

GRUPPUPPGIFT DA353A VT17

KOMPLETTERING 1

Grupp 1

Medlemmar:

Anas Abu Al-Soud

Henrik Fredlund

Philip Ekholm

Viktor Torki

Innehållsförteckning

Arbetsbeskrivning	4
Instruktioner för programstart.....	4
Systembeskrivning	5
Klassdiagram.....	6
Sekvensdiagram	7
Inläsning av filer och uppstart av program.....	7
Utlåning av media objekt.....	7
Returnering av media-objekt.....	7
Listning av utlånade media-objekt	7
Källkod	7
ArrayList	7
AVLNode.....	14
AVLTree	15
Book.....	23
BorrowedItemsView	24
CompareDateBorrowed	26
DVD	27
ItemsController.....	27
ItemsView	31
Lantagare.txt.....	33
LibraryModel.....	33
LinkedList	35
LinkedList	41
List	43
ListNode	45
LoginController.....	47
LoginView	49

Main.....	50
Media	50
Media.txt	51
Person	53
PersonModel	54
PriorityQueue.....	55
Queue	58
QueueException.....	59
SearchTree.....	60

Arbetsbeskrivning

Philip Ekholm

Arkitekten bakom hela systemet, har kommit på strukturen av systemet. Vi har Philip att tacka för datastrukturerna som användes för det här projektet såväl som mallen för dokumentet.

Anas Abu Al-Soud

Jobbat med inloggningssystemet där man säkerställer att ett visst personnummer är korrekt. Anas har också hållit på en hel del med testningen och säkerställt att alla funktioner fungerar som tänkt.

Henrik Fredlund

Arbetat med gränssnittet för biblioteket såväl som vissa funktioner i biblioteket. Har även byggt in en scroll-funktion hos vyerna.

Viktor Torki

Har jobbat med kompletteringen av projektet, gett bra förslag på vad som kan förbättras i arkitekturen, speciellt sedan en komplettering skulle skapas. Han har gett förslag på det nya upplägget som är fritt från kontroller-arv såväl som upplägg för filstruktur. Han har (även om kompletteringen slutligen inte krävde detta) hjälpt till med att uppdatera klassdiagrammet såväl som sekvensdiagrammen då vi kände de åtminstone bör vara uppdaterade med tanke på att vi gjorde en hel del förändringar. Vi anser att han på senare hand har hjälpt oss så pass mycket att han bör bli godkända på den här uppgiften tillsammans med oss andra, eller att ingen här blir godkänd.

Instruktioner för programstart

Programmet ska köras i Eclipse. Ni ska skapa ett nytt java-projekt via att ta:

file → new... → Java Project

Projektet kan döpas om till valfritt namn (så länge som Eclipse är okej med detta). För att köra projektet rekommenderas Java 1.8 Standard Edition och sedan kan man trycka "finish".

Ta sedan ut paketen ur "src"-mappen som kommer med projektet och dra in dem i src-mappen som ligger i ditt nyskapare Java-projekt.

För att projektet ska fungera som tänkt så är det också viktigt, utöver källkoden, att få in textfilerna som innehåller information kring diverse låntagare och böcker/DVD-skivor som tillhör biblioteket. För att inkludera dessa höger-klickar du på java projektet, och sedan tar mapp, kalla denna för filer. Ta sedan Lantagare.txt respektive Media.txt och lägg dessa i mappen.

För att köra programmet sedan så navigerar du till paketet common, och sedan dubbel-klickar på "Main.java". Du kan sedan köra detta med hjälp av gröna play-knappen ovanför. Nu bör programmet starta.

Systembeskrivning

Programmet är skrivet med 100 % standard java, vilket gör det enkelt att förstå helheten. Programmet är strukturerat enligt MVC-mönstret (Model, View, Controller) varav filerna är uppdelade i olika paket utifrån. Första versionen av detta program använde en del generalisering men då detta stötte på kritik är controller-delen nu arv-fri. Det som också är nytt är Models som inte fanns tidigare där strukturerna lagrar data.

För lagringsstruktur används allra mest AVL-träd då det tillåter snabb sökning såväl som sortering. Ett AVL-träd blir inte heller (till skillnad från standard BST) degenererat automatiskt utan hela tiden ombalanserar sig så fort nya element läggs till. Detta ger en viss prestandaförlust men vi finner detta litet jämfört med att sortera andra datastrukturer som oftast är linjära. Ett AVL-träd är också sorterat till skillnad från hashtabeller vilket är varför vi föredrar detta.

