```java
/**
package examples;

import java.io.File;

/**
 * @author lar02
 *
 */
public class BoyerMoore {

  char[] t; // text
  char[] p; // pattern
  int[] last = new int[256]; // last occurence table of a char

  int n; // length of text
  int m; // length of pattern

  int matchCounter;

  public BoyerMoore(char[] t, char[] p) {
    this.t = t;
    this.p = p;
    n = t.length;
    m = p.length;
    Arrays.fill(last, -1);
    for (int k = 0; k < m; k++)
    {
      last[p[k]] = k;
    }
  }

  public int match() {
    int i = m - 1; // pos in text
    int j = m - 1; // pos in pattern

    matchCounter = 0;

    while (i < n)
    {
      matchCounter++;
      if(t[i] == p[j])
      {
        if(j == 0)
          return i;
        i--;
        j--;
      }
      else
      {
        i = i + m - Math.min(j, last[t[i]] + 1);
        j = m - 1;
      }
    }

    return -1;
  }
```

```
63
64  public void setText(File file) throws IOException {
65    FileInputStream in = null;
66    int c = -1;
67
68    try
69    {
70      in = new FileInputStream(file);
71      int len = in.available();
72      t = new char[len + 1];
73      int i = 0;
74      while ((c = in.read()) != -1 && i < len)
75      {
76        char cb = (char) c;
77        // if (cb<=0 || cb>255) System.out.println("i: "+i+", cb: "+cb);
78        t[i++] = cb;
79      }
80      t[i++] = 0;// stopchar
81      n = t.length - 1;
82    }
83    finally
84    {
85      if(in != null)
86      {
87        in.close();
88      }
89    }
90  }
91
92  public int getMatchCount(){
93    return matchCounter;
94  }
95
96  /**
97   * @param args
98   */
99  public static void main(String[] args) {
100   String t = "a pattern rithm matching algorithm a pattern matching algorithm";
101   String p = "Dorfschulmeister";
102
103   BoyerMoore bm = new BoyerMoore(t.toCharArray(), p.toCharArray());
104
105   try
106   {
107     bm.setText(new File("resources/Goethe.txt"));
108   }
109   catch (IOException e)
110   {
111     // TODO Auto-generated catch block
112     e.printStackTrace();
113   }
114
115   long s = System.currentTimeMillis();
116
117   for (int i = 0; i < 1000; i++)
118     bm.match();
119
```

```
120     long e = System.currentTimeMillis();
121     System.out.println(e - s + " micro sec");
122     System.out.println(" found at " + bm.match());
123     System.out.println("Vergleiche: " +bm.getMatchCount());
124     System.out.println(t.indexOf(p));
125   }
126
127 }
128
```

```java
package examples;

import java.util.ArrayList;

public class GraphExamples<V, E> {

  static final private Object NUMBER = new Object();
  static final private Object VISITED = new Object();
  static final private Object DISCOVERY = new Object();
  // for dijkstra:
  static final private Object WEIGHT = new Object();
  static final private Object DISTANCE = new Object();
  static final private Object PQLOCATOR = new Object();

  // for kruskal
  static final private Object MSF = new Object();
  static final private Object CLUSTER = new Object();

  private Graph<V, E> g;
  private Vertex<V>[] vertexArray;

  public GraphExamples(Graph<V, E> g) {
    this.g = g;
  }

  public boolean isConnected() {
    Vertex<V> v = g.aVertex();
    visitDFS(v);// sets the attr. VISITED to all reachable vertices
    // do we have a VISITED attr. for all vertices?
    Iterator<Vertex<V>> it = g.vertices();
    int cnt = 0;
    while (it.hasNext())
    {
      Vertex<V> w = (Vertex<V>) it.next();
      if(w.has(VISITED))
      {
        cnt++;
        w.destroy(VISITED);
      }
    }

    return g.numberOfVertices() == cnt;
  }

  private void visitDFS(Vertex<V> v) {
    v.set(VISITED, null);
    Iterator<Edge<E>> edges = g.incidentEdges(v);
    while (edges.hasNext())
    {
      Edge<E> e = (Edge<E>) edges.next();
      Vertex<V> neighbour = g.opposite(e, v);
      if(!neighbour.has(VISITED))
      {
        visitDFS(neighbour);
      }
    }
  }
```

```
 61
 62  public int numberOfConnectedComponents() {
 63    Iterator<Vertex<V>> it = g.vertices();
 64    int cnt = 0;
 65
 66    while (it.hasNext())
 67    {
 68      Vertex<V> v = it.next();
 69      if(!v.has(VISITED))
 70      {
 71        cnt++;
 72        visitDFS(v);
 73      }
 74      v.destroy(VISITED);
 75    }
 76
 77    return cnt;
 78  }
 79
 80  private void setGW(Vertex<V> s) {
 81    // sets for all (reachable) vertices the attribute 's' to the value
 82    // 'g' where 'g' is the first vertex on the shortest path
 83    // to 's' (considering 'hopping' distance)
 84    s.set(s, s);
 85    LinkedList<Vertex<V>> q = new LinkedList<>();
 86    q.addLast(s);
 87
 88    while (!q.isEmpty())
 89    {
 90      Vertex<V> v = q.removeFirst();
 91      Iterator<Edge<E>> it = g.incidentEdges(v);
 92      while (it.hasNext())
 93      {
 94        Edge<E> e = (Edge<E>) it.next();
 95        Vertex<V> w = g.opposite(e, v);
 96        if(!w.has(s))
 97        {
 98          w.set(s, v);
 99          q.addLast(w);
100        }
101      }
102    }
103  }
104
105  public void setGateways() {
106    Iterator<Vertex<V>> it = g.vertices();
107    while (it.hasNext())
108    {
109      setGW(it.next());
110    }
111  }
112
113  public Vertex<V>[] shortestPath(Vertex<V> from, Vertex<V> to) {
114    ArrayList<Vertex<V>> al = new ArrayList<>();
115
116    if(from.has(to))
117    {
```

```
118       while (from != to)
119       {
120         al.add(from);
121         from = (Vertex<V>) from.get(to);
122       }
123       al.add(to);
124       return al.toArray(new Vertex[0]);
125     }
126     return null;
127   }
128
129   public int[][] getGatewayMatrix(int[][] ad) {
130     int n = ad.length;
131     int[][] dist = new int[n][n];
132     int[][] gw = new int[n][n];
133     for (int i = 0; i < n; i++)
134       for (int k = 0; k < n; k++)
135       {
136         dist[i][k] = ad[i][k];
137         if(i != k && ad[i][k] != 1)
138           dist[i][k] = n; // infinity!
139         gw[i][k] = -1;
140         if(ad[i][k] == 1)
141           gw[i][k] = k;
142       }
143     for (int k = 0; k < n; k++)
144       for (int i = 0; i < n; i++)
145         for (int j = 0; j < n; j++)
146         {
147           int newDist = dist[i][k] + dist[k][j];
148           if(newDist < dist[i][j])
149           {
150             dist[i][j] = newDist;
151             gw[i][j] = gw[i][k];
152           }
153         }
154     return gw;
155   }
156
157   public void setNumbers() {
158     vertexArray = new Vertex[g.numberOfVertices()];
159     Iterator<Vertex<V>> it = g.vertices();
160     int num = 0;
161     while (it.hasNext())
162     {
163       vertexArray[num] = it.next();
164       vertexArray[num].set(NUMBER, num++);
165     }
166   }
167
168   public int[][] getAdjacencyMatrix() {
169     setNumbers();
170     int n = g.numberOfVertices();
171     int[][] ad = new int[n][n];
172     boolean directed = g.isDirected();
173     Iterator<Edge<E>> it = g.edges();
174     while (it.hasNext())
```

```
175     {
176       Vertex<V>[] endPts = g.endVertices(it.next());
177       int i = (int) endPts[0].get(NUMBER);
178       int k = (int) endPts[1].get(NUMBER);
179       ad[i][k] = 1;
180       if(!directed)
181         ad[k][i] = 1;
182     }
183     return ad;
184   }
185
186   private void visitDFS(int[][] ad, int p, boolean[] visited) {
187     visited[p] = true;
188     for (int i = 0; i < ad.length; i++)
189     {
190       if(ad[p][i] == 1 && !visited[i])
191         visitDFS(ad, i, visited);
192     }
193   }
194
195   public int[] shortestPath(int[][] ad, int from, int to) {
196     // returns the vertex numbers of the shortest path
197     // (hopping distance) fromn 'from' to 'to' or 'null'
198     // if no path exists
199     int n = ad.length;
200     int[] visitedFrom = new int[n];
201     Arrays.fill(visitedFrom, -1);
202     visitedFrom[to] = to;
203     LinkedList<Integer> q = new LinkedList<>();
204     q.addLast(to); // we start at to (for directed graphs!)
205     boolean found = false;
206     while (!q.isEmpty() && !found)
207     {
208       int p = q.removeFirst();
209       for (int i = 0; i < n; i++)
210       {
211         // we take backwards direction!
