return {@code true} if this priority queue is empty;
     *         {@code false} otherwise
     */
    public boolean isEmpty() {
        return n == 0;
    }

    /**
     * Returns the number of keys on this priority queue.
```

```
     *
     * @return the number of keys on this priority queue
     */
    public int size() {
        return n;
    }

    /**
     * Returns true if the argument is an index on this priority queue.
     *
     * @param  i an index
     * @return {@code true} if {@code i} is an index on this priority queue;
     *         {@code false} otherwise
     * @throws IllegalArgumentException unless {@code 0 <= i < maxN}
     */
    public boolean contains(int i) {
        return keys[i] != null;
    }

    /**
     * Associates key with index {@code i}.
     *
     * @param  i an index
     * @param  key the key to associate with index {@code i}
     * @throws IllegalArgumentException unless {@code 0 <= i < maxN}
     * @throws IllegalArgumentException if there already is an item associated
     *         with index {@code i}
     */
    public void insert(int i, Key key) {
        if (contains(i)) throw new NoSuchElementException("index is already in priority queue");
        n++;
        keys[i] = key;
    }

    /**
     * Removes a minimum key and returns its associated index.
     *
     * @return an index associated with a minimum key
     * @throws NoSuchElementException if this priority queue is empty
     */
    public int delMin() {
        int min = minIndex();
        keys[min] = null;
        n--;
        return min;
    }

   /**
     * Returns a minimum key.
     *
     * @return a minimum key
     * @throws NoSuchElementException if this priority queue is empty
     */
    public Key minKey() {
        int min = minIndex();
        return keys[min];
    }

    /**
     * Returns an index associated with a minimum key.
     *
     * @return an index associated with a minimum key
     * @throws NoSuchElementException if this priority queue is empty
     */
    public int minIndex() {
        if (n == 0) throw new NoSuchElementException("Priority queue underflow");
        int min = -1;
        for (int i = 0; i < keys.length; i++) {
            if (keys[i] == null) continue;
            else if (min == -1) min = i;
            else if (keys[i].compareTo(keys[min]) < 0) min = i;
        }
        return min;
    }
    /**
     * Change the key associated with index {@code i} to the specified value.
     *
     * @param  i the index of the key to change
     * @param  key change the key assocated with index {@code i} to this key
     * @throws IllegalArgumentException unless {@code 0 <= i maxN}
     * @throws NoSuchElementException no key is associated with index {@code i}
     */
    public void changeKey(int i, Key key) {
        if (!contains(i)) throw new NoSuchElementException("index is not in the priority queue");
        keys[i] = key;
    }

    /**
     * Decrease the key associated with index {@code i} to the specified value.
     *
     * @param  i the index of the key to decrease
     * @param  key decrease the key assocated with index {@code i} to this key
```

```
     * @throws IllegalArgumentException unless {@code 0 <= i < maxN}
     * @throws IllegalArgumentException if {@code key >= keyOf(i)}
     * @throws NoSuchElementException no key is associated with index {@code i}
     */
    public void decreaseKey(int i, Key key) {
        if (!contains(i)) throw new NoSuchElementException("index is not in the priority queue");
        if (keys[i].compareTo(key) <= 0)
            throw new IllegalArgumentException("Calling decreaseKey() with given argument would not strictly decrease the key");
        keys[i] = key;
    }

    /**
     * Increase the key associated with index {@code i} to the specified value.
     *
     * @param  i the index of the key to increase
     * @param  key increase the key assocated with index {@code i} to this key
     * @throws IllegalArgumentException unless {@code 0 <= i < maxN}
     * @throws IllegalArgumentException if {@code key <= keyOf(i)}
     * @throws NoSuchElementException no key is associated with index {@code i}
     */
    public void increaseKey(int i, Key key) {
        if (!contains(i)) throw new NoSuchElementException("index is not in the priority queue");
        if (keys[i].compareTo(key) >= 0)
            throw new IllegalArgumentException("Calling increaseKey() with given argument would not strictly increase the key");
        keys[i] = key;
    }

}
```


Last updated: Fri Jul 7 09:52:30 EDT 2017.

IndexMinPQ.java

Below is the syntax highlighted version of IndexMinPQ.java from §2.4 Priority Queues.

```
/******************************************************************************
 *  Compilation:  javac IndexMinPQ.java
 *  Execution:    java IndexMinPQ
 *  Dependencies: StdOut.java
 *
 *  Minimum-oriented indexed PQ implementation using a binary heap.
 *
 ******************************************************************************/

import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 *  The {@code IndexMinPQ} class represents an indexed priority queue of generic keys.
 *  It supports the usual <em>insert</em> and <em>delete-the-minimum</em>
 *  operations, along with <em>delete</em> and <em>change-the-key</em>
 *  methods. In order to let the client refer to keys on the priority queue,
 *  an integer between {@code 0} and {@code maxN - 1}
 *  is associated with each key—the client uses this integer to specify
 *  which key to delete or change.
 *  It also supports methods for peeking at the minimum key,
 *  testing if the priority queue is empty, and iterating through
 *  the keys.
 *  <p>
 *  This implementation uses a binary heap along with an array to associate
 *  keys with integers in the given range.
 *  The <em>insert</em>, <em>delete-the-minimum</em>, <em>delete</em>,
 *  <em>change-key</em>, <em>decrease-key</em>, and <em>increase-key</em>
 *  operations take logarithmic time.
 *  The <em>is-empty</em>, <em>size</em>, <em>min-index</em>, <em>min-key</em>,
 *  and <em>key-of</em> operations take constant time.
 *  Construction takes time proportional to the specified capacity.
 *  <p>
 *  For additional documentation, see <a href="http://algs4.cs.princeton.edu/24pq">Section 2.4</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 *
 *  @param <Key> the generic type of key on this priority queue
 */
public class IndexMinPQ<Key extends Comparable<Key>> implements Iterable<Integer> {
    private int maxN;        // maximum number of elements on PQ
    private int n;           // number of elements on PQ
    private int[] pq;        // binary heap using 1-based indexing
    private int[] qp;        // inverse of pq - qp[pq[i]] = pq[qp[i]] = i
    private Key[] keys;      // keys[i] = priority of i

    /**
     * Initializes an empty indexed priority queue with indices between {@code 0}
     * and {@code maxN - 1}.
     * @param  maxN the keys on this priority queue are index from {@code 0}
     *         {@code maxN - 1}
     * @throws IllegalArgumentException if {@code maxN < 0}
     */
    public IndexMinPQ(int maxN) {
        if (maxN < 0) throw new IllegalArgumentException();
        this.maxN = maxN;
        n = 0;
        keys = (Key[]) new Comparable[maxN + 1];    // make this of length maxN??
        pq   = new int[maxN + 1];
        qp   = new int[maxN + 1];                   // make this of length maxN??
        for (int i = 0; i <= maxN; i++)
            qp[i] = -1;
    }

    /**
     * Returns true if this priority queue is empty.
     *
     * @return {@code true} if this priority queue is empty;
     *         {@code false} otherwise
     */
    public boolean isEmpty() {
        return n == 0;
    }

    /**
     * Is {@code i} an index on this priority queue?
     *
     * @param  i an index
     * @return {@code true} if {@code i} is an index on this priority queue;
     *         {@code false} otherwise
```

```
     * @throws IllegalArgumentException unless {@code 0 <= i < maxN}
     */
    public boolean contains(int i) {
        if (i < 0 || i >= maxN) throw new IllegalArgumentException();
        return qp[i] != -1;
    }

    /**
     * Returns the number of keys on this priority queue.
     *
     * @return the number of keys on this priority queue
     */
    public int size() {
        return n;
    }

    /**
     * Associates key with index {@code i}.
     *
     * @param  i an index
     * @param  key the key to associate with index {@code i}
     * @throws IllegalArgumentException unless {@code 0 <= i < maxN}
     * @throws IllegalArgumentException if there already is an item associated
     *         with index {@code i}
     */
    public void insert(int i, Key key) {
        if (i < 0 || i >= maxN) throw new IllegalArgumentException();
        if (contains(i)) throw new IllegalArgumentException("index is already in the priority queue");
        n++;
        qp[i] = n;
        pq[n] = i;
        keys[i] = key;
        swim(n);
    }

    /**
     * Returns an index associated with a minimum key.
     *
     * @return an index associated with a minimum key
     * @throws NoSuchElementException if this priority queue is empty
     */
    public int minIndex() {
        if (n == 0) throw new NoSuchElementException("Priority queue underflow");
        return pq[1];
    }

    /**
     * Returns a minimum key.
     *
     * @return a minimum key
     * @throws NoSuchElementException if this priority queue is empty
     */
    public Key minKey() {
        if (n == 0) throw new NoSuchElementException("Priority queue underflow");
        return keys[pq[1]];
    }

    /**
     * Removes a minimum key and returns its associated index.
     * @return an index associated with a minimum key
     * @throws NoSuchElementException if this priority queue is empty
     */
    public int delMin() {
        if (n == 0) throw new NoSuchElementException("Priority queue underflow");
        int min = pq[1];
        exch(1, n--);
        sink(1);
        assert min == pq[n+1];
        qp[min] = -1;        // delete
        keys[min] = null;    // to help with garbage collection
        pq[n+1] = -1;        // not needed
        return min;
    }

    /**
     * Returns the key associated with index {@code i}.
     *
     * @param  i the index of the key to return
     * @return the key associated with index {@code i}
     * @throws IllegalArgumentException unless {@code 0 <= i < maxN}
     * @throws NoSuchElementException no key is associated with index {@code i}
     */
    public Key keyOf(int i) {
        if (i < 0 || i >= maxN) throw new IllegalArgumentException();
        if (!contains(i)) throw new NoSuchElementException("index is not in the priority queue");
        else return keys[i];
    }

    /**
     * Change the key associated with index {@code i} to the specified value.
     *
     * @param  i the index of the key to change
```

```
     * @param  key change the key associated with index {@code i} to this key
     * @throws IllegalArgumentException unless {@code 0 <= i < maxN}
     * @throws NoSuchElementException no key is associated with index {@code i}
     */
    public void changeKey(int i, Key key) {
        if (i < 0 || i >= maxN) throw new IllegalArgumentException();
        if (!contains(i)) throw new NoSuchElementException("index is not in the priority queue");
        keys[i] = key;
        swim(qp[i]);
        sink(qp[i]);
    }

    /**
     * Change the key associated with index {@code i} to the specified value.
     *
     * @param  i the index of the key to change
     * @param  key change the key associated with index {@code i} to this key
     * @throws IllegalArgumentException unless {@code 0 <= i < maxN}
     * @deprecated Replaced by {@code changeKey(int, Key)}.
     */
    @Deprecated
    public void change(int i, Key key) {
        changeKey(i, key);
    }

    /**
     * Decrease the key associated with index {@code i} to the specified value.
     *
     * @param  i the index of the key to decrease
     * @param  key decrease the key associated with index {@code i} to this key
     * @throws IllegalArgumentException unless {@code 0 <= i < maxN}
     * @throws IllegalArgumentException if {@code key >= keyOf(i)}
     * @throws NoSuchElementException no key is associated with index {@code i}
     */
    public void decreaseKey(int i, Key key) {
        if (i < 0 || i >= maxN) throw new IllegalArgumentException();
        if (!contains(i)) throw new NoSuchElementException("index is not in the priority queue");
        if (keys[i].compareTo(key) <= 0)
            throw new IllegalArgumentException("Calling decreaseKey() with given argument would not strictly decrease the key");
        keys[i] = key;
        swim(qp[i]);
    }

    /**
     * Increase the key associated with index {@code i} to the specified value.
     *
     * @param  i the index of the key to increase
     * @param  key increase the key associated with index {@code i} to this key
     * @throws IllegalArgumentException unless {@code 0 <= i < maxN}
     * @throws IllegalArgumentException if {@code key <= keyOf(i)}
     * @throws NoSuchElementException no key is associated with index {@code i}
     */
    public void increaseKey(int i, Key key) {
        if (i < 0 || i >= maxN) throw new IllegalArgumentException();
        if (!contains(i)) throw new NoSuchElementException("index is not in the priority queue");
        if (keys[i].compareTo(key) >= 0)
            throw new IllegalArgumentException("Calling increaseKey() with given argument would not strictly increase the key");
        keys[i] = key;
        sink(qp[i]);
    }

    /**
     * Remove the key associated with index {@code i}.
     *
     * @param  i the index of the key to remove
     * @throws IllegalArgumentException unless {@code 0 <= i < maxN}
     * @throws NoSuchElementException no key is associated with index {@code i}
     */
    public void delete(int i) {
        if (i < 0 || i >= maxN) throw new IllegalArgumentException();
        if (!contains(i)) throw new NoSuchElementException("index is not in the priority queue");
        int index = qp[i];
        exch(index, n--);
        swim(index);
        sink(index);
        keys[i] = null;
        qp[i] = -1;
    }


   /***************************************************************************
    * General helper functions.
    ***************************************************************************/
    private boolean greater(int i, int j) {
        return keys[pq[i]].compareTo(keys[pq[j]]) > 0;
    }

    private void exch(int i, int j) {
        int swap = pq[i];
        pq[i] = pq[j];
        pq[j] = swap;
        qp[pq[i]] = i;
```

```
        qp[pq[j]] = j;
    }


   /***************************************************************************
    * Heap helper functions.
    ***************************************************************************/
    private void swim(int k) {
        while (k > 1 && greater(k/2, k)) {
            exch(k, k/2);
            k = k/2;
        }
    }

    private void sink(int k) {
        while (2*k <= n) {
            int j = 2*k;
            if (j < n && greater(j, j+1)) j++;
            if (!greater(k, j)) break;
            exch(k, j);
            k = j;
        }
    }


   /***************************************************************************
    * Iterators.
    ***************************************************************************/

    /**
     * Returns an iterator that iterates over the keys on the
     * priority queue in ascending order.
     * The iterator doesn't implement {@code remove()} since it's optional.
     *
     * @return an iterator that iterates over the keys in ascending order
     */
    public Iterator<Integer> iterator() { return new HeapIterator(); }

    private class HeapIterator implements Iterator<Integer> {
        // create a new pq
        private IndexMinPQ<Key> copy;

        // add all elements to copy of heap
        // takes linear time since already in heap order so no keys move
        public HeapIterator() {
            copy = new IndexMinPQ<Key>(pq.length - 1);
            for (int i = 1; i <= n; i++)
                copy.insert(pq[i], keys[pq[i]]);
        }

        public boolean hasNext()  { return !copy.isEmpty();                     }
        public void remove()      { throw new UnsupportedOperationException();  }

        public Integer next() {
            if (!hasNext()) throw new NoSuchElementException();
            return copy.delMin();
        }
    }


    /**
     * Unit tests the {@code IndexMinPQ} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        // insert a bunch of strings
        String[] strings = { "it", "was", "the", "best", "of", "times", "it", "was", "the", "worst" };

        IndexMinPQ<String> pq = new IndexMinPQ<String>(strings.length);
        for (int i = 0; i < strings.length; i++) {
            pq.insert(i, strings[i]);
        }

        // delete and print each key
        while (!pq.isEmpty()) {
            int i = pq.delMin();
            StdOut.println(i + " " + strings[i]);
        }
        StdOut.println();

        // reinsert the same strings
        for (int i = 0; i < strings.length; i++) {
            pq.insert(i, strings[i]);
        }

        // print each key using the iterator
        for (int i : pq) {
            StdOut.println(i + " " + strings[i]);
        }
        while (!pq.isEmpty()) {
            pq.delMin();
```

```
            }

        }
}
```


Last updated: Fri Jul 7 09:52:30 EDT 2017.

IndexMultiwayMinPQ.java

Below is the syntax highlighted version of IndexMultiwayMinPQ.java from §9.9 Miscellaneous.

```
/******************************************************************************
 *  Compilation: javac IndexMultiwayMinPQ.java
 *  Execution:
 *
 *  An inde  multiway heap.
 *
 ******************************************************************************/

import java.util.Comparator;
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 *  The IndexMultiwayMinPQ class represents an indexed priority queue of generic keys.
 *  It supports the usual insert and delete-the-minimum operations,
 *  along with delete and change-the-key methods.
 *  In order to let the client refer to keys on the priority queue,
 *  an integer between 0 and N-1 is associated with each key ; the client
 *  uses this integer to specify which key to delete or change.
 *  It also supports methods for peeking at the minimum key,
 *  testing if the priority queue is empty, and iterating through
 *  the keys.
 *
 *  This implementation uses a multiway heap along with an array to associate
 *  keys with integers in the given range.
 *  For simplified notations, logarithm in base d will be referred as log-d
 *  The delete-the-minimum, delete, change-key and increase-key operations
 *  take time proportional to d*log-d(n)
 *  The insert and decrease-key take time proportional to log-d(n)
 *  The is-empty, min-index, min-key, size, contains and key-of operations take constant time.
 *  Construction takes time proportional to the specified capacity.
 *
 *  The arrays used in this structure have the first d indices empty,
 *  it apparently helps with caching effects.
 *
 *  @author Tristan Claverie
 */
public class IndexMultiwayMinPQ<Key> implements Iterable<Integer> {
        private final int d;                                //Dimension of the heap
        private int n;                                      //Number of keys currently in the queue
        private int nmax;                                   //Maximum number of items in the queue
        private int[] pq;                                   //Multiway heap
        private int[] qp;                                   //Inverse of pq : qp[pq[i]] = pq[qp[i]] = i
        private Key[] keys;                                 //keys[i] = priority of i
        private final Comparator<Key> comp; //Comparator over the keys


    /**
     * Initializes an empty indexed priority queue with indices between {@code 0} to {@code N-1}
     * Worst case is O(n)
     * @param N number of keys in the priority queue, index from {@code 0} to {@code N-1}
     * @param D dimension of the heap
     * @throws java.lang.IllegalArgumentException if {@code N < 0}
     * @throws java.lang.IllegalArgumentException if {@code D < 2}
     */
        public IndexMultiwayMinPQ(int N, int D) {
                if (N < 0) throw new IllegalArgumentException("Maximum number of elements cannot be negative");
                if (D < 2) throw new IllegalArgumentException("Dimension should be 2 or over");
                this.d = D;
                nmax = N;
                pq = new int[nmax+D];
                qp = new int[nmax+D];
                keys = (Key[]) new Comparable[nmax+D];
                for (int i = 0; i < nmax+D; qp[i++] = -1);
                comp = new MyComparator();
        }

    /**
     * Initializes an empty indexed priority queue with indices between {@code 0} to {@code N-1}
     * Worst case is O(n)
     * @param N number of keys in the priority queue, index from {@code 0} to {@code N-1}
     * @param D dimension of the heap
     * @param C a Comparator over the keys
     * @throws java.lang.IllegalArgumentException if {@code N < 0}
     * @throws java.lang.IllegalArgumentException if {@code D < 2}
     */
        public IndexMultiwayMinPQ(int N, Comparator<Key> C, int D) {
                if (N < 0) throw new IllegalArgumentException("Maximum number of elements cannot be negative");
                if (D < 2) throw new IllegalArgumentException("Dimension should be 2 or over");
                this.d = D;
                nmax = N;
                pq = new int[nmax+D];
                qp = new int[nmax+D];
                keys = (Key[]) new Comparable[nmax+D];
                for (int i = 0; i < nmax+D; qp[i++] = -1);
                comp = C;
        }

        /**
         * Whether the priority queue is empty
```

```
     * Worst case is O(1)
     * @return true if the priority queue is empty, false if not
     */
    public boolean isEmpty() {
            return n == 0;
    }

    /**
     * Does the priority queue contains the index i ?
     * Worst case is O(1)
     * @param i an index
     * @throws java.lang.IllegalArgumentException if the specified index is invalid
     * @return true if i is on the priority queue, false if not
     */
    public boolean contains(int i) {
            if (i < 0 ||i >= nmax) throw new IllegalArgumentException();
            return qp[i+d] != -1;
    }

    /**
     * Number of elements currently on the priority queue
     * Worst case is O(1)
     * @return the number of elements on the priority queue
     */
    public int size() {
            return n;
    }

    /**
     * Associates a key with an index
     * Worst case is O(log-d(n))
     * @param i an index
     * @param key a Key associated with i
     * @throws java.lang.IllegalArgumentException if the specified index is invalid
     * @throws java.lang.IllegalArgumentException if the index is already in the queue
     */
    public void insert(int i, Key key) {
            if (i < 0 || i >= nmax) throw new IllegalArgumentException();
            if (contains(i)) throw new IllegalArgumentException("Index already there");
            keys[i+d] = key;
            pq[n+d] = i;
            qp[i+d] = n;
            swim(n++);
    }

    /**
     * Gets the index associated with the minimum key
     * Worst case is O(1)
     * @throws java.util.NoSuchElementException if the priority queue is empty
     * @return the index associated with the minimum key
     */
    public int minIndex() {
            if (isEmpty()) throw new NoSuchElementException("Priority queue is empty");
            return pq[d];
    }

    /**
     * Gets the minimum key currently in the queue
     * Worst case is O(1)
     * @throws java.util.NoSuchElementException if the priority queue is empty
     * @return the minimum key currently in the priority queue
     */
    public Key minKey() {
            if (isEmpty()) throw new NoSuchElementException("Priority queue is empty");
            return keys[pq[d]+d];
    }

    /**
     * Deletes the minimum key
     * Worst case is O(d*log-d(n))
     * @throws java.util.NoSuchElementException if the priority queue is empty
     * @return the index associated with the minimum key
     */
    public int delMin() {
            if (isEmpty()) throw new NoSuchElementException("Priority queue is empty");
            int min = pq[d];
            exch(0, --n);
            sink(0);
            qp[min+d] = -1;
            keys[pq[n+d]+d] = null;
            pq[n+d] = -1;
            return min;
    }

    /**
     * Gets the key associated with index i
     * Worst case is O(1)
     * @param i an index
     * @throws java.lang.IllegalArgumentException if the specified index is invalid
     * @throws java.lang.IllegalArgumentException if the index is not in the queue
     * @return the key associated with index i
     */
    public Key keyOf(int i) {
            if (i < 0 || i >= nmax) throw new IllegalArgumentException();
            if (! contains(i)) throw new NoSuchElementException("Specified index is not in the queue");
            return keys[i+d];
    }

    /**
     * Changes the key associated with index i to the given key
```

```
     * If the given key is greater, Worst case is O(d*log-d(n))
     * If the given key is lower,   Worst case is O(log-d(n))
     * @param i an index
     * @param key the key to associate with i
     * @throws java.lang.IllegalArgumentException if the specified index is invalid
     * @throws java.lang.IllegalArgumentException if the index has no key associated with
     */
    public void changeKey(int i, Key key) {
            if (i < 0 || i >= nmax) throw new IllegalArgumentException();
            if (! contains(i)) throw new NoSuchElementException("Specified index is not in the queue");
            Key tmp = keys[i+d];
            keys[i+d] = key;
            if (comp.compare(key, tmp) <= 0) { swim(qp[i+d]);}
            else                                                { sink(qp[i+d]);}
    }

    /**
     * Decreases the key associated with index i to the given key
     * Worst case is O(log-d(n))
     * @param i an index
     * @param key the key to associate with i
     * @throws java.lang.IllegalArgumentException if the specified index is invalid
     * @throws java.util.NoSuchElementException if the index has no key associated with
     * @throws java.lang.IllegalArgumentException if the given key is greater than the current key
     */
    public void decreaseKey(int i, Key key) {
            if (i < 0 || i >=nmax) throw new IllegalArgumentException();
            if (! contains(i)) throw new NoSuchElementException("Specified index is not in the queue");
            if (comp.compare(keys[i+d], key) <= 0) throw new IllegalArgumentException("Calling with this argument would not decrease th
            keys[i+d] = key;
            swim(qp[i+d]);
    }

    /**
     * Increases the key associated with index i to the given key
     * Worst case is O(d*log-d(n))
     * @param i an index
     * @param key the key to associate with i
     * @throws java.lang.IllegalArgumentException if the specified index is invalid
     * @throws java.util.NoSuchElementException if the index has no key associated with
     * @throws java.lang.IllegalArgumentException if the given key is lower than the current key
     */
    public void increaseKey(int i, Key key) {
            if (i < 0 || i >=nmax) throw new IllegalArgumentException();
            if (! contains(i)) throw new NoSuchElementException("Specified index is not in the queue");
            if (comp.compare(keys[i+d], key) >= 0) throw new IllegalArgumentException("Calling with this argument would not increase th
            keys[i+d] = key;
            sink(qp[i+d]);
    }

    /**
     * Deletes the key associated to the given index
     * Worst case is O(d*log-d(n))
     * @param i an index
     * @throws java.lang.IllegalArgumentException if the specified index is invalid
     * @throws java.util.NoSuchElementException if the given index has no key associated with
     */
    public void delete(int i) {
            if (i < 0 || i >= nmax) throw new IllegalArgumentException();
            if (! contains(i)) throw new NoSuchElementException("Specified index is not in the queue");
            int idx = qp[i+d];
            exch(idx, --n);
            swim(idx);
            sink(idx);
            keys[i+d] = null;
            qp[i+d] = -1;
    }

    /**************************
     * General helper functions
     *************************/

    //Compares two keys
    private boolean greater(int i, int j) {
            return comp.compare(keys[pq[i+d]+d], keys[pq[j+d]+d]) > 0;
    }

    //Exchanges two keys
    private void exch(int x, int y) {
            int i = x+d, j = y+d;
            int swap = pq[i];
            pq[i] = pq[j];
            pq[j] = swap;
            qp[pq[i]+d] = x;
            qp[pq[j]+d] = y;
    }

    /**************************
     * Functions for moving upward or downward
     *************************/

    //Moves upward
    private void swim(int i) {
            if (i > 0 && greater((i-1)/d, i)) {
                    exch(i, (i-1)/d);
                    swim((i-1)/d);
            }
    }

    //Moves downward
```

```
        private void sink(int i) {
                if (d*i+1 >= n) return;
                int min = minChild(i);
                while (min < n && greater(i, min)) {
                        exch(i, min);
                        i = min;
                        min = minChild(i);
                }
        }

        /**************************
         * Deletes the minimum child
         **************************/

        //Return the minimum child of i
        private int minChild(int i) {
                int loBound = d*i+1, hiBound = d*i+d;
                int min = loBound;
                for (int cur = loBound; cur <= hiBound; cur++) {
                        if (cur < n && greater(min, cur)) min = cur;
                }
                return min;
        }

        /**************************
         * Iterator
         **************************/

        /**
         * Gets an Iterator over the indexes in the priority queue in ascending order
         * The Iterator does not implement the remove() method
         * iterator() : Worst case is O(n)
         * next() :     Worst case is O(d*log-d(n))
         * hasNext() :  Worst case is O(1)
         * @return an Iterator over the indexes in the priority queue in ascending order
         */

        public Iterator<Integer> iterator() {
                return new MyIterator();
        }

        //Constructs an Iterator over the indices in linear time
        private class MyIterator implements Iterator<Integer> {
                IndexMultiwayMinPQ<Key> clone;

                public MyIterator() {
                        clone = new IndexMultiwayMinPQ<Key>(nmax, comp, d);
                        for (int i = 0; i < n; i++) {
                                clone.insert(pq[i+d], keys[pq[i+d]+d]);
                        }
                }

                public boolean hasNext() {
                        return !clone.isEmpty();
                }

                public Integer next() {
                        if (!hasNext()) throw new NoSuchElementException();
                        return clone.delMin();
                }

                public void remove() {
                        throw new UnsupportedOperationException();
                }
        }

        /**************************
         * Comparator
         **************************/

        //default Comparator
        private class MyComparator implements Comparator<Key> {
                @Override
                public int compare(Key key1, Key key2) {
                        return ((Comparable<Key>) key1).compareTo(key2);
                }
        }

}
```

IndexBinomialMinPQ.java

Below is the syntax highlighted version of IndexBinomialMinPQ.java from §9.9 Miscellaneous.

```
/******************************************************************************
 *  Compilation: javac IndexBinomialMinPQ.java
 *  Execution:
 *
 *  An index binomial heap.
 *
 ******************************************************************************/

import java.util.Comparator;
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 *  The IndexBinomialMinPQ class represents an indexed priority queue of generic keys.
 *  It supports the usual insert and delete-the-minimum operations,
 *  along with delete and change-the-key methods.
 *  In order to let the client refer to keys on the priority queue,
 *  an integer between 0 and N-1 is associated with each key ; the client
 *  uses this integer to specify which key to delete or change.
 *  It also supports methods for peeking at the minimum key,
 *  testing if the priority queue is empty, and iterating through
 *  the keys.
 *
 *  This implementation uses a binomial heap along with an array to associate
 *  keys with integers in the given range.
 *  The insert, delete-the-minimum, delete, change-key, decrease-key,
 *  increase-key and size operations take logarithmic time.
 *  The is-empty, min-index, min-key, and key-of operations take constant time.
 *  Construction takes time proportional to the specified capacity.
 *
 *  @author Tristan Claverie
 */
public class IndexBinomialMinPQ<Key> implements Iterable<Integer> {
        private Node<Key> head;                         //Head of the list of roots
        private Node<Key>[] nodes;                      //Array of indexed Nodes of the heap
        private int n;                                  //Maximum size of the tree
        private final Comparator<Key> comparator;       //Comparator over the keys

        //Represents a node of a Binomial Tree
        private class Node<Key> {
                Key key;                                //Key contained by the Node
                int order;                              //The order of the Binomial Tree rooted by this Node
                int index;                              //Index associated with the Key
                Node<Key> parent;                       //parent of this Node
                Node<Key> child, sibling;               //child and sibling of this Node
        }

    /**
     * Initializes an empty indexed priority queue with indices between {@code 0} to {@code N-1}
     * Worst case is O(n)
     * @param N number of keys in the priority queue, index from {@code 0} to {@code N-1}
     * @throws java.lang.IllegalArgumentException if {@code N < 0}
     */
        public IndexBinomialMinPQ(int N) {
                if (N < 0) throw new IllegalArgumentException("Cannot create a priority queue of negative size");
                comparator = new MyComparator();
                nodes = (Node<Key>[]) new Node[N];
                this.n = N;
        }

    /**
     * Initializes an empty indexed priority queue with indices between {@code 0} to {@code N-1}
     * Worst case is O(n)
     * @param N number of keys in the priority queue, index from {@code 0} to {@code N-1}
     * @param comparator a Comparator over the keys
     * @throws java.lang.IllegalArgumentException if {@code N < 0}
     */
        public IndexBinomialMinPQ(int N, Comparator<Key> comparator) {
                if (N < 0) throw new IllegalArgumentException("Cannot create a priority queue of negative size");
                this.comparator = comparator;
                nodes = (Node<Key>[]) new Node[N];
                this.n = N;
        }

        /**
         * Whether the priority queue is empty
         * Worst case is O(1)
         * @return true if the priority queue is empty, false if not
         */
        public boolean isEmpty() {
                return head == null;
        }

        /**
         * Does the priority queue contains the index i ?
         * Worst case is O(1)
         * @param i an index
         * @throws java.lang.IllegalArgumentException if the specified index is invalid
         * @return true if i is on the priority queue, false if not
         */
```

```
public boolean contains(int i) {
        if (i < 0 || i >= n) throw new IllegalArgumentException();
        else return nodes[i] != null;
}

/**
 * Number of elements currently on the priority queue
 * Worst case is O(log(n))
 * @return the number of elements on the priority queue
 */
public int size() {
        int result = 0, tmp;
        for (Node<Key> node = head; node != null; node = node.sibling) {
                if (node.order > 30) { throw new ArithmeticException("The number of elements cannot be evaluated, but the priority
                tmp =  1 << node.order;
                result |= tmp;
        }
        return result;
}

/**
 * Associates a key with an index
 * Worst case is O(log(n))
 * @param i an index
 * @param key a Key associated with i
 * @throws java.lang.IllegalArgumentException if the specified index is invalid
 * @throws java.lang.IllegalArgumentException if the index is already in the queue
 */
public void insert(int i, Key key) {
        if (i < 0 || i >= n) throw new IllegalArgumentException();
        if (contains(i)) throw new IllegalArgumentException("Specified index is already in the queue");
        Node<Key> x = new Node<Key>();
        x.key = key;
        x.index = i;
        x.order = 0;
        nodes[i] = x;
        IndexBinomialMinPQ<Key> H = new IndexBinomialMinPQ<Key>();
        H.head = x;
        head = union(H).head;
}

/**
 * Gets the index associated with the minimum key
 * Worst case is O(log(n))
 * @throws java.util.NoSuchElementException if the priority queue is empty
 * @return the index associated with the minimum key
 */

public int minIndex() {
        if (isEmpty()) throw new NoSuchElementException("Priority queue is empty");
        Node<Key> min = head;
        Node<Key> current = head;
        while (current.sibling != null) {
                min = (greater(min.key, current.sibling.key)) ? current.sibling : min;
                current = current.sibling;
        }
        return min.index;
}

/**
 * Gets the minimum key currently in the queue
 * Worst case is O(log(n))
 * @throws java.util.NoSuchElementException if the priority queue is empty
 * @return the minimum key currently in the priority queue
 */

public Key minKey() {
        if (isEmpty()) throw new NoSuchElementException("Priority queue is empty");
        Node<Key> min = head;
        Node<Key> current = head;
        while (current.sibling != null) {
                min = (greater(min.key, current.sibling.key)) ? current.sibling : min;
                current = current.sibling;
        }
        return min.key;
}

/**
 * Deletes the minimum key
 * Worst case is O(log(n))
 * @throws java.util.NoSuchElementException if the priority queue is empty
 * @return the index associated with the minimum key
 */

public int delMin() {
        if(isEmpty()) throw new NoSuchElementException("Priority queue is empty");
        Node<Key> min = eraseMin();
        Node<Key> x = (min.child == null) ? min : min.child;
        if (min.child != null) {
                min.child = null;
                Node<Key> prevx = null, nextx = x.sibling;
                while (nextx != null) {
                        x.parent = null; // for garbage collection
                        x.sibling = prevx;
                        prevx = x;
                        x = nextx;nextx = nextx.sibling;
                }
                x.parent = null;
                x.sibling = prevx;
                IndexBinomialMinPQ<Key> H = new IndexBinomialMinPQ<Key>();
```

```
                H.head = x;
                head = union(H).head;
        }
        return min.index;
}