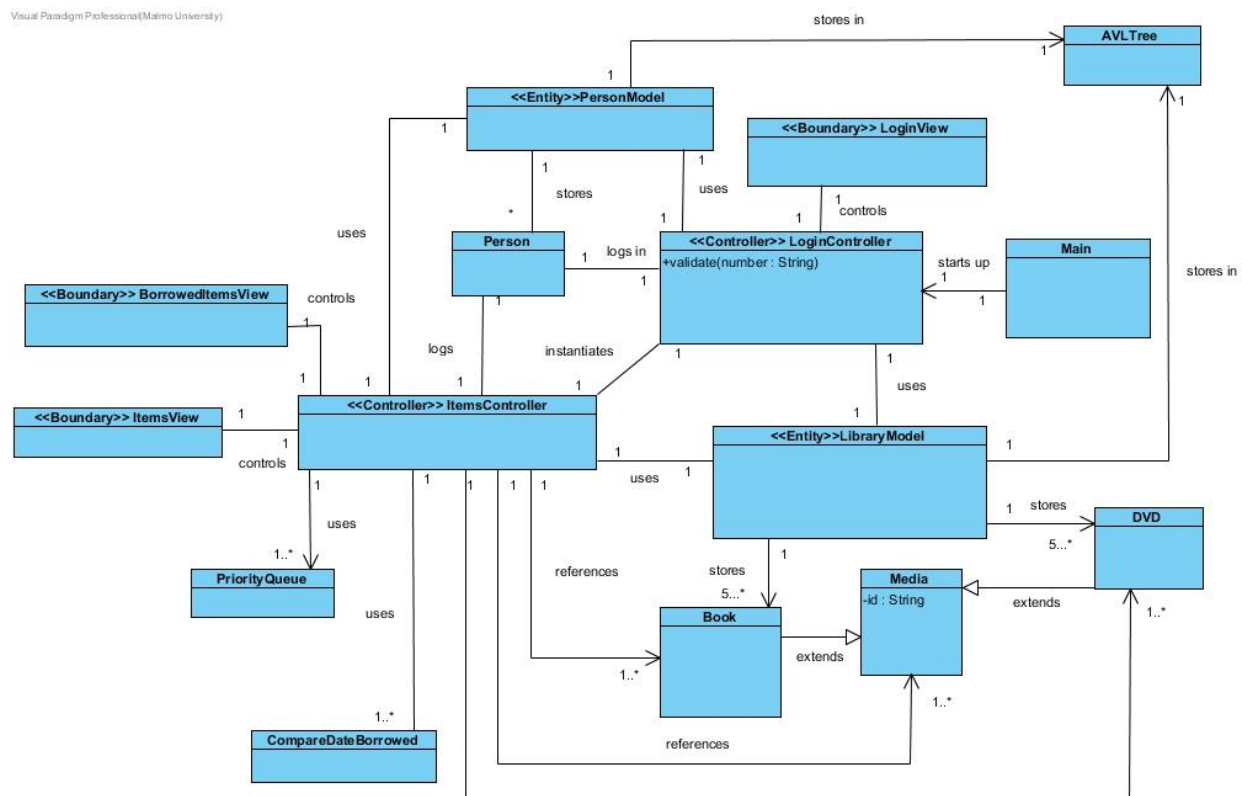
Vad som är nytt sedan tidigare är hur man håller koll på vilken media som är lånad tidigast resp. senast. Tidigare användes en ArrayList för att hålla koll på detta men det blev otillräckligt då man lade till möjligheten att logga ut och in dynamiskt. Det vi gjorde istället var att lägga till ytterligare en egenskap till Media, nämligen Date (standardklass i Java). När ett objekt lånas ut så skapas ett nytt date-objekt (som innehåller nuvarande tiden) och lagrar detta tillsammans med mediaobjektet. Detta lagras i sin tur i en PriorityQueue som sorterar utlånade mediaobjekt med hänsyn till vilken date som är senast (görs

mer specifikt i `CompareDateBorrowed.java`). Sätt att jämföra datum finns inbyggt i `Date` redan då den implementerar `Comparable`.

Första delen består av en inloggning där man enbart loggar in med ett personnummer. Vi fann lösenord till resp. användare överflödigt för den här uppgiften så vi tog bort detta. Skulle man vilja inkludera detta kan man använda sig av en hashing-funktion och sedan lagra hashen tillsammans med text-datan tillsammans med en viss person. Efter en inloggning kan man välja att gå vidare. Man kommer då vidare till huvuddelen av programmet där man kan låna resp. återlämna media. Man kan också välja att byta användare och därmed logga ut och låta någon annan logga in istället.

Klassdiagram

För att särskilja mellan diverse stereotyper står stereotypen (om applicerbar) inom dubbla malltaggar (<<>>). Större bild finns bifogad tillsammans med inlämningen.



Sekvensdiagram

Större bilder finns bifogade tillsammans med inlämningen. Vi vill tacka Viktor Torki för arbetet att uppdatera samtliga diagram för inlämningen.

Inläsning av filer och uppstart av program

Ansvarig: Philip Ekholm

Utlåning av media objekt

Ansvarig: Philip Ekholm

Returnering av media-objekt

Ansvarig: Henrik Fredlund

Listning av utlånade media-objekt

Ansvarig: Anas Abu Al-Soud

Källkod

Filändelse är .java om inget annat står givet. För framtida inlämningar av andra studenter tipsar vi om att använda ett verktyg för att skriva ut koden ordentligt för att underlätta arbetet. Exempel på ett sådant program är följande plugin för sublime-text: [Print to HTML](#)