212         if(ad[i][p] == 1 && visitedFrom[i] == -1)
213         {
214           visitedFrom[i] = p;
215           q.addLast(i);
216           if(i == from)
217             found = true;
218         }
219       }
220     }
221
222     if(visitedFrom[from] == -1)
223       return null;
224     int len = 2;
225     int p = from;
226     // get the length of the path
227     while (visitedFrom[p] != to)
228     {
229       len++;
230       p = visitedFrom[p];
231     }
```

```
232     // now we construct the path
233     int[] path = new int[len];
234     for (int i = 0; i < len; i++)
235     {
236       path[i] = from;
237       from = visitedFrom[from];
238     }
239     return path;
240   }
241
242   public boolean isConnected(int ad[][]) {
243     int n = ad.length;
244     boolean[] visited = new boolean[n];
245     visitDFS(ad, 0, visited);
246     for (boolean v : visited)
247       if(!v)
248         return false;
249     return true;
250   }
251
252   public void dijkstra(Vertex<V> s) {
253     // sets the attribute 's' of each vertex 'u' from wich
254     // we can reach 's' to 'g' where 'g' is the gateway
255     // of 'u' on the shortest path from 'u' to 's'
256     MyPriorityQueue<Double, Vertex<V>> pq = new MyPriorityQueue<>();
257     Iterator<Vertex<V>> it = g.vertices();
258
259     while (it.hasNext())
260     {
261       Vertex<V> v = it.next();
262       v.set(DISTANCE, Double.POSITIVE_INFINITY);
263       Locator<Double, Vertex<V>> loc = pq.insert(Double.POSITIVE_INFINITY, v);
264       v.set(PQLOCATOR, loc);
265     }
266     s.set(DISTANCE, 0.0);
267     pq.replaceKey((Locator<Double, Vertex<V>>) s.get(PQLOCATOR), 0.0);
268
269     while (!pq.isEmpty())
270     {
271       Vertex<V> u = pq.removeMin().element();
272       Iterator<Edge<E>> itEdge = g.incidentInEdges(u);
273       while (itEdge.hasNext())
274       {
275         Edge<E> e = itEdge.next();
276         double weight = 1.0;// default weight
277         if(e.has(WEIGHT))
278           weight = (Double) e.get(WEIGHT);
279         Vertex<V> z = g.opposite(e, u);
280         Double r = (Double) u.get(DISTANCE) + weight;
281
282         if(r < (Double) z.get(DISTANCE))
283         {
284           z.set(DISTANCE, r);
285           z.set(s, u); // set gateway
286           pq.replaceKey((Locator<Double, Vertex<V>>) z.get(PQLOCATOR), r);
287         }
288       }
```

```
289     }
290   }
291
292   public void kruskal() {
293     // gives the attribute MSF to each
294     // edge belonging to an minimal spanning tree
295
296     // create clusters, put the vertex in it
297     // and assign them to the vertices
298     Iterator<Vertex<V>> it = g.vertices();
299     while (it.hasNext())
300     {
301       Vertex<V> v = it.next();
302       ArrayList<Vertex<V>> cluster = new ArrayList<>();
303       cluster.add(v);
304       v.set(CLUSTER, cluster);
305     }
306     PriorityQueue<Double, Edge<E>> pq = new MyPriorityQueue<>();
307     Iterator<Edge<E>> edges = g.edges();
308
309     while (edges.hasNext())
310     {
311       Edge<E> e = edges.next();
312       double weight = (e.get(WEIGHT) == null) ? 1.0 : (Double) e.get(WEIGHT);
313       pq.insert(weight, e);
314     }
315
316     while (!pq.isEmpty())
317     {
318       Edge<E> e = pq.removeMin().element();
319       Vertex<V> v = g.origin(e);
320       Vertex<V> w = g.destination(e);
321
322       ArrayList<Vertex<V>> vCluster = (ArrayList<Vertex<V>>) v.get(CLUSTER);
323       ArrayList<Vertex<V>> wCluster = (ArrayList<Vertex<V>>) w.get(CLUSTER);
324
325       if(vCluster != wCluster)
326       {
327         e.set(MSF, null);
328         // merge clusters
329         if(vCluster.size() > wCluster.size())
330         {
331           for(Vertex<V> x : wCluster){
332             x.set(CLUSTER, vCluster);
333             vCluster.add(x);
334           }
335         }
336         else
337         {
338           for(Vertex<V> x : vCluster){
339             x.set(CLUSTER, wCluster);
340             wCluster.add(x);
341           }
342         }
343
344       }
345       //remove edge which is not in minimum spanning tree
```

```
346       else{
347         g.removeEdge(e);
348       }
349     }
350   }
351
352   /**
353    * @param args
354    *
355    */
356   /**
357    * @param args
358    *
359    */
360   public static void main(String[] args) {
361
362     // make an undirected graph
363     IncidenceListGraph<String, String> g = new IncidenceListGraph<>(false);
364     GraphExamples<String, String> ge = new GraphExamples<>(g);
365
366     Vertex vA = g.insertVertex("A");
367     Vertex vB = g.insertVertex("B");
368     Vertex vC = g.insertVertex("C");
369     Vertex vD = g.insertVertex("D");
370     Vertex vE = g.insertVertex("E");
371     Vertex vF = g.insertVertex("F");
372     Vertex vG = g.insertVertex("G");
373
374     Edge e_a = g.insertEdge(vA, vB, "a");
375     Edge e_b = g.insertEdge(vD, vC, "b");
376     Edge e_c = g.insertEdge(vD, vB, "c");
377     Edge e_d = g.insertEdge(vC, vB, "d");
378     Edge e_e = g.insertEdge(vC, vE, "e");
379     Edge e_f = g.insertEdge(vB, vE, "f");
380     e_f.set(WEIGHT, 7.0);
381     Edge e_g = g.insertEdge(vD, vE, "g");
382     Edge e_h = g.insertEdge(vE, vG, "h");
383     e_h.set(WEIGHT, 3.0);
384     Edge e_i = g.insertEdge(vG, vF, "i");
385     Edge e_j = g.insertEdge(vF, vE, "j");
386
387     System.out.println(g);
388     ge.setGateways();
389     System.out.print("Path: ");
390     Vertex<String>[] path = ge.shortestPath(vA, vG);
391     if(path == null)
392       System.out.println("no path");
393     else
394     {
395       for (Vertex<String> v : path)
396         System.out.print(v);
397     }
398     System.out.println();
399     ge.setNumbers();
400     int[][] ad = ge.getAdjacencyMatrix();
401     System.out.println(ge.isConnected(ad));
402     int[] spath = ge.shortestPath(ad, (int) vC.get(NUMBER), (int) vF.get(NUMBER));
```

```
403     if(spath == null)
404       System.out.println("no path");
405     else
406     {
407       for (int i = 0; i < spath.length; i++)
408       {
409         System.out.println(ge.vertexArray[spath[i]]);
410       }
411     }
412
413     int[][] gw = ge.getGatewayMatrix(ad);
414     int n = gw.length;
415     for (int i = 0; i < n; i++)
416       System.out.println(ge.vertexArray[i] + ", " + i);
417     for (int i = 0; i < n; i++)
418     {
419       System.out.println();
420       for (int k = 0; k < n; k++)
421       {
422         System.out.print(gw[i][k] + ", ");
423       }
424     }
425
426     // A__B F
427     // /|\ /|
428     // / | \ / |
429     // C__D__E__G
430     // \ /
431     // \___/
432     //
433   }
434 }
435
```

```java
package examples;

import java.util.HashMap;

/**
 * A implementation of the Graph interface based on incidence lists (at each vertex a list of
 * all the edges incident to this vertex are are stored).
 * @author ps
 *
 * @param <V> the type of the elements stored at the vertices of this graph
 * @param <E> the type of the elements stored at the edges of this graph
 */
public class IncidenceListGraph<V,E> implements Graph<V, E> {

  private HashSet <ILGVertex> vertices = new HashSet<ILGVertex>();
  private HashSet<ILGEdge> edges = new HashSet<ILGEdge>();
  private boolean isDirected = false;

  public IncidenceListGraph (boolean isDirected){
    this.isDirected = isDirected;
  }

  @Override
  public Vertex<V> aVertex() {
    if (numberOfVertices() > 0) return vertices.iterator().next();
    else return null;
  }

  @Override
  public int numberOfVertices() {
    return vertices.size();
  }

  @Override
  public int NumberOfEdges() {
    return edges.size();
  }

  @Override
  public boolean isDirected() {
    return isDirected;
  }

  @Override
  public Iterator<Vertex<V>> vertices() {
    final Iterator<ILGVertex> it = vertices.iterator();
    return new Iterator<Vertex<V>>() {

      @Override
      public boolean hasNext() {
        return it.hasNext();
      }

      @Override
      public Vertex<V> next() {
        return it.next();
      }
```

```
62
63      @Override
64      public void remove() {
65        it.remove();
66      }
67    };
68  }
69
70  @Override
71  public Iterator<Edge<E>> edges() {
72    final Iterator<ILGEdge> it = edges.iterator();
73    return new Iterator<Edge<E>>() {
74
75      @Override
76      public boolean hasNext() {
77        return it.hasNext();
78      }
79
80      @Override
81      public Edge<E> next() {
82        return it.next();
83      }
84
85      @Override
86      public void remove() {
87        it.remove();
88      }
89    };
90  }
91
92  @Override
93  public Iterator<Edge<E>> incidentEdges(Vertex<V> v) {
94    ILGVertex w = (ILGVertex) v;
95    if (w.thisGraph != this) throw new RuntimeException("Invalid Vertex!");
96    final Iterator<Entry<ILGVertex,ILGEdge>> it = w.iEdges.entrySet().iterator();
97    return new Iterator<Edge<E>>() {
98      @Override
99      public boolean hasNext() {
100       return it.hasNext();
101     }
102
103     @Override
104     public Edge<E> next() {
105       return it.next().getValue();
106     }
107
108     @Override
109     public void remove() {
110       it.remove();
111     }
112   };
113
114 }
115
116 @Override
117 public Iterator<Edge<E>> incidentInEdges(Vertex<V> v) {
118   ILGVertex w = (ILGVertex) v;
```

```
119     if (w.thisGraph != this) throw new RuntimeException("Invalid Vertex!");
120     if (! isDirected) throw new RuntimeException("undirected graph!");
121     final Iterator<Entry<ILGVertex,ILGEdge>> it = w.inIEdges.entrySet().iterator();