/**
 * Gets the key associated with index i
 * Worst case is O(1)
 * @param i an index
 * @throws java.lang.IllegalArgumentException if the specified index is invalid
 * @throws java.lang.IllegalArgumentException if the index is not in the queue
 * @return the key associated with index i
 */

public Key keyOf(int i) {
        if (i < 0 || i >= n) throw new IllegalArgumentException();
        if (!contains(i)) throw new IllegalArgumentException("Specified index is not in the queue");
        return nodes[i].key;
}

/**
 * Changes the key associated with index i to the given key
 * Worst case is O(log(n))
 * @param i an index
 * @param key the key to associate with i
 * @throws java.lang.IllegalArgumentException if the specified index is invalid
 * @throws java.lang.IllegalArgumentException if the index has no key associated with
 */

public void changeKey(int i, Key key) {
        if (i < 0 || i >= n)            throw new IllegalArgumentException();
        if (!contains(i))                       throw new IllegalArgumentException("Specified index is not in the queue");
        if (greater(nodes[i].key, key))  decreaseKey(i, key);
        else                                                    increaseKey(i, key);
}

/**
 * Decreases the key associated with index i to the given key
 * Worst case is O(log(n))
 * @param i an index
 * @param key the key to associate with i
 * @throws java.lang.IllegalArgumentException if the specified index is invalid
 * @throws java.util.NoSuchElementException if the index has no key associated with
 * @throws java.lang.IllegalArgumentException if the given key is greater than the current key
 */

public void decreaseKey(int i, Key key) {
        if (i < 0 || i >= n)            throw new IllegalArgumentException();
        if (!contains(i))                       throw new NoSuchElementException("Specified index is not in the queue");
        if (greater(key, nodes[i].key))  throw new IllegalArgumentException("Calling with this argument would not decrease the key"
        Node<Key> x = nodes[i];
        x.key = key;
        swim(i);
}

/**
 * Increases the key associated with index i to the given key
 * Worst case is O(log(n))
 * @param i an index
 * @param key the key to associate with i
 * @throws java.lang.IllegalArgumentException if the specified index is invalid
 * @throws java.util.NoSuchElementException if the index has no key associated with
 * @throws java.lang.IllegalArgumentException if the given key is lower than the current key
 */

public void increaseKey(int i, Key key) {
        if (i < 0 || i >= n)            throw new IllegalArgumentException();
        if (!contains(i))                       throw new NoSuchElementException("Specified index is not in the queue");
        if (greater(nodes[i].key, key))  throw new IllegalArgumentException("Calling with this argument would not increase the key"
        delete(i);
        insert(i, key);
}

/**
 * Deletes the key associated the given index
 * Worst case is O(log(n))
 * @param i an index
 * @throws java.lang.IllegalArgumentException if the specified index is invalid
 * @throws java.util.NoSuchElementException if the given index has no key associated with
 */

public void delete(int i) {
        if (i < 0 || i >= n)            throw new IllegalArgumentException();
        if (!contains(i))                       throw new NoSuchElementException("Specified index is not in the queue");
        toTheRoot(i);
        Node<Key> x = erase(i);
        if (x.child != null) {
                Node<Key> y = x;
                x = x.child;
                y.child = null;
                Node<Key> prevx = null, nextx = x.sibling;
                while (nextx != null) {
                        x.parent = null;
                        x.sibling = prevx;
                        prevx = x;
                        x = nextx; nextx = nextx.sibling;
                }
                x.parent = null;
                x.sibling = prevx;
```

```
                IndexBinomialMinPQ<Key> H = new IndexBinomialMinPQ<Key>();
                H.head = x;
                head = union(H).head;
        }
}

/**************************************************
 * General helper functions
 **************************************************/

//Compares two keys
private boolean greater(Key n, Key m) {
        if (n == null) return false;
        if (m == null) return true;
        return comparator.compare(n, m) > 0;
}

//Exchanges the positions of two nodes
private void exchange(Node<Key> x, Node<Key> y) {
        Key tempKey = x.key; x.key = y.key; y.key = tempKey;
        int tempInt = x.index; x.index = y.index; y.index = tempInt;
        nodes[x.index] = x;
        nodes[y.index] = y;
}

//Assuming root1 holds a greater key than root2, root2 becomes the new root
private void link(Node<Key> root1, Node<Key> root2) {
        root1.sibling = root2.child;
        root1.parent = root2;
        root2.child = root1;
        root2.order++;
}

/**************************************************
 * Functions for moving upward
 **************************************************/

//Moves a Node upward
private void swim(int i) {
        Node<Key> x = nodes[i];
        Node<Key> parent = x.parent;
        if (parent != null && greater(parent.key, x.key)) {
                exchange(x, parent);
                swim(i);
        }
}

//The key associated with i becomes the root of its Binomial Tree,
//regardless of the order relation defined for the keys
private void toTheRoot(int i) {
        Node<Key> x = nodes[i];
        Node<Key> parent = x.parent;
        if (parent != null) {
                exchange(x, parent);
                toTheRoot(i);
        }
}

/**************************************************
 * Functions for deleting a key
 **************************************************/

//Assuming the key associated with i is in the root list,
//deletes and return the node of index i
private Node<Key> erase(int i) {
        Node<Key> reference = nodes[i];
        Node<Key> x = head;
        Node<Key> previous = null;
        while (x != reference) {
                previous = x;
                x = x.sibling;
        }
        previous.sibling = x.sibling;
        if (x == head) head = head.sibling;
        nodes[i] = null;
        return x;
}

//Deletes and return the node containing the minimum key
private Node<Key> eraseMin() {
        Node<Key> min = head;
        Node<Key> previous = null;
        Node<Key> current = head;
        while (current.sibling != null) {
                if (greater(min.key, current.sibling.key)) {
                        previous = current;
                        min = current.sibling;
                }
                current = current.sibling;
        }
        previous.sibling = min.sibling;
        if (min == head) head = min.sibling;
        nodes[min.index] = null;
        return min;
}

/**************************************************
 * Functions for inserting a key in the heap
 **************************************************/
```

```
//Merges two root lists into one, there can be up to 2 Binomial Trees of same order
private Node<Key> merge(Node<Key> h, Node<Key> x, Node<Key> y) {
        if (x == null && y == null) return h;
        else if (x == null)             h.sibling = merge(y, null, y.sibling);
        else if (y == null)             h.sibling = merge(x, x.sibling, null);
        else if (x.order < y.order) h.sibling = merge(x, x.sibling, y);
        else                                            h.sibling = merge(y, x, y.sibling);
        return h;
}

//Merges two Binomial Heaps together and returns the resulting Binomial Heap
//It destroys the two Heaps in parameter, which should not be used any after.
//To guarantee logarithmic time, this function assumes the arrays are up-to-date
private IndexBinomialMinPQ<Key> union(IndexBinomialMinPQ<Key> heap) {
        this.head = merge(new Node<Key>(), this.head, heap.head).sibling;
        Node<Key> x = this.head;
        Node<Key> prevx = null, nextx = x.sibling;
        while (nextx != null) {
                if (x.order < nextx.order ||
                   (nextx.sibling != null && nextx.sibling.order == x.order)) {
                        prevx = x; x = nextx;
                } else if (greater(nextx.key, x.key)) {
                        x.sibling = nextx.sibling;
                        link(nextx, x);
                } else {
                        if (prevx == null) { this.head = nextx; }
                        else { prevx.sibling = nextx; }
                        link(x, nextx);
                        x = nextx;
                }
                nextx = x.sibling;
        }
        return this;
}

/*****************************************************************
 * Constructor
 *****************************************************************/

//Creates an empty heap
//The comparator is instanciated because it needs to,
//but won't be used by any heap created by this constructor
private IndexBinomialMinPQ() { comparator = null; }

/*****************************************************************
 * Iterator
 *****************************************************************/

/**
 * Gets an Iterator over the indexes in the priority queue in ascending order
 * The Iterator does not implement the remove() method
 * iterator() : Worst case is O(n)
 * next() :     Worst case is O(log(n))
 * hasNext() :  Worst case is O(1)
 * @return an Iterator over the indexes in the priority queue in ascending order
 */

public Iterator<Integer> iterator() {
        return new MyIterator();
}

private class MyIterator implements Iterator<Integer> {
        IndexBinomialMinPQ<Key> data;

        //Constructor clones recursively the elements in the queue
        //It takes linear time
        public MyIterator() {
                data = new IndexBinomialMinPQ<Key>(n, comparator);
                data.head = clone(head, null);
        }

        private Node<Key> clone(Node<Key> x, Node<Key> parent) {
                if (x == null) return null;
                Node<Key> node = new Node<Key>();
                node.index = x.index;
                node.key = x.key;
                data.nodes[node.index] = node;
                node.parent = parent;
                node.sibling = clone(x.sibling, parent);
                node.child = clone(x.child, node);
                return node;
        }

        public boolean hasNext() {
                return !data.isEmpty();
        }

        public Integer next() {
                if (!hasNext()) throw new NoSuchElementException();
                return data.delMin();
        }

        public void remove() {
                throw new UnsupportedOperationException();
        }
}

/***************************
 * Comparator
 ***************************/
```

```
        //default Comparator
        private class MyComparator implements Comparator<Key> {
                @Override
                public int compare(Key key1, Key key2) {
                        return ((Comparable<Key>) key1).compareTo(key2);
                }
        }

}
```


Last updated: Fri Jul 7 09:52:04 EDT 2017.

IndexFibonacciMinPQ.java

Below is the syntax highlighted version of IndexFibonacciMinPQ.java from §9.9 Miscellaneous.

```
/******************************************************************************
 *  Compilation: javac IndexFibonacciMinPQ.java
 *  Execution:
 *
 *  An index Fibonacci heap.
 *
 ******************************************************************************/

import java.util.Comparator;
import java.util.Iterator;
import java.util.HashMap;
import java.util.NoSuchElementException;

/*
 *  The IndexFibonacciMinPQ class represents an indexed priority queue of generic keys.
 *  It supports the usual insert and delete-the-minimum operations,
 *  along with delete and change-the-key methods.
 *  In order to let the client refer to keys on the priority queue,
 *  an integer between 0 and N-1 is associated with each key ; the client
 *  uses this integer to specify which key to delete or change.
 *  It also supports methods for peeking at the minimum key,
 *  testing if the priority queue is empty, and iterating through
 *  the keys.
 *
 *  This implementation uses a Fibonacci heap along with an array to associate
 *  keys with integers in the given range.
 *  The insert, size, is-empty, contains, minimum-index, minimum-key
 *  and key-of take constant time.
 *  The decrease-key operation takes amortized constant time.
 *  The delete, increase-key, delete-the-minimum, change-key take amortized logarithmic time.
 *  Construction takes time proportional to the specified capacity
 *
 *  @author Tristan Claverie
 */
public class IndexFibonacciMinPQ<Key> implements Iterable<Integer> {
        private Node<Key>[] nodes;                      //Array of Nodes in the heap
        private Node<Key> head;                         //Head of the circular root list
        private Node<Key> min;                          //Minimum Node in the heap
        private int size;                                       //Number of keys in the heap
        private int n;                                          //Maximum number of elements in the heap
        private final Comparator<Key> comp; //Comparator over the keys
        private HashMap<Integer, Node<Key>> table = new HashMap<Integer, Node<Key>>(); //Used for the consolidate operation

        //Represents a Node of a tree
        private class Node<Key> {
                Key key;                                                //Key of the Node
                int order;                                              //The order of the tree rooted by this Node
                int index;                                              //Index associated with the key
                Node<Key> prev, next;                   //siblings of the Node
                Node<Key> parent, child;                //parent and child of this Node
                boolean mark;                                   //Indicates if this Node already lost a child
        }

    /**
     * Initializes an empty indexed priority queue with indices between {@code 0} and {@code N-1}
     * Worst case is O(n)
     * @param N number of keys in the priority queue, index from {@code 0} to {@code N-1}
     * @throws java.lang.IllegalArgumentException if {@code N < 0}
     */
       public IndexFibonacciMinPQ(int N) {
                if (N < 0) throw new IllegalArgumentException("Cannot create a priority queue of negative size");
                n = N;
                nodes = (Node<Key>[]) new Node[n];
                comp = new MyComparator();
        }

    /**
     * Initializes an empty indexed priority queue with indices between {@code 0} and {@code N-1}
     * Worst case is O(n)
     * @param N number of keys in the priority queue, index from {@code 0} to {@code N-1}
     * @param C a Comparator over the keys
     * @throws java.lang.IllegalArgumentException if {@code N < 0}
     */
       public IndexFibonacciMinPQ(Comparator<Key> C, int N) {
                if (N < 0) throw new IllegalArgumentException("Cannot create a priority queue of negative size");
                n = N;
                nodes = (Node<Key>[]) new Node[n];
                comp = C;
        }

        /**
         * Whether the priority queue is empty
         * Worst case is O(1)
         * @return true if the priority queue is empty, false if not
         */

        public boolean isEmpty() {
                return size == 0;
        }
```

```
/**
 * Does the priority queue contains the index i ?
 * Worst case is O(1)
 * @param i an index
 * @throws java.lang.IllegalArgumentException if the specified index is invalid
 * @return true if i is on the priority queue, false if not
 */

public boolean contains(int i) {
        if (i < 0 || i >= n) throw new IllegalArgumentException();
        else                            return nodes[i] != null;
}

/**
 * Number of elements currently on the priority queue
 * Worst case is O(1)
 * @return the number of elements on the priority queue
 */

public int size() {
        return size;
}

/**
 * Associates a key with an index
 * Worst case is O(1)
 * @param i an index
 * @param key a Key associated with i
 * @throws java.lang.IllegalArgumentException if the specified index is invalid
 * @throws java.lang.IllegalArgumentException if the index is already in the queue
 */

public void insert(int i, Key key) {
        if (i < 0 || i >= n) throw new IllegalArgumentException();
        if (contains(i)) throw new IllegalArgumentException("Specified index is already in the queue");
        Node<Key> x = new Node<Key>();
        x.key = key;
        x.index = i;
        nodes[i] = x;
        size++;
        head = insert(x, head);
        if (min == null) min = head;
        else                    min = (greater(min.key, key)) ? head : min;
}

/**
 * Get the index associated with the minimum key
 * Worst case is O(1)
 * @throws java.util.NoSuchElementException if the priority queue is empty
 * @return the index associated with the minimum key
 */

public int minIndex() {
        if (isEmpty()) throw new NoSuchElementException("Priority queue is empty");
        return min.index;
}

/**
 * Get the minimum key currently in the queue
 * Worst case is O(1)
 * @throws java.util.NoSuchElementException if the priority queue is empty
 * @return the minimum key currently in the priority queue
 */

public Key minKey() {
        if (isEmpty()) throw new NoSuchElementException("Priority queue is empty");
        return min.key;
}

/**
 * Delete the minimum key
 * Worst case is O(log(n)) (amortized)
 * @throws java.util.NoSuchElementException if the priority queue is empty
 * @return the index associated with the minimum key
 */

public int delMin() {
        if (isEmpty()) throw new NoSuchElementException("Priority queue is empty");
        head = cut(min, head);
        Node<Key> x = min.child;
        int index = min.index;
        min.key = null;                                 //For garbage collection
        if (x != null) {
                do {
                        x.parent = null;
                        x = x.next;
                } while (x != min.child);
                head = meld(head, x);
                min.child = null;                       //For garbage collection
        }
        size--;
        if (!isEmpty()) consolidate();
        else                    min = null;
        nodes[index] = null;
        return index;
}

/**
 * Get the key associated with index i
 * Worst case is O(1)
```

```
     * @param i an index
     * @throws java.lang.IllegalArgumentException if the specified index is invalid
     * @throws java.util.NoSuchElementException if the index is not in the queue
     * @return the key associated with index i
     */

    public Key keyOf(int i) {
            if (i < 0 || i >= n) throw new IllegalArgumentException();
            if (!contains(i)) throw new NoSuchElementException("Specified index is not in the queue");
            return nodes[i].key;
    }

    /**
     * Changes the key associated with index i to the given key
     * If the given key is greater, Worst case is O(log(n))
     * If the given key is lower, Worst case is O(1) (amortized)
     * @param i an index
     * @param key the key to associate with i
     * @throws java.lang.IllegalArgumentException if the specified index is invalid
     * @throws java.util.NoSuchElementException if the index has no key associated with
     */

    public void changeKey(int i, Key key) {
            if (i < 0 || i >= n)            throw new IllegalArgumentException();
            if (!contains(i))                       throw new NoSuchElementException("Specified index is not in the queue");
            if (greater(key, nodes[i].key))  increaseKey(i, key);
            else                                                    decreaseKey(i, key);
    }

    /**
     * Decreases the key associated with index i to the given key
     * Worst case is O(1) (amortized).
     * @param i an index
     * @param key the key to associate with i
     * @throws java.lang.IllegalArgumentException if the specified index is invalid
     * @throws java.util.NoSuchElementException if the index has no key associated with
     * @throws java.lang.IllegalArgumentException if the given key is greater than the current key
     */

    public void decreaseKey(int i, Key key) {
            if (i < 0 || i >= n)            throw new IllegalArgumentException();
            if (!contains(i))                       throw new NoSuchElementException("Specified index is not in the queue");
            if (greater(key, nodes[i].key))  throw new IllegalArgumentException("Calling with this argument would not decrease the key"
            Node<Key> x = nodes[i];
            x.key = key;
            if (greater(min.key, key)) min = x;
            if (x.parent != null && greater(x.parent.key, key)) {
                    cut(i);
            }
    }

    /**
     * Increases the key associated with index i to the given key
     * Worst case is O(log(n))
     * @param i an index
     * @param key the key to associate with i
     * @throws java.lang.IllegalArgumentException if the specified index is invalid
     * @throws java.util.NoSuchElementException if the index has no key associated with
     * @throws java.lang.IllegalArgumentException if the given key is lower than the current key
     */

    public void increaseKey(int i, Key key) {
            if (i < 0 || i >= n)            throw new IllegalArgumentException();
            if (!contains(i))                       throw new NoSuchElementException("Specified index is not in the queue");
            if (greater(nodes[i].key, key))  throw new IllegalArgumentException("Calling with this argument would not increase the key"
            delete(i);
            insert(i, key);
    }

    /**
     * Deletes the key associated the given index
     * Worst case is O(log(n)) (amortized)
     * @param i an index
     * @throws java.lang.IllegalArgumentException if the specified index is invalid
     * @throws java.util.NoSuchElementException if the given index has no key associated with
     */

    public void delete(int i) {
            if (i < 0 || i >= n)            throw new IllegalArgumentException();
            if (!contains(i))                       throw new NoSuchElementException("Specified index is not in the queue");
            Node<Key> x = nodes[i];
            x.key = null;                           //For garbage collection
            if (x.parent != null) {
                    cut(i);
            }
            head = cut(x, head);
            if (x.child != null) {
                    Node<Key> child = x.child;
                    x.child = null;                 //For garbage collection
                    x = child;
                    do {
                            child.parent = null;
                            child = child.next;
                    } while (child != x);
                    head = meld(head, child);
            }
            if (!isEmpty()) consolidate();
            else                    min = null;
            nodes[i] = null;
            size--;
```

```
}

/************************************
 * General helper functions
 ************************************/

//Compares two keys
private boolean greater(Key n, Key m) {
        if (n == null) return false;
        if (m == null) return true;
        return comp.compare(n,  m) > 0;
}

//Assuming root1 holds a greater key than root2, root2 becomes the new root
private void link(Node<Key> root1, Node<Key> root2) {
        root1.parent = root2;
        root2.child = insert(root1, root2.child);
        root2.order++;
}

/************************************
 * Function for decreasing a key
 ************************************/

//Removes a Node from its parent's child list and insert it in the root list
//If the parent Node already lost a child, reshapes the heap accordingly
private void cut(int i) {
        Node<Key> x = nodes[i];
        Node<Key> parent = x.parent;
        parent.child = cut(x, parent.child);
        x.parent = null;
        parent.order--;
        head = insert(x, head);
        parent.mark = !parent.mark;
        if (!parent.mark && parent.parent != null) {
                cut(parent.index);}
}

/************************************
 * Function for consolidating all trees in the root list
 ************************************/

//Coalesces the roots, thus reshapes the heap
//Caching a HashMap improves greatly performances
private void consolidate() {
        table.clear();
        Node<Key> x = head;
        int maxOrder = 0;
        min = head;
        Node<Key> y = null, z = null;
        do {
                y = x;
                x = x.next;
                z = table.get(y.order);
                while (z != null) {
                        table.remove(y.order);
                        if (greater(y.key, z.key)) {
                                link(y, z);
                                y = z;
                        } else {
                                link(z, y);
                        }
                        z = table.get(y.order);
                }
                table.put(y.order, y);
                if (y.order > maxOrder) maxOrder = y.order;
        } while (x != head);
        head = null;
        for (Node<Key> n : table.values()) {
                min = greater(min.key, n.key) ? n : min;
                head = insert(n, head);
        }
}

/************************************
 * General helper functions for manipulating circular lists
 ************************************/

//Inserts a Node in a circular list containing head, returns a new head
private Node<Key> insert(Node<Key> x, Node<Key> head) {
        if (head == null) {
                x.prev = x;
                x.next = x;
        } else {
                head.prev.next = x;
                x.next = head;
                x.prev = head.prev;
                head.prev = x;
        }
        return x;
}

//Removes a tree from the list defined by the head pointer
private Node<Key> cut(Node<Key> x, Node<Key> head) {
        if (x.next == x) {
                x.next = null;
                x.prev = null;
                return null;
        } else {
                x.next.prev = x.prev;
```

```
                x.prev.next = x.next;
                Node<Key> res = x.next;
                x.next = null;
                x.prev = null;
                if (head == x)  return res;
                else                    return head;
            }
        }

        //Merges two lists together.
        private Node<Key> meld(Node<Key> x, Node<Key> y) {
            if (x == null) return y;
            if (y == null) return x;
            x.prev.next = y.next;
            y.next.prev = x.prev;
            x.prev = y;
            y.next = x;
            return x;
        }

        /***********************************
         * Iterator
         ***********************************/

        /**
         * Get an Iterator over the indexes in the priority queue in ascending order
         * The Iterator does not implement the remove() method
         * iterator() : Worst case is O(n)
         * next() :     Worst case is O(log(n)) (amortized)
         * hasNext() :  Worst case is O(1)
         * @return an Iterator over the indexes in the priority queue in ascending order
         */

        public Iterator<Integer> iterator() {
            return new MyIterator();
        }

        private class MyIterator implements Iterator<Integer> {
            private IndexFibonacciMinPQ<Key> copy;


            //Constructor takes linear time
            public MyIterator() {
                copy = new IndexFibonacciMinPQ<Key>(comp, n);
                for (Node<Key> x : nodes) {
                        if (x != null) copy.insert(x.index, x.key);
                }
            }

            public void remove() {
                throw new UnsupportedOperationException();
            }

            public boolean hasNext() {
                return !copy.isEmpty();
            }

            //Takes amortized logarithmic time
            public Integer next() {
                if (!hasNext()) throw new NoSuchElementException();
                return copy.delMin();
            }
        }

        /***************************
         * Comparator
         ***************************/

        //default Comparator
        private class MyComparator implements Comparator<Key> {
            @Override
            public int compare(Key key1, Key key2) {
                return ((Comparable<Key>) key1).compareTo(key2);
            }
        }

}
```

SequentialSearchST.java

Below is the syntax highlighted version of SequentialSearchST.java from §3.1 Elementary Symbol Tables.

```
/******************************************************************************
 *  Compilation:  javac SequentialSearchST.java
 *  Execution:    java SequentialSearchST
 *  Dependencies: StdIn.java StdOut.java
 *  Data files:   http://algs4.cs.princeton.edu/31elementary/tinyST.txt
 *
 *  Symbol table implementation with sequential search in an
 *  unordered linked list of key-value pairs.
 *
 *  % more tinyST.txt
 *  S E A R C H E X A M P L E
 *
 *  % java SequentialSearchST < tiny.txt
 *  L 11
 *  P 10
 *  M 9
 *  X 7
 *  H 5
 *  C 4
 *  R 3
 *  A 8
 *  E 12
 *  S 0
 *
 ******************************************************************************/

/**
 *  The {@code SequentialSearchST} class represents an (unordered)
 *  symbol table of generic key-value pairs.
 *  It supports the usual <em>put</em>, <em>get</em>, <em>contains</em>,
 *  <em>delete</em>, <em>size</em>, and <em>is-empty</em> methods.
 *  It also provides a <em>keys</em> method for iterating over all of the keys.
 *  A symbol table implements the <em>associative array</em> abstraction:
 *  when associating a value with a key that is already in the symbol table,
 *  the convention is to replace the old value with the new value.
 *  The class also uses the convention that values cannot be {@code null}. Setting the
 *  value associated with a key to {@code null} is equivalent to deleting the key
 *  from the symbol table.
 *  <p>
 *  This implementation uses a singly-linked list and sequential search.
 *  It relies on the {@code equals()} method to test whether two keys
 *  are equal. It does not call either the {@code compareTo()} or
 *  {@code hashCode()} method.
 *  The <em>put</em> and <em>delete</em> operations take linear time; the
 *  <em>get</em> and <em>contains</em> operations takes linear time in the worst case.
 *  The <em>size</em>, and <em>is-empty</em> operations take constant time.
 *  Construction takes constant time.
 *  <p>
 *  For additional documentation, see <a href="http://algs4.cs.princeton.edu/31elementary">Section 3.1</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class SequentialSearchST<Key, Value> {
    private int n;           // number of key-value pairs
    private Node first;      // the linked list of key-value pairs

    // a helper linked list data type
    private class Node {
        private Key key;
        private Value val;
        private Node next;

        public Node(Key key, Value val, Node next)  {
            this.key  = key;
            this.val  = val;
            this.next = next;
        }
```

```
}

/**
 * Initializes an empty symbol table.
 */
public SequentialSearchST() {
}

/**
 * Returns the number of key-value pairs in this symbol table.
 *
 * @return the number of key-value pairs in this symbol table
 */
public int size() {
    return n;
}

/**
 * Returns true if this symbol table is empty.
 *
 * @return {@code true} if this symbol table is empty;
 *         {@code false} otherwise
 */
public boolean isEmpty() {
    return size() == 0;
}

/**
 * Returns true if this symbol table contains the specified key.
 *
 * @param  key the key
 * @return {@code true} if this symbol table contains {@code key};
 *         {@code false} otherwise
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public boolean contains(Key key) {
    if (key == null) throw new IllegalArgumentException("argument to contains() is null");
    return get(key) != null;
}

/**
 * Returns the value associated with the given key in this symbol table.
 *
 * @param  key the key
 * @return the value associated with the given key if the key is in the symbol table
 *     and {@code null} if the key is not in the symbol table
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public Value get(Key key) {
    if (key == null) throw new IllegalArgumentException("argument to get() is null");
    for (Node x = first; x != null; x = x.next) {
        if (key.equals(x.key))
            return x.val;
    }
    return null;
}

/**
 * Inserts the specified key-value pair into the symbol table, overwriting the old
 * value with the new value if the symbol table already contains the specified key.
 * Deletes the specified key (and its associated value) from this symbol table
 * if the specified value is {@code null}.
 *
 * @param  key the key
 * @param  val the value
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public void put(Key key, Value val) {
    if (key == null) throw new IllegalArgumentException("first argument to put() is null");
    if (val == null) {
        delete(key);
        return;
    }

    for (Node x = first; x != null; x = x.next) {
        if (key.equals(x.key)) {
            x.val = val;
            return;
        }
```

```
        }
        first = new Node(key, val, first);
        n++;
    }

    /**
     * Removes the specified key and its associated value from this symbol table
     * (if the key is in this symbol table).
     *
     * @param  key the key
     * @throws IllegalArgumentException if {@code key} is {@code null}
     */
    public void delete(Key key) {
        if (key == null) throw new IllegalArgumentException("argument to delete() is null");
        first = delete(first, key);
    }

    // delete key in linked list beginning at Node x
    // warning: function call stack too large if table is large
    private Node delete(Node x, Key key) {
        if (x == null) return null;
        if (key.equals(x.key)) {
            n--;
            return x.next;
        }
        x.next = delete(x.next, key);
        return x;
    }


    /**
     * Returns all keys in the symbol table as an {@code Iterable}.
     * To iterate over all of the keys in the symbol table named {@code st},
     * use the foreach notation: {@code for (Key key : st.keys())}.
     *
     * @return all keys in the symbol table
     */
    public Iterable<Key> keys()  {
        Queue<Key> queue = new Queue<Key>();
        for (Node x = first; x != null; x = x.next)
            queue.enqueue(x.key);
        return queue;
    }


    /**
     * Unit tests the {@code SequentialSearchST} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        SequentialSearchST<String, Integer> st = new SequentialSearchST<String, Integer>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
            String key = StdIn.readString();
            st.put(key, i);
        }
        for (String s : st.keys())
            StdOut.println(s + " " + st.get(s));
    }
}
```

BinarySearchST.java

Below is the syntax highlighted version of BinarySearchST.java from §3.1 Elementary Symbol Tables.

```
/******************************************************************************
 *  Compilation:  javac BinarySearchST.java
 *  Execution:    java BinarySearchST
 *  Dependencies: StdIn.java StdOut.java
 *  Data files:   http://algs4.cs.princeton.edu/31elementary/tinyST.txt
 *
 *  Symbol table implementation with binary search in an ordered array.
 *
 *  % more tinyST.txt
 *  S E A R C H E X A M P L E
 *
 *  % java BinarySearchST < tinyST.txt
 *  A 8
 *  C 4
 *  E 12
 *  H 5
 *  L 11
 *  M 9
 *  P 10
 *  R 3
 *  S 0
 *  X 7
 *
 ******************************************************************************/

import java.util.NoSuchElementException;

/**
 *  The {@code BST} class represents an ordered symbol table of generic
 *  key-value pairs.
 *  It supports the usual <em>put</em>, <em>get</em>, <em>contains</em>,
 *  <em>delete</em>, <em>size</em>, and <em>is-empty</em> methods.
 *  It also provides ordered methods for finding the <em>minimum</em>,
 *  <em>maximum</em>, <em>floor</em>, <em>select</em>, and <em>ceiling</em>.
 *  It also provides a <em>keys</em> method for iterating over all of the keys.
 *  A symbol table implements the <em>associative array</em> abstraction:
 *  when associating a value with a key that is already in the symbol table,
 *  the convention is to replace the old value with the new value.
 *  Unlike {@link java.util.Map}, this class uses the convention that
 *  values cannot be {@code null}—setting the
 *  value associated with a key to {@code null} is equivalent to deleting the key
 *  from the symbol table.
 *  <p>
 *  This implementation uses a sorted array. It requires that
 *  the key type implements the {@code Comparable} interface and calls the
 *  {@code compareTo()} and method to compare two keys. It does not call either
 *  {@code equals()} or {@code hashCode()}.
 *  The <em>put</em> and <em>remove</em> operations each take linear time in
 *  the worst case; the <em>contains</em>, <em>ceiling</em>, <em>floor</em>,
 *  and <em>rank</em> operations take logarithmic time; the <em>size</em>,
 *  <em>is-empty</em>, <em>minimum</em>, <em>maximum</em>, and <em>select</em>
 *  operations take constant time. Construction takes constant time.
 *  <p>
 *  For additional documentation, see <a href="http://algs4.cs.princeton.edu/31elementary">Section 3.1</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *  For other implementations, see {@link ST}, {@link BST},
 *  {@link SequentialSearchST}, {@link RedBlackBST},
 *  {@link SeparateChainingHashST}, and {@link LinearProbingHashST},
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 */
public class BinarySearchST<Key extends Comparable<Key>, Value> {
    private static final int INIT_CAPACITY = 2;
    private Key[] keys;
    private Value[] vals;
    private int n = 0;

    /**
     * Initializes an empty symbol table.
     */
```

```
public BinarySearchST() {
    this(INIT_CAPACITY);
}

/**
 * Initializes an empty symbol table with the specified initial capacity.
 * @param capacity the maximum capacity
 */
public BinarySearchST(int capacity) {
    keys = (Key[]) new Comparable[capacity];
    vals = (Value[]) new Object[capacity];
}

// resize the underlying arrays
private void resize(int capacity) {
    assert capacity >= n;
    Key[]   tempk = (Key[])   new Comparable[capacity];
    Value[] tempv = (Value[]) new Object[capacity];
    for (int i = 0; i < n; i++) {
        tempk[i] = keys[i];
        tempv[i] = vals[i];
    }
    vals = tempv;
    keys = tempk;
}

/**
 * Returns the number of key-value pairs in this symbol table.
 *
 * @return the number of key-value pairs in this symbol table
 */
public int size() {
    return n;
}

/**
 * Returns true if this symbol table is empty.
 *
 * @return {@code true} if this symbol table is empty;
 *         {@code false} otherwise
 */
public boolean isEmpty() {
    return size() == 0;
}


/**
 * Does this symbol table contain the given key?
 *
 * @param  key the key
 * @return {@code true} if this symbol table contains {@code key} and
 *         {@code false} otherwise
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public boolean contains(Key key) {
    if (key == null) throw new IllegalArgumentException("argument to contains() is null");
    return get(key) != null;
}

/**
 * Returns the value associated with the given key in this symbol table.
 *
 * @param  key the key
 * @return the value associated with the given key if the key is in the symbol table
 *         and {@code null} if the key is not in the symbol table
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public Value get(Key key) {
    if (key == null) throw new IllegalArgumentException("argument to get() is null");
    if (isEmpty()) return null;
    int i = rank(key);
    if (i < n && keys[i].compareTo(key) == 0) return vals[i];
    return null;
}

/**
 * Returns the number of keys in this symbol table strictly less than {@code key}.
 *
 * @param  key the key
```

```
     * @return the number of keys in the symbol table strictly less than {@code key}
     * @throws IllegalArgumentException if {@code key} is {@code null}
     */
    public int rank(Key key) {
        if (key == null) throw new IllegalArgumentException("argument to rank() is null");

        int lo = 0, hi = n-1;
        while (lo <= hi) {
            int mid = lo + (hi - lo) / 2;
            int cmp = key.compareTo(keys[mid]);
            if      (cmp < 0) hi = mid - 1;
            else if (cmp > 0) lo = mid + 1;
            else return mid;
        }
        return lo;
    }