ArrayList

```
1 package collections;
2
3 import java.util.Iterator;
4 import java.util.NoSuchElementException;
5
6 /**
7  *   ArrayList E
8  *
9  *   A class that extends the functionality of list using a simple
   Array-data structure, implements the
```

```

10  *      interface List which states certain methods to be implemented.
Supports generics to work with any homogeneous datatype.
11  *
12  *      @author Rolf Axelsson
13  *      @author Philip Ekholm
14  */
15
16  public class ArrayList<E> implements List<E> {
17      /**
18       *      One array elements to keep track of actual data
19       *      size used to determine current length of list
20       */
21
22      private E[] elements;
23      private int size;
24
25      /**
26       *      grow will double the size of the current list if
27       *      number of elements exceed the number of cells available by
28       *      creating a new array twice the size
29       */
30
31      private void grow() {
32          E[] temp = (E[])new Object[2 * elements.length];
33          for(int i = 0; i < this.elements.length; i++){
34              temp[i] = this.elements[i];
35          }
36
37          this.elements = temp;
38      }
39
40      /**
41       *      Pass default size of 20 to base constructor if no argument
passed.
42       */
43
44      public ArrayList() {
45          this(20);
46      }
47
48      /**
49       *      Base constructor for this class, creates an initial capacity of
50       *      at least 1, will be extended later if too small.
51       *
52       *      @param initialCapacity size of the initial list
53       */
54
55      public ArrayList(int initialCapacity) {
56          initialCapacity = Math.max(1, initialCapacity);
57          elements = (E[])new Object[initialCapacity];
58      }
59
60      /**
61       *      Adds a new element to the list using target index.
62       *
63       *      @param index the target position to add the element to

```



```

64      *      @param element the element to be added to the list
65      *
66      *      @throws IndexOutOfBoundsException if invalid index is passed
67      */
68
69      public void add(int index, E element) {
70          if(index<0 || index>size)
71              throw new IndexOutOfBoundsException();
72          if(size==elements.length)
73              grow();
74          for(int i=size; i>index; i--) {
75              elements[i]=elements[i-1];
76          }
77          elements[index] = element;
78          size++;
79      }
80
81      /**
82      *      Adds a new element to the list, will be passed to {@code
add(int, <E>)}
83      *      with size, meaning it will add the element to the end of the
list.
84      *
85      *      @param element the element to be added to the list
86      */
87
88      public void add(E element) {
89          add(size,element);
90      }
91
92      /**
93      *      Adds a new element to the list, will be passed to {@code
add(int, <E>)}
94      *      with target index 0, meaning it will add the element to the
'top' of the list.
95      *
96      *      @param element the element to be added to the list
97      */
98
99      public void addFirst(E element) {
100          add(0, element);
101      }
102
103      /**
104      *      Same functionality as addLast.
105      *
106      *      @param element the element to be added to the list
107      */
108
109      public void addLast(E element) {
110          add(size, element);
111      }
112
113      /**
114      *      Will remove and return targeted object at passed index.
115      *

```

```

116      *      @param index target index where element can be found
117      *      @return the object to be removed from the list
118      *      @throws IndexOutOfBoundsException if invalid index is passed
119      */
120
121      public E remove(int index) {
122          E targetObject;
123
124          if(index < 0 || index > this.size){
125              throw new IndexOutOfBoundsException("Invalid index passed to
remove");
126          }
127          else{
128              targetObject = this.get(index);
129
130              for(int t = index; t < this.size; t++){
131                  this.elements[t] = this.elements[t + 1];
132              }
133
134              size--;
135          }
136
137          return targetObject;
138      }
139
140      /**
141       *      Removes the first element on the list with index 0.
142       *
143       *      @return result from remove with argument index = 0
144       */
145
146      public E removeFirst() {
147          return this.remove(0);
148      }
149
150      /**
151       *      Removes the last element on the list with index size.
152       *
153       *      @return result from remove with argument index = size
154       */
155
156      public E removeLast() {
157          return this.remove(this.size);
158      }
159
160      /**
161       *      Removes all elements in the list by looping through and
dereferencing object references.
162       */
163
164      public void clear() {
165          for(int i = 0; i < this.size; i++){
166              this.elements[i] = null;
167          }
168

```

```

169         size = 0;
170     }
171
172     /**
173      * Returns the object found at passed index, will throw exception
174      * if invalid index is passed.
175      *
176      * @param index target index where element can be found
177      * @return the object at given index
178      * @throws IndexOutOfBoundsException if invalid index is passed
179      */
180     public E get(int index) {
181         if(index < 0 || index > this.size){
182             throw new IndexOutOfBoundsException("Invalid index passed to
183 get");
184         }
185         return this.elements[index];
186     }
187
188     /**
189      * Set a new object at a given index in list.
190      *
191      * @param index target index where element can be found
192      * @param element the object to be set with
193      * @return the old object that was replaced
194      * @throws IndexOutOfBoundsException if invalid index is passed
195      */
196
197     public E set(int index, E element) {
198         E previousObj;
199
200         if(index < 0 || index > this.size){
201             throw new IndexOutOfBoundsException("Invalid index passed to
202 set");
203         }
204         else{
205             previousObj = this.elements[index];
206             this.elements[index] = element;
207         }
208         return previousObj;
209     }
210
211     /**
212      * Returns the index of a given object, will return -1 if not
213      * found.
214      *
215      * @param element the element being searched for
216      * @return result from method indexOf(int, E)
217      */
218
219     public int indexOf(E element) {
220         return this.indexOf(0, element);
221     }