122     return new Iterator<Edge<E>>() {
123       @Override
124       public boolean hasNext() {
125         return it.hasNext();
126       }
127
128       @Override
129       public Edge<E> next() {
130         return it.next().getValue();
131       }
132
133       @Override
134       public void remove() {
135         it.remove();
136       }
137     };
138
139   }
140
141   @Override
142   public Iterator<Edge<E>> incidentOutEdges(Vertex<V> v) {
143     ILGVertex w = (ILGVertex) v;
144     if (w.thisGraph != this) throw new RuntimeException("Invalid Vertex!");
145     if (! isDirected) throw new RuntimeException("undirected graph!");
146     final Iterator<Entry<ILGVertex,ILGEdge>> it = w.outIEdges.entrySet().iterator();
147     return new Iterator<Edge<E>>() {
148       @Override
149       public boolean hasNext() {
150         return it.hasNext();
151       }
152
153       @Override
154       public Edge<E> next() {
155         return it.next().getValue();
156       }
157
158       @Override
159       public void remove() {
160         it.remove();
161       }
162     };
163
164   }
165
166   @Override
167   public int degree(Vertex<V> v) {
168     ILGVertex w = (ILGVertex) v;
169     if (w.thisGraph != this) throw new RuntimeException("Invalid Vertex!");
170     return w.iEdges.size();
171   }
172
173   @Override
174   public int inDegree(Vertex<V> v) {
175     ILGVertex w = (ILGVertex) v;
```

```
176     if (w.thisGraph != this) throw new RuntimeException("Invalid Vertex!");
177     if (! isDirected) throw new RuntimeException("undirected graph!");
178     return w.inIEdges.size();
179   }
180
181   @Override
182   public int outDegree(Vertex<V> v) {
183     ILGVertex w = (ILGVertex) v;
184     if (w.thisGraph != this) throw new RuntimeException("Invalid Vertex!");
185     if (! isDirected) throw new RuntimeException("undirected graph!");
186     return w.outIEdges.size();
187   }
188
189   @Override
190   public Vertex<V> origin(Edge<E> e) {
191     ILGEdge iEdge = (ILGEdge) e;
192     if (iEdge.thisGraph != this) throw new RuntimeException("Invalid Edge!");
193     if (! isDirected) throw new RuntimeException("undirected graph!");
194     return iEdge.from;
195   }
196
197   @Override
198   public Vertex<V> destination(Edge<E> e) {
199     ILGEdge iEdge = (ILGEdge) e;
200     if (iEdge.thisGraph != this) throw new RuntimeException("Invalid Edge!");
201     if (! isDirected) throw new RuntimeException("undirected graph!");
202     return iEdge.to;
203   }
204
205   @Override
206   public Vertex<V>[] endVertices(Edge<E> e) {
207     ILGEdge iEdge = (ILGEdge) e;
208     if (iEdge.thisGraph != this) throw new RuntimeException("Invalid Edge!");
209     Vertex <V>  [] v = new Vertex[2];
210     v[0]=iEdge.from;
211     v[1]=iEdge.to;
212     return (Vertex<V> []) v;
213   }
214
215   @Override
216   public boolean areAdjacent(Vertex<V> v1, Vertex<V> v2) {
217     ILGVertex w1 = (ILGVertex) v1;
218     if (w1.thisGraph != this) throw new RuntimeException("Invalid Vertex!");
219     ILGVertex w2 = (ILGVertex) v2;
220     if (w2.thisGraph != this) throw new RuntimeException("Invalid Vertex!");
221     return (w1.iEdges.get(w2)!=null);
222   }
223
224   @Override
225   public Vertex<V> insertVertex(V elem) {
226     ILGVertex v = new ILGVertex(elem);
227     vertices.add(v);
228     return v;
229   }
230
231   @Override
232   public Edge<E> insertEdge(Vertex<V> from, Vertex<V> to, E elem) {
```

```
233     ILGVertex fromV = (ILGVertex) from;
234     if (fromV.thisGraph != this) throw new RuntimeException("Invalid Vertex!");
235     ILGVertex toV = (ILGVertex) to;
236     if (toV.thisGraph != this) throw new RuntimeException("Invalid Vertex!");
237     ILGEdge ed = new ILGEdge(elem, fromV, toV);
238     edges.add(ed);
239     return ed;
240   }
241
242   @Override
243   public E removeEdge(Edge<E> e) {
244     ILGEdge iEdge = (ILGEdge) e;
245     if (iEdge.thisGraph != this) throw new RuntimeException("Invalid Edge!");
246
247       if (isDirected){
248       iEdge.from.outIEdges.remove(iEdge.to);
249       iEdge.to.inIEdges.remove(iEdge.from);
250       }
251       iEdge.from.iEdges.remove(iEdge.to);
252       iEdge.to.iEdges.remove(iEdge.from);
253     edges.remove(iEdge);
254     iEdge.thisGraph = null;
255     return iEdge.element();
256   }
257
258   @Override
259   public V removeVertex(Vertex<V> v) {
260     ILGVertex w = (ILGVertex) v;
261     if (w.thisGraph != this) throw new RuntimeException("Invalid Vertex!");
262     // first we remove all edges!
263     Object [] el = new Object[degree(w)];
264     el = w.iEdges.entrySet().toArray();
265     for (Object e:el){
266       removeEdge((ILGEdge)((Map.Entry)e).getValue());
267     }
268     vertices.remove(w);
269     w.thisGraph=null;
270     return w.element;
271   }
272
273
274   /* (non-Javadoc)
275    * @see java.lang.Object#toString()
276    */
277   public String toString(){
278     StringBuffer sb = new StringBuffer();
279     String con = "---";
280     if (isDirected){
281       sb.append("Type: directed\n");
282       con = "-->";
283     }
284     else sb.append("Type: undirected\n");
285     sb.append("Vertices:\n");
286     Iterator<ILGVertex> it = vertices.iterator();
287     while (it.hasNext()){
288       ILGVertex v = it.next();
289       sb.append("  "+v.toString()+"\n");
```

```
290       Iterator<Edge<E>> eit;
291       if (! isDirected){
292         eit = incidentEdges(v);
293         if (eit.hasNext()) sb.append("       Incident Edges:\n");
294         while(eit.hasNext()){
295           Edge<E> e = eit.next();
296           sb.append("       "+e.toString()+"\n");
297         }
298       }
299       else {
300         eit = incidentOutEdges(v);
301         if (eit.hasNext()) sb.append("       outgoing Edges:\n");
302         while(eit.hasNext()){
303           Edge<E> e = eit.next();
304           sb.append("       "+e.toString()+"\n");
305         }
306         eit = incidentInEdges(v);
307         if (eit.hasNext()) sb.append("       incoming Edges:\n");
308         while(eit.hasNext()){
309           Edge<E> e = eit.next();
310           sb.append("       "+e.toString()+"\n");
311         }
312       }
313
314     }
315     sb.append("Edges:\n");
316     Iterator<ILGEdge> eit = edges.iterator();
317     while (eit.hasNext()){
318       ILGEdge ev= eit.next();
319       sb.append(ev.from.toString() +con+ev.to.toString()+"  "+ev.toString()+"\n");
320     }
321     return sb.toString();
322   }
323
324   @Override
325   public Vertex<V> opposite(Edge<E> e, Vertex<V> v) {
326     ILGVertex w = (ILGVertex) v;
327     if (w.thisGraph != this) throw new RuntimeException("Invalid Vertex!");
328     ILGEdge iEdge = (ILGEdge) e;
329     if (iEdge.thisGraph != this) throw new RuntimeException("Invalid Edge!");
330     if (iEdge.from==w) return iEdge.to;
331     else if (iEdge.to==w) return iEdge.from;
332     else throw new RuntimeException(w+" is not an endpoint of "+iEdge);
333   }
334
335
336   private class ILGVertex extends IGLDecorable implements Vertex<V>{
337     private V element;
338     private IncidenceListGraph<V,E> thisGraph = IncidenceListGraph.this;
339     private HashMap<ILGVertex,ILGEdge> iEdges;
340     private HashMap<ILGVertex,ILGEdge> inIEdges;
341     private HashMap<ILGVertex,ILGEdge> outIEdges;
342
343     private ILGVertex(V e){
344       iEdges = new HashMap<ILGVertex,ILGEdge>(4);
345       if (isDirected){
346         inIEdges = new HashMap<ILGVertex,ILGEdge>(4);
```

```
347         outIEdges = new HashMap<ILGVertex,ILGEdge>(4);
348       }
349       element=e;
350     }
351
352     @Override
353     public V element() {
354       return element;
355     }
356
357     public String toString(){
358       if (element == null) return "null";
359       else return element.toString();
360     }
361
362   }
363
364   private class ILGEdge extends IGLDecorable implements Edge<E>{
365     private E element;
366     private Object thisGraph = IncidenceListGraph.this;
367     private ILGVertex from;
368     private ILGVertex to;
369
370     ILGEdge(E e, ILGVertex from, ILGVertex to){
371       element=e;
372       this.from = from;
373       this.to = to;
374       if (isDirected){
375         if (from.outIEdges.containsKey(to)) throw new RuntimeException("Parallel edges not
    allowed!");
376         from.outIEdges.put(to,this);
377         to.inIEdges.put(from, this);
378       }
379       if (! isDirected && from.iEdges.containsKey(to)) throw new RuntimeException("Parallel
    edges not allowed!");
380       from.iEdges.put(to,this);
381       to.iEdges.put(from, this);
382     }
383
384     @Override
385     public E element() {
386       return element;
387     }
388
389     public String toString(){
390       if (element == null) return "null";
391       else return element.toString();
392     }
393   }
394
395   private class IGLDecorable implements Decorable {
396     private HashMap<Object,Object> attrs = new HashMap<Object,Object>(2);
397     private final Object DUMMY = new Object();
398     @Override
399     public Object get(Object attr) {
400       Object ret = attrs.get(attr);
401       if (ret==null) throw new RuntimeException("no attribute "+attr);
```

```
402       if (ret==DUMMY) ret=null;
403       return ret;
404     }
405
406     @Override
407     public boolean has(Object attr) {
408       Object o = attrs.get(attr);
409       return (o!=null);
410     }
411
412     @Override
413     public void set(Object attr, Object val) {