    /**
     * Inserts the specified key-value pair into the symbol table, overwriting the old
     * value with the new value if the symbol table already contains the specified key.
     * Deletes the specified key (and its associated value) from this symbol table
     * if the specified value is {@code null}.
     *
     * @param  key the key
     * @param  val the value
     * @throws IllegalArgumentException if {@code key} is {@code null}
     */
    public void put(Key key, Value val)  {
        if (key == null) throw new IllegalArgumentException("first argument to put() is null");

        if (val == null) {
            delete(key);
            return;
        }

        int i = rank(key);

        // key is already in table
        if (i < n && keys[i].compareTo(key) == 0) {
            vals[i] = val;
            return;
        }

        // insert new key-value pair
        if (n == keys.length) resize(2*keys.length);

        for (int j = n; j > i; j--)  {
            keys[j] = keys[j-1];
            vals[j] = vals[j-1];
        }
        keys[i] = key;
        vals[i] = val;
        n++;

        assert check();
    }

    /**
     * Removes the specified key and associated value from this symbol table
     * (if the key is in the symbol table).
     *
     * @param  key the key
     * @throws IllegalArgumentException if {@code key} is {@code null}
     */
    public void delete(Key key) {
        if (key == null) throw new IllegalArgumentException("argument to delete() is null");
        if (isEmpty()) return;

        // compute rank
        int i = rank(key);

        // key not in table
        if (i == n || keys[i].compareTo(key) != 0) {
            return;
        }

        for (int j = i; j < n-1; j++)  {
```

```
            keys[j] = keys[j+1];
            vals[j] = vals[j+1];
        }

        n--;
        keys[n] = null;  // to avoid loitering
        vals[n] = null;

        // resize if 1/4 full
        if (n > 0 && n == keys.length/4) resize(keys.length/2);

        assert check();
    }

    /**
     * Removes the smallest key and associated value from this symbol table.
     *
     * @throws NoSuchElementException if the symbol table is empty
     */
    public void deleteMin() {
        if (isEmpty()) throw new NoSuchElementException("Symbol table underflow error");
        delete(min());
    }

    /**
     * Removes the largest key and associated value from this symbol table.
     *
     * @throws NoSuchElementException if the symbol table is empty
     */
    public void deleteMax() {
        if (isEmpty()) throw new NoSuchElementException("Symbol table underflow error");
        delete(max());
    }


   /***************************************************************************
    *  Ordered symbol table methods.
    ***************************************************************************/

   /**
     * Returns the smallest key in this symbol table.
     *
     * @return the smallest key in this symbol table
     * @throws NoSuchElementException if this symbol table is empty
     */
    public Key min() {
        if (isEmpty()) throw new NoSuchElementException("called min() with empty symbol table");
        return keys[0];
    }

    /**
     * Returns the largest key in this symbol table.
     *
     * @return the largest key in this symbol table
     * @throws NoSuchElementException if this symbol table is empty
     */
    public Key max() {
        if (isEmpty()) throw new NoSuchElementException("called max() with empty symbol table");
        return keys[n-1];
    }

    /**
     * Return the kth smallest key in this symbol table.
     *
     * @param  k the order statistic
     * @return the {@code k}th smallest key in this symbol table
     * @throws IllegalArgumentException unless {@code k} is between 0 and
     *        <em>n</em>-1
     */
    public Key select(int k) {
        if (k < 0 || k >= size()) {
            throw new IllegalArgumentException("called select() with invalid argument: " + k);
        }
        return keys[k];
    }

    /**
     * Returns the largest key in this symbol table less than or equal to {@code key}.
     *
```

```
     * @param  key the key
     * @return the largest key in this symbol table less than or equal to {@code key}
     * @throws NoSuchElementException if there is no such key
     * @throws IllegalArgumentException if {@code key} is {@code null}
     */
    public Key floor(Key key) {
        if (key == null) throw new IllegalArgumentException("argument to floor() is null");
        int i = rank(key);
        if (i < n && key.compareTo(keys[i]) == 0) return keys[i];
        if (i == 0) return null;
        else return keys[i-1];
    }

    /**
     * Returns the smallest key in this symbol table greater than or equal to {@code key}.
     *
     * @param  key the key
     * @return the smallest key in this symbol table greater than or equal to {@code key}
     * @throws NoSuchElementException if there is no such key
     * @throws IllegalArgumentException if {@code key} is {@code null}
     */
    public Key ceiling(Key key) {
        if (key == null) throw new IllegalArgumentException("argument to ceiling() is null");
        int i = rank(key);
        if (i == n) return null;
        else return keys[i];
    }

    /**
     * Returns the number of keys in this symbol table in the specified range.
     *
     * @param lo minimum endpoint
     * @param hi maximum endpoint
     * @return the number of keys in this symbol table between {@code lo}
     *         (inclusive) and {@code hi} (inclusive)
     * @throws IllegalArgumentException if either {@code lo} or {@code hi}
     *         is {@code null}
     */
    public int size(Key lo, Key hi) {
        if (lo == null) throw new IllegalArgumentException("first argument to size() is null");
        if (hi == null) throw new IllegalArgumentException("second argument to size() is null");

        if (lo.compareTo(hi) > 0) return 0;
        if (contains(hi)) return rank(hi) - rank(lo) + 1;
        else              return rank(hi) - rank(lo);
    }

    /**
     * Returns all keys in this symbol table as an {@code Iterable}.
     * To iterate over all of the keys in the symbol table named {@code st},
     * use the foreach notation: {@code for (Key key : st.keys())}.
     *
     * @return all keys in this symbol table
     */
    public Iterable<Key> keys() {
        return keys(min(), max());
    }

    /**
     * Returns all keys in this symbol table in the given range,
     * as an {@code Iterable}.
     *
     * @param lo minimum endpoint
     * @param hi maximum endpoint
     * @return all keys in this symbol table between {@code lo}
     *         (inclusive) and {@code hi} (inclusive)
     * @throws IllegalArgumentException if either {@code lo} or {@code hi}
     *         is {@code null}
     */
    public Iterable<Key> keys(Key lo, Key hi) {
        if (lo == null) throw new IllegalArgumentException("first argument to keys() is null");
        if (hi == null) throw new IllegalArgumentException("second argument to keys() is null");

        Queue<Key> queue = new Queue<Key>();
        if (lo.compareTo(hi) > 0) return queue;
        for (int i = rank(lo); i < rank(hi); i++)
            queue.enqueue(keys[i]);
        if (contains(hi)) queue.enqueue(keys[rank(hi)]);
        return queue;
```

```
    }


   /***************************************************************************
    *  Check internal invariants.
    ***************************************************************************/

    private boolean check() {
        return isSorted() && rankCheck();
    }

    // are the items in the array in ascending order?
    private boolean isSorted() {
        for (int i = 1; i < size(); i++)
            if (keys[i].compareTo(keys[i-1]) < 0) return false;
        return true;
    }

    // check that rank(select(i)) = i
    private boolean rankCheck() {
        for (int i = 0; i < size(); i++)
            if (i != rank(select(i))) return false;
        for (int i = 0; i < size(); i++)
            if (keys[i].compareTo(select(rank(keys[i]))) != 0) return false;
        return true;
    }


    /**
     * Unit tests the {@code BinarySearchST} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        BinarySearchST<String, Integer> st = new BinarySearchST<String, Integer>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
            String key = StdIn.readString();
            st.put(key, i);
        }
        for (String s : st.keys())
            StdOut.println(s + " " + st.get(s));
    }
}
```

BST.java

Below is the syntax highlighted version of BST.java from §3.2 Binary Search Trees.

```
/******************************************************************************
 *  Compilation:  javac BST.java
 *  Execution:    java BST
 *  Dependencies: StdIn.java StdOut.java Queue.java
 *  Data files:   http://algs4.cs.princeton.edu/32bst/tinyST.txt
 *
 *  A symbol table implemented with a binary search tree.
 *
 *  % more tinyST.txt
 *  S E A R C H E X A M P L E
 *
 *  % java BST < tinyST.txt
 *  A 8
 *  C 4
 *  E 12
 *  H 5
 *  L 11
 *  M 9
 *  P 10
 *  R 3
 *  S 0
 *  X 7
 *
 ******************************************************************************/

import java.util.NoSuchElementException;

/**
 *  The {@code BST} class represents an ordered symbol table of generic
 *  key-value pairs.
 *  It supports the usual <em>put</em>, <em>get</em>, <em>contains</em>,
 *  <em>delete</em>, <em>size</em>, and <em>is-empty</em> methods.
 *  It also provides ordered methods for finding the <em>minimum</em>,
 *  <em>maximum</em>, <em>floor</em>, <em>select</em>, <em>ceiling</em>.
 *  It also provides a <em>keys</em> method for iterating over all of the keys.
 *  A symbol table implements the <em>associative array</em> abstraction:
 *  when associating a value with a key that is already in the symbol table,
 *  the convention is to replace the old value with the new value.
 *  Unlike {@link java.util.Map}, this class uses the convention that
 *  values cannot be {@code null}—setting the
 *  value associated with a key to {@code null} is equivalent to deleting the key
 *  from the symbol table.
 *  <p>
 *  This implementation uses an (unbalanced) binary search tree. It requires that
 *  the key type implements the {@code Comparable} interface and calls the
 *  {@code compareTo()} and method to compare two keys. It does not call either
 *  {@code equals()} or {@code hashCode()}.
 *  The <em>put</em>, <em>contains</em>, <em>remove</em>, <em>minimum</em>,
 *  <em>maximum</em>, <em>ceiling</em>, <em>floor</em>, <em>select</em>, and
 *  <em>rank</em>  operations each take
 *  linear time in the worst case, if the tree becomes unbalanced.
 *  The <em>size</em>, and <em>is-empty</em> operations take constant time.
 *  Construction takes constant time.
 *  <p>
 *  For additional documentation, see <a href="http://algs4.cs.princeton.edu/32bst">Section 3.2</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *  For other implementations, see {@link ST}, {@link BinarySearchST},
 *  {@link SequentialSearchST}, {@link RedBlackBST},
 *  {@link SeparateChainingHashST}, and {@link LinearProbingHashST},
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class BST<Key extends Comparable<Key>, Value> {
```

```
private Node root;             // root of BST

private class Node {
    private Key key;           // sorted by key
    private Value val;         // associated data
    private Node left, right;  // left and right subtrees
    private int size;          // number of nodes in subtree

    public Node(Key key, Value val, int size) {
        this.key = key;
        this.val = val;
        this.size = size;
    }
}

/**
 * Initializes an empty symbol table.
 */
public BST() {
}

/**
 * Returns true if this symbol table is empty.
 * @return {@code true} if this symbol table is empty; {@code false} otherwise
 */
public boolean isEmpty() {
    return size() == 0;
}

/**
 * Returns the number of key-value pairs in this symbol table.
 * @return the number of key-value pairs in this symbol table
 */
public int size() {
    return size(root);
}

// return number of key-value pairs in BST rooted at x
private int size(Node x) {
    if (x == null) return 0;
    else return x.size;
}

/**
 * Does this symbol table contain the given key?
 *
 * @param  key the key
 * @return {@code true} if this symbol table contains {@code key} and
 *         {@code false} otherwise
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public boolean contains(Key key) {
    if (key == null) throw new IllegalArgumentException("argument to contains() is null");
    return get(key) != null;
}

/**
 * Returns the value associated with the given key.
 *
 * @param  key the key
 * @return the value associated with the given key if the key is in the symbol table
 *         and {@code null} if the key is not in the symbol table
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public Value get(Key key) {
    return get(root, key);
}

private Value get(Node x, Key key) {
    if (key == null) throw new IllegalArgumentException("called get() with a null key");
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if      (cmp < 0) return get(x.left, key);
    else if (cmp > 0) return get(x.right, key);
```

```
    else              return x.val;
}

/**
 * Inserts the specified key-value pair into the symbol table, overwriting the old
 * value with the new value if the symbol table already contains the specified key.
 * Deletes the specified key (and its associated value) from this symbol table
 * if the specified value is {@code null}.
 *
 * @param  key the key
 * @param  val the value
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public void put(Key key, Value val) {
    if (key == null) throw new IllegalArgumentException("calledput() with a null key");
    if (val == null) {
        delete(key);
        return;
    }
    root = put(root, key, val);
    assert check();
}

private Node put(Node x, Key key, Value val) {
    if (x == null) return new Node(key, val, 1);
    int cmp = key.compareTo(x.key);
    if      (cmp < 0) x.left  = put(x.left,  key, val);
    else if (cmp > 0) x.right = put(x.right, key, val);
    else              x.val   = val;
    x.size = 1 + size(x.left) + size(x.right);
    return x;
}


/**
 * Removes the smallest key and associated value from the symbol table.
 *
 * @throws NoSuchElementException if the symbol table is empty
 */
public void deleteMin() {
    if (isEmpty()) throw new NoSuchElementException("Symbol table underflow");
    root = deleteMin(root);
    assert check();
}

private Node deleteMin(Node x) {
    if (x.left == null) return x.right;
    x.left = deleteMin(x.left);
    x.size = size(x.left) + size(x.right) + 1;
    return x;
}

/**
 * Removes the largest key and associated value from the symbol table.
 *
 * @throws NoSuchElementException if the symbol table is empty
 */
public void deleteMax() {
    if (isEmpty()) throw new NoSuchElementException("Symbol table underflow");
    root = deleteMax(root);
    assert check();
}

private Node deleteMax(Node x) {
    if (x.right == null) return x.left;
    x.right = deleteMax(x.right);
    x.size = size(x.left) + size(x.right) + 1;
    return x;
}

/**
 * Removes the specified key and its associated value from this symbol table
 * (if the key is in this symbol table).
 *
```

```
     * @param  key the key
     * @throws IllegalArgumentException if {@code key} is {@code null}
     */
    public void delete(Key key) {
        if (key == null) throw new IllegalArgumentException("called delete() with a null key");
        root = delete(root, key);
        assert check();
    }

    private Node delete(Node x, Key key) {
        if (x == null) return null;

        int cmp = key.compareTo(x.key);
        if      (cmp < 0) x.left  = delete(x.left,  key);
        else if (cmp > 0) x.right = delete(x.right, key);
        else {
            if (x.right == null) return x.left;
            if (x.left  == null) return x.right;
            Node t = x;
            x = min(t.right);
            x.right = deleteMin(t.right);
            x.left = t.left;
        }
        x.size = size(x.left) + size(x.right) + 1;
        return x;
    }


    /**
     * Returns the smallest key in the symbol table.
     *
     * @return the smallest key in the symbol table
     * @throws NoSuchElementException if the symbol table is empty
     */
    public Key min() {
        if (isEmpty()) throw new NoSuchElementException("called min() with empty symbol table");
        return min(root).key;
    }

    private Node min(Node x) {
        if (x.left == null) return x;
        else                return min(x.left);
    }

    /**
     * Returns the largest key in the symbol table.
     *
     * @return the largest key in the symbol table
     * @throws NoSuchElementException if the symbol table is empty
     */
    public Key max() {
        if (isEmpty()) throw new NoSuchElementException("called max() with empty symbol table");
        return max(root).key;
    }

    private Node max(Node x) {
        if (x.right == null) return x;
        else                 return max(x.right);
    }

    /**
     * Returns the largest key in the symbol table less than or equal to {@code key}.
     *
     * @param  key the key
     * @return the largest key in the symbol table less than or equal to {@code key}
     * @throws NoSuchElementException if there is no such key
     * @throws IllegalArgumentException if {@code key} is {@code null}
     */
    public Key floor(Key key) {
        if (key == null) throw new IllegalArgumentException("argument to floor() is null");
        if (isEmpty()) throw new NoSuchElementException("called floor() with empty symbol table");
        Node x = floor(root, key);
        if (x == null) return null;
        else return x.key;
```

```
}

private Node floor(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp == 0) return x;
    if (cmp <  0) return floor(x.left, key);
    Node t = floor(x.right, key);
    if (t != null) return t;
    else return x;
}

/**
 * Returns the smallest key in the symbol table greater than or equal to {@code key}.
 *
 * @param  key the key
 * @return the smallest key in the symbol table greater than or equal to {@code key}
 * @throws NoSuchElementException if there is no such key
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public Key ceiling(Key key) {
    if (key == null) throw new IllegalArgumentException("argument to ceiling() is null");
    if (isEmpty()) throw new NoSuchElementException("called ceiling() with empty symbol table");
    Node x = ceiling(root, key);
    if (x == null) return null;
    else return x.key;
}

private Node ceiling(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp == 0) return x;
    if (cmp < 0) {
        Node t = ceiling(x.left, key);
        if (t != null) return t;
        else return x;
    }
    return ceiling(x.right, key);
}

/**
 * Return the kth smallest key in the symbol table.
 *
 * @param  k the order statistic
 * @return the {@code k}th smallest key in the symbol table
 * @throws IllegalArgumentException unless {@code k} is between 0 and
 *        <em>n</em>-1
 */
public Key select(int k) {
    if (k < 0 || k >= size()) {
        throw new IllegalArgumentException("called select() with invalid argument: " + k);
    }
    Node x = select(root, k);
    return x.key;
}

// Return key of rank k.
private Node select(Node x, int k) {
    if (x == null) return null;
    int t = size(x.left);
    if      (t > k) return select(x.left,  k);
    else if (t < k) return select(x.right, k-t-1);
    else            return x;
}

/**
 * Return the number of keys in the symbol table strictly less than {@code key}.
 *
 * @param  key the key
 * @return the number of keys in the symbol table strictly less than {@code key}
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public int rank(Key key) {
    if (key == null) throw new IllegalArgumentException("argument to rank() is null");
```

```
    return rank(key, root);
}

// Number of keys in the subtree less than key.
private int rank(Key key, Node x) {
    if (x == null) return 0;
    int cmp = key.compareTo(x.key);
    if      (cmp < 0) return rank(key, x.left);
    else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);
    else              return size(x.left);
}

/**
 * Returns all keys in the symbol table as an {@code Iterable}.
 * To iterate over all of the keys in the symbol table named {@code st},
 * use the foreach notation: {@code for (Key key : st.keys())}.
 *
 * @return all keys in the symbol table
 */
public Iterable<Key> keys() {
    return keys(min(), max());
}

/**
 * Returns all keys in the symbol table in the given range,
 * as an {@code Iterable}.
 *
 * @param  lo minimum endpoint
 * @param  hi maximum endpoint
 * @return all keys in the symbol table between {@code lo}
 *         (inclusive) and {@code hi} (inclusive)
 * @throws IllegalArgumentException if either {@code lo} or {@code hi}
 *         is {@code null}
 */
public Iterable<Key> keys(Key lo, Key hi) {
    if (lo == null) throw new IllegalArgumentException("first argument to keys() is null");
    if (hi == null) throw new IllegalArgumentException("second argument to keys() is null");

    Queue<Key> queue = new Queue<Key>();
    keys(root, queue, lo, hi);
    return queue;
}

private void keys(Node x, Queue<Key> queue, Key lo, Key hi) {
    if (x == null) return;
    int cmplo = lo.compareTo(x.key);
    int cmphi = hi.compareTo(x.key);
    if (cmplo < 0) keys(x.left, queue, lo, hi);
    if (cmplo <= 0 && cmphi >= 0) queue.enqueue(x.key);
    if (cmphi > 0) keys(x.right, queue, lo, hi);
}

/**
 * Returns the number of keys in the symbol table in the given range.
 *
 * @param  lo minimum endpoint
 * @param  hi maximum endpoint
 * @return the number of keys in the symbol table between {@code lo}
 *         (inclusive) and {@code hi} (inclusive)
 * @throws IllegalArgumentException if either {@code lo} or {@code hi}
 *         is {@code null}
 */
public int size(Key lo, Key hi) {
    if (lo == null) throw new IllegalArgumentException("first argument to size() is null");
    if (hi == null) throw new IllegalArgumentException("second argument to size() is null");

    if (lo.compareTo(hi) > 0) return 0;
    if (contains(hi)) return rank(hi) - rank(lo) + 1;
    else              return rank(hi) - rank(lo);
}

/**
 * Returns the height of the BST (for debugging).
 *
```

```
     * @return the height of the BST (a 1-node tree has height 0)
     */
    public int height() {
        return height(root);
    }
    private int height(Node x) {
        if (x == null) return -1;
        return 1 + Math.max(height(x.left), height(x.right));
    }

    /**
     * Returns the keys in the BST in level order (for debugging).
     *
     * @return the keys in the BST in level order traversal
     */
    public Iterable<Key> levelOrder() {
        Queue<Key> keys = new Queue<Key>();
        Queue<Node> queue = new Queue<Node>();
        queue.enqueue(root);
        while (!queue.isEmpty()) {
            Node x = queue.dequeue();
            if (x == null) continue;
            keys.enqueue(x.key);
            queue.enqueue(x.left);
            queue.enqueue(x.right);
        }
        return keys;
    }

/*************************************************************************
     *  Check integrity of BST data structure.
     *************************************************************************/
    private boolean check() {
        if (!isBST())            StdOut.println("Not in symmetric order");
        if (!isSizeConsistent()) StdOut.println("Subtree counts not consistent");
        if (!isRankConsistent()) StdOut.println("Ranks not consistent");
        return isBST() && isSizeConsistent() && isRankConsistent();
    }

    // does this binary tree satisfy symmetric order?
    // Note: this test also ensures that data structure is a binary tree since order is strict
    private boolean isBST() {
        return isBST(root, null, null);
    }

    // is the tree rooted at x a BST with all keys strictly between min and max
    // (if min or max is null, treat as empty constraint)
    // Credit: Bob Dondero's elegant solution
    private boolean isBST(Node x, Key min, Key max) {
        if (x == null) return true;
        if (min != null && x.key.compareTo(min) <= 0) return false;
        if (max != null && x.key.compareTo(max) >= 0) return false;
        return isBST(x.left, min, x.key) && isBST(x.right, x.key, max);
    }

    // are the size fields correct?
    private boolean isSizeConsistent() { return isSizeConsistent(root); }
    private boolean isSizeConsistent(Node x) {
        if (x == null) return true;
        if (x.size != size(x.left) + size(x.right) + 1) return false;
        return isSizeConsistent(x.left) && isSizeConsistent(x.right);
    }

    // check that ranks are consistent
    private boolean isRankConsistent() {
        for (int i = 0; i < size(); i++)
            if (i != rank(select(i))) return false;
        for (Key key : keys())
            if (key.compareTo(select(rank(key))) != 0) return false;
        return true;
    }


    /**
```

```
     * Unit tests the {@code BST} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        BST<String, Integer> st = new BST<String, Integer>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
            String key = StdIn.readString();
            st.put(key, i);
        }

        for (String s : st.levelOrder())
            StdOut.println(s + " " + st.get(s));

        StdOut.println();

        for (String s : st.keys())
            StdOut.println(s + " " + st.get(s));
    }
}
```


Last updated: Fri Mar 10 21:01:06 EST 2017.

RedBlackBST.java

Below is the syntax highlighted version of RedBlackBST.java from §3.3 Balanced Search Trees.

```
/******************************************************************************
 *  Compilation:  javac RedBlackBST.java
 *  Execution:    java RedBlackBST < input.txt
 *  Dependencies: StdIn.java StdOut.java
 *  Data files:   http://algs4.cs.princeton.edu/33balanced/tinyST.txt
 *
 *  A symbol table implemented using a left-leaning red-black BST.
 *  This is the 2-3 version.
 *
 *  Note: commented out assertions because DrJava now enables assertions
 *        by default.
 *
 *  % more tinyST.txt
 *  S E A R C H E X A M P L E
 *
 *  % java RedBlackBST < tinyST.txt
 *  A 8
 *  C 4
 *  E 12
 *  H 5
 *  L 11
 *  M 9
 *  P 10
 *  R 3
 *  S 0
 *  X 7
 *
 ******************************************************************************/

import java.util.NoSuchElementException;

/**
 *  The {@code BST} class represents an ordered symbol table of generic
 *  key-value pairs.
 *  It supports the usual <em>put</em>, <em>get</em>, <em>contains</em>,
 *  <em>delete</em>, <em>size</em>, and <em>is-empty</em> methods.
 *  It also provides ordered methods for finding the <em>minimum</em>,
 *  <em>maximum</em>, <em>floor</em>, and <em>ceiling</em>.
 *  It also provides a <em>keys</em> method for iterating over all of the keys.
 *  A symbol table implements the <em>associative array</em> abstraction:
 *  when associating a value with a key that is already in the symbol table,
 *  the convention is to replace the old value with the new value.
 *  Unlike {@link java.util.Map}, this class uses the convention that
 *  values cannot be {@code null}—setting the
 *  value associated with a key to {@code null} is equivalent to deleting the key
 *  from the symbol table.
 *  <p>
 *  This implementation uses a left-leaning red-black BST. It requires that
 *  the key type implements the {@code Comparable} interface and calls the
 *  {@code compareTo()} and method to compare two keys. It does not call either
 *  {@code equals()} or {@code hashCode()}.
 *  The <em>put</em>, <em>contains</em>, <em>remove</em>, <em>minimum</em>,
 *  <em>maximum</em>, <em>ceiling</em>, and <em>floor</em> operations each take
 *  logarithmic time in the worst case, if the tree becomes unbalanced.
 *  The <em>size</em>, and <em>is-empty</em> operations take constant time.
 *  Construction takes constant time.
 *  <p>
 *  For additional documentation, see <a href="http://algs4.cs.princeton.edu/33balanced">Section 3.3</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *  For other implementations of the same API, see {@link ST}, {@link BinarySearchST},
 *  {@link SequentialSearchST}, {@link BST},
 *  {@link SeparateChainingHashST}, {@link LinearProbingHashST}, and {@link AVLTreeST}.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */

public class RedBlackBST<Key extends Comparable<Key>, Value> {
```

```
 private static final boolean RED   = true;
 private static final boolean BLACK = false;

 private Node root;     // root of the BST

 // BST helper node data type
 private class Node {
     private Key key;           // key
     private Value val;         // associated data
     private Node left, right;  // links to left and right subtrees
     private boolean color;     // color of parent link
     private int size;          // subtree count

     public Node(Key key, Value val, boolean color, int size) {
         this.key = key;
         this.val = val;
         this.color = color;
         this.size = size;
     }
 }

 /**
  * Initializes an empty symbol table.
  */
 public RedBlackBST() {
 }

/***************************************************************************
 *  Node helper methods.
 ***************************************************************************/
 // is node x red; false if x is null ?
 private boolean isRed(Node x) {
     if (x == null) return false;
     return x.color == RED;
 }

 // number of node in subtree rooted at x; 0 if x is null
 private int size(Node x) {
     if (x == null) return 0;
     return x.size;
 }


 /**
  * Returns the number of key-value pairs in this symbol table.
  * @return the number of key-value pairs in this symbol table
  */
 public int size() {
     return size(root);
 }

/**
  * Is this symbol table empty?
  * @return {@code true} if this symbol table is empty and {@code false} otherwise
  */
 public boolean isEmpty() {
     return root == null;
 }


/***************************************************************************
 *  Standard BST search.
 ***************************************************************************/

 /**
  * Returns the value associated with the given key.
  * @param key the key
  * @return the value associated with the given key if the key is in the symbol table
  *     and {@code null} if the key is not in the symbol table
  * @throws IllegalArgumentException if {@code key} is {@code null}
  */
 public Value get(Key key) {
     if (key == null) throw new IllegalArgumentException("argument to get() is null");
     return get(root, key);
 }

 // value associated with the given key in subtree rooted at x; null if no such key
```

```
 private Value get(Node x, Key key) {
     while (x != null) {
         int cmp = key.compareTo(x.key);
         if      (cmp < 0) x = x.left;
         else if (cmp > 0) x = x.right;
         else              return x.val;
     }
     return null;
 }

 /**
  * Does this symbol table contain the given key?
  * @param key the key
  * @return {@code true} if this symbol table contains {@code key} and
  *     {@code false} otherwise
  * @throws IllegalArgumentException if {@code key} is {@code null}
  */
 public boolean contains(Key key) {
     return get(key) != null;
 }

/***************************************************************************
 *  Red-black tree insertion.
 ***************************************************************************/

 /**
  * Inserts the specified key-value pair into the symbol table, overwriting the old
  * value with the new value if the symbol table already contains the specified key.
  * Deletes the specified key (and its associated value) from this symbol table
  * if the specified value is {@code null}.
  *
  * @param key the key
  * @param val the value
  * @throws IllegalArgumentException if {@code key} is {@code null}
  */
 public void put(Key key, Value val) {
     if (key == null) throw new IllegalArgumentException("first argument to put() is null");
     if (val == null) {
         delete(key);
         return;
     }

     root = put(root, key, val);
     root.color = BLACK;
     // assert check();
 }

 // insert the key-value pair in the subtree rooted at h
 private Node put(Node h, Key key, Value val) {
     if (h == null) return new Node(key, val, RED, 1);

     int cmp = key.compareTo(h.key);
     if      (cmp < 0) h.left  = put(h.left,  key, val);
     else if (cmp > 0) h.right = put(h.right, key, val);
     else              h.val   = val;

     // fix-up any right-leaning links
     if (isRed(h.right) && !isRed(h.left))      h = rotateLeft(h);
     if (isRed(h.left)  &&  isRed(h.left.left)) h = rotateRight(h);
     if (isRed(h.left)  &&  isRed(h.right))     flipColors(h);
     h.size = size(h.left) + size(h.right) + 1;

     return h;
 }

/***************************************************************************
 *  Red-black tree deletion.
 ***************************************************************************/

 /**
  * Removes the smallest key and associated value from the symbol table.
  * @throws NoSuchElementException if the symbol table is empty
  */
 public void deleteMin() {
     if (isEmpty()) throw new NoSuchElementException("BST underflow");

     // if both children of root are black, set root to red
     if (!isRed(root.left) && !isRed(root.right))
```

```
        root.color = RED;

    root = deleteMin(root);
    if (!isEmpty()) root.color = BLACK;
    // assert check();
}

// delete the key-value pair with the minimum key rooted at h
private Node deleteMin(Node h) {
    if (h.left == null)
        return null;

    if (!isRed(h.left) && !isRed(h.left.left))
        h = moveRedLeft(h);

    h.left = deleteMin(h.left);
    return balance(h);
}


/**
 * Removes the largest key and associated value from the symbol table.
 * @throws NoSuchElementException if the symbol table is empty
 */
public void deleteMax() {
    if (isEmpty()) throw new NoSuchElementException("BST underflow");

    // if both children of root are black, set root to red
    if (!isRed(root.left) && !isRed(root.right))
        root.color = RED;

    root = deleteMax(root);
    if (!isEmpty()) root.color = BLACK;
    // assert check();
}

// delete the key-value pair with the maximum key rooted at h
private Node deleteMax(Node h) {
    if (isRed(h.left))
        h = rotateRight(h);

    if (h.right == null)
        return null;

    if (!isRed(h.right) && !isRed(h.right.left))
        h = moveRedRight(h);

    h.right = deleteMax(h.right);

    return balance(h);
}

/**
 * Removes the specified key and its associated value from this symbol table
 * (if the key is in this symbol table).
 *
 * @param  key the key
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public void delete(Key key) {
    if (key == null) throw new IllegalArgumentException("argument to delete() is null");
    if (!contains(key)) return;

    // if both children of root are black, set root to red
    if (!isRed(root.left) && !isRed(root.right))
        root.color = RED;

    root = delete(root, key);
    if (!isEmpty()) root.color = BLACK;
    // assert check();
}

// delete the key-value pair with the given key rooted at h
private Node delete(Node h, Key key) {
    // assert get(h, key) != null;

    if (key.compareTo(h.key) < 0)  {
        if (!isRed(h.left) && !isRed(h.left.left))
```

```
            h = moveRedLeft(h);
        h.left = delete(h.left, key);
    }
    else {
        if (isRed(h.left))
            h = rotateRight(h);
        if (key.compareTo(h.key) == 0 && (h.right == null))
            return null;
        if (!isRed(h.right) && !isRed(h.right.left))
            h = moveRedRight(h);
        if (key.compareTo(h.key) == 0) {
            Node x = min(h.right);
            h.key = x.key;
            h.val = x.val;
            // h.val = get(h.right, min(h.right).key);
            // h.key = min(h.right).key;
            h.right = deleteMin(h.right);
        }
        else h.right = delete(h.right, key);
    }
    return balance(h);
}

/***************************************************************************
 *  Red-black tree helper functions.
 ***************************************************************************/

 // make a left-leaning link lean to the right
 private Node rotateRight(Node h) {
    // assert (h != null) && isRed(h.left);
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = x.right.color;
    x.right.color = RED;
    x.size = h.size;
    h.size = size(h.left) + size(h.right) + 1;
    return x;
 }

 // make a right-leaning link lean to the left
 private Node rotateLeft(Node h) {
    // assert (h != null) && isRed(h.right);
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = x.left.color;
    x.left.color = RED;
    x.size = h.size;
    h.size = size(h.left) + size(h.right) + 1;
    return x;
 }

 // flip the colors of a node and its two children
 private void flipColors(Node h) {
    // h must have opposite color of its two children
    // assert (h != null) && (h.left != null) && (h.right != null);
    // assert (!isRed(h) &&  isRed(h.left) &&  isRed(h.right))
    //    || (isRed(h)  && !isRed(h.left) && !isRed(h.right));
    h.color = !h.color;
    h.left.color = !h.left.color;
    h.right.color = !h.right.color;
 }

 // Assuming that h is red and both h.left and h.left.left
 // are black, make h.left or one of its children red.
 private Node moveRedLeft(Node h) {
    // assert (h != null);
    // assert isRed(h) && !isRed(h.left) && !isRed(h.left.left);

    flipColors(h);
    if (isRed(h.right.left)) {
        h.right = rotateRight(h.right);
        h = rotateLeft(h);
        flipColors(h);
    }
    return h;
 }
```

```
// Assuming that h is red and both h.right and h.right.left
// are black, make h.right or one of its children red.
private Node moveRedRight(Node h) {
    // assert (h != null);
    // assert isRed(h) && !isRed(h.right) && !isRed(h.right.left);
    flipColors(h);
    if (isRed(h.left.left)) {
        h = rotateRight(h);
        flipColors(h);
    }
    return h;
}

// restore red-black tree invariant
private Node balance(Node h) {
    // assert (h != null);

    if (isRed(h.right))                      h = rotateLeft(h);
    if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);
    if (isRed(h.left) && isRed(h.right))     flipColors(h);

    h.size = size(h.left) + size(h.right) + 1;
    return h;
}


/***************************************************************************
 *  Utility functions.
 ***************************************************************************/