```

```

220     }
221
222     /**
223      * Will return the index of a given object that has the same
224      * reference
225      * as passed object, give startIndex to improve search speed, will
226      * return -1 if not found.
227      * @param startIndex to start searching at to improve search
228      * performance
229      * @param element the object whos reference will be compared
230      * @return the index of object, -1 if not found
231      */
232     public int indexOf(int startIndex, E element) {
233         for(int i = startIndex; i < this.size; i++){
234             if(elements[i].equals(element)){
235                 return i;
236             }
237         }
238         return -1;
239     }
240
241     /**
242      * Returns the size of the element.
243      *
244      * @return the current size of the list
245      */
246
247     public int size() {
248         return this.size;
249     }
250
251     /**
252      * Overrides the toString from superclass and returns a listing of
253      * objects as a string.
254      * This is done via StringBuilder class.
255      * @see StringBuilder
256      * @return the list as a string.
257      */
258
259     public String toString() {
260         StringBuilder res = new StringBuilder("[ ");
261         for(int i=0; i<size; i++) {
262             res.append(elements[i]);
263             if(i<size-1)
264                 res.append("; ");
265         }
266         res.append(" ]");
267         return res.toString();
268     }
269
270     /**

```

```

271      *    A simple method to return an iterator object in order to loop
the list with
272      *    other means than index arithmetics. Currently uses a private
class, the lines
273      *    commented do this by passing an anonymous class implementing
the interface Iterator.
274      *
275      *    @return a new iterator object
276      */
277
278      public Iterator<E> iterator() {
279          return new Iter();
280      //      return new Iterator<E>() {
281      //          private int index=0;
282      //
283      //          public boolean hasNext() {
284      //              return index<size;
285      //          }
286      //
287      //          public E next() {
288      //              if(index==size)
289      //                  throw new NoSuchElementException();
290      //              return elements[index++];
291      //          }
292      //
293      //          public void remove() {
294      //              throw new UnsupportedOperationException();
295      //          }
296      //      };
297      }
298
299      /**
300      *    The private Iter class for passing Iterator objects, giving
means to loop through the list
301      */
302
303      private class Iter implements Iterator<E> {
304          private int index=0;
305
306          public boolean hasNext() {
307              return index<size;
308          }
309
310          public E next() {
311              if(index==size)
312                  throw new NoSuchElementException();
313              return elements[index++];
314          }
315
316          public void remove() {
317              throw new UnsupportedOperationException();
318          }
319      }
320 }

```

AVLNode

```
1 package collections;
2
3 /**
4  *   AVLNode
5  *
6  *   A wrapper class for storing data in a tree data-structure, works
7  *   by linking together nodes in an hierarchy, should be used together
with
8  *   AVLTree.
9  *
10  *   @author Rolf Axelsson
11  *   @author Philip Ekholm
12  */
13
14 class AVLNode<K,V> {
15     K key;
16     V value;
17     AVLNode<K,V> left;
18     AVLNode<K,V> right;
19     int height=0;
20
21     /**
22      *   Basic constructor for the class, assigns data stored together with
a key
23      *   as well as left and right children (does not need to be set
explicitly).
24      *
25      *   @param key key in which the node is identified with.
26      *   @param value the data object to be stored in the node.
27      *   @param left the left node to be assigned as left-child to this
node.
28      *   @param right the right node to be assigned as right-child to this
node.
29      */
30
31     public AVLNode( K key, V value, AVLNode<K,V> left, AVLNode<K,V>
right ) {
32         this.key = key;
33         this.value = value;
34         this.left = left;
35         this.right = right;
36     }
37
38     /**
39      *   Get the height from this node all the way down to the leaves of
40      *   the hierarchy, uses a recursive implementation to count the depth
41      *   of the structure.
42      *
43      *   @return the height of the structure downwards.
44      */
45
46     public int height() {
47         int leftD = -1, rightD = -1;
```

```

48         if( left != null )
49             leftD = left.height();
50         if( right != null )
51             rightD = right.height();
52         return 1 + Math.max( leftD, rightD );
53     }
54
55     /**
56      * Get the number of nodes from this node all the way down to the
57      * leaves of the hierarchy, uses a recursive implementation to count
58      * the number of nodes in the structure.
59      *
60      * @return the number of nodes in the structure downwards.
61      */
62
63     public int size() {
64         int leftS = 0, rightS = 0;
65         if( left != null )
66             leftS = left.size();
67         if( right != null )
68             rightS = right.size();
69         return 1 + leftS + rightS;
70     }
71
72     /**
73      * Get a console print of the current node (key and value) as well as
74      * for all the nodes downwards in the hierarchy. Algorithm uses a
recursive
75      * implementation to print all the nodes in the structure.
76      */
77
78     public void print() {
79         if( left != null )
80             left.print();
81         System.out.println(key + ": " + value);
82         if( right != null )
83             right.print();
84     }
85 }

```

AVLTree

```

1 package collections;
2 import java.util.Comparator;
3 import java.util.Iterator;
4 import java.util.NoSuchElementException;
5
6 /**
7  * AVLTree
8  *
9  * AVLTree is a written datastructure class for storing data in a non-
linear
10  * structure, while AVLTree is based on binary search trees it has the
11  * ability to auto balance the tree whenever the structure degenerates
more