414       Object value = DUMMY;
415       if (val != null) value = val;
416       attrs.put(attr, value);
417     }
418
419     @Override
420     public Object destroy(Object attr) {
421       Object ret = attrs.get(attr);
422       if (ret != null) attrs.remove(attr);
423       return ret;
424     }
425
426     @Override
427     public void clearAll() {
428       attrs.clear();
429     }
430
431   }
432
433 }
434
```

```java
/**
package examples;

import java.io.File;

/**
 * @author lar02
 *
 */
public class KMP {

  char[] t; // text
  char[] p; // pattern
  int[] prefix;

  int matchCounter;

  int n, m;

  public KMP(char[] t, char[] p) {
    this.t = t;
    this.p = p;
    n = t.length;
    m = p.length;

    prefix = new int[m];
    setFailure();
  }

  private void setFailure() {
    prefix[0] = 0;
    int i = 1;
    int j = 0;

    while (i < m)
    {
      if(p[i] == p[j])
      {
        prefix[i] = j + 1;
        i++;
        j++;
      }
      else if(j > 0)
      {
        j = prefix[j - 1];
      }
      else
      {
        prefix[i] = 0;
        i++;
      }
    }
  }

  public int match() {
    int i = 0;
    int j = 0;
```

```
62
63    matchCounter = 0;
64
65    while (i < n)
66    {
67      matchCounter++;
68      if(t[i] == p[j])
69      {
70        if(j == m - 1)
71        {
72          return i - j;
73        }
74        else
75        {
76          i++;
77          j++;
78        }
79      }
80      else
81      {
82        if(j > 0)
83          j = prefix[j - 1];
84        else
85          i++;
86      }
87    }
88
89    return -1;
90  }
91
92  public void setText(File file) throws IOException {
93    FileInputStream in = null;
94    int c = -1;
95
96    try
97    {
98      in = new FileInputStream(file);
99      int len = in.available();
100     t = new char[len + 1];
101     int i = 0;
102     while ((c = in.read()) != -1 && i < len)
103     {
104       char cb = (char) c;
105       // if (cb<=0 || cb>255) System.out.println("i: "+i+", cb: "+cb);
106       t[i++] = cb;
107     }
108     t[i++] = 0;// stopchar
109     n = t.length - 1;
110   }
111   finally
112   {
113     if(in != null)
114     {
115       in.close();
116     }
117   }
118 }
```

```
119
120   public int getMatchCount(){
121     return matchCounter;
122   }
123
124   /**
125    * @param args
126    */
127   public static void main(String[] args) {
128     String t = "a pattern matching algorithm";
129     String p = "Dorfschulmeister";
130     KMP bm = new KMP(t.toCharArray(), p.toCharArray());
131     try
132     {
133       bm.setText(new File("resources/Goethe.txt"));
134     }
135     catch (IOException e)
136     {
137       // TODO Auto-generated catch block
138       e.printStackTrace();
139     }
140     long s = System.currentTimeMillis();
141     for (int i = 0; i < 1000; i++)
142       bm.match();
143     long e = System.currentTimeMillis();
144     System.out.println(e - s + " micro sec");
145     System.out.println("Vergleiche: " + bm.getMatchCount());
146     System.out.println(" found at " + bm.match());
147     System.out.println(t.indexOf(p));
148   }
149
150 }
151
```

```java
/**
package examples;

import java.util.HashSet;

/**
 * @author ps
 */
public class LCS {

  char[] x, y;
  int n, m;
  int[][] lcs;

  public LCS(String sX, String sY){
    x = sX.toCharArray();
    y = sY.toCharArray();
    n = x.length;
    m = y.length;
    lcs = new int[n+1][m+1];
    for(int i = 1; i <= n; i++)
      for(int k = 1; k <= m; k++)
        if(x[i-1] == y[k-1])
          lcs[i][k] = lcs[i-1][k-1]+1;
        else
          lcs[i][k] = Math.max(lcs[i][k-1], lcs[i-1][k]);
  }

  private String getLCSubSequence(){
    return getLCSubSequence(n, m);
  }

  private String getLCSubSequence(int i, int k){
    if(i == 0 || k == 0)
      return "";

    if(lcs[i-1][k] == lcs[i][k])
      return getLCSubSequence(i-1, k);
    else if(lcs[i][k-1] == lcs[i][k])
      return getLCSubSequence(i, k-1);
    else
      // Bei diagonalem Schritt Buchstabe (aus X) merken => ï¿½bereinstimmung
      return getLCSubSequence(i-1, k-1)+x[i-1];
  }

  public String getEditString(){
    return getEditString(n, m, lcs[n][m]);
  }

  private String getEditString(int i, int k, int len){
    // to do ......
    return "";
  }

  public Set<String> getAllSubseqs(){
    Set<String> al = new HashSet<>();
    getLCSubSequences(al, "", n, m);
```

```
61    return al;
62  }
63
64  public void getLCSubSequences(Set<String> list, String seq, int i, int k){
65    if(i < 1 || k < 1)
66    {
67      list.add(seq);
68      return;
69    }
70
71    if(lcs[i-1][k] == lcs[i][k])
72      getLCSubSequences(list, seq, i-1, k);
73    if(lcs[i][k-1] == lcs[i][k])
74      getLCSubSequences(list, seq, i, k-1);
75    if(x[i-1]==y[k-1]){
76      seq = x[i-1]+seq;
77      getLCSubSequences(list, seq, i-1, k-1);
78    }
79  }
80
81  static public void main(String[] argv){
82    LCS lc = new LCS("SENDEN", "DRESEN");
83    System.out.println(lc.getLCSubSequence());
84    System.out.println(lc.getAllSubseqs());
85    System.out.println(lc.getEditString());
86  }
87}
```

```java
/**
package examples;

import java.util.ArrayList;

/**
 * @author ps
 */
public class MyAVLTree<K extends Comparable<? super K>, E> implements OrderedDictionary<K, E>
 {

  class AVLNode implements Locator<K, E> {

    AVLNode parent, left, right;
    Object creator = MyAVLTree.this;
    E elem;
    K key;
    int height;

    /*
     * (non-Javadoc)
     *
     * @see examples.Position#element()
     */
    @Override
    public E element() {
      return elem;
    }

    /*
     * (non-Javadoc)
     *
     * @see examples.Locator#key()
     */
    @Override
    public K key() {
      return key;
    }

    boolean isExternal() {
      return left == null; // is also true for right
    }

    boolean isLeftChild() {
      return parent != null && parent.left == this;
    }

    boolean isRightChild() {
      return parent != null && parent.right == this;
    }

    void expand(K key, E elem) {
      this.elem = elem;
      this.key = key;
      left = new AVLNode();
      right = new AVLNode();
      left.parent = this;
```

```
61      right.parent = this;
62      height = 1;
63    }
64  }
65
66  // istance variables:
67  private AVLNode root = new AVLNode();
68  private int size;
69
70  private AVLNode checkAndCast(Locator<K, E> p) {
71    try
72    {
73      AVLNode n = (AVLNode) p;
74      if(n.creator == null)
75        throw new RuntimeException(" allready removed locator!");
76      if(n.creator != this)
77        throw new RuntimeException(" locator belongs to another AVLTree instance");
78
79      return n;
80    }
81    catch (ClassCastException e)
82    {
83      throw new RuntimeException(" locator belongs to another container-type ");
84    }
85  }
86
87  /*
88   * (non-Javadoc)
89   *
90   * @see examples.OrderedDictionary#size()
91   */
92  @Override
93  public int size() {
94    return size;
95  }
96
97  public Locator<K, E> find(K key) {
98    // returns the leftmost occurence of
99    // 'key' or null
100
101   AVLNode n = root;
102   AVLNode match = null;
103
104   while (!n.isExternal())
105   {
106     int comp = key.compareTo(n.key);
107     if(comp == 0)
108     {
109       match = n;
110       n = n.left;
111     }
112     else if(comp > 0)
113       n = n.right;
114     else
115       n = n.left;
116   }
117
```

```
118     return match;
119   }
120
121   /*
122    * (non-Javadoc)
123    *
124    * @see examples.OrderedDictionary#findAll(java.lang.Comparable)
125    */
126   @Override
127   public Locator<K, E>[] findAll(K key) {
128     return null;
129   }
130
131   /**
132    * @param n
133    * @param al
134    */
135   private void findAll(K key, AVLNode n, ArrayList<Locator<K, E>> al) {
136   }
137
138   /*
139    * (non-Javadoc)
140    *
141    * @see examples.OrderedDictionary#insert(java.lang.Comparable,
142    * java.lang.Object)
143    */
144   @Override
145   public Locator<K, E> insert(K key, E o) {
146     AVLNode n = root;
147     while (!n.isExternal())
148     {
149       if(n.key.compareTo(key) >= 0)
150         n = n.left;
151       else
152         n = n.right;
153     }
154     n.expand(key, o);
155     adjustHeightAboveAndRebalance(n);
156     size++;
157
158     return n;
159   }
160
161   private void adjustHeightAboveAndRebalance(AVLNode n) {
162     // correct height of all parents
163
164     int height = 1;
165     n.height = height;
166     n = n.parent;
167     while (n != null)
168     {
169       boolean unbalanced = Math.abs(n.left.height - n.right.height) > 1;
170       if(unbalanced)
171         n = restructure(n);
172
173       height++;
174       n.height = height;
```

```
175       n = n.parent;
176     }
177   }
178
179   /*
180    * (non-Javadoc)
181    *
182    * @see examples.OrderedDictionary#remove(examples.Locator)
183    */
184   @Override
185   public void remove(Locator<K, E> loc) {
186     AVLNode n = checkAndCast(loc);
187     AVLNode w = null;
188
189     if(n.left.isExternal() || n.right.isExternal())
190       w = removeAboveExternal(n);
191     else
192     {
193
194     }
195     adjustHeightAboveAndRebalance(w);
196     size--;
197     n.creator = null;
198   }
199
200   /**
201    * @param n
202    */
203   private AVLNode removeAboveExternal(AVLNode n) {
204     // returns the node which replaces n
205     AVLNode w;
206
207     if(n.left.isExternal())
208     {
209       w = n.right;
210       w.parent = n.parent;
211       if(n.parent.left == n)
212         n.parent.left = w;