/**
 * Returns the height of the BST (for debugging).
 * @return the height of the BST (a 1-node tree has height 0)
 */
public int height() {
    return height(root);
}
private int height(Node x) {
    if (x == null) return -1;
    return 1 + Math.max(height(x.left), height(x.right));
}

/***************************************************************************
 *  Ordered symbol table methods.
 ***************************************************************************/

/**
 * Returns the smallest key in the symbol table.
 * @return the smallest key in the symbol table
 * @throws NoSuchElementException if the symbol table is empty
 */
public Key min() {
    if (isEmpty()) throw new NoSuchElementException("called min() with empty symbol table");
    return min(root).key;
}

// the smallest key in subtree rooted at x; null if no such key
private Node min(Node x) {
    // assert x != null;
    if (x.left == null) return x;
    else                return min(x.left);
}

/**
 * Returns the largest key in the symbol table.
 * @return the largest key in the symbol table
 * @throws NoSuchElementException if the symbol table is empty
 */
public Key max() {
    if (isEmpty()) throw new NoSuchElementException("called max() with empty symbol table");
    return max(root).key;
}

// the largest key in the subtree rooted at x; null if no such key
private Node max(Node x) {
    // assert x != null;
    if (x.right == null) return x;
```

```
    else                return max(x.right);
}


/**
 * Returns the largest key in the symbol table less than or equal to {@code key}.
 * @param key the key
 * @return the largest key in the symbol table less than or equal to {@code key}
 * @throws NoSuchElementException if there is no such key
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public Key floor(Key key) {
    if (key == null) throw new IllegalArgumentException("argument to floor() is null");
    if (isEmpty()) throw new NoSuchElementException("called floor() with empty symbol table");
    Node x = floor(root, key);
    if (x == null) return null;
    else           return x.key;
}

// the largest key in the subtree rooted at x less than or equal to the given key
private Node floor(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp == 0) return x;
    if (cmp < 0)  return floor(x.left, key);
    Node t = floor(x.right, key);
    if (t != null) return t;
    else           return x;
}

/**
 * Returns the smallest key in the symbol table greater than or equal to {@code key}.
 * @param key the key
 * @return the smallest key in the symbol table greater than or equal to {@code key}
 * @throws NoSuchElementException if there is no such key
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public Key ceiling(Key key) {
    if (key == null) throw new IllegalArgumentException("argument to ceiling() is null");
    if (isEmpty()) throw new NoSuchElementException("called ceiling() with empty symbol table");
    Node x = ceiling(root, key);
    if (x == null) return null;
    else           return x.key;
}

// the smallest key in the subtree rooted at x greater than or equal to the given key
private Node ceiling(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp == 0) return x;
    if (cmp > 0)  return ceiling(x.right, key);
    Node t = ceiling(x.left, key);
    if (t != null) return t;
    else           return x;
}

/**
 * Return the kth smallest key in the symbol table.
 * @param k the order statistic
 * @return the {@code k}th smallest key in the symbol table
 * @throws IllegalArgumentException unless {@code k} is between 0 and
 *     <em>n</em>-1
 */
public Key select(int k) {
    if (k < 0 || k >= size()) {
        throw new IllegalArgumentException("called select() with invalid argument: " + k);
    }
    Node x = select(root, k);
    return x.key;
}

// the key of rank k in the subtree rooted at x
private Node select(Node x, int k) {
    // assert x != null;
    // assert k >= 0 && k < size(x);
    int t = size(x.left);
    if      (t > k) return select(x.left,  k);
    else if (t < k) return select(x.right, k-t-1);
```

```
        else            return x;
    }

    /**
     * Return the number of keys in the symbol table strictly less than {@code key}.
     * @param key the key
     * @return the number of keys in the symbol table strictly less than {@code key}
     * @throws IllegalArgumentException if {@code key} is {@code null}
     */
    public int rank(Key key) {
        if (key == null) throw new IllegalArgumentException("argument to rank() is null");
        return rank(key, root);
    }

    // number of keys less than key in the subtree rooted at x
    private int rank(Key key, Node x) {
        if (x == null) return 0;
        int cmp = key.compareTo(x.key);
        if      (cmp < 0) return rank(key, x.left);
        else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);
        else              return size(x.left);
    }

   /***********************************************************************
    *  Range count and range search.
    ***********************************************************************/

    /**
     * Returns all keys in the symbol table as an {@code Iterable}.
     * To iterate over all of the keys in the symbol table named {@code st},
     * use the foreach notation: {@code for (Key key : st.keys())}.
     * @return all keys in the symbol table as an {@code Iterable}
     */
    public Iterable<Key> keys() {
        if (isEmpty()) return new Queue<Key>();
        return keys(min(), max());
    }

    /**
     * Returns all keys in the symbol table in the given range,
     * as an {@code Iterable}.
     *
     * @param  lo minimum endpoint
     * @param  hi maximum endpoint
     * @return all keys in the sybol table between {@code lo}
     *    (inclusive) and {@code hi} (inclusive) as an {@code Iterable}
     * @throws IllegalArgumentException if either {@code lo} or {@code hi}
     *    is {@code null}
     */
    public Iterable<Key> keys(Key lo, Key hi) {
        if (lo == null) throw new IllegalArgumentException("first argument to keys() is null");
        if (hi == null) throw new IllegalArgumentException("second argument to keys() is null");

        Queue<Key> queue = new Queue<Key>();
        // if (isEmpty() || lo.compareTo(hi) > 0) return queue;
        keys(root, queue, lo, hi);
        return queue;
    }

    // add the keys between lo and hi in the subtree rooted at x
    // to the queue
    private void keys(Node x, Queue<Key> queue, Key lo, Key hi) {
        if (x == null) return;
        int cmplo = lo.compareTo(x.key);
        int cmphi = hi.compareTo(x.key);
        if (cmplo < 0) keys(x.left, queue, lo, hi);
        if (cmplo <= 0 && cmphi >= 0) queue.enqueue(x.key);
        if (cmphi > 0) keys(x.right, queue, lo, hi);
    }

    /**
     * Returns the number of keys in the symbol table in the given range.
     *
     * @param  lo minimum endpoint
     * @param  hi maximum endpoint
     * @return the number of keys in the sybol table between {@code lo}
     *    (inclusive) and {@code hi} (inclusive)
     * @throws IllegalArgumentException if either {@code lo} or {@code hi}
```

```
     *    is {@code null}
     */
    public int size(Key lo, Key hi) {
        if (lo == null) throw new IllegalArgumentException("first argument to size() is null");
        if (hi == null) throw new IllegalArgumentException("second argument to size() is null");

        if (lo.compareTo(hi) > 0) return 0;
        if (contains(hi)) return rank(hi) - rank(lo) + 1;
        else              return rank(hi) - rank(lo);
    }


    /*************************************************************************
     *  Check integrity of red-black tree data structure.
     *************************************************************************/
    private boolean check() {
        if (!isBST())            StdOut.println("Not in symmetric order");
        if (!isSizeConsistent()) StdOut.println("Subtree counts not consistent");
        if (!isRankConsistent()) StdOut.println("Ranks not consistent");
        if (!is23())             StdOut.println("Not a 2-3 tree");
        if (!isBalanced())       StdOut.println("Not balanced");
        return isBST() && isSizeConsistent() && isRankConsistent() && is23() && isBalanced();
    }

    // does this binary tree satisfy symmetric order?
    // Note: this test also ensures that data structure is a binary tree since order is strict
    private boolean isBST() {
        return isBST(root, null, null);
    }

    // is the tree rooted at x a BST with all keys strictly between min and max
    // (if min or max is null, treat as empty constraint)
    // Credit: Bob Dondero's elegant solution
    private boolean isBST(Node x, Key min, Key max) {
        if (x == null) return true;
        if (min != null && x.key.compareTo(min) <= 0) return false;
        if (max != null && x.key.compareTo(max) >= 0) return false;
        return isBST(x.left, min, x.key) && isBST(x.right, x.key, max);
    }

    // are the size fields correct?
    private boolean isSizeConsistent() { return isSizeConsistent(root); }
    private boolean isSizeConsistent(Node x) {
        if (x == null) return true;
        if (x.size != size(x.left) + size(x.right) + 1) return false;
        return isSizeConsistent(x.left) && isSizeConsistent(x.right);
    }

    // check that ranks are consistent
    private boolean isRankConsistent() {
        for (int i = 0; i < size(); i++)
            if (i != rank(select(i))) return false;
        for (Key key : keys())
            if (key.compareTo(select(rank(key))) != 0) return false;
        return true;
    }

    // Does the tree have no red right links, and at most one (left)
    // red links in a row on any path?
    private boolean is23() { return is23(root); }
    private boolean is23(Node x) {
        if (x == null) return true;
        if (isRed(x.right)) return false;
        if (x != root && isRed(x) && isRed(x.left))
            return false;
        return is23(x.left) && is23(x.right);
    }

    // do all paths from root to leaf have same number of black edges?
    private boolean isBalanced() {
        int black = 0;     // number of black links on path from root to min
        Node x = root;
        while (x != null) {
            if (!isRed(x)) black++;
            x = x.left;
        }
        return isBalanced(root, black);
    }
```

```
    // does every path from the root to a leaf have the given number of black links?
    private boolean isBalanced(Node x, int black) {
        if (x == null) return black == 0;
        if (!isRed(x)) black--;
        return isBalanced(x.left, black) && isBalanced(x.right, black);
    }


    /**
     * Unit tests the {@code RedBlackBST} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        RedBlackBST<String, Integer> st = new RedBlackBST<String, Integer>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
            String key = StdIn.readString();
            st.put(key, i);
        }
        for (String s : st.keys())
            StdOut.println(s + " " + st.get(s));
        StdOut.println();
    }
}
```


Last updated: Sat Jan 28 06:55:20 EST 2017.

SeparateChainingHashST.java

Below is the syntax highlighted version of SeparateChainingHashST.java from §3.4 Hash Tables.

```
/******************************************************************************
 *  Compilation:  javac SeparateChainingHashST.java
 *  Execution:    java SeparateChainingHashST < input.txt
 *  Dependencies: StdIn.java StdOut.java
 *  Data files:   http://algs4.cs.princeton.edu/34hash/tinyST.txt
 *
 *  A symbol table implemented with a separate-chaining hash table.
 *
 ******************************************************************************/

/**
 *  The {@code SeparateChainingHashST} class represents a symbol table of generic
 *  key-value pairs.
 *  It supports the usual <em>put</em>, <em>get</em>, <em>contains</em>,
 *  <em>delete</em>, <em>size</em>, and <em>is-empty</em> methods.
 *  It also provides a <em>keys</em> method for iterating over all of the keys.
 *  A symbol table implements the <em>associative array</em> abstraction:
 *  when associating a value with a key that is already in the symbol table,
 *  the convention is to replace the old value with the new value.
 *  Unlike {@link java.util.Map}, this class uses the convention that
 *  values cannot be {@code null}—setting the
 *  value associated with a key to {@code null} is equivalent to deleting the key
 *  from the symbol table.
 *  <p>
 *  This implementation uses a separate chaining hash table. It requires that
 *  the key type overrides the {@code equals()} and {@code hashCode()} methods.
 *  The expected time per <em>put</em>, <em>contains</em>, or <em>remove</em>
 *  operation is constant, subject to the uniform hashing assumption.
 *  The <em>size</em>, and <em>is-empty</em> operations take constant time.
 *  Construction takes constant time.
 *  <p>
 *  For additional documentation, see <a href="http://algs4.cs.princeton.edu/34hash">Section 3.4</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *  For other implementations, see {@link ST}, {@link BinarySearchST},
 *  {@link SequentialSearchST}, {@link BST}, {@link RedBlackBST}, and
 *  {@link LinearProbingHashST},
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class SeparateChainingHashST<Key, Value> {
    private static final int INIT_CAPACITY = 4;

    private int n;                                // number of key-value pairs
    private int m;                                // hash table size
    private SequentialSearchST<Key, Value>[] st;  // array of linked-list symbol tables


    /**
     * Initializes an empty symbol table.
     */
    public SeparateChainingHashST() {
        this(INIT_CAPACITY);
    }

    /**
     * Initializes an empty symbol table with {@code m} chains.
     * @param m the initial number of chains
     */
    public SeparateChainingHashST(int m) {
        this.m = m;
        st = (SequentialSearchST<Key, Value>[]) new SequentialSearchST[m];
        for (int i = 0; i < m; i++)
            st[i] = new SequentialSearchST<Key, Value>();
    }
```

```
// resize the hash table to have the given number of chains,
// rehashing all of the keys
private void resize(int chains) {
    SeparateChainingHashST<Key, Value> temp = new SeparateChainingHashST<Key, Value>(chains);
    for (int i = 0; i < m; i++) {
        for (Key key : st[i].keys()) {
            temp.put(key, st[i].get(key));
        }
    }
    this.m  = temp.m;
    this.n  = temp.n;
    this.st = temp.st;
}

// hash value between 0 and m-1
private int hash(Key key) {
    return (key.hashCode() & 0x7fffffff) % m;
}

/**
 * Returns the number of key-value pairs in this symbol table.
 *
 * @return the number of key-value pairs in this symbol table
 */
public int size() {
    return n;
}

/**
 * Returns true if this symbol table is empty.
 *
 * @return {@code true} if this symbol table is empty;
 *         {@code false} otherwise
 */
public boolean isEmpty() {
    return size() == 0;
}

/**
 * Returns true if this symbol table contains the specified key.
 *
 * @param  key the key
 * @return {@code true} if this symbol table contains {@code key};
 *         {@code false} otherwise
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public boolean contains(Key key) {
    if (key == null) throw new IllegalArgumentException("argument to contains() is null");
    return get(key) != null;
}

/**
 * Returns the value associated with the specified key in this symbol table.
 *
 * @param  key the key
 * @return the value associated with {@code key} in the symbol table;
 *         {@code null} if no such value
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public Value get(Key key) {
    if (key == null) throw new IllegalArgumentException("argument to get() is null");
    int i = hash(key);
    return st[i].get(key);
}

/**
 * Inserts the specified key-value pair into the symbol table, overwriting the old
 * value with the new value if the symbol table already contains the specified key.
 * Deletes the specified key (and its associated value) from this symbol table
 * if the specified value is {@code null}.
 *
 * @param  key the key
 * @param  val the value
 * @throws IllegalArgumentException if {@code key} is {@code null}
```

```
     */
    public void put(Key key, Value val) {
        if (key == null) throw new IllegalArgumentException("first argument to put() is null");
        if (val == null) {
            delete(key);
            return;
        }

        // double table size if average length of list >= 10
        if (n >= 10*m) resize(2*m);

        int i = hash(key);
        if (!st[i].contains(key)) n++;
        st[i].put(key, val);
    }

    /**
     * Removes the specified key and its associated value from this symbol table
     * (if the key is in this symbol table).
     *
     * @param  key the key
     * @throws IllegalArgumentException if {@code key} is {@code null}
     */
    public void delete(Key key) {
        if (key == null) throw new IllegalArgumentException("argument to delete() is null");

        int i = hash(key);
        if (st[i].contains(key)) n--;
        st[i].delete(key);

        // halve table size if average length of list <= 2
        if (m > INIT_CAPACITY && n <= 2*m) resize(m/2);
    }

    // return keys in symbol table as an Iterable
    public Iterable<Key> keys() {
        Queue<Key> queue = new Queue<Key>();
        for (int i = 0; i < m; i++) {
            for (Key key : st[i].keys())
                queue.enqueue(key);
        }
        return queue;
    }


    /**
     * Unit tests the {@code SeparateChainingHashST} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        SeparateChainingHashST<String, Integer> st = new SeparateChainingHashST<String, Integer>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
            String key = StdIn.readString();
            st.put(key, i);
        }

        // print keys
        for (String s : st.keys())
            StdOut.println(s + " " + st.get(s));

    }

}
```

LinearProbingHashST.java

Below is the syntax highlighted version of LinearProbingHashST.java from §3.4 Hash Tables.

```
/******************************************************************************
 *  Compilation:  javac LinearProbingHashST.java
 *  Execution:    java LinearProbingHashST < input.txt
 *  Dependencies: StdIn.java StdOut.java
 *  Data files:   http://algs4.cs.princeton.edu/34hash/tinyST.txt
 *
 *  Symbol-table implementation with linear-probing hash table.
 *
 ******************************************************************************/

/**
 *  The {@code LinearProbingHashST} class represents a symbol table of generic
 *  key-value pairs.
 *  It supports the usual <em>put</em>, <em>get</em>, <em>contains</em>,
 *  <em>delete</em>, <em>size</em>, and <em>is-empty</em> methods.
 *  It also provides a <em>keys</em> method for iterating over all of the keys.
 *  A symbol table implements the <em>associative array</em> abstraction:
 *  when associating a value with a key that is already in the symbol table,
 *  the convention is to replace the old value with the new value.
 *  Unlike {@link java.util.Map}, this class uses the convention that
 *  values cannot be {@code null}—setting the
 *  value associated with a key to {@code null} is equivalent to deleting the key
 *  from the symbol table.
 *  <p>
 *  This implementation uses a linear probing hash table. It requires that
 *  the key type overrides the {@code equals()} and {@code hashCode()} methods.
 *  The expected time per <em>put</em>, <em>contains</em>, or <em>remove</em>
 *  operation is constant, subject to the uniform hashing assumption.
 *  The <em>size</em>, and <em>is-empty</em> operations take constant time.
 *  Construction takes constant time.
 *  <p>
 *  For additional documentation, see <a href="http://algs4.cs.princeton.edu/34hash">Section 3.4</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *  For other implementations, see {@link ST}, {@link BinarySearchST},
 *  {@link SequentialSearchST}, {@link BST}, {@link RedBlackBST}, and
 *  {@link SeparateChainingHashST},
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class LinearProbingHashST<Key, Value> {
    private static final int INIT_CAPACITY = 4;

    private int n;           // number of key-value pairs in the symbol table
    private int m;           // size of linear probing table
    private Key[] keys;      // the keys
    private Value[] vals;    // the values


    /**
     * Initializes an empty symbol table.
     */
    public LinearProbingHashST() {
        this(INIT_CAPACITY);
    }

    /**
     * Initializes an empty symbol table with the specified initial capacity.
     *
     * @param capacity the initial capacity
     */
    public LinearProbingHashST(int capacity) {
        m = capacity;
        n = 0;
        keys = (Key[])   new Object[m];
```

```
    vals = (Value[]) new Object[m];
}

/**
 * Returns the number of key-value pairs in this symbol table.
 *
 * @return the number of key-value pairs in this symbol table
 */
public int size() {
    return n;
}

/**
 * Returns true if this symbol table is empty.
 *
 * @return {@code true} if this symbol table is empty;
 *         {@code false} otherwise
 */
public boolean isEmpty() {
    return size() == 0;
}

/**
 * Returns true if this symbol table contains the specified key.
 *
 * @param  key the key
 * @return {@code true} if this symbol table contains {@code key};
 *         {@code false} otherwise
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public boolean contains(Key key) {
    if (key == null) throw new IllegalArgumentException("argument to contains() is null");
    return get(key) != null;
}

// hash function for keys - returns value between 0 and M-1
private int hash(Key key) {
    return (key.hashCode() & 0x7fffffff) % m;
}

// resizes the hash table to the given capacity by re-hashing all of the keys
private void resize(int capacity) {
    LinearProbingHashST<Key, Value> temp = new LinearProbingHashST<Key, Value>(capacity);
    for (int i = 0; i < m; i++) {
        if (keys[i] != null) {
            temp.put(keys[i], vals[i]);
        }
    }
    keys = temp.keys;
    vals = temp.vals;
    m    = temp.m;
}

/**
 * Inserts the specified key-value pair into the symbol table, overwriting the old
 * value with the new value if the symbol table already contains the specified key.
 * Deletes the specified key (and its associated value) from this symbol table
 * if the specified value is {@code null}.
 *
 * @param  key the key
 * @param  val the value
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public void put(Key key, Value val) {
    if (key == null) throw new IllegalArgumentException("first argument to put() is null");

    if (val == null) {
        delete(key);
        return;
    }

    // double table size if 50% full
    if (n >= m/2) resize(2*m);

    int i;
```

```
    for (i = hash(key); keys[i] != null; i = (i + 1) % m) {
        if (keys[i].equals(key)) {
            vals[i] = val;
            return;
        }
    }
    keys[i] = key;
    vals[i] = val;
    n++;
}

/**
 * Returns the value associated with the specified key.
 * @param key the key
 * @return the value associated with {@code key};
 *         {@code null} if no such value
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public Value get(Key key) {
    if (key == null) throw new IllegalArgumentException("argument to get() is null");
    for (int i = hash(key); keys[i] != null; i = (i + 1) % m)
        if (keys[i].equals(key))
            return vals[i];
    return null;
}

/**
 * Removes the specified key and its associated value from this symbol table
 * (if the key is in this symbol table).
 *
 * @param  key the key
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public void delete(Key key) {
    if (key == null) throw new IllegalArgumentException("argument to delete() is null");
    if (!contains(key)) return;

    // find position i of key
    int i = hash(key);
    while (!key.equals(keys[i])) {
        i = (i + 1) % m;
    }

    // delete key and associated value
    keys[i] = null;
    vals[i] = null;

    // rehash all keys in same cluster
    i = (i + 1) % m;
    while (keys[i] != null) {
        // delete keys[i] an vals[i] and reinsert
        Key   keyToRehash = keys[i];
        Value valToRehash = vals[i];
        keys[i] = null;
        vals[i] = null;
        n--;
        put(keyToRehash, valToRehash);
        i = (i + 1) % m;
    }

    n--;

    // halves size of array if it's 12.5% full or less
    if (n > 0 && n <= m/8) resize(m/2);

    assert check();
}

/**
 * Returns all keys in this symbol table as an {@code Iterable}.
 * To iterate over all of the keys in the symbol table named {@code st},
 * use the foreach notation: {@code for (Key key : st.keys())}.
 *
 * @return all keys in this symbol table
 */
```

```
    public Iterable<Key> keys() {
        Queue<Key> queue = new Queue<Key>();
        for (int i = 0; i < m; i++)
            if (keys[i] != null) queue.enqueue(keys[i]);
        return queue;
    }

    // integrity check - don't check after each put() because
    // integrity not maintained during a delete()
    private boolean check() {

        // check that hash table is at most 50% full
        if (m < 2*n) {
            System.err.println("Hash table size m = " + m + "; array size n = " + n);
            return false;
        }

        // check that each key in table can be found by get()
        for (int i = 0; i < m; i++) {
            if (keys[i] == null) continue;
            else if (get(keys[i]) != vals[i]) {
                System.err.println("get[" + keys[i] + "] = " + get(keys[i]) + "; vals[i] = " + vals[i]);
                return false;
            }
        }
        return true;
    }


    /**
     * Unit tests the {@code LinearProbingHashST} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        LinearProbingHashST<String, Integer> st = new LinearProbingHashST<String, Integer>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
            String key = StdIn.readString();
            st.put(key, i);
        }

        // print keys
        for (String s : st.keys())
            StdOut.println(s + " " + st.get(s));
    }
}
```

DepthFirstPaths.java

Below is the syntax highlighted version of DepthFirstPaths.java from §4.1 Undirected Graphs.

```
/******************************************************************************
 *  Compilation:  javac DepthFirstPaths.java
 *  Execution:    java DepthFirstPaths G s
 *  Dependencies: Graph.java Stack.java StdOut.java
 *  Data files:   http://algs4.cs.princeton.edu/41graph/tinyCG.txt
 *                http://algs4.cs.princeton.edu/41graph/tinyG.txt
 *                http://algs4.cs.princeton.edu/41graph/mediumG.txt
 *                http://algs4.cs.princeton.edu/41graph/largeG.txt
 *
 *  Run depth first search on an undirected graph.
 *  Runs in O(E + V) time.
 *
 *  %  java Graph tinyCG.txt
 *  6 8
 *  0: 2 1 5
 *  1: 0 2
 *  2: 0 1 3 4
 *  3: 5 4 2
 *  4: 3 2
 *  5: 3 0
 *
 *  % java DepthFirstPaths tinyCG.txt 0
 *  0 to 0:  0
 *  0 to 1:  0-2-1
 *  0 to 2:  0-2
 *  0 to 3:  0-2-3
 *  0 to 4:  0-2-3-4
 *  0 to 5:  0-2-3-5
 *
 ******************************************************************************/

/**
 *  The {@code DepthFirstPaths} class represents a data type for finding
 *  paths from a source vertex <em>s</em> to every other vertex
 *  in an undirected graph.
 *  <p>
 *  This implementation uses depth-first search.
 *  The constructor takes time proportional to <em>V</em> + <em>E</em>,
 *  where <em>V</em> is the number of vertices and <em>E</em> is the number of edges.
 *  It uses extra space (not including the graph) proportional to <em>V</em>.
 *  <p>
 *  For additional documentation, see <a href="http://algs4.cs.princeton.edu/41graph">Section 4.1</a>
 *  of <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class DepthFirstPaths {
    private boolean[] marked;    // marked[v] = is there an s-v path?
    private int[] edgeTo;        // edgeTo[v] = last edge on s-v path
    private final int s;         // source vertex

    /**
     * Computes a path between {@code s} and every other vertex in graph {@code G}.
     * @param G the graph
     * @param s the source vertex
     * @throws IllegalArgumentException unless {@code 0 <= s < V}
     */
    public DepthFirstPaths(Graph G, int s) {
        this.s = s;
        edgeTo = new int[G.V()];
        marked = new boolean[G.V()];
        validateVertex(s);
        dfs(G, s);
    }
```

```java
// depth first search from v
private void dfs(Graph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
        if (!marked[w]) {
            edgeTo[w] = v;
            dfs(G, w);
        }
    }
}

/**
 * Is there a path between the source vertex {@code s} and vertex {@code v}?
 * @param v the vertex
 * @return {@code true} if there is a path, {@code false} otherwise
 * @throws IllegalArgumentException unless {@code 0 <= v < V}
 */
public boolean hasPathTo(int v) {
    validateVertex(v);
    return marked[v];
}

/**
 * Returns a path between the source vertex {@code s} and vertex {@code v}, or
 * {@code null} if no such path.
 * @param  v the vertex
 * @return the sequence of vertices on a path between the source vertex
 *         {@code s} and vertex {@code v}, as an Iterable
 * @throws IllegalArgumentException unless {@code 0 <= v < V}
 */
public Iterable<Integer> pathTo(int v) {
    validateVertex(v);
    if (!hasPathTo(v)) return null;
    Stack<Integer> path = new Stack<Integer>();
    for (int x = v; x != s; x = edgeTo[x])
        path.push(x);
    path.push(s);
    return path;
}

// throw an IllegalArgumentException unless {@code 0 <= v < V}
private void validateVertex(int v) {
    int V = marked.length;
    if (v < 0 || v >= V)
        throw new IllegalArgumentException("vertex " + v + " is not between 0 and " + (V-1));
}

/**
 * Unit tests the {@code DepthFirstPaths} data type.
 *
 * @param args the command-line arguments
 */
public static void main(String[] args) {
    In in = new In(args[0]);
    Graph G = new Graph(in);
    int s = Integer.parseInt(args[1]);
    DepthFirstPaths dfs = new DepthFirstPaths(G, s);

    for (int v = 0; v < G.V(); v++) {
        if (dfs.hasPathTo(v)) {
            StdOut.printf("%d to %d:  ", s, v);
            for (int x : dfs.pathTo(v)) {
                if (x == s) StdOut.print(x);
                else        StdOut.print("-" + x);
            }
            StdOut.println();
        }

        else {
            StdOut.printf("%d to %d:  not connected\n", s, v);
        }

    }
}
```

```
}
```


Last updated: Sat Nov 19 06:48:07 EST 2016.

Cycle.java

Below is the syntax highlighted version of Cycle.java from §4.1 Undirected Graphs.

```
/******************************************************************************
 *  Compilation:  javac Cycle.java
 *  Execution:    java  Cycle filename.txt
 *  Dependencies: Graph.java Stack.java In.java StdOut.java
 *  Data files:   http://algs4.cs.princeton.edu/41graph/tinyG.txt
 *                http://algs4.cs.princeton.edu/41graph/mediumG.txt
 *                http://algs4.cs.princeton.edu/41graph/largeG.txt
 *
 *  Identifies a cycle.
 *  Runs in O(E + V) time.
 *
 *  % java Cycle tinyG.txt
 *  3 4 5 3
 *
 *  % java Cycle mediumG.txt
 *  15 0 225 15
 *
 *  % java Cycle largeG.txt
 *  996673 762 840164 4619 785187 194717 996673
 *
 ******************************************************************************/

/**
 *  The {@code Cycle} class represents a data type for
 *  determining whether an undirected graph has a cycle.
 *  The <em>hasCycle</em> operation determines whether the graph has
 *  a cycle and, if so, the <em>cycle</em> operation returns one.
 *  <p>
 *  This implementation uses depth-first search.
 *  The constructor takes time proportional to <em>V</em> + <em>E</em>
 *  (in the worst case),
 *  where <em>V</em> is the number of vertices and <em>E</em> is the number of edges.
 *  Afterwards, the <em>hasCycle</em> operation takes constant time;
 *  the <em>cycle</em> operation takes time proportional
 *  to the length of the cycle.
 *  <p>
 *  For additional documentation, see <a href="http://algs4.cs.princeton.edu/41graph">Section 4.1</a>
 *  of <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class Cycle {
    private boolean[] marked;
    private int[] edgeTo;
    private Stack<Integer> cycle;

    /**
     * Determines whether the undirected graph {@code G} has a cycle and,
     * if so, finds such a cycle.
     *
     * @param G the undirected graph
     */
    public Cycle(Graph G) {
        if (hasSelfLoop(G)) return;
        if (hasParallelEdges(G)) return;
        marked = new boolean[G.V()];
        edgeTo = new int[G.V()];
        for (int v = 0; v < G.V(); v++)
            if (!marked[v])
                dfs(G, -1, v);
    }


    // does this graph have a self loop?
```

```
// side effect: initialize cycle to be self loop
private boolean hasSelfLoop(Graph G) {
    for (int v = 0; v < G.V(); v++) {
        for (int w : G.adj(v)) {
            if (v == w) {
                cycle = new Stack<Integer>();
                cycle.push(v);
                cycle.push(v);
                return true;
            }
        }
    }
    return false;
}

// does this graph have two parallel edges?
// side effect: initialize cycle to be two parallel edges
private boolean hasParallelEdges(Graph G) {
    marked = new boolean[G.V()];

    for (int v = 0; v < G.V(); v++) {

        // check for parallel edges incident to v
        for (int w : G.adj(v)) {
            if (marked[w]) {
                cycle = new Stack<Integer>();
                cycle.push(v);
                cycle.push(w);
                cycle.push(v);
                return true;
            }
            marked[w] = true;
        }

        // reset so marked[v] = false for all v
        for (int w : G.adj(v)) {
            marked[w] = false;
        }
    }
    return false;
}

/**
 * Returns true if the graph {@code G} has a cycle.
 *
 * @return {@code true} if the graph has a cycle; {@code false} otherwise
 */
public boolean hasCycle() {
    return cycle != null;
}

 /**
 * Returns a cycle in the graph {@code G}.
 * @return a cycle if the graph {@code G} has a cycle,
 *         and {@code null} otherwise
 */
public Iterable<Integer> cycle() {
    return cycle;
}

private void dfs(Graph G, int u, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {

        // short circuit if cycle already found
        if (cycle != null) return;

        if (!marked[w]) {
            edgeTo[w] = v;
            dfs(G, v, w);
        }

        // check for cycle (but disregard reverse of edge leading to v)
        else if (w != u) {
            cycle = new Stack<Integer>();
```

```
                for (int x = v; x != w; x = edgeTo[x]) {
                    cycle.push(x);
                }
                cycle.push(w);
                cycle.push(v);
            }
        }
    }

    /**
     * Unit tests the {@code Cycle} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        In in = new In(args[0]);
        Graph G = new Graph(in);
        Cycle finder = new Cycle(G);
        if (finder.hasCycle()) {
            for (int v : finder.cycle()) {
                StdOut.print(v + " ");
            }
            StdOut.println();
        }
        else {
            StdOut.println("Graph is acyclic");
        }
    }


}
```

DirectedCycle.java

Below is the syntax highlighted version of DirectedCycle.java from §4.2 Directed Graphs.