```

```

12 *   than a balance factor of -1 or +1. Uses wrapper objects AVLNode for
storing
13 *   data in.
14 *
15 *   @author Philip Ekholm
16 *   @created 2017-03-04
17 */
18
19 public class AVLTree<K,V> implements SearchTree<K,V> {
20     private Comparator<K> comparator;
21     private AVLNode<K,V> tree;
22
23     /**
24      *   If no custom comparator implementation has been made
25      *   a default comparator as inner class will be instantiated.
26      */
27
28     public AVLTree() {
29         comparator = new Comp();
30     }
31
32     /**
33      *   Custom comparator can be implemented depending
34      *   on how keys should be compared against each other.
35      *
36      *   @param comp class that implements Comparator with overridden
compare.
37      */
38
39     public AVLTree( Comparator<K> comp ) {
40         comparator = comp;
41     }
42
43     /**
44      *   Return the root of the whole AVL-structure.
45      *
46      *   @return root of the AVL-Structure.
47      */
48
49     public AVLNode<K,V> root() {
50         return tree;
51     }
52
53     /**
54      *   find a certain AVLNode using passed key, if found the value of
the
55      *   node will be returned, otherwise null will be returned.
56      *
57      *   @param key key of the target node to be obtained.
58      *   @return the value of the node if found, otherwise null.
59      */
60
61     public V get(K key) {
62         AVLNode<K,V> node = find( key );
63         if (node!=null)

```



```

64         return node.value;
65     return null;
66 }
67
68 /**
69  * put (insert) a new node into the AVL-structure using a unique key
70  * as well as a value (data). Make sure the key is unique to the
71  * AVL-structure to avoid errors.
72  *
73  * @param key key identifier unique for the created node.
74  * @param value the data value to pass to the newly created node.
75  */
76
77 public void put(K key, V value) {
78     tree = put(tree, key, value);
79 }
80
81 /**
82  * Remove a node from the AVL-structure by passing the key of the
83  * target node.
84  *
85  * @param key key identifier unique for the created node.
86  * @return the value of the node that was removed if successful,
87  * otherwise
88  * returns null.
89  */
90
91 public V remove(K key) {
92     V value = get( key );
93     if (value != null) {
94         tree = remove(tree, key);
95     }
96     return value;
97 }
98
99 /**
100  * Check if a certain node with passed key can be found in the
101  * AVL-structure.
102  *
103  * @param key key identifier unique for the created node.
104  * @return true if the node with specific key could be found,
105  * otherwise false.
106  */
107
108 public boolean contains( K key ) {
109     return find( key ) != null;
110 }
111
112 /**
113  * Returns the height of the AVL-structure.
114  *
115  * @return the height of the AVL-structure.
116  */
117
118 public int height() {

```

```

115         return height( tree );
116     }
117
118     private int height( AVLNode<K,V> node ) {
119         if( node == null )
120             return -1;
121         return 1 + Math.max( height( node.left ), height( node.right ) );
122     }
123
124     /**
125      *   Return a new instance of an iterator for iterating all the
values of
126      *   nodes in the AVL-structure.
127      *
128      *   @return new instance of iterator for values.
129      */
130
131     public Iterator<V> iterator() {
132         return new IterValues();
133     }
134
135     /**
136      *   Return a new instance of an iterator for iterating all the keys
of
137      *   nodes in the AVL-structure.
138      *
139      *   @return new instance of iterator for keys.
140      */
141
142     public Iterator<K> iteratorKeys() {
143         return new IterKeys();
144     }
145
146     private AVLNode<K,V> find(K key) {
147         int res;
148         AVLNode<K,V> node=tree;
149         while( ( node != null ) && ( ( res = comparator.compare( key,
node.key ) ) != 0 ) ) {
150             if( res < 0 )
151                 node = node.left;
152             else
153                 node = node.right;
154         }
155         return node;
156     }
157
158     private AVLNode<K,V> put(AVLNode<K,V> node, K key, V value) {
159         if( node == null ) {
160             node = new AVLNode<K,V>( key, value, null, null );
161         } else {
162             if(comparator.compare(key,node.key)<0) {
163                 node.left = put(node.left,key,value);
164                 node = this.balanceLeft(node);
165             } else if(comparator.compare(key,node.key)>0) {
166                 node.right = put(node.right,key,value);

```

```

167         node = this.balanceRight (node);
168     }
169 }
170 return node;
171 }
172
173 private AVLNode<K,V> remove(AVLNode<K,V> node, K key) {
174     int compare = comparator.compare(key,node.key);
175     if(compare==0) {
176         if(node.left==null && node.right==null)
177             node = null;
178         else if(node.left!=null && node.right==null)
179             node = node.left;
180         else if(node.left==null && node.right!=null)
181             node = node.right;
182         else {
183             AVLNode<K,V> min = getMin(node.right);
184             min.right = remove(node.right,min.key);
185             min.left = node.left;
186             node = min;
187         }
188     } else if(compare<0) {
189         node.left = remove(node.left, key);
190     } else {
191         node.right = remove(node.right, key);
192     }
193     node = this.balanceNode (node);
194     return node;
195 }
196
197 private AVLNode<K,V> getMin(AVLNode<K,V> node) {
198     while(node.left!=null)
199         node = node.left;
200     return node;
201 }
202
203 /**
204  *   Return the size ("length") of the AVL-structure, will be
205  *   calculated using recursion.
206  *
207  *   @return the size of the tree.
208  */
209
210 @Override
211 public int size() {
212     return this.size(tree);
213 }
214
215 // Laboration 13
216 private int size(AVLNode<K, V> node) {
217     int l = 0, r = 0;
218
219     if(this.tree == null) {
220         return 0;
221     }