213       else if(n.parent.right == n)
214         n.parent.right = w;
215       else
216         root = w;
217     }
218     else
219     {
220       w = n.left;
221       w.parent = n.parent;
222       if(n.parent.left == n)
223         n.parent.left = w;
224       else if(n.parent.right == n)
225         n.parent.right = w;
226       else
227         root = w;
228     }
229
230     return w;
231   }
```

```java

/*
 * (non-Javadoc)
 *
 * @see examples.OrderedDictionary#closestBefore(java.lang.Comparable)
 */
@Override
public Locator<K, E> closestBefore(K key) {
  // TODO Auto-generated method stub
  return null;
}

/*
 * (non-Javadoc)
 *
 * @see examples.OrderedDictionary#closestAfter(java.lang.Comparable)
 */
@Override
public Locator<K, E> closestAfter(K key) {
  // TODO Auto-generated method stub
  return null;
}

/*
 * (non-Javadoc)
 *
 * @see examples.OrderedDictionary#next(examples.Locator)
 */
@Override
public Locator<K, E> next(Locator<K, E> loc) {
  AVLNode n = checkAndCast(loc);

  if(n.left != null){
    return n;
  }
  else if(n.right != null){
    n = n.right;
    while(n.left != null){
      n = n.left;
    }

  }
  return null;
}

/*
 * (non-Javadoc)
 *
 * @see examples.OrderedDictionary#previous(examples.Locator)
 */
@Override
public Locator<K, E> previous(Locator<K, E> loc) {
  // TODO Auto-generated method stub
  return null;
}

/*
```

```
289   * (non-Javadoc)
290   *
291   * @see examples.OrderedDictionary#min()
292   */
293  @Override
294  public Locator<K, E> min() {
295    // TODO Auto-generated method stub
296    return null;
297  }
298
299  /*
300   * (non-Javadoc)
301   *
302   * @see examples.OrderedDictionary#max()
303   */
304  @Override
305  public Locator<K, E> max() {
306    // TODO Auto-generated method stub
307    return null;
308  }
309
310  /*
311   * (non-Javadoc)
312   *
313   * @see examples.OrderedDictionary#sortedLocators()
314   */
315  @Override
316  public Iterator<Locator<K, E>> sortedLocators() {
317    // TODO Auto-generated method stub
318    return null;
319  }
320
321  private void print(AVLNode r, String in) {
322    if(r.isExternal())
323      return;
324    print(r.right, in + "..");
325    System.out.println(in + r.key + "(h=" + r.height + ")");
326    print(r.left, in + "..");
327  }
328
329  public void print() {
330    print(root, "");
331  }
332
333  private AVLNode restructure(AVLNode n) {
334    // cnt++;
335    // n is unbalanced
336    // returns the node that takes the position of n
337    AVLNode p = n.parent, z = n, x = null, y = null, a = null, b = null, c = null, t1 = null,
   t2 = null;
338    // t0 and t3 never change their parent,
339    // that's why we don't need them
340    if(z.left.height > z.right.height)
341    {
342      // z
343      // /
344      // y
```

```
345       c = z;
346       y = z.left;
347       if(y.left.height >= y.right.height)
348       {
349         // in case we have two equal branches
350         // concidering the length we take alway s the single
351         // rotation
352         // z
353         // /
354         // y
355         // /
356         // x
357         x = y.left;
358         t1 = x.right;
359         t2 = y.right;
360         b = y;
361         a = x;
362       }
363       else
364       {
365         // z
366         // /
367         // y
368         // \
369         // x
370         x = y.right;
371         t1 = x.left;
372         t2 = x.right;
373         a = y;
374         b = x;
375       }
376     }
377     else
378     {
379       // z
380       // \
381       // y
382       a = z;
383       y = z.right;
384       if(y.right.height >= y.left.height)
385       {
386         // z
387         // \
388         // y
389         // \
390         // x
391         x = y.right;
392         b = y;
393         c = x;
394         t1 = y.left;
395         t2 = x.left;
396       }
397       else
398       {
399         // z
400         // \
401         // y
```

```
402         // /
403         // x
404         x = y.left;
405         b = x;
406         c = y;
407         t1 = x.left;
408         t2 = x.right;
409       }
410     }
411     // umhaengen
412     b.parent = p;
413     if(p != null)
414     {
415       if(p.left == z)
416       {
417         p.left = b;
418       }
419       else
420         p.right = b;
421     }
422     else
423     {
424       root = b;
425     }
426     b.right = c;
427     b.left = a;
428     // und umgekehrt
429     a.parent = b;
430     c.parent = b;
431
432     // subtrees:
433     a.right = t1;
434     t1.parent = a;
435     c.left = t2;
436     t2.parent = c;
437
438     a.height = Math.max(a.left.height, a.right.height) + 1;
439     c.height = Math.max(c.left.height, c.right.height) + 1;
440     // now we can calculate the height of b
441     b.height = Math.max(b.left.height, b.right.height) + 1;
442     return b;
443   }
444
445   public static void main(String[] argv) {
446     MyAVLTree<Integer, String> t = new MyAVLTree<>();
447     Random rand = new Random(3434534);
448     int n = 10;
449     Locator<Integer, String>[] locs = new Locator[n];
450     long time1 = System.currentTimeMillis();
451     for (int i = 0; i < n; i++)
452     {
453       int k = rand.nextInt(n);
454       // System.out.println("insert key: " + k);
455       locs[i] = t.insert(k, "" + i);
456       // locs[i]=t.insert(i, "bla");
457     }
458     t.print();
```

```
459     System.out.println(t.find(4).key());
460     // for (int i=0;i<n/2;i++) {
461     // t.remove(t.find(locs[i].key()));
462     // }
463   }
464
465 }
466
```

```java
package examples;

import java.util.Iterator;

public class MyLinkedList<E> implements List<E> {

  class LNode implements Position<E> {

    E elem;
    LNode next, prev;
    Object creator = MyLinkedList.this; // pointer to outer instance

    @Override
    public E element() {
      return elem;
    }

  }

  private LNode first, last;
  private int size;

  @Override
  public Position<E> first() {
    return first;
  }

  @Override
  public Position<E> last() {
    return last;
  }

  @Override
  public boolean isFirst(Position<E> p) {
    return castToLNode(p) == first;
  }

  @Override
  public boolean isLast(Position<E> p) {
    return castToLNode(p) == last;
  }

  @Override
  public Position<E> next(Position<E> p) {
    return castToLNode(p).next;
  }

  private LNode castToLNode(Position<E> p) {
    LNode n;

    try
    {
      n = (LNode) p;
    }
    catch (ClassCastException e)
    {
      throw new RuntimeException("This is not a Position belonging to MyLinkedList");
```

```
60      }
61      if(n.creator == null) throw new RuntimeException("position was allready deleted!");
62      if(n.creator != this) throw new RuntimeException("position belongs to another List
    instance!");
63
64      return n;
65    }
66
67    @Override
68    public Position<E> previous(Position<E> p) {
69      return castToLNode(p).prev;
70    }
71
72    @Override
73    public E replaceElement(Position<E> p, E o) {
74      LNode old = castToLNode(p);
75      old.creator = null; // invalidate old node
76      LNode n = new LNode();
77      n.elem = o;
78      n.next = old.next;
79      n.prev = old.prev;
80
81      if(old == first)
82      {
83        first = n;
84        if(first.next != null) first.next.prev = n;
85        return old.elem;
86      }
87      if(old == last)
88      {
89        last = n;
90        if(last.prev != null) last.prev.next = n;
91        return old.elem;
92      }
93
94      old.next.prev = n;
95      old.prev.next = n;
96
97      return old.elem;
98    }
99
100   @Override
101   public Position<E> insertFirst(E o) {
102     LNode n = new LNode();
103     n.elem = o;
104     n.next = first;
105     if(first != null)
106       first.prev = n;
107     else
108       last = n;
109
110     size++;
111     first = n;
112
113     return n;
114   }
115
```

```java
@Override
public Position<E> insertLast(E o) {
  LNode n = new LNode();
  n.elem = o;
  n.prev = last;

  if(last != null)
    last.next = n;
  else
    first = n;

  size++;
  last = n;

  return n;
}

@Override
public Position<E> insertBefore(Position<E> p, E o) {
  LNode old = castToLNode(p);
  LNode n = new LNode();
  n.elem = o;
  n.next = old;
  n.prev = old.prev;

  if(first == old)
    first = n;
  if(old.prev != null)
    old.prev.next = n;

  old.prev = n;
  size++;

  return n;
}

@Override
public Position<E> insertAfter(Position<E> p, E o) {
  LNode old = castToLNode(p);
  LNode n = new LNode();
  n.elem = o;
  n.next = old.next;
  n.prev = old;

  if(last == old) last = n;

  if(old.next != null)
  {
    old.next.prev = n;
  }
  old.next = n;
  size++;

  return n;
}

@Override
```

```
173  public void remove(Position<E> p) {
174    if(size == 0) throw new RuntimeException("List is empty!");
175
176    LNode n = castToLNode(p);
177    size--;
178    n.creator = null; // invalidate p
179
180    if(n == first)
181    {
182      first = n.next;
183      if(first != null) first.prev = null;
184    }
185    else if(n == last)
186    {
187      last = n.prev;
188      if(last != null) last.next = null;
189    }
190    else
191    {
192      n.prev.next = n.next;
193      n.next.prev = n.prev;
194    }
195
196  }
197
198  @Override
199  public Iterator<Position<E>> positions() {
200    return new Iterator<Position<E>>() {
201
202      LNode current = first;
203
204      @Override
205      public boolean hasNext() {
206        return current != null;
207      }
208
209      @Override
210      public Position<E> next() {
211        LNode ret = current;