```
/******************************************************************************
 *  Compilation:  javac DirectedCycle.java
 *  Execution:    java DirectedCycle input.txt
 *  Dependencies: Digraph.java Stack.java StdOut.java In.java
 *  Data files:   http://algs4.cs.princeton.edu/42digraph/tinyDG.txt
 *                http://algs4.cs.princeton.edu/42digraph/tinyDAG.txt
 *
 *  Finds a directed cycle in a digraph.
 *  Runs in O(E + V) time.
 *
 *  % java DirectedCycle tinyDG.txt
 *  Directed cycle: 3 5 4 3
 *
 *  %  java DirectedCycle tinyDAG.txt
 *  No directed cycle
 *
 ******************************************************************************/

/**
 *  The {@code DirectedCycle} class represents a data type for
 *  determining whether a digraph has a directed cycle.
 *  The <em>hasCycle</em> operation determines whether the digraph has
 *  a directed cycle and, and of so, the <em>cycle</em> operation
 *  returns one.
 *  <p>
 *  This implementation uses depth-first search.
 *  The constructor takes time proportional to <em>V</em> + <em>E</em>
 *  (in the worst case),
 *  where <em>V</em> is the number of vertices and <em>E</em> is the number of edges.
 *  Afterwards, the <em>hasCycle</em> operation takes constant time;
 *  the <em>cycle</em> operation takes time proportional
 *  to the length of the cycle.
 *  <p>
 *  See {@link Topological} to compute a topological order if the
 *  digraph is acyclic.
 *  <p>
 *  For additional documentation,
 *  see <a href="http://algs4.cs.princeton.edu/42digraph">Section 4.2</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class DirectedCycle {
    private boolean[] marked;        // marked[v] = has vertex v been marked?
    private int[] edgeTo;            // edgeTo[v] = previous vertex on path to v
    private boolean[] onStack;       // onStack[v] = is vertex on the stack?
    private Stack<Integer> cycle;    // directed cycle (or null if no such cycle)

    /**
     * Determines whether the digraph {@code G} has a directed cycle and, if so,
     * finds such a cycle.
     * @param G the digraph
     */
    public DirectedCycle(Digraph G) {
        marked  = new boolean[G.V()];
        onStack = new boolean[G.V()];
        edgeTo  = new int[G.V()];
        for (int v = 0; v < G.V(); v++)
            if (!marked[v] && cycle == null) dfs(G, v);
```

```
}

// check that algorithm computes either the topological order or finds a directed cycle
private void dfs(Digraph G, int v) {
    onStack[v] = true;
    marked[v] = true;
    for (int w : G.adj(v)) {

        // short circuit if directed cycle found
        if (cycle != null) return;

        // found new vertex, so recur
        else if (!marked[w]) {
            edgeTo[w] = v;
            dfs(G, w);
        }

        // trace back directed cycle
        else if (onStack[w]) {
            cycle = new Stack<Integer>();
            for (int x = v; x != w; x = edgeTo[x]) {
                cycle.push(x);
            }
            cycle.push(w);
            cycle.push(v);
            assert check();
        }
    }
    onStack[v] = false;
}

/**
 * Does the digraph have a directed cycle?
 * @return {@code true} if the digraph has a directed cycle, {@code false} otherwise
 */
public boolean hasCycle() {
    return cycle != null;
}

/**
 * Returns a directed cycle if the digraph has a directed cycle, and {@code null} otherwise.
 * @return a directed cycle (as an iterable) if the digraph has a directed cycle,
 *    and {@code null} otherwise
 */
public Iterable<Integer> cycle() {
    return cycle;
}


// certify that digraph has a directed cycle if it reports one
private boolean check() {

    if (hasCycle()) {
        // verify cycle
        int first = -1, last = -1;
        for (int v : cycle()) {
            if (first == -1) first = v;
            last = v;
        }
        if (first != last) {
            System.err.printf("cycle begins with %d and ends with %d\n", first, last);
            return false;
        }
    }


    return true;
}

/**
```

```
     * Unit tests the {@code DirectedCycle} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        In in = new In(args[0]);
        Digraph G = new Digraph(in);

        DirectedCycle finder = new DirectedCycle(G);
        if (finder.hasCycle()) {
            StdOut.print("Directed cycle: ");
            for (int v : finder.cycle()) {
                StdOut.print(v + " ");
            }
            StdOut.println();
        }

        else {
            StdOut.println("No directed cycle");
        }
        StdOut.println();
    }

}
```


Last updated: Tue Aug 30 10:09:18 EDT 2016.

Topological.java

Below is the syntax highlighted version of Topological.java from §4.2 Directed Graphs.

```
/******************************************************************************
 *  Compilation:  javac Topological.java
 *  Execution:    java  Topological filename.txt delimiter
 *  Dependencies: Digraph.java DepthFirstOrder.java DirectedCycle.java
 *                EdgeWeightedDigraph.java EdgeWeightedDirectedCycle.java
 *                SymbolDigraph.java
 *  Data files:   http://algs4.cs.princeton.edu/42digraph/jobs.txt
 *
 *  Compute topological ordering of a DAG or edge-weighted DAG.
 *  Runs in O(E + V) time.
 *
 *  % java Topological jobs.txt "/"
 *  Calculus
 *  Linear Algebra
 *  Introduction to CS
 *  Advanced Programming
 *  Algorithms
 *  Theoretical CS
 *  Artificial Intelligence
 *  Robotics
 *  Machine Learning
 *  Neural Networks
 *  Databases
 *  Scientific Computing
 *  Computational Biology
 *
 ******************************************************************************/

/**
 *  The {@code Topological} class represents a data type for
 *  determining a topological order of a directed acyclic graph (DAG).
 *  Recall, a digraph has a topological order if and only if it is a DAG.
 *  The <em>hasOrder</em> operation determines whether the digraph has
 *  a topological order, and if so, the <em>order</em> operation
 *  returns one.
 *  <p>
 *  This implementation uses depth-first search.
 *  The constructor takes time proportional to <em>V</em> + <em>E</em>
 *  (in the worst case),
 *  where <em>V</em> is the number of vertices and <em>E</em> is the number of edges.
 *  Afterwards, the <em>hasOrder</em> and <em>rank</em> operations takes constant time;
 *  the <em>order</em> operation takes time proportional to <em>V</em>.
 *  <p>
 *  See {@link DirectedCycle}, {@link DirectedCycleX}, and
 *  {@link EdgeWeightedDirectedCycle} to compute a
 *  directed cycle if the digraph is not a DAG.
 *  See {@link TopologicalX} for a nonrecursive queue-based algorithm
 *  to compute a topological order of a DAG.
 *  <p>
 *  For additional documentation,
 *  see <a href="http://algs4.cs.princeton.edu/42digraph">Section 4.2</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class Topological {
    private Iterable<Integer> order;  // topological order
    private int[] rank;               // rank[v] = position of vertex v in topological order
```

```
/**
 * Determines whether the digraph {@code G} has a topological order and, if so,
 * finds such a topological order.
 * @param G the digraph
 */
public Topological(Digraph G) {
    DirectedCycle finder = new DirectedCycle(G);
    if (!finder.hasCycle()) {
        DepthFirstOrder dfs = new DepthFirstOrder(G);
        order = dfs.reversePost();
        rank = new int[G.V()];
        int i = 0;
        for (int v : order)
            rank[v] = i++;
    }
}

/**
 * Determines whether the edge-weighted digraph {@code G} has a topological
 * order and, if so, finds such an order.
 * @param G the edge-weighted digraph
 */
public Topological(EdgeWeightedDigraph G) {
    EdgeWeightedDirectedCycle finder = new EdgeWeightedDirectedCycle(G);
    if (!finder.hasCycle()) {
        DepthFirstOrder dfs = new DepthFirstOrder(G);
        order = dfs.reversePost();
    }
}

/**
 * Returns a topological order if the digraph has a topologial order,
 * and {@code null} otherwise.
 * @return a topological order of the vertices (as an interable) if the
 *    digraph has a topological order (or equivalently, if the digraph is a DAG),
 *    and {@code null} otherwise
 */
public Iterable<Integer> order() {
    return order;
}

/**
 * Does the digraph have a topological order?
 * @return {@code true} if the digraph has a topological order (or equivalently,
 *    if the digraph is a DAG), and {@code false} otherwise
 */
public boolean hasOrder() {
    return order != null;
}

/**
 * Does the digraph have a topological order?
 * @return {@code true} if the digraph has a topological order (or equivalently,
 *    if the digraph is a DAG), and {@code false} otherwise
 * @deprecated Replaced by {@link #hasOrder()}.
 */
@Deprecated
public boolean isDAG() {
    return hasOrder();
}

/**
 * The the rank of vertex {@code v} in the topological order;
 * -1 if the digraph is not a DAG
 *
 * @param v the vertex
 * @return the position of vertex {@code v} in a topological order
 *    of the digraph; -1 if the digraph is not a DAG
 * @throws IllegalArgumentException unless {@code 0 <= v < V}
 */
public int rank(int v) {
```

```
        validateVertex(v);
        if (hasOrder()) return rank[v];
        else            return -1;
    }

    // throw an IllegalArgumentException unless {@code 0 <= v < V}
    private void validateVertex(int v) {
        int V = rank.length;
        if (v < 0 || v >= V)
            throw new IllegalArgumentException("vertex " + v + " is not between 0 and " + (V-1));
    }

    /**
     * Unit tests the {@code Topological} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        String filename  = args[0];
        String delimiter = args[1];
        SymbolDigraph sg = new SymbolDigraph(filename, delimiter);
        Topological topological = new Topological(sg.digraph());
        for (int v : topological.order()) {
            StdOut.println(sg.nameOf(v));
        }
    }

}
```


Last updated: Sat Jul 15 19:24:43 EDT 2017.

Bipartite.java

Below is the syntax highlighted version of Bipartite.java from §4.1 Undirected Graphs.

```
/******************************************************************************
 *  Compilation:  javac Bipartite.java
 *  Execution:    java  Bipartite V E F
 *  Dependencies: Graph.java
 *  Data files:   http://algs4.cs.princeton.edu/41graph/tinyG.txt
 *                http://algs4.cs.princeton.edu/41graph/mediumG.txt
 *                http://algs4.cs.princeton.edu/41graph/largeG.txt
 *
 *  Given a graph, find either (i) a bipartition or (ii) an odd-length cycle.
 *  Runs in O(E + V) time.
 *
 *
 ******************************************************************************/


/**
 *  The {@code Bipartite} class represents a data type for
 *  determining whether an undirected graph is bipartite or whether
 *  it has an odd-length cycle.
 *  The <em>isBipartite</em> operation determines whether the graph is
 *  bipartite. If so, the <em>color</em> operation determines a
 *  bipartition; if not, the <em>oddCycle</em> operation determines a
 *  cycle with an odd number of edges.
 *  <p>
 *  This implementation uses depth-first search.
 *  The constructor takes time proportional to <em>V</em> + <em>E</em>
 *  (in the worst case),
 *  where <em>V</em> is the number of vertices and <em>E</em> is the number of edges.
 *  Afterwards, the <em>isBipartite</em> and <em>color</em> operations
 *  take constant time; the <em>oddCycle</em> operation takes time proportional
 *  to the length of the cycle.
 *  See {@link BipartiteX} for a nonrecursive version that uses breadth-first
 *  search.
 *  <p>
 *  For additional documentation, see <a href="http://algs4.cs.princeton.edu/41graph">Section 4.1</a>
 *  of <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class Bipartite {
    private boolean isBipartite;   // is the graph bipartite?
    private boolean[] color;       // color[v] gives vertices on one side of bipartition
    private boolean[] marked;      // marked[v] = true if v has been visited in DFS
    private int[] edgeTo;          // edgeTo[v] = last edge on path to v
    private Stack<Integer> cycle;  // odd-length cycle

    /**
     * Determines whether an undirected graph is bipartite and finds either a
     * bipartition or an odd-length cycle.
     *
     * @param  G the graph
     */
    public Bipartite(Graph G) {
        isBipartite = true;
        color  = new boolean[G.V()];
        marked = new boolean[G.V()];
        edgeTo = new int[G.V()];

        for (int v = 0; v < G.V(); v++) {
            if (!marked[v]) {
                dfs(G, v);
            }
        }
        assert check(G);
    }

    private void dfs(Graph G, int v) {
        marked[v] = true;
        for (int w : G.adj(v)) {

            // short circuit if odd-length cycle found
```

```
        if (cycle != null) return;

        // found uncolored vertex, so recur
        if (!marked[w]) {
            edgeTo[w] = v;
            color[w] = !color[v];
            dfs(G, w);
        }

        // if v-w create an odd-length cycle, find it
        else if (color[w] == color[v]) {
            isBipartite = false;
            cycle = new Stack<Integer>();
            cycle.push(w);  // don't need this unless you want to include start vertex twice
            for (int x = v; x != w; x = edgeTo[x]) {
                cycle.push(x);
            }
            cycle.push(w);
        }
    }
}

/**
 * Returns true if the graph is bipartite.
 *
 * @return {@code true} if the graph is bipartite; {@code false} otherwise
 */
public boolean isBipartite() {
    return isBipartite;
}

/**
 * Returns the side of the bipartite that vertex {@code v} is on.
 *
 * @param  v the vertex
 * @return the side of the bipartition that vertex {@code v} is on; two vertices
 *         are in the same side of the bipartition if and only if they have the
 *         same color
 * @throws IllegalArgumentException unless {@code 0 <= v < V}
 * @throws UnsupportedOperationException if this method is called when the graph
 *         is not bipartite
 */
public boolean color(int v) {
    validateVertex(v);
    if (!isBipartite)
        throw new UnsupportedOperationException("graph is not bipartite");
    return color[v];
}

/**
 * Returns an odd-length cycle if the graph is not bipartite, and
 * {@code null} otherwise.
 *
 * @return an odd-length cycle if the graph is not bipartite
 *         (and hence has an odd-length cycle), and {@code null}
 *         otherwise
 */
public Iterable<Integer> oddCycle() {
    return cycle;
}

private boolean check(Graph G) {
    // graph is bipartite
    if (isBipartite) {
        for (int v = 0; v < G.V(); v++) {
            for (int w : G.adj(v)) {
                if (color[v] == color[w]) {
                    System.err.printf("edge %d-%d with %d and %d in same side of bipartition\n", v, w, v, w);
                    return false;
                }
            }
        }
    }

    // graph has an odd-length cycle
    else {
        // verify cycle
        int first = -1, last = -1;
        for (int v : oddCycle()) {
            if (first == -1) first = v;
            last = v;
        }
```

```
            if (first != last) {
                System.err.printf("cycle begins with %d and ends with %d\n", first, last);
                return false;
            }
        }

        return true;
    }

    // throw an IllegalArgumentException unless {@code 0 <= v < V}
    private void validateVertex(int v) {
        int V = marked.length;
        if (v < 0 || v >= V)
            throw new IllegalArgumentException("vertex " + v + " is not between 0 and " + (V-1));
    }

    /**
     * Unit tests the {@code Bipartite} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        int V1 = Integer.parseInt(args[0]);
        int V2 = Integer.parseInt(args[1]);
        int E  = Integer.parseInt(args[2]);
        int F  = Integer.parseInt(args[3]);

        // create random bipartite graph with V1 vertices on left side,
        // V2 vertices on right side, and E edges; then add F random edges
        Graph G = GraphGenerator.bipartite(V1, V2, E);
        for (int i = 0; i < F; i++) {
            int v = StdRandom.uniform(V1 + V2);
            int w = StdRandom.uniform(V1 + V2);
            G.addEdge(v, w);
        }

        StdOut.println(G);


        Bipartite b = new Bipartite(G);
        if (b.isBipartite()) {
            StdOut.println("Graph is bipartite");
            for (int v = 0; v < G.V(); v++) {
                StdOut.println(v + ": " + b.color(v));
            }
        }
        else {
            StdOut.print("Graph has an odd-length cycle: ");
            for (int x : b.oddCycle()) {
                StdOut.print(x + " ");
            }
            StdOut.println();
        }
    }


}
```

CC.java

Below is the syntax highlighted version of CC.java from §4.1 Undirected Graphs.

```
/******************************************************************************
 *  Compilation:  javac CC.java
 *  Execution:    java CC filename.txt
 *  Dependencies: Graph.java StdOut.java Queue.java
 *  Data files:   http://algs4.cs.princeton.edu/41graph/tinyG.txt
 *                http://algs4.cs.princeton.edu/41graph/mediumG.txt
 *                http://algs4.cs.princeton.edu/41graph/largeG.txt
 *
 *  Compute connected components using depth first search.
 *  Runs in O(E + V) time.
 *
 *  % java CC tinyG.txt
 *  3 components
 *  0 1 2 3 4 5 6
 *  7 8
 *  9 10 11 12
 *
 *  % java CC mediumG.txt
 *  1 components
 *  0 1 2 3 4 5 6 7 8 9 10 ...
 *
 *  % java -Xss50m CC largeG.txt
 *  1 components
 *  0 1 2 3 4 5 6 7 8 9 10 ...
 *
 *  Note: This implementation uses a recursive DFS. To avoid needing
 *        a potentially very large stack size, replace with a non-recurisve
 *        DFS ala NonrecursiveDFS.java.
 *
 ******************************************************************************/

/**
 *  The {@code CC} class represents a data type for
 *  determining the connected components in an undirected graph.
 *  The <em>id</em> operation determines in which connected component
 *  a given vertex lies; the <em>connected</em> operation
 *  determines whether two vertices are in the same connected component;
 *  the <em>count</em> operation determines the number of connected
 *  components; and the <em>size</em> operation determines the number
 *  of vertices in the connect component containing a given vertex.

 *  The <em>component identifier</em> of a connected component is one of the
 *  vertices in the connected component: two vertices have the same component
 *  identifier if and only if they are in the same connected component.

 *  <p>
 *  This implementation uses depth-first search.
 *  The constructor takes time proportional to <em>V</em> + <em>E</em>
 *  (in the worst case),
 *  where <em>V</em> is the number of vertices and <em>E</em> is the number of edges.
 *  Afterwards, the <em>id</em>, <em>count</em>, <em>connected</em>,
 *  and <em>size</em> operations take constant time.
 *  <p>
 *  For additional documentation, see <a href="http://algs4.cs.princeton.edu/41graph">Section 4.1</a>
 *  of <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class CC {
    private boolean[] marked;   // marked[v] = has vertex v been marked?
    private int[] id;           // id[v] = id of connected component containing v
    private int[] size;         // size[id] = number of vertices in given component
    private int count;          // number of connected components
```

```
/**
 * Computes the connected components of the undirected graph {@code G}.
 *
 * @param G the undirected graph
 */
public CC(Graph G) {
    marked = new boolean[G.V()];
    id = new int[G.V()];
    size = new int[G.V()];
    for (int v = 0; v < G.V(); v++) {
        if (!marked[v]) {
            dfs(G, v);
            count++;
        }
    }
}

/**
 * Computes the connected components of the edge-weighted graph {@code G}.
 *
 * @param G the edge-weighted graph
 */
public CC(EdgeWeightedGraph G) {
    marked = new boolean[G.V()];
    id = new int[G.V()];
    size = new int[G.V()];
    for (int v = 0; v < G.V(); v++) {
        if (!marked[v]) {
            dfs(G, v);
            count++;
        }
    }
}

// depth-first search for a Graph
private void dfs(Graph G, int v) {
    marked[v] = true;
    id[v] = count;
    size[count]++;
    for (int w : G.adj(v)) {
        if (!marked[w]) {
            dfs(G, w);
        }
    }
}

// depth-first search for an EdgeWeightedGraph
private void dfs(EdgeWeightedGraph G, int v) {
    marked[v] = true;
    id[v] = count;
    size[count]++;
    for (Edge e : G.adj(v)) {
        int w = e.other(v);
        if (!marked[w]) {
            dfs(G, w);
        }
    }
}


/**
 * Returns the component id of the connected component containing vertex {@code v}.
 *
 * @param  v the vertex
 * @return the component id of the connected component containing vertex {@code v}
 * @throws IllegalArgumentException unless {@code 0 <= v < V}
 */
public int id(int v) {
    validateVertex(v);
    return id[v];
}

/**
 * Returns the number of vertices in the connected component containing vertex {@code v}.
 *
```

```
     * @param  v the vertex
     * @return the number of vertices in the connected component containing vertex {@code v}
     * @throws IllegalArgumentException unless {@code 0 <= v < V}
     */
    public int size(int v) {
        validateVertex(v);
        return size[id[v]];
    }

    /**
     * Returns the number of connected components in the graph {@code G}.
     *
     * @return the number of connected components in the graph {@code G}
     */
    public int count() {
        return count;
    }

    /**
     * Returns true if vertices {@code v} and {@code w} are in the same
     * connected component.
     *
     * @param  v one vertex
     * @param  w the other vertex
     * @return {@code true} if vertices {@code v} and {@code w} are in the same
     *         connected component; {@code false} otherwise
     * @throws IllegalArgumentException unless {@code 0 <= v < V}
     * @throws IllegalArgumentException unless {@code 0 <= w < V}
     */
    public boolean connected(int v, int w) {
        validateVertex(v);
        validateVertex(w);
        return id(v) == id(w);
    }

    /**
     * Returns true if vertices {@code v} and {@code w} are in the same
     * connected component.
     *
     * @param  v one vertex
     * @param  w the other vertex
     * @return {@code true} if vertices {@code v} and {@code w} are in the same
     *         connected component; {@code false} otherwise
     * @throws IllegalArgumentException unless {@code 0 <= v < V}
     * @throws IllegalArgumentException unless {@code 0 <= w < V}
     * @deprecated Replaced by {@link #connected(int, int)}.
     */
    @Deprecated
    public boolean areConnected(int v, int w) {
        validateVertex(v);
        validateVertex(w);
        return id(v) == id(w);
    }

    // throw an IllegalArgumentException unless {@code 0 <= v < V}
    private void validateVertex(int v) {
        int V = marked.length;
        if (v < 0 || v >= V)
            throw new IllegalArgumentException("vertex " + v + " is not between 0 and " + (V-1));
    }

    /**
     * Unit tests the {@code CC} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        In in = new In(args[0]);
        Graph G = new Graph(in);
        CC cc = new CC(G);

        // number of connected components
        int m = cc.count();
        StdOut.println(m + " components");
```

```
        // compute list of vertices in each connected component
        Queue<Integer>[] components = (Queue<Integer>[]) new Queue[m];
        for (int i = 0; i < m; i++) {
            components[i] = new Queue<Integer>();
        }
        for (int v = 0; v < G.V(); v++) {
            components[cc.id(v)].enqueue(v);
        }

        // print results
        for (int i = 0; i < m; i++) {
            for (int v : components[i]) {
                StdOut.print(v + " ");
            }
            StdOut.println();
        }
    }
}
```

KosarajuSharirSCC.java

Below is the syntax highlighted version of KosarajuSharirSCC.java from §4.2 Directed Graphs.

```
/******************************************************************************
 *  Compilation:  javac KosarajuSharirSCC.java
 *  Execution:    java KosarajuSharirSCC filename.txt
 *  Dependencies: Digraph.java TransitiveClosure.java StdOut.java In.java
 *  Data files:   http://algs4.cs.princeton.edu/42digraph/tinyDG.txt
 *                http://algs4.cs.princeton.edu/42digraph/mediumDG.txt
 *                http://algs4.cs.princeton.edu/42digraph/largeDG.txt
 *
 *  Compute the strongly-connected components of a digraph using the
 *  Kosaraju-Sharir algorithm.
 *
 *  Runs in O(E + V) time.
 *
 *  % java KosarajuSharirSCC tinyDG.txt
 *  5 strong components
 *  1
 *  0 2 3 4 5
 *  9 10 11 12
 *  6 8
 *  7
 *
 *  % java KosarajuSharirSCC mediumDG.txt
 *  10 strong components
 *  21
 *  2 5 6 8 9 11 12 13 15 16 18 19 22 23 25 26 28 29 30 31 32 33 34 35 37 38 39 40 42 43 44 46 47 48 49
 *  14
 *  3 4 17 20 24 27 36
 *  41
 *  7
 *  45
 *  1
 *  0
 *  10
 *
 *  % java -Xss50m KosarajuSharirSCC mediumDG.txt
 *  25 strong components
 *  7 11 32 36 61 84 95 116 121 128 230   ...
 *  28 73 80 104 115 143 149 164 184 185  ...
 *  38 40 200 201 207 218 286 387 418 422 ...
 *  12 14 56 78 87 103 216 269 271 272    ...
 *  42 48 112 135 160 217 243 246 273 346 ...
 *  46 76 96 97 224 237 297 303 308 309   ...
 *  9 15 21 22 27 90 167 214 220 225 227  ...
 *  74 99 133 146 161 166 202 205 245 262 ...
 *  43 83 94 120 125 183 195 206 244 254  ...
 *  1 13 54 91 92 93 106 140 156 194 208  ...
 *  10 39 67 69 131 144 145 154 168 258   ...
 *  6 52 66 113 118 122 139 147 212 213   ...
 *  8 127 150 182 203 204 249 367 400 432 ...
 *  63 65 101 107 108 136 169 170 171 173 ...
 *  55 71 102 155 159 198 228 252 325 419 ...
 *  4 25 34 58 70 152 172 196 199 210 226 ...
 *  2 44 50 88 109 138 141 178 197 211    ...
 *  57 89 129 162 174 179 188 209 238 276 ...
 *  33 41 49 119 126 132 148 181 215 221  ...
 *  3 18 23 26 35 64 105 124 157 186 251  ...
 *  5 16 17 20 31 47 81 98 158 180 187    ...
 *  24 29 51 59 75 82 100 114 117 134 151 ...
 *  30 45 53 60 72 85 111 130 137 142 163 ...
 *  19 37 62 77 79 110 153 352 353 361    ...
 *  0 68 86 123 165 176 193 239 289 336   ...
 *
 ******************************************************************************/

/**
```

```
 *  The {@code KosarajuSharirSCC} class represents a data type for
 *  determining the strong components in a digraph.
 *  The <em>id</em> operation determines in which strong component
 *  a given vertex lies; the <em>areStronglyConnected</em> operation
 *  determines whether two vertices are in the same strong component;
 *  and the <em>count</em> operation determines the number of strong
 *  components.

 *  The <em>component identifier</em> of a component is one of the
 *  vertices in the strong component: two vertices have the same component
 *  identifier if and only if they are in the same strong component.

 *  <p>
 *  This implementation uses the Kosaraju-Sharir algorithm.
 *  The constructor takes time proportional to <em>V</em> + <em>E</em>
 *  (in the worst case),
 *  where <em>V</em> is the number of vertices and <em>E</em> is the number of edges.
 *  Afterwards, the <em>id</em>, <em>count</em>, and <em>areStronglyConnected</em>
 *  operations take constant time.
 *  For alternate implementations of the same API, see
 *  {@link TarjanSCC} and {@link GabowSCC}.
 *  <p>
 *  For additional documentation,
 *  see <a href="http://algs4.cs.princeton.edu/42digraph">Section 4.2</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class KosarajuSharirSCC {
    private boolean[] marked;     // marked[v] = has vertex v been visited?
    private int[] id;             // id[v] = id of strong component containing v
    private int count;            // number of strongly-connected components

    /**
     * Computes the strong components of the digraph {@code G}.
     * @param G the digraph
     */
    public KosarajuSharirSCC(Digraph G) {

        // compute reverse postorder of reverse graph
        DepthFirstOrder dfs = new DepthFirstOrder(G.reverse());

        // run DFS on G, using reverse postorder to guide calculation
        marked = new boolean[G.V()];
        id = new int[G.V()];
        for (int v : dfs.reversePost()) {
            if (!marked[v]) {
                dfs(G, v);
                count++;
            }
        }

        // check that id[] gives strong components
        assert check(G);
    }

    // DFS on graph G
    private void dfs(Digraph G, int v) {
        marked[v] = true;
        id[v] = count;
        for (int w : G.adj(v)) {
            if (!marked[w]) dfs(G, w);
        }
    }

    /**
     * Returns the number of strong components.
     * @return the number of strong components
     */
    public int count() {
        return count;
    }

    /**
```

```
     * Are vertices {@code v} and {@code w} in the same strong component?
     * @param  v one vertex
     * @param  w the other vertex
     * @return {@code true} if vertices {@code v} and {@code w} are in the same
     *         strong component, and {@code false} otherwise
     * @throws IllegalArgumentException unless {@code 0 <= v < V}
     * @throws IllegalArgumentException unless {@code 0 <= w < V}
     */
    public boolean stronglyConnected(int v, int w) {
        validateVertex(v);
        validateVertex(w);
        return id[v] == id[w];
    }

    /**
     * Returns the component id of the strong component containing vertex {@code v}.
     * @param  v the vertex
     * @return the component id of the strong component containing vertex {@code v}
     * @throws IllegalArgumentException unless {@code 0 <= s < V}
     */
    public int id(int v) {
        validateVertex(v);
        return id[v];
    }

    // does the id[] array contain the strongly connected components?
    private boolean check(Digraph G) {
        TransitiveClosure tc = new TransitiveClosure(G);
        for (int v = 0; v < G.V(); v++) {
            for (int w = 0; w < G.V(); w++) {
                if (stronglyConnected(v, w) != (tc.reachable(v, w) && tc.reachable(w, v)))
                    return false;
            }
        }
        return true;
    }

    // throw an IllegalArgumentException unless {@code 0 <= v < V}
    private void validateVertex(int v) {
        int V = marked.length;
        if (v < 0 || v >= V)
            throw new IllegalArgumentException("vertex " + v + " is not between 0 and " + (V-1));
    }

    /**
     * Unit tests the {@code KosarajuSharirSCC} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        In in = new In(args[0]);
        Digraph G = new Digraph(in);
        KosarajuSharirSCC scc = new KosarajuSharirSCC(G);

        // number of connected components
        int m = scc.count();
        StdOut.println(m + " strong components");

        // compute list of vertices in each strong component
        Queue<Integer>[] components = (Queue<Integer>[]) new Queue[m];
        for (int i = 0; i < m; i++) {
            components[i] = new Queue<Integer>();
        }
        for (int v = 0; v < G.V(); v++) {
            components[scc.id(v)].enqueue(v);
        }

        // print results
        for (int i = 0; i < m; i++) {
            for (int v : components[i]) {
                StdOut.print(v + " ");
            }
            StdOut.println();
        }
```

```
        }

}
```


Last updated: Sat Nov 19 09:05:38 EST 2016.

EulerianCycle.java

Below is the syntax highlighted version of EulerianCycle.java from §4.1 Undirected Graphs.

```
/******************************************************************************
 *  Compilation:  javac EulerianCycle.java
 *  Execution:    java  EulerianCycle V E
 *  Dependencies: Graph.java Stack.java StdOut.java
 *
 *  Find an Eulerian cycle in a graph, if one exists.
 *
 *  Runs in O(E + V) time.
 *
 *  This implementation is tricker than the one for digraphs because
 *  when we use edge v-w from v's adjacency list, we must be careful
 *  not to use the second copy of the edge from w's adjaceny list.
 *
 ******************************************************************************/

/**
 *  The {@code EulerianCycle} class represents a data type
 *  for finding an Eulerian cycle or path in a graph.
 *  An <em>Eulerian cycle</em> is a cycle (not necessarily simple) that
 *  uses every edge in the graph exactly once.
 *  <p>
 *  This implementation uses a nonrecursive depth-first search.
 *  The constructor runs in O(<Em>E</em> + <em>V</em>) time,
 *  and uses O(<em>E</em> + <em>V</em>) extra space, where <em>E</em> is the
 *  number of edges and <em>V</em> the number of vertices
 *  All other methods take O(1) time.
 *  <p>
 *  To compute Eulerian paths in graphs, see {@link EulerianPath}.
 *  To compute Eulerian cycles and paths in digraphs, see
 *  {@link DirectedEulerianCycle} and {@link DirectedEulerianPath}.
 *  <p>
 *  For additional documentation,
 *  see <a href="http://algs4.cs.princeton.edu/41graph">Section 4.1</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 *  @author Nate Liu
 */
public class EulerianCycle {
    private Stack<Integer> cycle = new Stack<Integer>();  // Eulerian cycle; null if no such cycle

    // an undirected edge, with a field to indicate whether the edge has already been used
    private static class Edge {
        private final int v;
        private final int w;
        private boolean isUsed;

        public Edge(int v, int w) {
            this.v = v;
            this.w = w;
            isUsed = false;
        }

        // returns the other vertex of the edge
        public int other(int vertex) {
            if      (vertex == v) return w;
            else if (vertex == w) return v;
            else throw new IllegalArgumentException("Illegal endpoint");
        }
    }
```

```
/**
 * Computes an Eulerian cycle in the specified graph, if one exists.
 *
 * @param G the graph
 */
public EulerianCycle(Graph G) {

    // must have at least one edge
    if (G.E() == 0) return;

    // necessary condition: all vertices have even degree
    // (this test is needed or it might find an Eulerian path instead of cycle)
    for (int v = 0; v < G.V(); v++)
        if (G.degree(v) % 2 != 0)
            return;

    // create local view of adjacency lists, to iterate one vertex at a time
    // the helper Edge data type is used to avoid exploring both copies of an edge v-w
    Queue<Edge>[] adj = (Queue<Edge>[]) new Queue[G.V()];
    for (int v = 0; v < G.V(); v++)
        adj[v] = new Queue<Edge>();

    for (int v = 0; v < G.V(); v++) {
        int selfLoops = 0;
        for (int w : G.adj(v)) {
            // careful with self loops
            if (v == w) {
                if (selfLoops % 2 == 0) {
                    Edge e = new Edge(v, w);
                    adj[v].enqueue(e);
                    adj[w].enqueue(e);
                }
                selfLoops++;
            }
            else if (v < w) {
                Edge e = new Edge(v, w);
                adj[v].enqueue(e);
                adj[w].enqueue(e);
            }
        }
    }

    // initialize stack with any non-isolated vertex
    int s = nonIsolatedVertex(G);
    Stack<Integer> stack = new Stack<Integer>();
    stack.push(s);

    // greedily search through edges in iterative DFS style
    cycle = new Stack<Integer>();
    while (!stack.isEmpty()) {
        int v = stack.pop();
        while (!adj[v].isEmpty()) {
            Edge edge = adj[v].dequeue();
            if (edge.isUsed) continue;
            edge.isUsed = true;
            stack.push(v);
            v = edge.other(v);
        }
        // push vertex with no more leaving edges to cycle
        cycle.push(v);
    }

    // check if all edges are used
    if (cycle.size() != G.E() + 1)
        cycle = null;

    assert certifySolution(G);
}

/**
```

```
     * Returns the sequence of vertices on an Eulerian cycle.
     *
     * @return the sequence of vertices on an Eulerian cycle;
     *         {@code null} if no such cycle
     */
    public Iterable<Integer> cycle() {
        return cycle;
    }

    /**
     * Returns true if the graph has an Eulerian cycle.
     *
     * @return {@code true} if the graph has an Eulerian cycle;
     *         {@code false} otherwise
     */
    public boolean hasEulerianCycle() {
        return cycle != null;
    }

    // returns any non-isolated vertex; -1 if no such vertex
    private static int nonIsolatedVertex(Graph G) {
        for (int v = 0; v < G.V(); v++)
            if (G.degree(v) > 0)
                return v;
        return -1;
    }

    /***************************************************************************
     *
     *  The code below is solely for testing correctness of the data type.
     *
     ***************************************************************************/

    // Determines whether a graph has an Eulerian cycle using necessary
    // and sufficient conditions (without computing the cycle itself):
    //    - at least one edge
    //    - degree(v) is even for every vertex v
    //    - the graph is connected (ignoring isolated vertices)
    private static boolean satisfiesNecessaryAndSufficientConditions(Graph G) {

        // Condition 0: at least 1 edge
        if (G.E() == 0) return false;

        // Condition 1: degree(v) is even for every vertex
        for (int v = 0; v < G.V(); v++)
            if (G.degree(v) % 2 != 0)
                return false;

        // Condition 2: graph is connected, ignoring isolated vertices
        int s = nonIsolatedVertex(G);
        BreadthFirstPaths bfs = new BreadthFirstPaths(G, s);
        for (int v = 0; v < G.V(); v++)
            if (G.degree(v) > 0 && !bfs.hasPathTo(v))
                return false;

        return true;
    }

    // check that solution is correct
    private boolean certifySolution(Graph G) {

        // internal consistency check
        if (hasEulerianCycle() == (cycle() == null)) return false;

        // hashEulerianCycle() returns correct value
        if (hasEulerianCycle() != satisfiesNecessaryAndSufficientConditions(G)) return false;

        // nothing else to check if no Eulerian cycle
        if (cycle == null) return true;

        // check that cycle() uses correct number of edges
```

```
        if (cycle.size() != G.E() + 1) return false;

        // check that cycle() is a cycle of G
        // TODO

        // check that first and last vertices in cycle() are the same
        int first = -1, last = -1;
        for (int v : cycle()) {
            if (first == -1) first = v;
            last = v;
        }
        if (first != last) return false;

        return true;
    }

    private static void unitTest(Graph G, String description) {
        StdOut.println(description);
        StdOut.println("-------------------------------------");
        StdOut.print(G);

        EulerianCycle euler = new EulerianCycle(G);

        StdOut.print("Eulerian cycle: ");
        if (euler.hasEulerianCycle()) {
            for (int v : euler.cycle()) {
                StdOut.print(v + " ");
            }
            StdOut.println();
        }
        else {
            StdOut.println("none");
        }
        StdOut.println();
    }


    /**
     * Unit tests the {@code EulerianCycle} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        int V = Integer.parseInt(args[0]);
        int E = Integer.parseInt(args[1]);

        // Eulerian cycle
        Graph G1 = GraphGenerator.eulerianCycle(V, E);
        unitTest(G1, "Eulerian cycle");

        // Eulerian path
        Graph G2 = GraphGenerator.eulerianPath(V, E);
        unitTest(G2, "Eulerian path");

        // empty graph
        Graph G3 = new Graph(V);
        unitTest(G3, "empty graph");

        // self loop
        Graph G4 = new Graph(V);
        int v4 = StdRandom.uniform(V);
        G4.addEdge(v4, v4);
        unitTest(G4, "single self loop");

        // union of two disjoint cycles
        Graph H1 = GraphGenerator.eulerianCycle(V/2, E/2);
        Graph H2 = GraphGenerator.eulerianCycle(V - V/2, E - E/2);
        int[] perm = new int[V];
        for (int i = 0; i < V; i++)
            perm[i] = i;
        StdRandom.shuffle(perm);
```

```
        Graph G5 = new Graph(V);
        for (int v = 0; v < H1.V(); v++)
            for (int w : H1.adj(v))
                G5.addEdge(perm[v], perm[w]);
        for (int v = 0; v < H2.V(); v++)
            for (int w : H2.adj(v))
                G5.addEdge(perm[V/2 + v], perm[V/2 + w]);
        unitTest(G5, "Union of two disjoint cycles");

        // random digraph
        Graph G6 = GraphGenerator.simple(V, E);
        unitTest(G6, "simple graph");
    }
}
```

DirectedEulerianCycle.java

Below is the syntax highlighted version of DirectedEulerianCycle.java from §4.2 Directed Graphs.