```

```

222         if (node.left != null) {
223             l = this.size(node.left);
224         }
225         if (node.right != null) {
226             r = this.size(node.right);
227         }
228
229         return 1 + r + l;
230     }
231
232
233     public List<K> keys() {
234         ArrayList<K> list = new ArrayList<K>();
235
236         Iterator<K> iter = new IterKeys();
237
238         while (iter.hasNext()) {
239             list.add(iter.next());
240         }
241
242         return list;
243     }
244
245     public List<V> values() {
246         ArrayList<V> list = new ArrayList<V>();
247
248         Iterator<V> iter = this.iterator();
249
250         while (iter.hasNext()) {
251             list.add(iter.next());
252         }
253
254         return list;
255     }
256
257     /**
258      *   Operation not supported in this implementation.
259      *
260      *   @throws UnsupportedOperationException since operation is
261      *   unsupported.
262      */
263     public V first() throws UnsupportedOperationException {
264         throw new UnsupportedOperationException();
265     }
266
267     /**
268      *   Operation not supported in this implementation.
269      *
270      *   @throws UnsupportedOperationException since operation is
271      *   unsupported.
272      */
273     public V last() throws UnsupportedOperationException {
274         throw new UnsupportedOperationException();
275     }

```

```

275
276  /**
277   *   When the tree is instantiated either a class that implements
278   *   Comparator can be passed to constructor, otherwise this class
279   *   will be passed which tries to typecast the passed arguments
280   *   to Comparable.
281   *
282   *   If typecasting fails then ClassCastException will be thrown.
283   */
284
285  private class Comp implements Comparator<K> {
286      public int compare( K key1, K key2 ) {
287          Comparable<K> k1 = ( Comparable<K> )key1;
288          return k1.compareTo( key2 );
289      }
290  }
291
292  private AVLNode<K,V> balanceNode (AVLNode<K,V> node) {
293      if (node!=null) {
294          node = balanceLeft (node);
295          node = balanceRight (node);
296      }
297      return node;
298  }
299
300  private AVLNode<K,V> balanceLeft (AVLNode<K,V> node) {
301      if ((this.height (node.left) - this.height (node.right) == 2)){
302          if (this.height (node.left.left) - this.height (node.left.right)
303 == -1) {
304              node.left = this.rotateLeft (node.left);
305              node = this.rotateRight (node);
306          }
307          else {
308              node = this.rotateRight (node);
309          }
310      }
311      return node;
312  }
313
314  private AVLNode<K,V> balanceRight (AVLNode<K,V> node) {
315      if (this.height (node.left) - this.height (node.right) == -2){
316          if (this.height (node.right.left) -
317 this.height (node.right.right) == 1){
318              node.right = this.rotateRight (node.right);
319              node = this.rotateLeft (node);
320          }
321          else {
322              node = this.rotateLeft (node);
323          }
324      }
325      return node;
326  }
327

```

```

328 private AVLNode<K,V> rotateLeft(AVLNode<K,V> node) {
329     AVLNode<K, V> newRoot = node.right;
330     AVLNode<K, V> leftChild = newRoot.left;
331
332     newRoot.left = node;
333     node.right = leftChild;
334
335     return newRoot;
336 }
337
338 private AVLNode<K,V> rotateRight(AVLNode<K,V> node) {
339     AVLNode<K, V> newRoot = node.left;
340     AVLNode<K, V> rightChild = newRoot.right;
341     newRoot.right = node;
342     node.left = rightChild;
343
344     return newRoot;
345 }
346
347 /**
348  * The iteration class for iterating values.
349  */
350
351 private class IterValues implements Iterator<V> {
352     ArrayList<V> list = new ArrayList<V>();
353     int index = -1;
354
355     public IterValues() {
356         inOrder(tree);
357     }
358
359     private void inOrder(AVLNode<K,V> node) {
360         if (node != null) {
361             inOrder(node.left);
362             list.add(node.value);
363             inOrder(node.right);
364         }
365     }
366
367     public boolean hasNext() {
368         return index < list.size() - 1;
369     }
370
371     public V next() {
372         if (!hasNext())
373             throw new NoSuchElementException();
374         index++;
375         return list.get(index);
376     }
377
378     public void remove() {
379         throw new UnsupportedOperationException();
380     }
381 }
382

```

```

383  /**
384   *   The iteration class for iterating values.
385   */
386
387  private class IterKeys implements Iterator<K> {
388      ArrayList<K> list = new ArrayList<K>();
389      int index = -1;
390
391      public IterKeys() {
392          inOrder(tree);
393      }
394
395      private void inOrder(AVLNode<K,V> node) {
396          if(node!=null) {
397              inOrder(node.left);
398              list.add(node.key);
399              inOrder(node.right);
400          }
401      }
402
403      public boolean hasNext() {
404          return index<list.size()-1;
405      }
406
407      public K next() {
408          if(!hasNext())
409              throw new NoSuchElementException();
410          index++;
411          return list.get(index);
412      }
413
414      public void remove() {
415          throw new UnsupportedOperationException();
416      }
417  }
418 }

```

Book

```

1 package library;
2
3 /**
4  *   The Book class for a book.
5  */
6
7 public class Book extends Media{
8     private String author, bookTitle;
9
10    public Book(String id, int year, String author, String bookTitle){
11        super(id, year);
12        this.author = author;
13        this.bookTitle = bookTitle;
14    }