212        current = current.next;
213        return ret;
214      }
215
216      @Override
217      public void remove() {
218        throw new NotImplementedException();
219      }
220    };
221  }
222
223  @Override
224  public Iterator<E> elements() {
225    return new Iterator<E>() {
226
227      LNode current = first;
228
229      @Override
```

```
230       public boolean hasNext() {
231         return current != null;
232       }
233
234       @Override
235       public E next() {
236         E elem = current.elem;
237         current = current.next;
238         return elem;
239       }
240
241       @Override
242       public void remove() {
243         throw new NotImplementedException();
244       }
245     };
246   }
247
248   @Override
249   public int size() {
250     return size;
251   }
252
253   @Override
254   public boolean isEmpty() {
255     return size == 0;
256   }
257
258   public static void main(String[] args) {
259     List<String> li = new MyLinkedList<>();
260     System.out.println("insert hans");
261     Position<String> p5 = li.insertFirst("hans");
262     li.insertBefore(p5, "before");
263     li.insertAfter(p5, "after");
264     // System.out.println(li.last().element());
265     // System.out.println("remove " + li.last().element());
266     // li.remove(li.last());
267     // Position<String> p4 = li.insertFirst("heiri");
268     // Position<String> p = li.insertFirst("susi");
269     // li.insertFirst("heidi");
270     Position<String> p1 = li.first();
271     while (p1 != null)
272     {
273       System.out.println(p1.element());
274       p1 = li.next(p1);
275     }
276     // System.out.println("-------");
277     // li.replaceElement(p, "raffi");
278     // li.insertBefore(p4, "danae");
279     // Position<String> p3 = li.first();
280     // while (p3 != null)
281     // {
282     // System.out.println(p3.element());
283     // p3 = li.next(p3);
284     // }
285     // System.out.println("remove " + li.last().element());
286     // li.remove(li.first());
```

```
287     // Position<String> p2 = li.first();
288     // while (p2 != null)
289     // {
290     // System.out.println(p2.element());
291     // p2 = li.next(p2);
292     // }
293   }
294
295 }
296
```

```java
/**
package examples;

import java.util.Arrays;

/**
 * @author ps Implements an array-heap based priority-queue with Locators
 */
public class MyPriorityQueue<K extends Comparable<? super K>, E> implements PriorityQueue<K, E>
{

  class PQLocator<K1 extends Comparable<? super K1>, E1> implements Locator<K1, E1>
  {

    K1 key;
    E1 elem;
    Object creator = MyPriorityQueue.this;
    int pos; // position in the heap-array

    /*
     * (non-Javadoc)
     *
     * @see examples.Position#element()
     */
    @Override
    public E1 element()
    {
      return elem;
    }

    /*
     * (non-Javoadoc)
     *
     * @see examples.Locator#key()
     */
    @Override
    public K1 key()
    {
      return key;
    }

  }

  private PQLocator<K, E>[] locs = (PQLocator<K, E>[]) new PQLocator[256];
  private int size = 1; // we start at 1 because the navigation is simpler

  /*
   * (non-Javadoc)
   *
   * @see examples.PriorityQueue#showMin()
   */
  @Override
  public Locator<K, E> showMin()
  {
    return locs[1];
  }
```

```
60
61  /*
62   * (non-Javadoc)
63   *
64   * @see examples.PriorityQueue#removeMin()
65   */
66  @Override
67  public Locator<K, E> removeMin()
68  {
69    Locator<K, E> ret = showMin();
70    remove(ret);
71    return ret;
72  }
73
74  /*
75   * (non-Javadoc)
76   *
77   * @see examples.PriorityQueue#insert(java.lang.Comparable, java.lang.Object)
78   */
79  @Override
80  public Locator<K, E> insert(K key, E element)
81  {
82    PQLocator<K, E> n = new PQLocator<>();
83    n.key = key;
84    n.elem = element;
85    if(size == locs.length)
86      expand();
87    locs[size] = n;
88    n.pos = size;
89    upHeap(size);
90    size++;
91    return n;
92  }
93
94  /**
95   *
96   */
97  private void expand()
98  {
99    locs = Arrays.copyOf(locs, locs.length * 2);
100 }
101
102 /**
103  * @param i
104  */
105 private void upHeap(int i)
106 {
107   int parent = i / 2;
108   while(i > 0 && parent > 0)
109   {
110     if(locs[i].key.compareTo(locs[parent].key) < 0)
111     {
112       swap(parent, i);
113       i = parent;
114       parent = i / 2;
115     }
116     else
```

```
117         i = 0;
118       }
119     }
120
121     /**
122      * @param i
123      */
124     private void downHeap(int i)
125     {
126
127     }
128
129     /**
130      * @param i
131      * @param k
132      */
133     private void swap(int i, int k)
134     {
135       PQLocator<K, E> tmp = locs[i];
136       locs[i] = locs[k];
137       locs[k] = tmp;
138       // do'nt forget the 'pos' values:
139       locs[i].pos = i;
140       locs[k].pos = k;
141     }
142
143     /*
144      * (non-Javadoc)
145      *
146      * @see examples.PriorityQueue#remove(examples.Locator)
147      */
148     @Override
149     public void remove(Locator<K, E> loc)
150     {
151       PQLocator<K, E> l = (PQLocator<K, E>) loc;
152       if(l.creator != this)
153         throw new RuntimeException("invalid locator");
154       int pos = l.pos;
155       swap(pos, --size);
156       l.creator = null; // invalidate node
157       upHeap(pos);
158       System.out.println("isHeap? " + isHeap());
159       downHeap(pos);
160     }
161
162     /*
163      * (non-Javadoc)
164      *
165      * @see examples.PriorityQueue#replaceKey(examples.Locator,
166      * java.lang.Comparable)
167      */
168     @Override
169     public void replaceKey(Locator<K, E> loc, K newKey)
170     {
171       PQLocator<K, E> l = (PQLocator<K, E>) loc;
172       if(l.creator != this)
173         throw new RuntimeException("invalid locator" + loc.element());
```

```
174     int comp = l.key.compareTo(newKey);
175     l.key = newKey;
176     if(comp < 0)
177       downHeap(l.pos);
178     else if(comp > 0)
179       upHeap(l.pos);
180   }
181
182   /*
183    * (non-Javadoc)
184    *
185    * @see examples.PriorityQueue#isEmpty()
186    */
187   @Override
188   public boolean isEmpty()
189   {
190     return size == 1;
191   }
192
193   /*
194    * (non-Javado
195    *
196    * @see examples.PriorityQueue#size()
197    */
198   @Override
199   public int size()
200   {
201     return size;
202   }
203
204   private boolean isHeap()
205   {
206     for(int i = 2; i < size; i++)
207     {
208       if(locs[i].key.compareTo(locs[i / 2].key) < 0)
209         return false;
210     }
211     return true;
212   }
213
214   static public void main(String[] argv)
215   {
216     int N = 10;
217     MyPriorityQueue<Double, String> pq = new MyPriorityQueue<>();
218     Locator<Double, String>[] locs = new Locator[N];
219     Random ra = new Random(63465);
220     for(int i = 0; i < N / 2; i++)
221       locs[i] = pq.insert(ra.nextDouble(), null);
222     for(int i = 0; i < N / 2; i++)
223       locs[i + N / 2] = pq.insert(ra.nextDouble(), null);
224     for(int i = 0; i < N / 2; i++)
225       pq.removeMin();
226     System.out.println(pq.isHeap());
227
228   }
229 }
230
```

```java
/**
package examples;

import java.io.File;

/**
 * @author ps
 *
 */
public class MySuffixTree {


  class Node {
    int start;
    int end;

    Node(int s, int e){
      start=s;
      end=e;
    }

    public String toString(){
      return start+":"+end;
    }
  }

  private MyTree<Node> tree = new MyTree<>();
  private char [] t; // text of this suffix tree
  private int n; // last pos in text
//  private char [] p; // pattern

  public MySuffixTree(String txt){

    t = new char[txt.length()+1];

    txt.getChars(0,txt.length(),t,0);
    n=t.length-1;
    t[n]=0; //
    tree.createRoot(new Node(-1,-1)); // dummy node
    for (int i=0;i<n;i++){
      this.insertSuffix(i);
    }
  }

  public void setText(File file) throws IOException{
    FileInputStream in = null;
    int c =-1;

        try {
            in = new FileInputStream(file);
            int len = in.available();
            t = new char[len+1];
            int i=0;
            while ((c = in.read()) != -1 && i<len) {
              char cb = (char) c;
              // if (cb<=0 || cb>255) System.out.println("i: "+i+", cb: "+cb);
              t[i++] = cb;
```

```
65            }
66            t[i++]=0;// stopchar
67            n = t.length-1;
68        } finally {
69            if (in != null) {
70                in.close();
71            }
72        }
73    }
74
75
76  public MySuffixTree(File file){
77    try {
78      setText(file);
79    } catch (IOException e) {
80      // TODO Auto-generated catch block
81      e.printStackTrace();
82    }
83    n = t.length-1;
84    tree.createRoot(new Node(-1,-1)); // dummy node
85    for (int i=0;i<=n;i++){
86      this.insertSuffix(i);
87    }
88  }
89
90
91  private void insertSuffix(int pos){
92    Position<Node> po = tree.root();
93    boolean f = true;
94    while (f){
95      f=false;
96      Iterator<Position<Node>> it = tree.childrenPositions(po);
97      while (it.hasNext()){
98        Position<Node> v = it.next();
99        Node n = v.element(); // child
100       int i = n.start;
101       if (t[i]==t[pos]){
102         // we got the right node
103         pos++;
104         i++;
105         // try to match as many chars as possible
106         while (i<=n.end){
107           if (t[i] != t[pos]) {
108             break;
109           }
110           i++;
111           pos++;
112         }
113         // did we match all ?