```
/******************************************************************************
 *  Compilation:  javac DirectedEulerianCycle.java
 *  Execution:    java DirectedEulerianCycle V E
 *  Dependencies: Digraph.java Stack.java StdOut.java
 *                BreadthFirstPaths.java
 *                DigraphGenerator.java StdRandom.java
 *
 *  Find an Eulerian cycle in a digraph, if one exists.
 *
 ******************************************************************************/

import java.util.Iterator;

/**
 *  The {@code DirectedEulerianCycle} class represents a data type
 *  for finding an Eulerian cycle or path in a digraph.
 *  An <em>Eulerian cycle</em> is a cycle (not necessarily simple) that
 *  uses every edge in the digraph exactly once.
 *  <p>
 *  This implementation uses a nonrecursive depth-first search.
 *  The constructor runs in O(<Em>E</em> + <em>V</em>) time,
 *  and uses O(<em>V</em>) extra space, where <em>E</em> is the
 *  number of edges and <em>V</em> the number of vertices
 *  All other methods take O(1) time.
 *  <p>
 *  To compute Eulerian paths in digraphs, see {@link DirectedEulerianPath}.
 *  To compute Eulerian cycles and paths in undirected graphs, see
 *  {@link EulerianCycle} and {@link EulerianPath}.
 *  <p>
 *  For additional documentation,
 *  see <a href="http://algs4.cs.princeton.edu/42digraph">Section 4.2</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 *  @author Nate Liu
 */
public class DirectedEulerianCycle {
    private Stack<Integer> cycle = null;  // Eulerian cycle; null if no such cylce

    /**
     * Computes an Eulerian cycle in the specified digraph, if one exists.
     *
     * @param G the digraph
     */
    public DirectedEulerianCycle(Digraph G) {

        // must have at least one edge
        if (G.E() == 0) return;

        // necessary condition: indegree(v) = outdegree(v) for each vertex v
        // (without this check, DFS might return a path instead of a cycle)
        for (int v = 0; v < G.V(); v++)
            if (G.outdegree(v) != G.indegree(v))
                return;

        // create local view of adjacency lists, to iterate one vertex at a time
```

```
        Iterator<Integer>[] adj = (Iterator<Integer>[]) new Iterator[G.V()];
        for (int v = 0; v < G.V(); v++)
            adj[v] = G.adj(v).iterator();

        // initialize stack with any non-isolated vertex
        int s = nonIsolatedVertex(G);
        Stack<Integer> stack = new Stack<Integer>();
        stack.push(s);

        // greedily add to putative cycle, depth-first search style
        cycle = new Stack<Integer>();
        while (!stack.isEmpty()) {
            int v = stack.pop();
            while (adj[v].hasNext()) {
                stack.push(v);
                v = adj[v].next();
            }
            // add vertex with no more leaving edges to cycle
            cycle.push(v);
        }

        // check if all edges have been used
        // (in case there are two or more vertex-disjoint Eulerian cycles)
        if (cycle.size() != G.E() + 1)
            cycle = null;

        assert certifySolution(G);
    }

    /**
     * Returns the sequence of vertices on an Eulerian cycle.
     *
     * @return the sequence of vertices on an Eulerian cycle;
     *         {@code null} if no such cycle
     */
    public Iterable<Integer> cycle() {
        return cycle;
    }

    /**
     * Returns true if the digraph has an Eulerian cycle.
     *
     * @return {@code true} if the digraph has an Eulerian cycle;
     *         {@code false} otherwise
     */
    public boolean hasEulerianCycle() {
        return cycle != null;
    }

    // returns any non-isolated vertex; -1 if no such vertex
    private static int nonIsolatedVertex(Digraph G) {
        for (int v = 0; v < G.V(); v++)
            if (G.outdegree(v) > 0)
                return v;
        return -1;
    }


    /***************************************************************************
     *
     *  The code below is solely for testing correctness of the data type.
     *
     ***************************************************************************/

    // Determines whether a digraph has an Eulerian cycle using necessary
    // and sufficient conditions (without computing the cycle itself):
    //    - at least one edge
```

```
//    - indegree(v) = outdegree(v) for every vertex
//    - the graph is connected, when viewed as an undirected graph
//      (ignoring isolated vertices)
private static boolean satisfiesNecessaryAndSufficientConditions(Digraph G) {

    // Condition 0: at least 1 edge
    if (G.E() == 0) return false;

    // Condition 1: indegree(v) == outdegree(v) for every vertex
    for (int v = 0; v < G.V(); v++)
        if (G.outdegree(v) != G.indegree(v))
            return false;

    // Condition 2: graph is connected, ignoring isolated vertices
    Graph H = new Graph(G.V());
    for (int v = 0; v < G.V(); v++)
        for (int w : G.adj(v))
            H.addEdge(v, w);

    // check that all non-isolated vertices are conneted
    int s = nonIsolatedVertex(G);
    BreadthFirstPaths bfs = new BreadthFirstPaths(H, s);
    for (int v = 0; v < G.V(); v++)
        if (H.degree(v) > 0 && !bfs.hasPathTo(v))
            return false;

    return true;
}

// check that solution is correct
private boolean certifySolution(Digraph G) {

    // internal consistency check
    if (hasEulerianCycle() == (cycle() == null)) return false;

    // hashEulerianCycle() returns correct value
    if (hasEulerianCycle() != satisfiesNecessaryAndSufficientConditions(G)) return false;

    // nothing else to check if no Eulerian cycle
    if (cycle == null) return true;

    // check that cycle() uses correct number of edges
    if (cycle.size() != G.E() + 1) return false;

    // check that cycle() is a directed cycle of G
    // TODO

    return true;
}


private static void unitTest(Digraph G, String description) {
    StdOut.println(description);
    StdOut.println("-------------------------------------");
    StdOut.print(G);

    DirectedEulerianCycle euler = new DirectedEulerianCycle(G);

    StdOut.print("Eulerian cycle: ");
    if (euler.hasEulerianCycle()) {
        for (int v : euler.cycle()) {
            StdOut.print(v + " ");
        }
        StdOut.println();
    }
    else {
        StdOut.println("none");
```

```
        }
        StdOut.println();
    }


    /**
     * Unit tests the {@code DirectedEulerianCycle} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        int V = Integer.parseInt(args[0]);
        int E = Integer.parseInt(args[1]);

        // Eulerian cycle
        Digraph G1 = DigraphGenerator.eulerianCycle(V, E);
        unitTest(G1, "Eulerian cycle");

        // Eulerian path
        Digraph G2 = DigraphGenerator.eulerianPath(V, E);
        unitTest(G2, "Eulerian path");

        // empty digraph
        Digraph G3 = new Digraph(V);
        unitTest(G3, "empty digraph");

        // self loop
        Digraph G4 = new Digraph(V);
        int v4 = StdRandom.uniform(V);
        G4.addEdge(v4, v4);
        unitTest(G4, "single self loop");

        // union of two disjoint cycles
        Digraph H1 = DigraphGenerator.eulerianCycle(V/2, E/2);
        Digraph H2 = DigraphGenerator.eulerianCycle(V - V/2, E - E/2);
        int[] perm = new int[V];
        for (int i = 0; i < V; i++)
            perm[i] = i;
        StdRandom.shuffle(perm);
        Digraph G5 = new Digraph(V);
        for (int v = 0; v < H1.V(); v++)
            for (int w : H1.adj(v))
                G5.addEdge(perm[v], perm[w]);
        for (int v = 0; v < H2.V(); v++)
            for (int w : H2.adj(v))
                G5.addEdge(perm[V/2 + v], perm[V/2 + w]);
        unitTest(G5, "Union of two disjoint cycles");

        // random digraph
        Digraph G6 = DigraphGenerator.simple(V, E);
        unitTest(G6, "simple digraph");

        // 4-vertex digraph
        Digraph G7 = new Digraph(new In("eulerianD.txt"));
        unitTest(G7, "4-vertex Eulerian digraph");
    }

}
```

TransitiveClosure.java

Below is the syntax highlighted version of TransitiveClosure.java from §4.2 Directed Graphs.

```
/******************************************************************************
 *  Compilation:  javac TransitiveClosure.java
 *  Execution:    java TransitiveClosure filename.txt
 *  Dependencies: Digraph.java DepthFirstDirectedPaths.java In.java StdOut.java
 *  Data files:   http://algs4.cs.princeton.edu/42digraph/tinyDG.txt
 *
 *  Compute transitive closure of a digraph and support
 *  reachability queries.
 *
 *  Preprocessing time: O(V(E + V)) time.
 *  Query time: O(1).
 *  Space: O(V^2).
 *
 *  % java TransitiveClosure tinyDG.txt
 *         0  1  2  3  4  5  6  7  8  9 10 11 12
 *  ------------------------------------------
 *    0:   T  T  T  T  T  T
 *    1:      T
 *    2:   T  T  T  T  T  T
 *    3:   T  T  T  T  T  T
 *    4:   T  T  T  T  T  T
 *    5:   T  T  T  T  T  T
 *    6:   T  T  T  T  T  T  T        T  T  T  T
 *    7:   T  T  T  T  T  T  T  T  T  T  T  T  T
 *    8:   T  T  T  T  T  T  T  T  T  T  T  T  T
 *    9:   T  T  T  T  T  T           T  T  T  T
 *   10:   T  T  T  T  T  T           T  T  T  T
 *   11:   T  T  T  T  T  T           T  T  T  T
 *   12:   T  T  T  T  T  T           T  T  T  T
 *
 ******************************************************************************/

/**
 *  The {@code TransitiveClosure} class represents a data type for
 *  computing the transitive closure of a digraph.
 *  <p>
 *  This implementation runs depth-first search from each vertex.
 *  The constructor takes time proportional to <em>V</em>(<em>V</em> + <em>E</em>)
 *  (in the worst case) and uses space proportional to <em>V</em><sup>2</sup>,
 *  where <em>V</em> is the number of vertices and <em>E</em> is the number of edges.
 *  <p>
 *  For large digraphs, you may want to consider a more sophisticated algorithm.
 *  <a href = "http://www.cs.hut.fi/~enu/thesis.html">Nuutila</a> proposes two
 *  algorithm for the problem (based on strong components and an interval representation)
 *  that runs in <em>E</em> + <em>V</em> time on typical digraphs.
 *
 *  For additional documentation,
 *  see <a href="http://algs4.cs.princeton.edu/42digraph">Section 4.2</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class TransitiveClosure {
    private DirectedDFS[] tc;  // tc[v] = reachable from v

    /**
     * Computes the transitive closure of the digraph {@code G}.
     * @param G the digraph
     */
```

```
    public TransitiveClosure(Digraph G) {
        tc = new DirectedDFS[G.V()];
        for (int v = 0; v < G.V(); v++)
            tc[v] = new DirectedDFS(G, v);
    }

    /**
     * Is there a directed path from vertex {@code v} to vertex {@code w} in the digraph?
     * @param  v the source vertex
     * @param  w the target vertex
     * @return {@code true} if there is a directed path from {@code v} to {@code w},
     *         {@code false} otherwise
     * @throws IllegalArgumentException unless {@code 0 <= v < V}
     * @throws IllegalArgumentException unless {@code 0 <= w < V}
     */
    public boolean reachable(int v, int w) {
        validateVertex(v);
        validateVertex(w);
        return tc[v].marked(w);
    }

    // throw an IllegalArgumentException unless {@code 0 <= v < V}
    private void validateVertex(int v) {
        int V = tc.length;
        if (v < 0 || v >= V)
            throw new IllegalArgumentException("vertex " + v + " is not between 0 and " + (V-1));
    }

    /**
     * Unit tests the {@code TransitiveClosure} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        In in = new In(args[0]);
        Digraph G = new Digraph(in);

        TransitiveClosure tc = new TransitiveClosure(G);

        // print header
        StdOut.print("     ");
        for (int v = 0; v < G.V(); v++)
            StdOut.printf("%3d", v);
        StdOut.println();
        StdOut.println("-------------------------------------------");

        // print transitive closure
        for (int v = 0; v < G.V(); v++) {
            StdOut.printf("%3d: ", v);
            for (int w = 0; w < G.V(); w++) {
                if (tc.reachable(v, w)) StdOut.printf("  T");
                else                    StdOut.printf("   ");
            }
            StdOut.println();
        }
    }

}
```

KruskalMST.java

Below is the syntax highlighted version of KruskalMST.java from §4.3 Minimum Spanning Trees.

```
/******************************************************************************
 *  Compilation:  javac KruskalMST.java
 *  Execution:    java  KruskalMST filename.txt
 *  Dependencies: EdgeWeightedGraph.java Edge.java Queue.java
 *                UF.java In.java StdOut.java
 *  Data files:   http://algs4.cs.princeton.edu/43mst/tinyEWG.txt
 *                http://algs4.cs.princeton.edu/43mst/mediumEWG.txt
 *                http://algs4.cs.princeton.edu/43mst/largeEWG.txt
 *
 *  Compute a minimum spanning forest using Kruskal's algorithm.
 *
 *  %  java KruskalMST tinyEWG.txt
 *  0-7 0.16000
 *  2-3 0.17000
 *  1-7 0.19000
 *  0-2 0.26000
 *  5-7 0.28000
 *  4-5 0.35000
 *  6-2 0.40000
 *  1.81000
 *
 *  % java KruskalMST mediumEWG.txt
 *  168-231 0.00268
 *  151-208 0.00391
 *  7-157   0.00516
 *  122-205 0.00647
 *  8-152   0.00702
 *  156-219 0.00745
 *  28-198  0.00775
 *  38-126  0.00845
 *  10-123  0.00886
 *  ...
 *  10.46351
 *
 ******************************************************************************/

/**
 *  The {@code KruskalMST} class represents a data type for computing a
 *  <em>minimum spanning tree</em> in an edge-weighted graph.
 *  The edge weights can be positive, zero, or negative and need not
 *  be distinct. If the graph is not connected, it computes a <em>minimum
 *  spanning forest</em>, which is the union of minimum spanning trees
 *  in each connected component. The {@code weight()} method returns the
 *  weight of a minimum spanning tree and the {@code edges()} method
 *  returns its edges.
 *  <p>
 *  This implementation uses <em>Krusal's algorithm</em> and the
 *  union-find data type.
 *  The constructor takes time proportional to <em>E</em> log <em>E</em>
 *  and extra space (not including the graph) proportional to <em>V</em>,
 *  where <em>V</em> is the number of vertices and <em>E</em> is the number of edges.
 *  Afterwards, the {@code weight()} method takes constant time
 *  and the {@code edges()} method takes time proportional to <em>V</em>.
 *  <p>
 *  For additional documentation,
 *  see <a href="http://algs4.cs.princeton.edu/43mst">Section 4.3</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *  For alternate implementations, see {@link LazyPrimMST}, {@link PrimMST},
 *  and {@link BoruvkaMST}.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class KruskalMST {
    private static final double FLOATING_POINT_EPSILON = 1E-12;
```

```
private double weight;                          // weight of MST
private Queue<Edge> mst = new Queue<Edge>();  // edges in MST

/**
 * Compute a minimum spanning tree (or forest) of an edge-weighted graph.
 * @param G the edge-weighted graph
 */
public KruskalMST(EdgeWeightedGraph G) {
    // more efficient to build heap by passing array of edges
    MinPQ<Edge> pq = new MinPQ<Edge>();
    for (Edge e : G.edges()) {
        pq.insert(e);
    }

    // run greedy algorithm
    UF uf = new UF(G.V());
    while (!pq.isEmpty() && mst.size() < G.V() - 1) {
        Edge e = pq.delMin();
        int v = e.either();
        int w = e.other(v);
        if (!uf.connected(v, w)) { // v-w does not create a cycle
            uf.union(v, w);  // merge v and w components
            mst.enqueue(e);  // add edge e to mst
            weight += e.weight();
        }
    }

    // check optimality conditions
    assert check(G);
}

/**
 * Returns the edges in a minimum spanning tree (or forest).
 * @return the edges in a minimum spanning tree (or forest) as
 *    an iterable of edges
 */
public Iterable<Edge> edges() {
    return mst;
}

/**
 * Returns the sum of the edge weights in a minimum spanning tree (or forest).
 * @return the sum of the edge weights in a minimum spanning tree (or forest)
 */
public double weight() {
    return weight;
}

// check optimality conditions (takes time proportional to E V lg* V)
private boolean check(EdgeWeightedGraph G) {

    // check total weight
    double total = 0.0;
    for (Edge e : edges()) {
        total += e.weight();
    }
    if (Math.abs(total - weight()) > FLOATING_POINT_EPSILON) {
        System.err.printf("Weight of edges does not equal weight(): %f vs. %f\n", total, weight());
        return false;
    }

    // check that it is acyclic
    UF uf = new UF(G.V());
    for (Edge e : edges()) {
        int v = e.either(), w = e.other(v);
        if (uf.connected(v, w)) {
            System.err.println("Not a forest");
            return false;
        }
        uf.union(v, w);
    }

    // check that it is a spanning forest
    for (Edge e : G.edges()) {
```

```
            int v = e.either(), w = e.other(v);
            if (!uf.connected(v, w)) {
                System.err.println("Not a spanning forest");
                return false;
            }
        }

        // check that it is a minimal spanning forest (cut optimality conditions)
        for (Edge e : edges()) {

            // all edges in MST except e
            uf = new UF(G.V());
            for (Edge f : mst) {
                int x = f.either(), y = f.other(x);
                if (f != e) uf.union(x, y);
            }

            // check that e is min weight edge in crossing cut
            for (Edge f : G.edges()) {
                int x = f.either(), y = f.other(x);
                if (!uf.connected(x, y)) {
                    if (f.weight() < e.weight()) {
                        System.err.println("Edge " + f + " violates cut optimality conditions");
                        return false;
                    }
                }
            }

        }

        return true;
    }


    /**
     * Unit tests the {@code KruskalMST} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        In in = new In(args[0]);
        EdgeWeightedGraph G = new EdgeWeightedGraph(in);
        KruskalMST mst = new KruskalMST(G);
        for (Edge e : mst.edges()) {
            StdOut.println(e);
        }
        StdOut.printf("%.5f\n", mst.weight());
    }

}
```


Last updated: Tue Aug 30 10:09:18 EDT 2016.

PrimMST.java

Below is the syntax highlighted version of PrimMST.java from §4.3 Minimum Spanning Trees.

```
/******************************************************************************
 *  Compilation:  javac PrimMST.java
 *  Execution:    java PrimMST filename.txt
 *  Dependencies: EdgeWeightedGraph.java Edge.java Queue.java
 *                IndexMinPQ.java UF.java In.java StdOut.java
 *  Data files:   http://algs4.cs.princeton.edu/43mst/tinyEWG.txt
 *                http://algs4.cs.princeton.edu/43mst/mediumEWG.txt
 *                http://algs4.cs.princeton.edu/43mst/largeEWG.txt
 *
 *  Compute a minimum spanning forest using Prim's algorithm.
 *
 *  %  java PrimMST tinyEWG.txt
 *  1-7 0.19000
 *  0-2 0.26000
 *  2-3 0.17000
 *  4-5 0.35000
 *  5-7 0.28000
 *  6-2 0.40000
 *  0-7 0.16000
 *  1.81000
 *
 *  % java PrimMST mediumEWG.txt
 *  1-72   0.06506
 *  2-86   0.05980
 *  3-67   0.09725
 *  4-55   0.06425
 *  5-102  0.03834
 *  6-129  0.05363
 *  7-157  0.00516
 *  ...
 *  10.46351
 *
 *  % java PrimMST largeEWG.txt
 *  ...
 *  647.66307
 *
 ******************************************************************************/

/**
 *  The {@code PrimMST} class represents a data type for computing a
 *  <em>minimum spanning tree</em> in an edge-weighted graph.
 *  The edge weights can be positive, zero, or negative and need not
 *  be distinct. If the graph is not connected, it computes a <em>minimum
 *  spanning forest</em>, which is the union of minimum spanning trees
 *  in each connected component. The {@code weight()} method returns the
 *  weight of a minimum spanning tree and the {@code edges()} method
 *  returns its edges.
 *  <p>
 *  This implementation uses <em>Prim's algorithm</em> with an indexed
 *  binary heap.
 *  The constructor takes time proportional to <em>E</em> log <em>V</em>
 *  and extra space (not including the graph) proportional to <em>V</em>,
 *  where <em>V</em> is the number of vertices and <em>E</em> is the number of edges.
 *  Afterwards, the {@code weight()} method takes constant time
 *  and the {@code edges()} method takes time proportional to <em>V</em>.
 *  <p>
 *  For additional documentation,
 *  see <a href="http://algs4.cs.princeton.edu/43mst">Section 4.3</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *  For alternate implementations, see {@link LazyPrimMST}, {@link KruskalMST},
 *  and {@link BoruvkaMST}.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class PrimMST {
    private static final double FLOATING_POINT_EPSILON = 1E-12;

    private Edge[] edgeTo;        // edgeTo[v] = shortest edge from tree vertex to non-tree vertex
```

```
private double[] distTo;      // distTo[v] = weight of shortest such edge
private boolean[] marked;     // marked[v] = true if v on tree, false otherwise
private IndexMinPQ<Double> pq;

/**
 * Compute a minimum spanning tree (or forest) of an edge-weighted graph.
 * @param G the edge-weighted graph
 */
public PrimMST(EdgeWeightedGraph G) {
    edgeTo = new Edge[G.V()];
    distTo = new double[G.V()];
    marked = new boolean[G.V()];
    pq = new IndexMinPQ<Double>(G.V());
    for (int v = 0; v < G.V(); v++)
        distTo[v] = Double.POSITIVE_INFINITY;

    for (int v = 0; v < G.V(); v++)      // run from each vertex to find
        if (!marked[v]) prim(G, v);      // minimum spanning forest

    // check optimality conditions
    assert check(G);
}

// run Prim's algorithm in graph G, starting from vertex s
private void prim(EdgeWeightedGraph G, int s) {
    distTo[s] = 0.0;
    pq.insert(s, distTo[s]);
    while (!pq.isEmpty()) {
        int v = pq.delMin();
        scan(G, v);
    }
}

// scan vertex v
private void scan(EdgeWeightedGraph G, int v) {
    marked[v] = true;
    for (Edge e : G.adj(v)) {
        int w = e.other(v);
        if (marked[w]) continue;         // v-w is obsolete edge
        if (e.weight() < distTo[w]) {
            distTo[w] = e.weight();
            edgeTo[w] = e;
            if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
            else                pq.insert(w, distTo[w]);
        }
    }
}

/**
 * Returns the edges in a minimum spanning tree (or forest).
 * @return the edges in a minimum spanning tree (or forest) as
 *    an iterable of edges
 */
public Iterable<Edge> edges() {
    Queue<Edge> mst = new Queue<Edge>();
    for (int v = 0; v < edgeTo.length; v++) {
        Edge e = edgeTo[v];
        if (e != null) {
            mst.enqueue(e);
        }
    }
    return mst;
}

/**
 * Returns the sum of the edge weights in a minimum spanning tree (or forest).
 * @return the sum of the edge weights in a minimum spanning tree (or forest)
 */
public double weight() {
    double weight = 0.0;
    for (Edge e : edges())
        weight += e.weight();
    return weight;
}


// check optimality conditions (takes time proportional to E V lg* V)
private boolean check(EdgeWeightedGraph G) {
```

```
        // check weight
        double totalWeight = 0.0;
        for (Edge e : edges()) {
            totalWeight += e.weight();
        }
        if (Math.abs(totalWeight - weight()) > FLOATING_POINT_EPSILON) {
            System.err.printf("Weight of edges does not equal weight(): %f vs. %f\n", totalWeight, weight());
            return false;
        }

        // check that it is acyclic
        UF uf = new UF(G.V());
        for (Edge e : edges()) {
            int v = e.either(), w = e.other(v);
            if (uf.connected(v, w)) {
                System.err.println("Not a forest");
                return false;
            }
            uf.union(v, w);
        }

        // check that it is a spanning forest
        for (Edge e : G.edges()) {
            int v = e.either(), w = e.other(v);
            if (!uf.connected(v, w)) {
                System.err.println("Not a spanning forest");
                return false;
            }
        }

        // check that it is a minimal spanning forest (cut optimality conditions)
        for (Edge e : edges()) {

            // all edges in MST except e
            uf = new UF(G.V());
            for (Edge f : edges()) {
                int x = f.either(), y = f.other(x);
                if (f != e) uf.union(x, y);
            }

            // check that e is min weight edge in crossing cut
            for (Edge f : G.edges()) {
                int x = f.either(), y = f.other(x);
                if (!uf.connected(x, y)) {
                    if (f.weight() < e.weight()) {
                        System.err.println("Edge " + f + " violates cut optimality conditions");
                        return false;
                    }
                }
            }

        }

        return true;
    }

    /**
     * Unit tests the {@code PrimMST} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        In in = new In(args[0]);
        EdgeWeightedGraph G = new EdgeWeightedGraph(in);
        PrimMST mst = new PrimMST(G);
        for (Edge e : mst.edges()) {
            StdOut.println(e);
        }
        StdOut.printf("%.5f\n", mst.weight());
    }


}
```


Last updated: Tue Aug 30 10:09:18 EDT 2016.

BoruvkaMST.java

Below is the syntax highlighted version of BoruvkaMST.java from §4.3 Minimum Spanning Trees.

```
/******************************************************************************
 *  Compilation:  javac BoruvkaMST.java
 *  Execution:    java BoruvkaMST filename.txt
 *  Dependencies: EdgeWeightedGraph.java Edge.java Bag.java
 *                UF.java In.java StdOut.java
 *  Data files:   http://algs4.cs.princeton.edu/43mst/tinyEWG.txt
 *                http://algs4.cs.princeton.edu/43mst/mediumEWG.txt
 *                http://algs4.cs.princeton.edu/43mst/largeEWG.txt
 *
 *  Compute a minimum spanning forest using Boruvka's algorithm.
 *
 *  % java BoruvkaMST tinyEWG.txt
 *  0-2 0.26000
 *  6-2 0.40000
 *  5-7 0.28000
 *  4-5 0.35000
 *  2-3 0.17000
 *  1-7 0.19000
 *  0-7 0.16000
 *  1.81000
 *
 ******************************************************************************/

/**
 *  The {@code BoruvkaMST} class represents a data type for computing a
 *  <em>minimum spanning tree</em> in an edge-weighted graph.
 *  The edge weights can be positive, zero, or negative and need not
 *  be distinct. If the graph is not connected, it computes a <em>minimum
 *  spanning forest</em>, which is the union of minimum spanning trees
 *  in each connected component. The {@code weight()} method returns the
 *  weight of a minimum spanning tree and the {@code edges()} method
 *  returns its edges.
 *  <p>
 *  This implementation uses <em>Boruvka's algorithm</em> and the union-find
 *  data type.
 *  The constructor takes time proportional to <em>E</em> log <em>V</em>
 *  and extra space (not including the graph) proportional to <em>V</em>,
 *  where <em>V</em> is the number of vertices and <em>E</em> is the number of edges.
 *  Afterwards, the {@code weight()} method takes constant time
 *  and the {@code edges()} method takes time proportional to <em>V</em>.
 *  <p>
 *  For additional documentation,
 *  see <a href="http://algs4.cs.princeton.edu/43mst">Section 4.3</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *  For alternate implementations, see {@link LazyPrimMST}, {@link PrimMST},
 *  and {@link KruskalMST}.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class BoruvkaMST {
    private static final double FLOATING_POINT_EPSILON = 1E-12;

    private Bag<Edge> mst = new Bag<Edge>();    // edges in MST
    private double weight;                      // weight of MST

    /**
     * Compute a minimum spanning tree (or forest) of an edge-weighted graph.
     * @param G the edge-weighted graph
     */
    public BoruvkaMST(EdgeWeightedGraph G) {
        UF uf = new UF(G.V());

        // repeat at most log V times or until we have V-1 edges
        for (int t = 1; t < G.V() && mst.size() < G.V() - 1; t = t + t) {

            // foreach tree in forest, find closest edge
            // if edge weights are equal, ties are broken in favor of first edge in G.edges()
            Edge[] closest = new Edge[G.V()];
```

```
            for (Edge e : G.edges()) {
                int v = e.either(), w = e.other(v);
                int i = uf.find(v), j = uf.find(w);
                if (i == j) continue;   // same tree
                if (closest[i] == null || less(e, closest[i])) closest[i] = e;
                if (closest[j] == null || less(e, closest[j])) closest[j] = e;
            }

            // add newly discovered edges to MST
            for (int i = 0; i < G.V(); i++) {
                Edge e = closest[i];
                if (e != null) {
                    int v = e.either(), w = e.other(v);
                    // don't add the same edge twice
                    if (!uf.connected(v, w)) {
                        mst.add(e);
                        weight += e.weight();
                        uf.union(v, w);
                    }
                }
            }
        }

        // check optimality conditions
        assert check(G);
    }

    /**
     * Returns the edges in a minimum spanning tree (or forest).
     * @return the edges in a minimum spanning tree (or forest) as
     *    an iterable of edges
     */
    public Iterable<Edge> edges() {
        return mst;
    }


    /**
     * Returns the sum of the edge weights in a minimum spanning tree (or forest).
     * @return the sum of the edge weights in a minimum spanning tree (or forest)
     */
    public double weight() {
        return weight;
    }

    // is the weight of edge e strictly less than that of edge f?
    private static boolean less(Edge e, Edge f) {
        return e.weight() < f.weight();
    }

    // check optimality conditions (takes time proportional to E V lg* V)
    private boolean check(EdgeWeightedGraph G) {

        // check weight
        double totalWeight = 0.0;
        for (Edge e : edges()) {
            totalWeight += e.weight();
        }
        if (Math.abs(totalWeight - weight()) > FLOATING_POINT_EPSILON) {
            System.err.printf("Weight of edges does not equal weight(): %f vs. %f\n", totalWeight, weight());
            return false;
        }

        // check that it is acyclic
        UF uf = new UF(G.V());
        for (Edge e : edges()) {
            int v = e.either(), w = e.other(v);
            if (uf.connected(v, w)) {
                System.err.println("Not a forest");
                return false;
            }
            uf.union(v, w);
        }

        // check that it is a spanning forest
        for (Edge e : G.edges()) {
            int v = e.either(), w = e.other(v);
            if (!uf.connected(v, w)) {
                System.err.println("Not a spanning forest");
```

```
                return false;
            }
        }

        // check that it is a minimal spanning forest (cut optimality conditions)
        for (Edge e : edges()) {

            // all edges in MST except e
            uf = new UF(G.V());
            for (Edge f : mst) {
                int x = f.either(), y = f.other(x);
                if (f != e) uf.union(x, y);
            }

            // check that e is min weight edge in crossing cut
            for (Edge f : G.edges()) {
                int x = f.either(), y = f.other(x);
                if (!uf.connected(x, y)) {
                    if (f.weight() < e.weight()) {
                        System.err.println("Edge " + f + " violates cut optimality conditions");
                        return false;
                    }
                }
            }

        }

        return true;
    }

    /**
     * Unit tests the {@code BoruvkaMST} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        In in = new In(args[0]);
        EdgeWeightedGraph G = new EdgeWeightedGraph(in);
        BoruvkaMST mst = new BoruvkaMST(G);
        for (Edge e : mst.edges()) {
            StdOut.println(e);
        }
        StdOut.printf("%.5f\n", mst.weight());
    }

}
```


Last updated: Tue Aug 30 10:09:18 EDT 2016.

BreadthFirstPaths.java

Below is the syntax highlighted version of BreadthFirstPaths.java from §4.1 Undirected Graphs.