```

```

15
16     /**
17      *      @return the author of the book.
18      */
19
20     public String getAuthor() {
21         return author;
22     }
23
24     /**
25      *      @return the title of the book.
26      */
27
28     public String getBookTitle() {
29         return bookTitle;
30     }
31
32     /**
33      *      @return the data about the book as a string.
34      */
35
36     public String toString(){
37         return bookTitle + ", " + author + ", " + super.getYear() + ", ID:
" + super.getId();
38     }
39 }

```

BorrowedItemsView

```

1 package views;
2
3 import java.awt.Dimension;
4 import java.awt.FlowLayout;
5 import java.awt.event.ActionEvent;
6 import java.awt.event.ActionListener;
7
8 import javax.swing.JButton;
9 import javax.swing.JLabel;
10 import javax.swing.JPanel;
11 import javax.swing.JScrollPane;
12 import javax.swing.JTextArea;
13 import javax.swing.JTextField;
14 import javax.swing.ScrollPaneConstants;
15
16 import controllers.ItemsController;
17
18 /**
19  *      BorrowedItemsView
20  *
21  *      The view for currently borrowed media objects.
22  *      From here borrowed objects can be listed and returned.
23  */
24

```



```

25 public class BorrowedItemsView extends JPanel{
26     private static final long serialVersionUID = 1L;
27     private ItemsController controller;
28     private JLabel      personnrLabel = new JLabel("Va`lkommen!"),
29                       returnLabel = new JLabel("A`terla`mna: ");
30     private JTextArea currentItems = new JTextArea();
31     private JTextField returnField = new JTextField();
32     private JButton returnItemButton = new JButton("A`terla`mna!"),
33             changeUserButton = new JButton("Byt anvndare");
34     private JScrollPane scrollPane = new JScrollPane(currentItems);
35
36     /**
37      *     controller will be passed since communication is necessary.
38      *
39      *     @param controller the controller controlling the
40      *     current view (ItemsController)
41      */
42
43     public BorrowedItemsView(ItemsController controller){
44         this.setLayout(new FlowLayout());
45         this.setPreferredSize(new Dimension(500, 400));
46         currentItems.setEditable(false);
47         this.controller = controller;
48
49         scrollPane.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLL
50         BAR_AS_NEEDED);
51
52         this.setDimensions();
53         this.setActionListeners();
54         this.addComponent();
55
56     private void setDimensions(){
57         personnrLabel.setPreferredSize(new Dimension(200, 25));
58         returnLabel.setPreferredSize(new Dimension(100, 25));
59         changeUserButton.setPreferredSize(new Dimension(130, 25));
60         scrollPane.setPreferredSize(new Dimension(500, 300));
61         currentItems.setPreferredSize(new Dimension(500, 300));
62         returnField.setPreferredSize(new Dimension(100, 25));
63     }
64
65     private void setActionListeners(){
66         AL buttonListener = new AL();
67
68         returnItemButton.addActionListener(buttonListener);
69         changeUserButton.addActionListener(buttonListener);
70     }
71
72     private class AL implements ActionListener{
73         @Override
74         public void actionPerformed(ActionEvent e) {
75             BorrowedItemsView outerClass = BorrowedItemsView.this;
76
77             if(e.getSource() == returnItemButton){

```

```

78         controller.returnItem(outerClass.returnField.getText());
79     }
80     else if(e.getSource() == changeUserButton){
81         //controller.changeUser(loginField.getText());
82         controller.logOut();
83     }
84 }
85 }
86
87 public void setWelcomeText(String text){
88     personnrLabel.setText(text);
89 }
90
91 private void addComponents(){
92     this.add(personnrLabel);
93     this.add(changeUserButton);
94     this.add(scrollPane);
95     this.add(returnLabel);
96     this.add(returnField);
97     this.add(returnItemButton);
98 }
99
100 public void setBorrowedItems(String items){
101     this.currentItems.setText(items);
102 }
103 }

```

CompareDateBorrowed

```

1 package library;
2
3 import java.util.Comparator;
4
5 /**
6  * Used for PriorityQueue in order to sort media-objects
7  * corresponding to time borrowed. Used for sorting borrowed objects
8  * first.
9  */
10
11 public class CompareDateBorrowed implements Comparator<Media>{
12
13     @Override
14     public int compare(Media o1, Media o2) {
15         if(o1.getDateBorrowed() != null && o2.getDateBorrowed() != null){
16             return o1.getDateBorrowed().compareTo(o2.getDateBorrowed());
17         }
18
19         return 0;
20     }
21 }
22 }

```

DVD

```
1 package library;
2
3 /**
4  *      DVD
5  *
6  *      Contains info to be associated with a DVD.
7  */
8
9 public class DVD extends Media{
10     private String name;
11     private String[] actors;
12
13     public DVD(String id, int year, String name, String[] actors) {
14         super(id, year);
15         this.name = name;
16         this.actors = actors;
17     }
18
19     public String getName() {
20         return name;
21     }
22
23     /**
24     * Return the actors as a copy of the array.
25     *
26     * @return all the actors as an array
27     */
28
29     public String[] getActors() {
30         String[] newArray = new String[actors.length];
31
32         for(int i = 0; i < newArray.length; i++){
33             newArray[i] = actors[i];
34         }
35
36         return newArray;
37     }
38
39     public String toString(){
40         return name + ", " + super.getYear() + ", ID: " + super.getId();
41     }
42 }
```