114         if (i>n.end){
115           // yes, we matched all of this node
116           // so we break the while hasNext() loop and continue one level deeper
117           po=v;
118           f = true;
119           break; // while hasNext()
120         }
121         else {
```

```
122             // we split here
123             // i points to the first pos which does not match
124             Node newN = new Node(n.start,i-1);
125             Position<Node> newP = tree.insertParent(v,newN);
126             n.start=i;
127             tree.addChild(newP, new Node(pos,this.n));// add the new branch
128             return;
129           }
130         }
131       }
132     }
133     // we add a new child with the rest of the suffix
134     tree.addChild(po, new Node(pos,n));
135   }
136
137   public ArrayList<Integer> search(char[] p){
138     ArrayList<Integer> al = new ArrayList<Integer>();
139     Position<Node> po = tree.root();
140     boolean f = true;
141     int j = 0;  // we start at this position of the pattern
142     int len = p.length; // we have to match this many chars
143     while (f){ // loop over all levels
144       f=false;
145       Iterator<Position<Node>> it = tree.childrenPositions(po);
146       while (it.hasNext()){
147         Position<Node> v = it.next();
148         Node nd = (Node)v.element();
149         int i = nd.start;
150         if (t[i]==p[j]){
151           // we got the right node
152           int x = nd.end-i+1; // #of chars stored at this node
153           // is this node shorter than the rest of the patern?
154           if (len <= x){
155             // yes. Either we match all chars of p or there is no match at all
156             while(j<p.length-1){
157               if (p[++j] != t[++i]) return al;
158             }
159             // we found a pattern that ends at t[i]
160             //  System.out.println();
161             int offset = nd.end-i+p.length-1;
162             findBranchLengths(al,v,offset);
163             return al;
164           }
165           else {
166             //No. Therefore all of the chars stored
167             // at this node should match and we continue one level deeper
168             while(i<nd.end){
169               if (p[++j] != t[++i]) return al;
170             }
171             // everything of this node matched
172             len = len - x;
173             j++; // next j
174             po=v;
175             f=true;
176             break; // break while hasNext() --> so we go one level deeper
177           }
178         }
```

```
179       }
180     }
181     return al;
182   }
183 
184 
185   /**
186    * @param al
187    * @param p
188    * @param offset
189    */
190   private void findBranchLengths(ArrayList<Integer> al, Position<Node> p, int offset) {
191     if (tree.numberOfChildren(p)==0) al.add(this.n-offset);
192     else {
193       Iterator<Position<Node>> it = tree.childrenPositions(p);
194       while (it.hasNext()){
195         Position<Node> pc = it.next();
196         Node nd = (Node)pc.element();
197         int len = nd.end-nd.start+1;
198         findBranchLengths(al, pc, offset+len);
199       }
200     }
201   }
202 
203   /**
204    * @param args
205    */
206   public static void main(String[] args) {
207 
208     MySuffixTree st = new MySuffixTree(new File("resources/Goethe.txt"));
209 
210     //MySuffixTree st = new MySuffixTree("abracadabra");
211                                        //01234567890123456789012
212     long ts = System.currentTimeMillis();
213     //st.tree.print();
214     // repeat the search 1'000 times
215     ArrayList<Integer> al = null;
216     for (int i=0;i<1000;i++){
217       al = (st.search("Dorfschulmeister".toCharArray()));
218     }
219     long te= System.currentTimeMillis();
220     TreeSet<Integer> s = new TreeSet<>();
221     s.addAll(al);
222     // st.tree.print();
223     System.out.println("pos: "+s+" time: "+(te-ts)+" micro s");
224   }
225 
226 }
227 
```

```java
package examples;

import java.util.ArrayList;

public class MyTree<E> implements Tree<E> {

  class TNode implements Position<E> {

    TNode parent;
    E elem;
    MyLinkedList<TNode> children = new MyLinkedList<>();
    Position<TNode> mySiblingPos;
    Object creator = MyTree.this;

    @Override
    public E element() {
      return elem;
    }

  }

  private TNode root;
  private int size;

  private TNode castToTNode(Position<E> p) {
    TNode n;
    try
    {
      n = (TNode) p;
    }
    catch (ClassCastException e)
    {
      throw new RuntimeException("This is not a Position belonging to MyTree");
    }
    if(n.creator == null) throw new RuntimeException("position was allready deleted!");
    if(n.creator != this) throw new RuntimeException("position belongs to another MyLinkedList instance!");
    return n;
  }

  @Override
  public Position<E> root() {
    return root;
  }

  @Override
  public Position<E> createRoot(E o) {
    if(root != null) throw new RuntimeException("already a root node present");
    TNode n = new TNode();
    n.elem = o;
    size++;
    root = n;
    return n;
  }

  @Override
  public Position<E> parent(Position<E> child) {
```

```
60      return castToTNode(child).parent;
61    }
62
63    @Override
64    public Iterator<Position<E>> childrenPositions(Position<E> parent) {
65      final TNode p = castToTNode(parent);
66      return new Iterator<Position<E>>() {
67        Iterator<TNode> it = p.children.elements();
68
69        @Override
70        public boolean hasNext() {
71          return it.hasNext();
72        }
73
74        @Override
75        public Position<E> next() {
76          return it.next();
77        }
78
79        @Override
80        public void remove() {
81          throw new NotImplementedException();
82        }
83      };
84    }
85
86    @Override
87    public Iterator<E> childrenElements(Position<E> parent) {
88      final TNode p = castToTNode(parent);
89      return new Iterator<E>() {
90        Iterator<TNode> it = p.children.elements();
91
92        @Override
93        public boolean hasNext() {
94          return it.hasNext();
95        }
96
97        @Override
98        public E next() {
99          return it.next().elem;
100       }
101
102       @Override
103       public void remove() {
104         throw new NotImplementedException();
105       }
106     };
107   }
108
109   @Override
110   public int numberOfChildren(Position<E> parent) {
111     TNode p = castToTNode(parent);
112     return p.children.size();
113   }
114
115   @Override
116   public Position<E> insertParent(Position<E> p, E o) {
```

```
117     // TODO Auto-generated method stub
118     return null;
119   }
120
121   @Override
122   public Position<E> addChild(Position<E> parent, E o) {
123     TNode par = castToTNode(parent);
124     TNode child = new TNode();
125     child.elem = o;
126     child.parent = par;
127     child.mySiblingPos = par.children.insertLast(child);
128     size++;
129     return child;
130   }
131
132   @Override
133   public Position<E> addChildAt(int pos, Position<E> parent, E o) {
134     TNode par = castToTNode(parent);
135     TNode child = new TNode();
136     child.elem = o;
137     child.parent = par;
138
139     if(pos >= par.children.size())
140     {
141       child.mySiblingPos = par.children.insertLast(child);
142     }
143     else
144     {
145       Position<TNode> position = getSiblingPosition(pos, par);
146       if(position == null) throw new RuntimeException("Position not found");
147       child.mySiblingPos = par.children.insertBefore(position, child);
148     }
149
150     size++;
151     return null;
152   }
153
154   private Position<TNode> getSiblingPosition(int pos, TNode parent) {
155     Iterator<Position<TNode>> it = parent.children.positions();
156     int counter = 0;
157
158     while (it.hasNext())
159     {
160       if(counter == pos) return it.next();
161       it.next();
162       counter++;
163     }
164     return null;
165   }
166
167   @Override
168   public Position<E> addSiblingAfter(Position<E> sibling, E o) {
169     TNode sib = castToTNode(sibling);
170     if(sib == root) throw new RuntimeException("root can not have siblings");
171     TNode n = new TNode();
172     n.parent = sib.parent;
173     n.elem = o;
```

```
174     n.mySiblingPos = sib.parent.children.insertAfter(sib.mySiblingPos, n);
175     size++;
176     return n;
177   }
178
179   @Override
180   public Position<E> addSiblingBefore(Position<E> sibling, E o) {
181     TNode sib = castToTNode(sibling);
182     if(sib == root) throw new RuntimeException("root can not have siblings");
183     TNode n = new TNode();
184     n.parent = sib.parent;
185     n.elem = o;
186     n.mySiblingPos = sib.parent.children.insertBefore(sib.mySiblingPos, n);
187     size++;
188     return null;
189   }
190
191   @Override
192   public void remove(Position<E> p) {
193     TNode n = castToTNode(p);
194     size--;
195     n.creator = null; //invalidate node
196
197     if(n==root)
198       root = null;
199     else
200       n.parent.children.remove(n.mySiblingPos);
201   }
202
203   @Override
204   public boolean isExternal(Position<E> p) {
205     return castToTNode(p).children.size() == 0;
206   }
207
208   @Override
209   public boolean isInternal(Position<E> p) {
210     return castToTNode(p).children.size() > 0;
211   }
212
213   @Override
214   public int size() {
215     return size;
216   }
217
218   @Override
219   public E replaceElement(Position<E> p, E o) {
220     TNode n = castToTNode(p);
221     E temp = n.elem;
222     n.elem = o;
223     return temp;
224   }
225
226   public void print() {
227     print(root, "");
228   }
229
230   /**
```

```
231   * @param root2
232   */
233  private void print(TNode p, String indent) {
234    // print the subtree originating at p
235    System.out.println(indent + p.elem);
236    Iterator<TNode> it = p.children.elements();
237    while (it.hasNext())
238    {
239      print(it.next(), indent + "   ");
240    }