```
/******************************************************************************
 *  Compilation:  javac BreadthFirstPaths.java
 *  Execution:    java BreadthFirstPaths G s
 *  Dependencies: Graph.java Queue.java Stack.java StdOut.java
 *  Data files:   http://algs4.cs.princeton.edu/41graph/tinyCG.txt
 *                http://algs4.cs.princeton.edu/41graph/tinyG.txt
 *                http://algs4.cs.princeton.edu/41graph/mediumG.txt
 *                http://algs4.cs.princeton.edu/41graph/largeG.txt
 *
 *  Run breadth first search on an undirected graph.
 *  Runs in O(E + V) time.
 *
 *  %  java Graph tinyCG.txt
 *  6 8
 *  0: 2 1 5
 *  1: 0 2
 *  2: 0 1 3 4
 *  3: 5 4 2
 *  4: 3 2
 *  5: 3 0
 *
 *  %  java BreadthFirstPaths tinyCG.txt 0
 *  0 to 0 (0):  0
 *  0 to 1 (1):  0-1
 *  0 to 2 (1):  0-2
 *  0 to 3 (2):  0-2-3
 *  0 to 4 (2):  0-2-4
 *  0 to 5 (1):  0-5
 *
 *  %  java BreadthFirstPaths largeG.txt 0
 *  0 to 0 (0):  0
 *  0 to 1 (418):  0-932942-474885-82707-879889-971961-...
 *  0 to 2 (323):  0-460790-53370-594358-780059-287921-...
 *  0 to 3 (168):  0-713461-75230-953125-568284-350405-...
 *  0 to 4 (144):  0-460790-53370-310931-440226-380102-...
 *  0 to 5 (566):  0-932942-474885-82707-879889-971961-...
 *  0 to 6 (349):  0-932942-474885-82707-879889-971961-...
 *
 ******************************************************************************/


/**
 *  The {@code BreadthFirstPaths} class represents a data type for finding
 *  shortest paths (number of edges) from a source vertex <em>s</em>
 *  (or a set of source vertices)
 *  to every other vertex in an undirected graph.
 *  <p>
 *  This implementation uses breadth-first search.
 *  The constructor takes time proportional to <em>V</em> + <em>E</em>,
 *  where <em>V</em> is the number of vertices and <em>E</em> is the number of edges.
 *  It uses extra space (not including the graph) proportional to <em>V</em>.
 *  <p>
 *  For additional documentation, see <a href="http://algs4.cs.princeton.edu/41graph">Section 4.1</a>
 *  of <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class BreadthFirstPaths {
    private static final int INFINITY = Integer.MAX_VALUE;
    private boolean[] marked;  // marked[v] = is there an s-v path
    private int[] edgeTo;      // edgeTo[v] = previous edge on shortest s-v path
    private int[] distTo;      // distTo[v] = number of edges shortest s-v path

    /**
```

```
     * Computes the shortest path between the source vertex {@code s}
     * and every other vertex in the graph {@code G}.
     * @param G the graph
     * @param s the source vertex
     * @throws IllegalArgumentException unless {@code 0 <= s < V}
     */
    public BreadthFirstPaths(Graph G, int s) {
        marked = new boolean[G.V()];
        distTo = new int[G.V()];
        edgeTo = new int[G.V()];
        validateVertex(s);
        bfs(G, s);

        assert check(G, s);
    }

    /**
     * Computes the shortest path between any one of the source vertices in {@code sources}
     * and every other vertex in graph {@code G}.
     * @param G the graph
     * @param sources the source vertices
     * @throws IllegalArgumentException unless {@code 0 <= s < V} for each vertex
     *         {@code s} in {@code sources}
     */
    public BreadthFirstPaths(Graph G, Iterable<Integer> sources) {
        marked = new boolean[G.V()];
        distTo = new int[G.V()];
        edgeTo = new int[G.V()];
        for (int v = 0; v < G.V(); v++)
            distTo[v] = INFINITY;
        validateVertices(sources);
        bfs(G, sources);
    }


    // breadth-first search from a single source
    private void bfs(Graph G, int s) {
        Queue<Integer> q = new Queue<Integer>();
        for (int v = 0; v < G.V(); v++)
            distTo[v] = INFINITY;
        distTo[s] = 0;
        marked[s] = true;
        q.enqueue(s);

        while (!q.isEmpty()) {
            int v = q.dequeue();
            for (int w : G.adj(v)) {
                if (!marked[w]) {
                    edgeTo[w] = v;
                    distTo[w] = distTo[v] + 1;
                    marked[w] = true;
                    q.enqueue(w);
                }
            }
        }
    }

    // breadth-first search from multiple sources
    private void bfs(Graph G, Iterable<Integer> sources) {
        Queue<Integer> q = new Queue<Integer>();
        for (int s : sources) {
            marked[s] = true;
            distTo[s] = 0;
            q.enqueue(s);
        }
        while (!q.isEmpty()) {
            int v = q.dequeue();
            for (int w : G.adj(v)) {
                if (!marked[w]) {
                    edgeTo[w] = v;
                    distTo[w] = distTo[v] + 1;
                    marked[w] = true;
                    q.enqueue(w);
                }
            }
```

```
    }
}

/**
 * Is there a path between the source vertex {@code s} (or sources) and vertex {@code v}?
 * @param v the vertex
 * @return {@code true} if there is a path, and {@code false} otherwise
 * @throws IllegalArgumentException unless {@code 0 <= v < V}
 */
public boolean hasPathTo(int v) {
    validateVertex(v);
    return marked[v];
}

/**
 * Returns the number of edges in a shortest path between the source vertex {@code s}
 * (or sources) and vertex {@code v}?
 * @param v the vertex
 * @return the number of edges in a shortest path
 * @throws IllegalArgumentException unless {@code 0 <= v < V}
 */
public int distTo(int v) {
    validateVertex(v);
    return distTo[v];
}

/**
 * Returns a shortest path between the source vertex {@code s} (or sources)
 * and {@code v}, or {@code null} if no such path.
 * @param  v the vertex
 * @return the sequence of vertices on a shortest path, as an Iterable
 * @throws IllegalArgumentException unless {@code 0 <= v < V}
 */
public Iterable<Integer> pathTo(int v) {
    validateVertex(v);
    if (!hasPathTo(v)) return null;
    Stack<Integer> path = new Stack<Integer>();
    int x;
    for (x = v; distTo[x] != 0; x = edgeTo[x])
        path.push(x);
    path.push(x);
    return path;
}


// check optimality conditions for single source
private boolean check(Graph G, int s) {

    // check that the distance of s = 0
    if (distTo[s] != 0) {
        StdOut.println("distance of source " + s + " to itself = " + distTo[s]);
        return false;
    }

    // check that for each edge v-w dist[w] <= dist[v] + 1
    // provided v is reachable from s
    for (int v = 0; v < G.V(); v++) {
        for (int w : G.adj(v)) {
            if (hasPathTo(v) != hasPathTo(w)) {
                StdOut.println("edge " + v + "-" + w);
                StdOut.println("hasPathTo(" + v + ") = " + hasPathTo(v));
                StdOut.println("hasPathTo(" + w + ") = " + hasPathTo(w));
                return false;
            }
            if (hasPathTo(v) && (distTo[w] > distTo[v] + 1)) {
                StdOut.println("edge " + v + "-" + w);
                StdOut.println("distTo[" + v + "] = " + distTo[v]);
                StdOut.println("distTo[" + w + "] = " + distTo[w]);
                return false;
            }
        }
    }

    // check that v = edgeTo[w] satisfies distTo[w] = distTo[v] + 1
    // provided v is reachable from s
```

```
        for (int w = 0; w < G.V(); w++) {
            if (!hasPathTo(w) || w == s) continue;
            int v = edgeTo[w];
            if (distTo[w] != distTo[v] + 1) {
                StdOut.println("shortest path edge " + v + "-" + w);
                StdOut.println("distTo[" + v + "] = " + distTo[v]);
                StdOut.println("distTo[" + w + "] = " + distTo[w]);
                return false;
            }
        }

        return true;
    }

    // throw an IllegalArgumentException unless {@code 0 <= v < V}
    private void validateVertex(int v) {
        int V = marked.length;
        if (v < 0 || v >= V)
            throw new IllegalArgumentException("vertex " + v + " is not between 0 and " + (V-1));
    }

    // throw an IllegalArgumentException unless {@code 0 <= v < V}
    private void validateVertices(Iterable<Integer> vertices) {
        if (vertices == null) {
            throw new IllegalArgumentException("argument is null");
        }
        int V = marked.length;
        for (int v : vertices) {
            if (v < 0 || v >= V) {
                throw new IllegalArgumentException("vertex " + v + " is not between 0 and " + (V-1));
            }
        }
    }

    /**
     * Unit tests the {@code BreadthFirstPaths} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        In in = new In(args[0]);
        Graph G = new Graph(in);
        // StdOut.println(G);

        int s = Integer.parseInt(args[1]);
        BreadthFirstPaths bfs = new BreadthFirstPaths(G, s);

        for (int v = 0; v < G.V(); v++) {
            if (bfs.hasPathTo(v)) {
                StdOut.printf("%d to %d (%d):  ", s, v, bfs.distTo(v));
                for (int x : bfs.pathTo(v)) {
                    if (x == s) StdOut.print(x);
                    else        StdOut.print("-" + x);
                }
                StdOut.println();
            }

            else {
                StdOut.printf("%d to %d (-):  not connected\n", s, v);
            }

        }
    }


}
```

DijkstraSP.java

Below is the syntax highlighted version of DijkstraSP.java from §4.4 Shortest Paths.

```
/******************************************************************************
 *  Compilation:  javac DijkstraSP.java
 *  Execution:    java DijkstraSP input.txt s
 *  Dependencies: EdgeWeightedDigraph.java IndexMinPQ.java Stack.java DirectedEdge.java
 *  Data files:   http://algs4.cs.princeton.edu/44sp/tinyEWD.txt
 *                http://algs4.cs.princeton.edu/44sp/mediumEWD.txt
 *                http://algs4.cs.princeton.edu/44sp/largeEWD.txt
 *
 *  Dijkstra's algorithm. Computes the shortest path tree.
 *  Assumes all weights are nonnegative.
 *
 *  % java DijkstraSP tinyEWD.txt 0
 *  0 to 0 (0.00)
 *  0 to 1 (1.05)  0->4  0.38   4->5  0.35   5->1  0.32
 *  0 to 2 (0.26)  0->2  0.26
 *  0 to 3 (0.99)  0->2  0.26   2->7  0.34   7->3  0.39
 *  0 to 4 (0.38)  0->4  0.38
 *  0 to 5 (0.73)  0->4  0.38   4->5  0.35
 *  0 to 6 (1.51)  0->2  0.26   2->7  0.34   7->3  0.39   3->6  0.52
 *  0 to 7 (0.60)  0->2  0.26   2->7  0.34
 *
 *  % java DijkstraSP mediumEWD.txt 0
 *  0 to 0 (0.00)
 *  0 to 1 (0.71)  0->44  0.06   44->93  0.07   ...  107->1  0.07
 *  0 to 2 (0.65)  0->44  0.06   44->231  0.10  ...  42->2  0.11
 *  0 to 3 (0.46)  0->97  0.08   97->248  0.09  ...  45->3  0.12
 *  0 to 4 (0.42)  0->44  0.06   44->93  0.07   ...  77->4  0.11
 *  ...
 *
 ******************************************************************************/


/**
 *  The {@code DijkstraSP} class represents a data type for solving the
 *  single-source shortest paths problem in edge-weighted digraphs
 *  where the edge weights are nonnegative.
 *  <p>
 *  This implementation uses Dijkstra's algorithm with a binary heap.
 *  The constructor takes time proportional to <em>E</em> log <em>V</em>,
 *  where <em>V</em> is the number of vertices and <em>E</em> is the number of edges.
 *  Afterwards, the {@code distTo()} and {@code hasPathTo()} methods take
 *  constant time and the {@code pathTo()} method takes time proportional to the
 *  number of edges in the shortest path returned.
 *  <p>
 *  For additional documentation,
 *  see <a href="http://algs4.cs.princeton.edu/44sp">Section 4.4</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class DijkstraSP {
    private double[] distTo;          // distTo[v] = distance  of shortest s->v path
    private DirectedEdge[] edgeTo;    // edgeTo[v] = last edge on shortest s->v path
    private IndexMinPQ<Double> pq;    // priority queue of vertices

    /**
     * Computes a shortest-paths tree from the source vertex {@code s} to every other
     * vertex in the edge-weighted digraph {@code G}.
     *
     * @param  G the edge-weighted digraph
```

```
     * @param  s the source vertex
     * @throws IllegalArgumentException if an edge weight is negative
     * @throws IllegalArgumentException unless {@code 0 <= s < V}
     */
    public DijkstraSP(EdgeWeightedDigraph G, int s) {
        for (DirectedEdge e : G.edges()) {
            if (e.weight() < 0)
                throw new IllegalArgumentException("edge " + e + " has negative weight");
        }

        distTo = new double[G.V()];
        edgeTo = new DirectedEdge[G.V()];

        validateVertex(s);

        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;

        // relax vertices in order of distance from s
        pq = new IndexMinPQ<Double>(G.V());
        pq.insert(s, distTo[s]);
        while (!pq.isEmpty()) {
            int v = pq.delMin();
            for (DirectedEdge e : G.adj(v))
                relax(e);
        }

        // check optimality conditions
        assert check(G, s);
    }

    // relax edge e and update pq if changed
    private void relax(DirectedEdge e) {
        int v = e.from(), w = e.to();
        if (distTo[w] > distTo[v] + e.weight()) {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
            if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
            else                pq.insert(w, distTo[w]);
        }
    }

    /**
     * Returns the length of a shortest path from the source vertex {@code s} to vertex {@code v}.
     * @param  v the destination vertex
     * @return the length of a shortest path from the source vertex {@code s} to vertex {@code v};
     *         {@code Double.POSITIVE_INFINITY} if no such path
     * @throws IllegalArgumentException unless {@code 0 <= v < V}
     */
    public double distTo(int v) {
        validateVertex(v);
        return distTo[v];
    }

    /**
     * Returns true if there is a path from the source vertex {@code s} to vertex {@code v}.
     *
     * @param  v the destination vertex
     * @return {@code true} if there is a path from the source vertex
     *         {@code s} to vertex {@code v}; {@code false} otherwise
     * @throws IllegalArgumentException unless {@code 0 <= v < V}
     */
    public boolean hasPathTo(int v) {
        validateVertex(v);
        return distTo[v] < Double.POSITIVE_INFINITY;
    }

    /**
     * Returns a shortest path from the source vertex {@code s} to vertex {@code v}.
     *
```

```
     * @param  v the destination vertex
     * @return a shortest path from the source vertex {@code s} to vertex {@code v}
     *         as an iterable of edges, and {@code null} if no such path
     * @throws IllegalArgumentException unless {@code 0 <= v < V}
     */
    public Iterable<DirectedEdge> pathTo(int v) {
        validateVertex(v);
        if (!hasPathTo(v)) return null;
        Stack<DirectedEdge> path = new Stack<DirectedEdge>();
        for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()]) {
            path.push(e);
        }
        return path;
    }


    // check optimality conditions:
    // (i) for all edges e:            distTo[e.to()] <= distTo[e.from()] + e.weight()
    // (ii) for all edge e on the SPT: distTo[e.to()] == distTo[e.from()] + e.weight()
    private boolean check(EdgeWeightedDigraph G, int s) {

        // check that edge weights are nonnegative
        for (DirectedEdge e : G.edges()) {
            if (e.weight() < 0) {
                System.err.println("negative edge weight detected");
                return false;
            }
        }

        // check that distTo[v] and edgeTo[v] are consistent
        if (distTo[s] != 0.0 || edgeTo[s] != null) {
            System.err.println("distTo[s] and edgeTo[s] inconsistent");
            return false;
        }
        for (int v = 0; v < G.V(); v++) {
            if (v == s) continue;
            if (edgeTo[v] == null && distTo[v] != Double.POSITIVE_INFINITY) {
                System.err.println("distTo[] and edgeTo[] inconsistent");
                return false;
            }
        }

        // check that all edges e = v->w satisfy distTo[w] <= distTo[v] + e.weight()
        for (int v = 0; v < G.V(); v++) {
            for (DirectedEdge e : G.adj(v)) {
                int w = e.to();
                if (distTo[v] + e.weight() < distTo[w]) {
                    System.err.println("edge " + e + " not relaxed");
                    return false;
                }
            }
        }

        // check that all edges e = v->w on SPT satisfy distTo[w] == distTo[v] + e.weight()
        for (int w = 0; w < G.V(); w++) {
            if (edgeTo[w] == null) continue;
            DirectedEdge e = edgeTo[w];
            int v = e.from();
            if (w != e.to()) return false;
            if (distTo[v] + e.weight() != distTo[w]) {
                System.err.println("edge " + e + " on shortest path not tight");
                return false;
            }
        }
        return true;
    }

    // throw an IllegalArgumentException unless {@code 0 <= v < V}
    private void validateVertex(int v) {
        int V = distTo.length;
        if (v < 0 || v >= V)
```

```
            throw new IllegalArgumentException("vertex " + v + " is not between 0 and " + (V-1));
    }

    /**
     * Unit tests the {@code DijkstraSP} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        In in = new In(args[0]);
        EdgeWeightedDigraph G = new EdgeWeightedDigraph(in);
        int s = Integer.parseInt(args[1]);

        // compute shortest paths
        DijkstraSP sp = new DijkstraSP(G, s);


        // print shortest path
        for (int t = 0; t < G.V(); t++) {
            if (sp.hasPathTo(t)) {
                StdOut.printf("%d to %d (%.2f)  ", s, t, sp.distTo(t));
                for (DirectedEdge e : sp.pathTo(t)) {
                    StdOut.print(e + "   ");
                }
                StdOut.println();
            }
            else {
                StdOut.printf("%d to %d         no path\n", s, t);
            }
        }
    }

}
```


Last updated: Sat Nov 19 09:09:51 EST 2016.

BellmanFordSP.java

Below is the syntax highlighted version of BellmanFordSP.java from §4.4 Shortest Paths.

```
/******************************************************************************
 *  Compilation:  javac BellmanFordSP.java
 *  Execution:    java BellmanFordSP filename.txt s
 *  Dependencies: EdgeWeightedDigraph.java DirectedEdge.java Queue.java
 *                EdgeWeightedDirectedCycle.java
 *  Data files:   http://algs4.cs.princeton.edu/44sp/tinyEWDn.txt
 *                http://algs4.cs.princeton.edu/44sp/mediumEWDnc.txt
 *
 *  Bellman-Ford shortest path algorithm. Computes the shortest path tree in
 *  edge-weighted digraph G from vertex s, or finds a negative cost cycle
 *  reachable from s.
 *
 *  % java BellmanFordSP tinyEWDn.txt 0
 *  0 to 0 ( 0.00)
 *  0 to 1 ( 0.93)  0->2  0.26   2->7  0.34   7->3  0.39   3->6  0.52   6->4 -1.25   4->5  0.35   5->1  0.32
 *  0 to 2 ( 0.26)  0->2  0.26
 *  0 to 3 ( 0.99)  0->2  0.26   2->7  0.34   7->3  0.39
 *  0 to 4 ( 0.26)  0->2  0.26   2->7  0.34   7->3  0.39   3->6  0.52   6->4 -1.25
 *  0 to 5 ( 0.61)  0->2  0.26   2->7  0.34   7->3  0.39   3->6  0.52   6->4 -1.25   4->5  0.35
 *  0 to 6 ( 1.51)  0->2  0.26   2->7  0.34   7->3  0.39   3->6  0.52
 *  0 to 7 ( 0.60)  0->2  0.26   2->7  0.34
 *
 *  % java BellmanFordSP tinyEWDnc.txt 0
 *  4->5  0.35
 *  5->4 -0.66
 *
 *
 ******************************************************************************/

/**
 *  The {@code BellmanFordSP} class represents a data type for solving the
 *  single-source shortest paths problem in edge-weighted digraphs with
 *  no negative cycles.
 *  The edge weights can be positive, negative, or zero.
 *  This class finds either a shortest path from the source vertex <em>s</em>
 *  to every other vertex or a negative cycle reachable from the source vertex.
 *  <p>
 *  This implementation uses the Bellman-Ford-Moore algorithm.
 *  The constructor takes time proportional to <em>V</em> (<em>V</em> + <em>E</em>)
 *  in the worst case, where <em>V</em> is the number of vertices and <em>E</em>
 *  is the number of edges.
 *  Afterwards, the {@code distTo()}, {@code hasPathTo()}, and {@code hasNegativeCycle()}
 *  methods take constant time; the {@code pathTo()} and {@code negativeCycle()}
 *  method takes time proportional to the number of edges returned.
 *  <p>
 *  For additional documentation,
 *  see <a href="http://algs4.cs.princeton.edu/44sp">Section 4.4</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class BellmanFordSP {
    private double[] distTo;               // distTo[v] = distance  of shortest s->v path
    private DirectedEdge[] edgeTo;         // edgeTo[v] = last edge on shortest s->v path
    private boolean[] onQueue;             // onQueue[v] = is v currently on the queue?
    private Queue<Integer> queue;          // queue of vertices to relax
    private int cost;                      // number of calls to relax()
    private Iterable<DirectedEdge> cycle;  // negative cycle (or null if no such cycle)

    /**
     * Computes a shortest paths tree from {@code s} to every other vertex in
     * the edge-weighted digraph {@code G}.
     * @param G the acyclic digraph
     * @param s the source vertex
     * @throws IllegalArgumentException unless {@code 0 <= s < V}
     */
    public BellmanFordSP(EdgeWeightedDigraph G, int s) {
```

```
        distTo  = new double[G.V()];
        edgeTo  = new DirectedEdge[G.V()];
        onQueue = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;

        // Bellman-Ford algorithm
        queue = new Queue<Integer>();
        queue.enqueue(s);
        onQueue[s] = true;
        while (!queue.isEmpty() && !hasNegativeCycle()) {
            int v = queue.dequeue();
            onQueue[v] = false;
            relax(G, v);
        }

        assert check(G, s);
    }

    // relax vertex v and put other endpoints on queue if changed
    private void relax(EdgeWeightedDigraph G, int v) {
        for (DirectedEdge e : G.adj(v)) {
            int w = e.to();
            if (distTo[w] > distTo[v] + e.weight()) {
                distTo[w] = distTo[v] + e.weight();
                edgeTo[w] = e;
                if (!onQueue[w]) {
                    queue.enqueue(w);
                    onQueue[w] = true;
                }
            }
            if (cost++ % G.V() == 0) {
                findNegativeCycle();
                if (hasNegativeCycle()) return;  // found a negative cycle
            }
        }
    }

    /**
     * Is there a negative cycle reachable from the source vertex {@code s}?
     * @return {@code true} if there is a negative cycle reachable from the
     *    source vertex {@code s}, and {@code false} otherwise
     */
    public boolean hasNegativeCycle() {
        return cycle != null;
    }

    /**
     * Returns a negative cycle reachable from the source vertex {@code s}, or {@code null}
     * if there is no such cycle.
     * @return a negative cycle reachable from the soruce vertex {@code s}
     *    as an iterable of edges, and {@code null} if there is no such cycle
     */
    public Iterable<DirectedEdge> negativeCycle() {
        return cycle;
    }

    // by finding a cycle in predecessor graph
    private void findNegativeCycle() {
        int V = edgeTo.length;
        EdgeWeightedDigraph spt = new EdgeWeightedDigraph(V);
        for (int v = 0; v < V; v++)
            if (edgeTo[v] != null)
                spt.addEdge(edgeTo[v]);

        EdgeWeightedDirectedCycle finder = new EdgeWeightedDirectedCycle(spt);
        cycle = finder.cycle();
    }

    /**
     * Returns the length of a shortest path from the source vertex {@code s} to vertex {@code v}.
     * @param  v the destination vertex
     * @return the length of a shortest path from the source vertex {@code s} to vertex {@code v};
     *         {@code Double.POSITIVE_INFINITY} if no such path
     * @throws UnsupportedOperationException if there is a negative cost cycle reachable
     *         from the source vertex {@code s}
     * @throws IllegalArgumentException unless {@code 0 <= v < V}
```

```
     */
    public double distTo(int v) {
        validateVertex(v);
        if (hasNegativeCycle())
            throw new UnsupportedOperationException("Negative cost cycle exists");
        return distTo[v];
    }

    /**
     * Is there a path from the source {@code s} to vertex {@code v}?
     * @param  v the destination vertex
     * @return {@code true} if there is a path from the source vertex
     *         {@code s} to vertex {@code v}, and {@code false} otherwise
     * @throws IllegalArgumentException unless {@code 0 <= v < V}
     */
    public boolean hasPathTo(int v) {
        validateVertex(v);
        return distTo[v] < Double.POSITIVE_INFINITY;
    }

    /**
     * Returns a shortest path from the source {@code s} to vertex {@code v}.
     * @param  v the destination vertex
     * @return a shortest path from the source {@code s} to vertex {@code v}
     *         as an iterable of edges, and {@code null} if no such path
     * @throws UnsupportedOperationException if there is a negative cost cycle reachable
     *         from the source vertex {@code s}
     * @throws IllegalArgumentException unless {@code 0 <= v < V}
     */
    public Iterable<DirectedEdge> pathTo(int v) {
        validateVertex(v);
        if (hasNegativeCycle())
            throw new UnsupportedOperationException("Negative cost cycle exists");
        if (!hasPathTo(v)) return null;
        Stack<DirectedEdge> path = new Stack<DirectedEdge>();
        for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()]) {
            path.push(e);
        }
        return path;
    }

    // check optimality conditions: either
    // (i) there exists a negative cycle reacheable from s
    //     or
    // (ii)  for all edges e = v->w:            distTo[w] <= distTo[v] + e.weight()
    // (ii') for all edges e = v->w on the SPT: distTo[w] == distTo[v] + e.weight()
    private boolean check(EdgeWeightedDigraph G, int s) {

        // has a negative cycle
        if (hasNegativeCycle()) {
            double weight = 0.0;
            for (DirectedEdge e : negativeCycle()) {
                weight += e.weight();
            }
            if (weight >= 0.0) {
                System.err.println("error: weight of negative cycle = " + weight);
                return false;
            }
        }

        // no negative cycle reachable from source
        else {

            // check that distTo[v] and edgeTo[v] are consistent
            if (distTo[s] != 0.0 || edgeTo[s] != null) {
                System.err.println("distanceTo[s] and edgeTo[s] inconsistent");
                return false;
            }
            for (int v = 0; v < G.V(); v++) {
                if (v == s) continue;
                if (edgeTo[v] == null && distTo[v] != Double.POSITIVE_INFINITY) {
                    System.err.println("distTo[] and edgeTo[] inconsistent");
                    return false;
                }
            }

            // check that all edges e = v->w satisfy distTo[w] <= distTo[v] + e.weight()
            for (int v = 0; v < G.V(); v++) {
```

```
                for (DirectedEdge e : G.adj(v)) {
                    int w = e.to();
                    if (distTo[v] + e.weight() < distTo[w]) {
                        System.err.println("edge " + e + " not relaxed");
                        return false;
                    }
                }
            }

            // check that all edges e = v->w on SPT satisfy distTo[w] == distTo[v] + e.weight()
            for (int w = 0; w < G.V(); w++) {
                if (edgeTo[w] == null) continue;
                DirectedEdge e = edgeTo[w];
                int v = e.from();
                if (w != e.to()) return false;
                if (distTo[v] + e.weight() != distTo[w]) {
                    System.err.println("edge " + e + " on shortest path not tight");
                    return false;
                }
            }
        }

        StdOut.println("Satisfies optimality conditions");
        StdOut.println();
        return true;
    }

    // throw an IllegalArgumentException unless {@code 0 <= v < V}
    private void validateVertex(int v) {
        int V = distTo.length;
        if (v < 0 || v >= V)
            throw new IllegalArgumentException("vertex " + v + " is not between 0 and " + (V-1));
    }

    /**
     * Unit tests the {@code BellmanFordSP} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        In in = new In(args[0]);
        int s = Integer.parseInt(args[1]);
        EdgeWeightedDigraph G = new EdgeWeightedDigraph(in);

        BellmanFordSP sp = new BellmanFordSP(G, s);

        // print negative cycle
        if (sp.hasNegativeCycle()) {
            for (DirectedEdge e : sp.negativeCycle())
                StdOut.println(e);
        }

        // print shortest paths
        else {
            for (int v = 0; v < G.V(); v++) {
                if (sp.hasPathTo(v)) {
                    StdOut.printf("%d to %d (%5.2f)  ", s, v, sp.distTo(v));
                    for (DirectedEdge e : sp.pathTo(v)) {
                        StdOut.print(e + "   ");
                    }
                    StdOut.println();
                }
                else {
                    StdOut.printf("%d to %d           no path\n", s, v);
                }
            }
        }

    }

}
```

FloydWarshall.java

Below is the syntax highlighted version of FloydWarshall.java from §4.4 Shortest Paths.

```
/******************************************************************************
 *  Compilation:  javac FloydWarshall.java
 *  Execution:    java FloydWarshall V E
 *  Dependencies: AdjMatrixEdgeWeightedDigraph.java
 *
 *  Floyd-Warshall all-pairs shortest path algorithm.
 *
 *  % java FloydWarshall 100 500
 *
 *  Should check for negative cycles during triple loop; otherwise
 *  intermediate numbers can get exponentially large.
 *  Reference: "The Floyd-Warshall algorithm on graphs with negative cycles"
 *  by Stefan Hougardy
 *
 ******************************************************************************/


/**
 *  The {@code FloydWarshall} class represents a data type for solving the
 *  all-pairs shortest paths problem in edge-weighted digraphs with
 *  no negative cycles.
 *  The edge weights can be positive, negative, or zero.
 *  This class finds either a shortest path between every pair of vertices
 *  or a negative cycle.
 *  <p>
 *  This implementation uses the Floyd-Warshall algorithm.
 *  The constructor takes time proportional to <em>V</em><sup>3</sup> in the
 *  worst case, where <em>V</em> is the number of vertices.
 *  Afterwards, the {@code dist()}, {@code hasPath()}, and {@code hasNegativeCycle()}
 *  methods take constant time; the {@code path()} and {@code negativeCycle()}
 *  method takes time proportional to the number of edges returned.
 *  <p>
 *  For additional documentation,
 *  see <a href="http://algs4.cs.princeton.edu/44sp">Section 4.4</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class FloydWarshall {
    private boolean hasNegativeCycle;  // is there a negative cycle?
    private double[][] distTo;         // distTo[v][w] = length of shortest v->w path
    private DirectedEdge[][] edgeTo;   // edgeTo[v][w] = last edge on shortest v->w path

    /**
     * Computes a shortest paths tree from each vertex to to every other vertex in
     * the edge-weighted digraph {@code G}. If no such shortest path exists for
     * some pair of vertices, it computes a negative cycle.
     * @param G the edge-weighted digraph
     */
    public FloydWarshall(AdjMatrixEdgeWeightedDigraph G) {
        int V = G.V();
        distTo = new double[V][V];
        edgeTo = new DirectedEdge[V][V];

        // initialize distances to infinity
        for (int v = 0; v < V; v++) {
            for (int w = 0; w < V; w++) {
                distTo[v][w] = Double.POSITIVE_INFINITY;
            }
```

```
    }

    // initialize distances using edge-weighted digraph's
    for (int v = 0; v < G.V(); v++) {
        for (DirectedEdge e : G.adj(v)) {
            distTo[e.from()][e.to()] = e.weight();
            edgeTo[e.from()][e.to()] = e;
        }
        // in case of self-loops
        if (distTo[v][v] >= 0.0) {
            distTo[v][v] = 0.0;
            edgeTo[v][v] = null;
        }
    }

    // Floyd-Warshall updates
    for (int i = 0; i < V; i++) {
        // compute shortest paths using only 0, 1, ..., i as intermediate vertices
        for (int v = 0; v < V; v++) {
            if (edgeTo[v][i] == null) continue;  // optimization
            for (int w = 0; w < V; w++) {
                if (distTo[v][w] > distTo[v][i] + distTo[i][w]) {
                    distTo[v][w] = distTo[v][i] + distTo[i][w];
                    edgeTo[v][w] = edgeTo[i][w];
                }
            }
            // check for negative cycle
            if (distTo[v][v] < 0.0) {
                hasNegativeCycle = true;
                return;
            }
        }
    }
    assert check(G);
}

/**
 * Is there a negative cycle?
 * @return {@code true} if there is a negative cycle, and {@code false} otherwise
 */
public boolean hasNegativeCycle() {
    return hasNegativeCycle;
}

/**
 * Returns a negative cycle, or {@code null} if there is no such cycle.
 * @return a negative cycle as an iterable of edges,
 * or {@code null} if there is no such cycle
 */
public Iterable<DirectedEdge> negativeCycle() {
    for (int v = 0; v < distTo.length; v++) {
        // negative cycle in v's predecessor graph
        if (distTo[v][v] < 0.0) {
            int V = edgeTo.length;
            EdgeWeightedDigraph spt = new EdgeWeightedDigraph(V);
            for (int w = 0; w < V; w++)
                if (edgeTo[v][w] != null)
                    spt.addEdge(edgeTo[v][w]);
            EdgeWeightedDirectedCycle finder = new EdgeWeightedDirectedCycle(spt);
            assert finder.hasCycle();
            return finder.cycle();
        }
    }
    return null;
}

/**
 * Is there a path from the vertex {@code s} to vertex {@code t}?
 * @param  s the source vertex
 * @param  t the destination vertex
 * @return {@code true} if there is a path from vertex {@code s}
```

```
     *         to vertex {@code t}, and {@code false} otherwise
     * @throws IllegalArgumentException unless {@code 0 <= s < V}
     * @throws IllegalArgumentException unless {@code 0 <= t < V}
     */
    public boolean hasPath(int s, int t) {
        validateVertex(s);
        validateVertex(t);
        return distTo[s][t] < Double.POSITIVE_INFINITY;
    }

    /**
     * Returns the length of a shortest path from vertex {@code s} to vertex {@code t}.
     * @param  s the source vertex
     * @param  t the destination vertex
     * @return the length of a shortest path from vertex {@code s} to vertex {@code t};
     *         {@code Double.POSITIVE_INFINITY} if no such path
     * @throws UnsupportedOperationException if there is a negative cost cycle
     * @throws IllegalArgumentException unless {@code 0 <= v < V}
     */
    public double dist(int s, int t) {
        validateVertex(s);
        validateVertex(t);
        if (hasNegativeCycle())
            throw new UnsupportedOperationException("Negative cost cycle exists");
        return distTo[s][t];
    }

    /**
     * Returns a shortest path from vertex {@code s} to vertex {@code t}.
     * @param  s the source vertex
     * @param  t the destination vertex
     * @return a shortest path from vertex {@code s} to vertex {@code t}
     *         as an iterable of edges, and {@code null} if no such path
     * @throws UnsupportedOperationException if there is a negative cost cycle
     * @throws IllegalArgumentException unless {@code 0 <= v < V}
     */
    public Iterable<DirectedEdge> path(int s, int t) {
        validateVertex(s);
        validateVertex(t);
        if (hasNegativeCycle())
            throw new UnsupportedOperationException("Negative cost cycle exists");
        if (!hasPath(s, t)) return null;
        Stack<DirectedEdge> path = new Stack<DirectedEdge>();
        for (DirectedEdge e = edgeTo[s][t]; e != null; e = edgeTo[s][e.from()]) {
            path.push(e);
        }
        return path;
    }

    // check optimality conditions
    private boolean check(AdjMatrixEdgeWeightedDigraph G) {

        // no negative cycle
        if (!hasNegativeCycle()) {
            for (int v = 0; v < G.V(); v++) {
                for (DirectedEdge e : G.adj(v)) {
                    int w = e.to();
                    for (int i = 0; i < G.V(); i++) {
                        if (distTo[i][w] > distTo[i][v] + e.weight()) {
                            System.err.println("edge " + e + " is eligible");
                            return false;
                        }
                    }
                }
            }
        }
        return true;
    }

    // throw an IllegalArgumentException unless {@code 0 <= v < V}
    private void validateVertex(int v) {
```

```
    int V = distTo.length;
    if (v < 0 || v >= V)
        throw new IllegalArgumentException("vertex " + v + " is not between 0 and " + (V-1));
}

/**
 * Unit tests the {@code FloydWarshall} data type.
 *
 * @param args the command-line arguments
 */
public static void main(String[] args) {

    // random graph with V vertices and E edges, parallel edges allowed
    int V = Integer.parseInt(args[0]);
    int E = Integer.parseInt(args[1]);
    AdjMatrixEdgeWeightedDigraph G = new AdjMatrixEdgeWeightedDigraph(V);
    for (int i = 0; i < E; i++) {
        int v = StdRandom.uniform(V);
        int w = StdRandom.uniform(V);
        double weight = Math.round(100 * (StdRandom.uniform() - 0.15)) / 100.0;
        if (v == w) G.addEdge(new DirectedEdge(v, w, Math.abs(weight)));
        else G.addEdge(new DirectedEdge(v, w, weight));
    }

    StdOut.println(G);

    // run Floyd-Warshall algorithm
    FloydWarshall spt = new FloydWarshall(G);

    // print all-pairs shortest path distances
    StdOut.printf("  ");
    for (int v = 0; v < G.V(); v++) {
        StdOut.printf("%6d ", v);
    }
    StdOut.println();
    for (int v = 0; v < G.V(); v++) {
        StdOut.printf("%3d: ", v);
        for (int w = 0; w < G.V(); w++) {
            if (spt.hasPath(v, w)) StdOut.printf("%6.2f ", spt.dist(v, w));
            else StdOut.printf("  Inf ");
        }
        StdOut.println();
    }

    // print negative cycle
    if (spt.hasNegativeCycle()) {
        StdOut.println("Negative cost cycle:");
        for (DirectedEdge e : spt.negativeCycle())
            StdOut.println(e);
        StdOut.println();
    }

    // print all-pairs shortest paths
    else {
        for (int v = 0; v < G.V(); v++) {
            for (int w = 0; w < G.V(); w++) {
                if (spt.hasPath(v, w)) {
                    StdOut.printf("%d to %d (%5.2f)  ", v, w, spt.dist(v, w));
                    for (DirectedEdge e : spt.path(v, w))
                        StdOut.print(e + "  ");
                    StdOut.println();
                }
                else {
                    StdOut.printf("%d to %d no path\n", v, w);
                }
            }
        }
    }

}
```

```
}
```


Last updated: Sat Nov 19 09:18:43 EST 2016.

FordFulkerson.java

Below is the syntax highlighted version of FordFulkerson.java from §6.4 Maxflow.

```
/******************************************************************************
 *  Compilation:  javac FordFulkerson.java
 *  Execution:    java FordFulkerson V E
 *  Dependencies: FlowNetwork.java FlowEdge.java Queue.java
 *  Data files:   http://algs4.cs.princeton.edu/65maxflow/tinyFN.txt
 *
 *  Ford-Fulkerson algorithm for computing a max flow and
 *  a min cut using shortest augmenting path rule.
 *
 ******************************************************************************/

/**
 *  The {@code FordFulkerson} class represents a data type for computing a
 *  <em>maximum st-flow</em> and <em>minimum st-cut</em> in a flow
 *  network.
 *  <p>
 *  This implementation uses the <em>Ford-Fulkerson</em> algorithm with
 *  the <em>shortest augmenting path</em> heuristic.
 *  The constructor takes time proportional to <em>E V</em> (<em>E</em> + <em>V</em>)
 *  in the worst case and extra space (not including the network)
 *  proportional to <em>V</em>, where <em>V</em> is the number of vertices
 *  and <em>E</em> is the number of edges. In practice, the algorithm will
 *  run much faster.
 *  Afterwards, the {@code inCut()} and {@code value()} methods take
 *  constant time.
 *  <p>
 *  If the capacities and initial flow values are all integers, then this
 *  implementation guarantees to compute an integer-valued maximum flow.
 *  If the capacities and floating-point numbers, then floating-point
 *  roundoff error can accumulate.
 *  <p>
 *  For additional documentation,
 *  see <a href="http://algs4.cs.princeton.edu/64maxflow">Section 6.4</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class FordFulkerson {
    private static final double FLOATING_POINT_EPSILON = 1E-11;

    private final int V;          // number of vertices
    private boolean[] marked;     // marked[v] = true iff s->v path in residual graph
    private FlowEdge[] edgeTo;    // edgeTo[v] = last edge on shortest residual s->v path
    private double value;         // current value of max flow

    /**
     * Compute a maximum flow and minimum cut in the network {@code G}
     * from vertex {@code s} to vertex {@code t}.
     *
     * @param  G the flow network
     * @param  s the source vertex
     * @param  t the sink vertex
     * @throws IllegalArgumentException unless {@code 0 <= s < V}
     * @throws IllegalArgumentException unless {@code 0 <= t < V}
     * @throws IllegalArgumentException if {@code s == t}
     * @throws IllegalArgumentException if initial flow is infeasible
     */
    public FordFulkerson(FlowNetwork G, int s, int t) {
        V = G.V();
        validate(s);
        validate(t);
        if (s == t)               throw new IllegalArgumentException("Source equals sink");
        if (!isFeasible(G, s, t)) throw new IllegalArgumentException("Initial flow is infeasible");

        // while there exists an augmenting path, use it
        value = excess(G, t);
        while (hasAugmentingPath(G, s, t)) {
```

```
            // compute bottleneck capacity
            double bottle = Double.POSITIVE_INFINITY;
            for (int v = t; v != s; v = edgeTo[v].other(v)) {
                bottle = Math.min(bottle, edgeTo[v].residualCapacityTo(v));
            }

            // augment flow
            for (int v = t; v != s; v = edgeTo[v].other(v)) {
                edgeTo[v].addResidualFlowTo(v, bottle);
            }

            value += bottle;
        }

        // check optimality conditions
        assert check(G, s, t);
    }

    /**
     * Returns the value of the maximum flow.
     *
     * @return the value of the maximum flow
     */
    public double value()  {
        return value;
    }

    /**
     * Returns true if the specified vertex is on the {@code s} side of the mincut.
     *
     * @param  v vertex
     * @return {@code true} if vertex {@code v} is on the {@code s} side of the micut;
     *         {@code false} otherwise
     * @throws IllegalArgumentException unless {@code 0 <= v < V}
     */
    public boolean inCut(int v)  {
        validate(v);
        return marked[v];
    }

    // throw an IllegalArgumentException if v is outside prescibed range
    private void validate(int v)  {
        if (v < 0 || v >= V)
            throw new IllegalArgumentException("vertex " + v + " is not between 0 and " + (V-1));
    }


    // is there an augmenting path?
    // if so, upon termination edgeTo[] will contain a parent-link representation of such a path
    // this implementation finds a shortest augmenting path (fewest number of edges),
    // which performs well both in theory and in practice
    private boolean hasAugmentingPath(FlowNetwork G, int s, int t) {
        edgeTo = new FlowEdge[G.V()];
        marked = new boolean[G.V()];

        // breadth-first search
        Queue<Integer> queue = new Queue<Integer>();
        queue.enqueue(s);
        marked[s] = true;
        while (!queue.isEmpty() && !marked[t]) {
            int v = queue.dequeue();

            for (FlowEdge e : G.adj(v)) {
                int w = e.other(v);

                // if residual capacity from v to w
                if (e.residualCapacityTo(w) > 0) {
                    if (!marked[w]) {
                        edgeTo[w] = e;
                        marked[w] = true;
                        queue.enqueue(w);
                    }
                }
            }
        }

        // is there an augmenting path?
        return marked[t];
    }
```

```
// return excess flow at vertex v
private double excess(FlowNetwork G, int v) {
    double excess = 0.0;
    for (FlowEdge e : G.adj(v)) {
        if (v == e.from()) excess -= e.flow();
        else               excess += e.flow();
    }
    return excess;
}

// return excess flow at vertex v
private boolean isFeasible(FlowNetwork G, int s, int t) {

    // check that capacity constraints are satisfied
    for (int v = 0; v < G.V(); v++) {
        for (FlowEdge e : G.adj(v)) {
            if (e.flow() < -FLOATING_POINT_EPSILON || e.flow() > e.capacity() + FLOATING_POINT_EPSILON) {
                System.err.println("Edge does not satisfy capacity constraints: " + e);
                return false;
            }
        }
    }

    // check that net flow into a vertex equals zero, except at source and sink
    if (Math.abs(value + excess(G, s)) > FLOATING_POINT_EPSILON) {
        System.err.println("Excess at source = " + excess(G, s));
        System.err.println("Max flow         = " + value);
        return false;
    }
    if (Math.abs(value - excess(G, t)) > FLOATING_POINT_EPSILON) {
        System.err.println("Excess at sink   = " + excess(G, t));
        System.err.println("Max flow         = " + value);
        return false;
    }
    for (int v = 0; v < G.V(); v++) {
        if (v == s || v == t) continue;
        else if (Math.abs(excess(G, v)) > FLOATING_POINT_EPSILON) {
            System.err.println("Net flow out of " + v + " doesn't equal zero");
            return false;
        }
    }
    return true;
}



// check optimality conditions
private boolean check(FlowNetwork G, int s, int t) {

    // check that flow is feasible
    if (!isFeasible(G, s, t)) {
        System.err.println("Flow is infeasible");
        return false;
    }

    // check that s is on the source side of min cut and that t is not on source side
    if (!inCut(s)) {
        System.err.println("source " + s + " is not on source side of min cut");
        return false;
    }
    if (inCut(t)) {
        System.err.println("sink " + t + " is on source side of min cut");
        return false;
    }

    // check that value of min cut = value of max flow
    double mincutValue = 0.0;
    for (int v = 0; v < G.V(); v++) {
        for (FlowEdge e : G.adj(v)) {
            if ((v == e.from()) && inCut(e.from()) && !inCut(e.to()))
                mincutValue += e.capacity();
        }
    }

    if (Math.abs(mincutValue - value) > FLOATING_POINT_EPSILON) {
        System.err.println("Max flow value = " + value + ", min cut value = " + mincutValue);
```

```
            return false;
        }

        return true;
    }


    /**
     * Unit tests the {@code FordFulkerson} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {

        // create flow network with V vertices and E edges
        int V = Integer.parseInt(args[0]);
        int E = Integer.parseInt(args[1]);
        int s = 0, t = V-1;
        FlowNetwork G = new FlowNetwork(V, E);
        StdOut.println(G);

        // compute maximum flow and minimum cut
        FordFulkerson maxflow = new FordFulkerson(G, s, t);
        StdOut.println("Max flow from " + s + " to " + t);
        for (int v = 0; v < G.V(); v++) {
            for (FlowEdge e : G.adj(v)) {
                if ((v == e.from()) && e.flow() > 0)
                    StdOut.println("   " + e);
            }
        }

        // print min-cut
        StdOut.print("Min cut: ");
        for (int v = 0; v < G.V(); v++) {
            if (maxflow.inCut(v)) StdOut.print(v + " ");
        }
        StdOut.println();

        StdOut.println("Max flow value = " +  maxflow.value());
    }

}
```

HopcroftKarp.java

Below is the syntax highlighted version of HopcroftKarp.java from §6.5 Reductions.

```
/******************************************************************************
 *  Compilation:  javac HopcroftKarp.java
 *  Execution:    java HopcroftKarp V1 V2 E
 *  Dependencies: FordFulkerson.java FlowNetwork.java FlowEdge.java
 *                BipartiteX.java
 *
 *  Find a maximum cardinality matching (and minimum cardinality vertex cover)
 *  in a bipartite graph using Hopcroft-Karp algorithm.
 *
 ******************************************************************************/

import java.util.Iterator;

/**
 *  The {@code HopcroftKarp} class represents a data type for computing a
 *  <em>maximum (cardinality) matching</em> and a
 *  <em>minimum (cardinality) vertex cover</em> in a bipartite graph.
 *  A <em>bipartite graph</em> in a graph whose vertices can be partitioned
 *  into two disjoint sets such that every edge has one endpoint in either set.
 *  A <em>matching</em> in a graph is a subset of its edges with no common
 *  vertices. A <em>maximum matching</em> is a matching with the maximum number
 *  of edges.
 *  A <em>perfect matching</em> is a matching which matches all vertices in the graph.
 *  A <em>vertex cover</em> in a graph is a subset of its vertices such that
 *  every edge is incident to at least one vertex. A <em>minimum vertex cover</em>
 *  is a vertex cover with the minimum number of vertices.
 *  By Konig's theorem, in any biparite
 *  graph, the maximum number of edges in matching equals the minimum number
 *  of vertices in a vertex cover.
 *  The maximum matching problem in <em>nonbipartite</em> graphs is
 *  also important, but all known algorithms for this more general problem
 *  are substantially more complicated.
 *  <p>
 *  This implementation uses the <em>Hopcroft-Karp algorithm</em>.
 *  The order of growth of the running time in the worst case is
 *  (<em>E</em> + <em>V</em>) sqrt(<em>V</em>),
 *  where <em>E</em> is the number of edges and <em>V</em> is the number
 *  of vertices in the graph. It uses extra space (not including the graph)
 *  proportional to <em>V</em>.
 *  <p>
 *  See also {@link BipartiteMatching}, which solves the problem in
 *  O(<em>E V</em>) time using the <em>alternating path algorithm</em>
 *  and <a href = "http://algs4.cs.princeton.edu/65reductions/BipartiteMatchingToMaxflow.java.html">BipartiteMatchingToMaxflow</a>,
 *  which solves the problem in O(<em>E V</em>) time via a reduction
 *  to the maxflow problem.
 *  <p>
 *  For additional documentation, see
 *  <a href="http://algs4.cs.princeton.edu/65reductions">Section 6.5</a>
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class HopcroftKarp {
    private static final int UNMATCHED = -1;

    private final int V;                 // number of vertices in the graph
    private BipartiteX bipartition;      // the bipartition
    private int cardinality;             // cardinality of current matching
    private int[] mate;                  // mate[v] =  w if v-w is an edge in current matching
                                         //         = -1 if v is not in current matching
    private boolean[] inMinVertexCover;  // inMinVertexCover[v] = true iff v is in min vertex cover
    private boolean[] marked;            // marked[v] = true iff v is reachable via alternating path
    private int[] distTo;                // distTo[v] = number of edges on shortest path to v

    /**
     * Determines a maximum matching (and a minimum vertex cover)
     * in a bipartite graph.
     *
     * @param  G the bipartite graph
     * @throws IllegalArgumentException if {@code G} is not bipartite
     */
    public HopcroftKarp(Graph G) {
        bipartition = new BipartiteX(G);
        if (!bipartition.isBipartite()) {
            throw new IllegalArgumentException("graph is not bipartite");
        }

        // initialize empty matching
        this.V = G.V();
        mate = new int[V];
        for (int v = 0; v < V; v++)
            mate[v] = UNMATCHED;

        // the call to hasAugmentingPath() provides enough info to reconstruct level graph
```

```
    while (hasAugmentingPath(G)) {

        // to be able to iterate over each adjacency list, keeping track of which
        // vertex in each adjacency list needs to be explored next
        Iterator<Integer>[] adj = (Iterator<Integer>[]) new Iterator[G.V()];
        for (int v = 0; v < G.V(); v++)
            adj[v] = G.adj(v).iterator();

        // for each unmatched vertex s on one side of bipartition
        for (int s = 0; s < V; s++) {
            if (isMatched(s) || !bipartition.color(s)) continue;   // or use distTo[s] == 0

            // find augmenting path from s using nonrecursive DFS
            Stack<Integer> path = new Stack<Integer>();
            path.push(s);
            while (!path.isEmpty()) {
                int v = path.peek();

                // retreat, no more edges in level graph leaving v
                if (!adj[v].hasNext())
                    path.pop();

                // advance
                else {
                    // process edge v-w only if it is an edge in level graph
                    int w = adj[v].next();
                    if (!isLevelGraphEdge(v, w)) continue;

                    // add w to augmenting path
                    path.push(w);

                    // augmenting path found: update the matching
                    if (!isMatched(w)) {
                        // StdOut.println("augmenting path: " + toString(path));

                        while (!path.isEmpty()) {
                            int x = path.pop();
                            int y = path.pop();
                            mate[x] = y;
                            mate[y] = x;
                        }
                        cardinality++;
                    }
                }
            }
        }
    }

    // also find a min vertex cover
    inMinVertexCover = new boolean[V];
    for (int v = 0; v < V; v++) {
        if (bipartition.color(v) && !marked[v]) inMinVertexCover[v] = true;
        if (!bipartition.color(v) && marked[v]) inMinVertexCover[v] = true;
    }

    assert certifySolution(G);
}

// string representation of augmenting path (chop off last vertex)
private static String toString(Iterable<Integer> path) {
    StringBuilder sb = new StringBuilder();
    for (int v : path)
        sb.append(v + "-");
    String s = sb.toString();
    s = s.substring(0, s.lastIndexOf('-'));
    return s;
}

// is the edge v-w in the level graph?
private boolean isLevelGraphEdge(int v, int w) {
    return (distTo[w] == distTo[v] + 1) && isResidualGraphEdge(v, w);
}

// is the edge v-w a forward edge not in the matching or a reverse edge in the matching?
private boolean isResidualGraphEdge(int v, int w) {
    if ((mate[v] != w) &&  bipartition.color(v)) return true;
    if ((mate[v] == w) && !bipartition.color(v)) return true;
    return false;
}

/*
 * is there an augmenting path?
 *   - if so, upon termination adj[] contains the level graph;
 *   - if not, upon termination marked[] specifies those vertices reachable via an alternating
 *     path from one side of the bipartition
 *
 * an alternating path is a path whose edges belong alternately to the matching and not
 * to the matching
 *
 * an augmenting path is an alternating path that starts and ends at unmatched vertices
 */
private boolean hasAugmentingPath(Graph G) {

    // shortest path distances
    marked = new boolean[V];
```

```
        distTo = new int[V];
        for (int v = 0; v < V; v++)
            distTo[v] = Integer.MAX_VALUE;

        // breadth-first search (starting from all unmatched vertices on one side of bipartition)
        Queue<Integer> queue = new Queue<Integer>();
        for (int v = 0; v < V; v++) {
            if (bipartition.color(v) && !isMatched(v)) {
                queue.enqueue(v);
                marked[v] = true;
                distTo[v] = 0;
            }
        }

        // run BFS until an augmenting path is found
        // (and keep going until all vertices at that distance are explored)
        boolean hasAugmentingPath = false;
        while (!queue.isEmpty()) {
            int v = queue.dequeue();
            for (int w : G.adj(v)) {

                // forward edge not in matching or backwards edge in matching
                if (isResidualGraphEdge(v, w)) {
                    if (!marked[w]) {
                        distTo[w] = distTo[v] + 1;
                        marked[w] = true;
                        if (!isMatched(w))
                            hasAugmentingPath = true;

                        // stop enqueuing vertices once an alternating path has been discovered
                        // (no vertex on same side will be marked if its shortest path distance longer)
                        if (!hasAugmentingPath) queue.enqueue(w);
                    }
                }
            }
        }

        return hasAugmentingPath;
    }

    /**
     * Returns the vertex to which the specified vertex is matched in
     * the maximum matching computed by the algorithm.
     *
     * @param  v the vertex
     * @return the vertex to which vertex {@code v} is matched in the
     *         maximum matching; {@code -1} if the vertex is not matched
     * @throws IllegalArgumentException unless {@code 0 <= v < V}
     *
     */
    public int mate(int v) {
        validate(v);
        return mate[v];
    }

    /**
     * Returns true if the specified vertex is matched in the maximum matching
     * computed by the algorithm.
     *
     * @param  v the vertex
     * @return {@code true} if vertex {@code v} is matched in maximum matching;
     *         {@code false} otherwise
     * @throws IllegalArgumentException unless {@code 0 <= v < V}
     *
     */
    public boolean isMatched(int v) {
        validate(v);
        return mate[v] != UNMATCHED;
    }

    /**
     * Returns the number of edges in any maximum matching.
     *
     * @return the number of edges in any maximum matching
     */
    public int size() {
        return cardinality;
    }

    /**
     * Returns true if the graph contains a perfect matching.
     * That is, the number of edges in a maximum matching is equal to one half
     * of the number of vertices in the graph (so that every vertex is matched).
     *
     * @return {@code true} if the graph contains a perfect matching;
     *         {@code false} otherwise
     */
    public boolean isPerfect() {
        return cardinality * 2 == V;
    }

    /**
     * Returns true if the specified vertex is in the minimum vertex cover
     * computed by the algorithm.
     *
```

```
     * @param  v the vertex
     * @return {@code true} if vertex {@code v} is in the minimum vertex cover;
     *         {@code false} otherwise
     * @throws IllegalArgumentException unless {@code 0 <= v < V}
     */
    public boolean inMinVertexCover(int v) {
        validate(v);
        return inMinVertexCover[v];
    }

    // throw an exception if vertex is invalid
    private void validate(int v) {
        if (v < 0 || v >= V)
            throw new IllegalArgumentException("vertex " + v + " is not between 0 and " + (V-1));
    }

    /***************************************************************************
     *
     *  The code below is solely for testing correctness of the data type.
     *
     ***************************************************************************/

    // check that mate[] and inVertexCover[] define a max matching and min vertex cover, respectively
    private boolean certifySolution(Graph G) {

        // check that mate(v) = w iff mate(w) = v
        for (int v = 0; v < V; v++) {
            if (mate(v) == -1) continue;
            if (mate(mate(v)) != v) return false;
        }

        // check that size() is consistent with mate()
        int matchedVertices = 0;
        for (int v = 0; v < V; v++) {
            if (mate(v) != -1) matchedVertices++;
        }
        if (2*size() != matchedVertices) return false;

        // check that size() is consistent with minVertexCover()
        int sizeOfMinVertexCover = 0;
        for (int v = 0; v < V; v++)
            if (inMinVertexCover(v)) sizeOfMinVertexCover++;
        if (size() != sizeOfMinVertexCover) return false;

        // check that mate() uses each vertex at most once
        boolean[] isMatched = new boolean[V];
        for (int v = 0; v < V; v++) {
            int w = mate[v];
            if (w == -1) continue;
            if (v == w) return false;
            if (v >= w) continue;
            if (isMatched[v] || isMatched[w]) return false;
            isMatched[v] = true;
            isMatched[w] = true;
        }

        // check that mate() uses only edges that appear in the graph
        for (int v = 0; v < V; v++) {
            if (mate(v) == -1) continue;
            boolean isEdge = false;
            for (int w : G.adj(v)) {
                if (mate(v) == w) isEdge = true;
            }
            if (!isEdge) return false;
        }

        // check that inMinVertexCover() is a vertex cover
        for (int v = 0; v < V; v++)
            for (int w : G.adj(v))
                if (!inMinVertexCover(v) && !inMinVertexCover(w)) return false;

        return true;
    }

    /**
     * Unit tests the {@code HopcroftKarp} data type.
     * Takes three command-line arguments {@code V1}, {@code V2}, and {@code E};
     * creates a random bipartite graph with {@code V1} + {@code V2} vertices
     * and {@code E} edges; computes a maximum matching and minimum vertex cover;
     * and prints the results.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {

        int V1 = Integer.parseInt(args[0]);
        int V2 = Integer.parseInt(args[1]);
        int E  = Integer.parseInt(args[2]);
        Graph G = GraphGenerator.bipartite(V1, V2, E);
        if (G.V() < 1000) StdOut.println(G);

        HopcroftKarp matching = new HopcroftKarp(G);

        // print maximum matching
        StdOut.printf("Number of edges in max matching        = %d\n", matching.size());
```

```
        StdOut.printf("Number of vertices in min vertex cover = %d\n", matching.size());
        StdOut.printf("Graph has a perfect matching           = %b\n", matching.isPerfect());
        StdOut.println();

        if (G.V() >= 1000) return;

        StdOut.print("Max matching: ");
        for (int v = 0; v < G.V(); v++) {
            int w = matching.mate(v);
            if (matching.isMatched(v) && v < w)  // print each edge only once
                StdOut.print(v + "-" + w + " ");
        }
        StdOut.println();

        // print minimum vertex cover
        StdOut.print("Min vertex cover: ");
        for (int v = 0; v < G.V(); v++)
            if (matching.inMinVertexCover(v))
                StdOut.print(v + " ");
        StdOut.println();
    }

}
```

AssignmentProblem.java

Below is the syntax highlighted version of AssignmentProblem.java from §6.5 Reductions.

```
/******************************************************************************
 *  Compilation:  javac AssignmentProblem.java
 *  Execution:    java AssignmentProblem n
 *  Dependencies: DijkstraSP.java DirectedEdge.java
 *
 *  Solve an n-by-n assignment problem in n^3 log n time using the
 *  successive shortest path algorithm.
 *
 ******************************************************************************/

/**
 *  The {@code AssignmentProblem} class represents a data type for computing
 *  an optimal solution to an <em>n</em>-by-<em>n</em> <em>assignment problem</em>.
 *  The assignment problem is to find a minimum weight matching in an
 *  edge-weighted complete bipartite graph.
 *  <p>
 *  The data type supplies methods for determining the optimal solution
 *  and the corresponding dual solution.
 *  <p>
 *  This implementation uses the <em>successive shortest paths algorithm</em>.
 *  The order of growth of the running time in the worst case is
 *  O(<em>n</em>^3 log <em>n</em>) to solve an <em>n</em>-by-<em>n</em>
 *  instance.
 *  <p>
 *  For additional documentation, see
 *  <a href="http://algs4.cs.princeton.edu/65reductions">Section 6.5</a>
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class AssignmentProblem {
    private static final double FLOATING_POINT_EPSILON = 1E-14;
    private static final int UNMATCHED = -1;

    private int n;              // number of rows and columns
    private double[][] weight;  // the n-by-n cost matrix
    private double minWeight;   // minimum value of any weight
    private double[] px;        // px[i] = dual variable for row i
    private double[] py;        // py[j] = dual variable for col j
    private int[] xy;           // xy[i] = j means i-j is a match
    private int[] yx;           // yx[j] = i means i-j is a match

    /**
     * Determines an optimal solution to the assignment problem.
     *
     * @param  weight the <em>n</em>-by-<em>n</em> matrix of weights
     * @throws IllegalArgumentException unless all weights are nonnegative
     * @throws IllegalArgumentException if {@code weight} is {@code null}
     */
    public AssignmentProblem(double[][] weight) {
        if (weight == null) throw new IllegalArgumentException("constructor argument is null");

        n = weight.length;
        this.weight = new double[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (Double.isNaN(weight[i][j]))
                    throw new IllegalArgumentException("weight " + i + "-" + j  + " is NaN");
                if (weight[i][j] < minWeight) minWeight = weight[i][j];
                this.weight[i][j] = weight[i][j];
            }
        }

        // dual variables
        px = new double[n];
        py = new double[n];

        // initial matching is empty
        xy = new int[n];
        yx = new int[n];
        for (int i = 0; i < n; i++)
```

```
            xy[i] = UNMATCHED;
        for (int j = 0; j < n; j++)
            yx[j] = UNMATCHED;

        // add n edges to matching
        for (int k = 0; k < n; k++) {
            assert isDualFeasible();
            assert isComplementarySlack();
            augment();
        }
        assert certifySolution();
    }

    // find shortest augmenting path and upate
    private void augment() {

        // build residual graph
        EdgeWeightedDigraph G = new EdgeWeightedDigraph(2*n+2);
        int s = 2*n, t = 2*n+1;
        for (int i = 0; i < n; i++) {
            if (xy[i] == UNMATCHED)
                G.addEdge(new DirectedEdge(s, i, 0.0));
        }
        for (int j = 0; j < n; j++) {
            if (yx[j] == UNMATCHED)
                G.addEdge(new DirectedEdge(n+j, t, py[j]));
        }
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (xy[i] == j) G.addEdge(new DirectedEdge(n+j, i, 0.0));
                else            G.addEdge(new DirectedEdge(i, n+j, reducedCost(i, j)));
            }
        }

        // compute shortest path from s to every other vertex
        DijkstraSP spt = new DijkstraSP(G, s);

        // augment along alternating path
        for (DirectedEdge e : spt.pathTo(t)) {
            int i = e.from(), j = e.to() - n;
            if (i < n) {
                xy[i] = j;
                yx[j] = i;
            }
        }

        // update dual variables
        for (int i = 0; i < n; i++)
            px[i] += spt.distTo(i);
        for (int j = 0; j < n; j++)
            py[j] += spt.distTo(n+j);
    }

    // reduced cost of i-j
    // (subtracting off minWeight reweights all weights to be non-negative)
    private double reducedCost(int i, int j) {
        double reducedCost = (weight[i][j] - minWeight) + px[i] - py[j];

        // to avoid issues with floating-point precision
        double magnitude = Math.abs(weight[i][j]) + Math.abs(px[i]) + Math.abs(py[j]);
        if (Math.abs(reducedCost) <= FLOATING_POINT_EPSILON * magnitude) return 0.0;

        assert reducedCost >= 0.0;
        return reducedCost;
    }

    /**
     * Returns the dual optimal value for the specified row.
     *
     * @param  i the row index
     * @return the dual optimal value for row {@code i}
     * @throws IllegalArgumentException unless {@code 0 <= i < n}
     *
     */
    // dual variable for row i
    public double dualRow(int i) {
        validate(i);
        return px[i];
    }

    /**
     * Returns the dual optimal value for the specified column.
```

```
     *
     * @param  j the column index
     * @return the dual optimal value for column {@code j}
     * @throws IllegalArgumentException unless {@code 0 <= j < n}
     *
     */
    public double dualCol(int j) {
        validate(j);
        return py[j];
    }

    /**
     * Returns the column associated with the specified row in the optimal solution.
     *
     * @param  i the row index
     * @return the column matched to row {@code i} in the optimal solution
     * @throws IllegalArgumentException unless {@code 0 <= i < n}
     *
     */
    public int sol(int i) {
        validate(i);
        return xy[i];
    }

    /**
     * Returns the total weight of the optimal solution
     *
     * @return the total weight of the optimal solution
     *
     */
    public double weight() {
        double total = 0.0;
        for (int i = 0; i < n; i++) {
            if (xy[i] != UNMATCHED)
                total += weight[i][xy[i]];
        }
        return total;
    }

    private void validate(int i) {
        if (i < 0 || i >= n) throw new IllegalArgumentException("index is not between 0 and " + (n-1) + ": " + i);
    }


    /***************************************************************************
     *
     *  The code below is solely for testing correctness of the data type.
     *
     ***************************************************************************/

    // check that dual variables are feasible
    private boolean isDualFeasible() {
        // check that all edges have >= 0 reduced cost
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (reducedCost(i, j) < 0) {
                    StdOut.println("Dual variables are not feasible");
                    return false;
                }
            }
        }
        return true;
    }

    // check that primal and dual variables are complementary slack
    private boolean isComplementarySlack() {

        // check that all matched edges have 0-reduced cost
        for (int i = 0; i < n; i++) {
            if ((xy[i] != UNMATCHED) && (reducedCost(i, xy[i]) != 0)) {
                StdOut.println("Primal and dual variables are not complementary slack");
                return false;
            }
        }
        return true;
    }

    // check that primal variables are a perfect matching
    private boolean isPerfectMatching() {

        // check that xy[] is a perfect matching
        boolean[] perm = new boolean[n];
```

```
        for (int i = 0; i < n; i++) {
            if (perm[xy[i]]) {
                StdOut.println("Not a perfect matching");
                return false;
            }
            perm[xy[i]] = true;
        }

        // check that xy[] and yx[] are inverses
        for (int j = 0; j < n; j++) {
            if (xy[yx[j]] != j) {
                StdOut.println("xy[] and yx[] are not inverses");
                return false;
            }
        }
        for (int i = 0; i < n; i++) {
            if (yx[xy[i]] != i) {
                StdOut.println("xy[] and yx[] are not inverses");
                return false;
            }
        }

        return true;
    }

    // check optimality conditions
    private boolean certifySolution() {
        return isPerfectMatching() && isDualFeasible() && isComplementarySlack();
    }

    /**
     * Unit tests the {@code AssignmentProblem} data type.
     * Takes a command-line argument n; creates a random n-by-n matrix;
     * solves the n-by-n assignment problem; and prints the optimal
     * solution.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {

        // create random n-by-n matrix
        int n = Integer.parseInt(args[0]);
        double[][] weight = new double[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                weight[i][j] = StdRandom.uniform(900) + 100;  // 3 digits
            }
        }

        // solve assignment problem
        AssignmentProblem assignment = new AssignmentProblem(weight);
        StdOut.printf("weight = %.0f\n", assignment.weight());
        StdOut.println();

        // print n-by-n matrix and optimal solution
        if (n >= 20) return;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (j == assignment.sol(i))
                    StdOut.printf("*%.0f ", weight[i][j]);
                else
                    StdOut.printf(" %.0f ", weight[i][j]);
            }
            StdOut.println();
        }
    }

}
```