241  }
242
243  public int height() {
244    return height(root);
245  }
246
247  private int height(TNode p) {
248    // System.out.println(max);
249    int h = 0;
250    Iterator<TNode> it = p.children.elements();
251    while (it.hasNext())
252    {
253      h = Math.max(height(it.next()), h);
254    }
255    return h + 1;
256  }
257
258  public ArrayList<Position<E>> externalNodes() {
259    ArrayList<Position<E>> list = new ArrayList<Position<E>>();
260    return externalNodes(root, list);
261  }
262
263  private ArrayList<Position<E>> externalNodes(TNode p, ArrayList<Position<E>> list) {
264    Iterator<TNode> it = p.children.elements();
265    if(it.hasNext() == false) list.add(p);
266    while (it.hasNext())
267    {
268      externalNodes(it.next(), list);
269    }
270    return list;
271  }
272
273  class Helper {
274    TNode n;
275    int depth = -1;
276  }
277
278  public Position<E> deepestNode() {
279    Helper he = new Helper();
280
281    deepestNode(root, he.depth, he);
282    return he.n;
283  }
284
285  private void deepestNode(TNode n, int currentDepth, Helper he) {
286    int depth = currentDepth + 1;
287
```

```
288     if(isExternal(n))
289     {
290       if(currentDepth > he.depth)
291       {
292         he.depth = currentDepth;
293         he.n = n;
294       }
295       return;
296     }
297     Iterator<TNode> it = n.children.elements();
298 
299     while (it.hasNext())
300     {
301       deepestNode(it.next(), depth, he);
302     }
303     return;
304   }
305 
306   public static void main(String[] args) {
307     MyTree<String> t = new MyTree<>();
308     Position<String> pA = t.createRoot("A");
309     Position<String> pB = t.addChild(pA, "B");
310     t.addChild(pA, "C");
311     Position<String> pD = t.addChild(pA, "D");
312     t.addChild(pB, "E");
313     t.addChild(pB, "F");
314     Position<String> pG = t.addChild(pD, "G");
315     t.addChild(pG, "X");
316     t.addChild(pG, "Y");
317     t.addChildAt(3, pA, "Z");
318     t.addChildAt(3, pA, "ZZ");
319     t.print();
320     System.out.println("--------------------");
321     t.remove(pD);
322     t.print();
323     System.out.println("--------------------");
324     System.out.println("height: " + t.height());
325     System.out.println("--------------------");
326     System.out.println("external nodes: ");
327     ArrayList<Position<String>> all = t.externalNodes();
328     for (Position<String> r : all)
329       System.out.println(r.element());
330     System.out.println("--------------------");
331     Position<String> deepest = t.deepestNode();
332     System.out.println("deepest node: " + deepest.element());
333     System.out.println("--------------------");
334     System.out.println("number of children: " + t.numberOfChildren(pA));
335   }
336 }
337 
```

```java
package examples;

import java.lang.management.ManagementFactory;

/**
 * @author ps Various sort programs for int arrays
 */
public class SortTest {

  public static long cnt;
  static Random rand = new Random();

  public static void heapSort(int[] a) {
    int n = a.length;
    // first we make 'a' a max-heap:
    for (int i = 1; i < n; i++)
      upHeap(a, i);

    System.out.println("heap?" + checkHeap(a, n));
    // Now we remove the max element and move it to the
    // last position of the array 'a'. repair the heap with 'downHeap'
    // Heap has now only n-1 elements.
    // Repeat until all elements are at place.
    for (int i = n - 1; i >= 0; i--)
    {
      swap(a, i, 0);
      downHeap(a, 0, i);
    }

  }

  /**
   * @param a
   * @param i
   * @param len
   */
  private static void downHeap(int[] a, int i, int len) {
    // assume a [i..len-1] is a heap, but the element
    // at position i possibly violates the heap condition.
    // swap a[i] with its bigger child until a[i..len-1] is a heap.
    int left = i * 2 + 1;
    while (left < len)
    {
      int right = left + 1;
      int current = left;

      if(right < len && a[left] < a[right])
        current = right;

      if(a[i] >= a[current])
        break;

      swap(a, i, current);
      i = current;
      left = i * 2 + 1;
    }
  }
```

```
60
61  /**
62   * @param a
63   *          heap
64   * @param i
65   *          position in heap
66   */
67  private static void upHeap(int[] a, int i) {
68    // swap i with parent if parent bigger until top element reached
69
70    // assume a[0..i-1] is a heap. swap element
71    // at position i with its parent and so on
72    // until a[0..i] is a max-heap
73    int p = (i - 1) / 2;
74    while (i > 0)
75    {
76      if(a[p] < a[i])
77      {
78        swap(a, p, i);
79        i = p;
80        p = (i - 1) / 2;
81      }
82      else
83        i = 0;
84    }
85  }
86
87  static boolean checkHeap(int a[], int len) {
88    for (int i = a.length - 1; i > 0; i--)
89    {
90      if(a[i] > a[(i - 1) / 2])
91        return false;
92    }
93    return true;
94  }
95
96  /**
97   * @param a
98   *          int aray
99   * @return 'true' if 'a' is sorted
100  */
101 public static boolean sortCheck(int[] a) {
102   for (int i = 0; i < a.length - 1; i++)
103   {
104     if(a[i] > a[i + 1])
105       return false;
106   }
107   return true;
108 }
109
110 /**
111  * @param a
112  *          int aray
113  * @return 'true' if 'a' is sorted
114  */
115 public static boolean sortCheck(int[] a, int to) {
116   for (int i = 0; i < a.length - 1 && i <= to; i++)
```

```
117     {
118       if(a[i] > a[i + 1])
119         return false;
120     }
121     return true;
122   }
123 
124   /**
125    * Non optimized bubble sort for an int array
126    *
127    * @param a
128    */
129   public static void bubbleSort(int[] a) {
130     cnt = 0;
131     int m = a.length - 1;
132     for (int i = m; i > 0; i--)
133     {
134       for (int k = 0; k < i; k++)
135       {
136         if(a[k] > a[k + 1])
137           swap(a, k, k + 1);
138       }
139       // now a[i] is on its final position!
140     }
141   }
142 
143   /**
144    * swap the array elements a[i] and a[k]
145    *
146    * @param a
147    *          int array
148    * @param i
149    *          position in the array 'a'
150    * @param k
151    *          position in the array 'a'
152    */
153   static void swap(int[] a, int i, int k) {
154     int tmp = a[i];
155     a[i] = a[k];
156     a[k] = tmp;
157     cnt++;
158   }
159 
160   /**
161    * Wrapper which calls the recursive version of the quick sort program
162    *
163    * @param a
164    *          the int array to be sorted
165    */
166   public static void quickSort(int[] a) {
167     qSort(a, 0, a.length - 1);
168   }
169 
170   /**
171    * recursive version of quick sort (sorts the range a[from..to] of the int
172    * array 'a')
173    *
```

```
174   * @param a
175   * @param from
176   * @param to
177   */
178  private static void qSort(int[] a, int from, int to) {
179    if(from >= to)
180      return; // nothing to do if sequence has length 1 or less
181    int piv = partition(a, from, to);
182    // now a[to..piv-1] <= a[piv] and
183    // a[piv+1..to]>=a[piv]
184    qSort(a, from, piv - 1);
185    qSort(a, piv + 1, to);
186  }
187
188  public static void quickSelect(int[] a, int rank) {
189    // after return of this method the elements a[0]..a[rank-1]
190    // are all smaller or equal to a[rank]
191    // and the remaining elements a[rank+1]..a[a.lenght-1] are all
192    // bigger or equal to a[rank]
193    // to be completed:
194    // ----------------
195    qSelect(a, 0, a.length - 1, rank);
196  }
197
198  private static void qSelect(int[] a, int from, int to, int p) {
199    int piv = partition(a, from, to);
200    if(piv == p)
201      return;
202    else if(piv < p)
203      qSelect(a, piv+1, to, p);
204    else
205      qSelect(a, from, piv-1, p);
206  }
207
208  /**
209   * partitions the range such that all of the elements in the range
210   * a[from..piv-1] are smaller than a[piv] and all elements in the range
211   * a[piv+1..to] are greater or equal than a[piv]
212   *
213   * @param a
214   * @param from
215   * @param to
216   * @return the position 'piv' of the pivot
217   */
218  private static int partition(int[] a, int from, int to) {
219    // take a random pivot and put it at the end
220    // of the range
221    // (necessary if data not random)
222    if(from < to)
223      swap(a, rand.nextInt(to - from) + from, to);
224
225    int pivot = a[to];
226    int left = from - 1;
227    int right = to;
228
229    while (true)
230    {
```

```
231       while (a[++left] < pivot); // stoppt bei Tauschkandidat von links
232       while (a[--right] >= pivot)
233       {
234         if(right == from)
235           break;
236       }
237       if(right <= left)
238         break;
239       swap(a, left, right);
240     }
241
242     swap(a, to, left);
243     return left; // return the final position of the pivot (to be changed!)
244   }
245
246   public static void main(String[] args) {
247     long t1 = 0, t2 = 0, te1 = 0, te2 = 0, eTime = 0, time = 0;
248     int n = 10;
249     // we need a random generator
250     Random rand = new Random();
251     // rand.setSeed(54326346); // initialize always in the same state
252     ThreadMXBean threadBean = ManagementFactory.getThreadMXBean();
253     // new array
254     int[] a = new int[n];
255     // fill it randomly
256     for (int i = 0; i < a.length; i++)
257       a[i] = rand.nextInt(n);
258     cnt = 0; // for statistcs reasons
259     // get Time
260     te1 = System.currentTimeMillis();
261     t1 = threadBean.getCurrentThreadCpuTime();
262     quickSelect(a, 5);
263     te2 = System.currentTimeMillis();
264     t2 = threadBean.getCurrentThreadCpuTime();
265     time = t2 - t1;
266     eTime = te2 - te1;
267     System.out.println("CPU-Time usage: " + time / 1000000.0 + " ms");
268     System.out.println("elapsed time: " + eTime + " ms");
269     System.out.println("sorted? " + sortCheck(a));
270     System.out.println("swap operation needed: " + cnt);
271   }
272
273 }
274
```