ItemsController

```
1 package controllers;
2
3 import java.util.Date;
4 import java.util.Iterator;
5
6 import javax.swing.JFrame;
```

```

7 import javax.swing.JOptionPane;
8
9 import collections.PriorityQueue;
10 import models.LibraryModel;
11 import models.Person;
12 import models.PersonModel;
13 import views.BorrowedItemsView;
14 import views.ItemsView;
15 import library.Book;
16 import library.CompareDateBorrowed;
17 import library.DVD;
18 import library.Media;
19
20 /**
21  * ItemsController
22  *
23  * ItemsController is responsible for the exchange of items.
24  * It controls two views in this case. It also extends the
25  * LibraryController responsible for loading in media files.
26  */
27
28 public class ItemsController{
29     private ItemsView itemsView = new ItemsView(this);
30     private BorrowedItemsView borrowedItemsView = new
BorrowedItemsView(this);
31     private Person currentlyLoggedIn;
32     private PersonModel pm;
33     private LibraryModel lm;
34     private JFrame itemsFrame = new JFrame(),
35         borrowedItemsFrame = new JFrame();
36
37     /**
38      * File path is sent upwards to the GeneralController.
39      * The old frame is reused in the borrowed-window, and the
40      * person currently logged in will be passed upwards.
41      *
42      * @param filePath filePath for the persons (Lantagare.txt)
43      * @param oldFrame the old frame used for the login.
44      * @param currentlyLoggedIn the person that is currently
45      * logged into the system.
46      */
47
48     public ItemsController(PersonModel personsModel, LibraryModel
libraryModel, Person currentlyLoggedIn) {
49         this.currentlyLoggedIn = currentlyLoggedIn;
50         this.pm = personsModel;
51         this.lm = libraryModel;
52
53         this.listItems();
54         this.openUpWindows();
55
56         borrowedItemsView.setWelcomeText("Välkommen, " +
currentlyLoggedIn.getName());
57     }
58

```

```

59      /**
60      *      Refresh the views with new items,
61      *      and decide which ones are borrowed
62      *      and not borrowed.
63      */
64
65      public void listItems() {
66          String      books = "",
67                     DVDs = "",
68                     borrowedObjects = "";
69
70          Iterator<Media> iter = lm.iterator();
71          PriorityQueue<Media> pq = new PriorityQueue<Media>(new
CompareDateBorrowed());
72
73          while(iter.hasNext()) {
74              Media media = iter.next();
75
76              if(media instanceof Book && media.getBorrowedBy() == null) {
77                  books += media.toString() + "\n";
78              }
79              else if(media instanceof DVD && media.getBorrowedBy() ==
null) {
80                  DVDs += media.toString() + "\n";
81              }
82              else if(media.getBorrowedBy().equals(currentlyLoggedIn)) {
83                  pq.enqueue(media);
84              }
85          }
86
87          while(pq.size() > 0) {
88              borrowedObjects += pq.dequeue().toString() + "\n";
89          }
90
91          borrowedItemsView.setBorrowedItems(borrowedObjects);
92          itemsView.setItems(books, DVDs);
93      }
94
95      private void openUpWindows() {
96
97          itemsFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
98
99          borrowedItemsFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
100
101          itemsFrame.add(itemsView);
102          borrowedItemsFrame.add(borrowedItemsView);
103
104          itemsFrame.pack();
105          itemsFrame.setVisible(true);
106          itemsFrame.setLocation(50, 50);
107          borrowedItemsFrame.pack();
108          borrowedItemsFrame.setVisible(true);
109      }
110

```

```

111     /**
112     * Borrow a certain media-object from the collection.
113     *
114     * @param id the id of the media object to be borrowed.
115     */
116
117     public void borrow(String id){
118         if(lm.contains(id)){
119             Media media = lm.get(id);
120
121             media.setBorrowedBy(currentlyLoggedIn);
122             media.setDateBorrowed(new Date());
123         }
124
125         this.listItems();
126     }
127
128     /**
129     * Return an item currently borrowed to the collection.
130     *
131     * @param id the id of the media-object to be returned.
132     */
133
134     public void returnItem(String id){
135         if(lm.contains(id)){
136             Media media = lm.get(id);
137             media.setBorrowedBy(null);
138             media.setDateBorrowed(null);
139         }
140
141         this.listItems();
142     }
143
144     /**
145     * Change the user that is currently logged in.
146     *
147     * @param personId personnr to be used for verification.
148     */
149
150     public void changeUser(String personId){
151         if(pm.contains(personId)){
152             int result = JOptionPane.showConfirmDialog(null, "Byte av
153 användare gick, vill du fortsätta?");
154
155             if(result == JOptionPane.OK_OPTION){
156                 itemsFrame.setVisible(false);
157                 borrowedItemsFrame.setVisible(false);
158                 Person loggedInPerson = pm.get(personId);
159                 new ItemsController(pm, lm, loggedInPerson);
160             }
161             else{
162                 JOptionPane.showMessageDialog(null, "Byte av användare gick
163 ej, försök igen.");
164             }
165         }
166     }

```

```

164     }
165
166     /**
167      *   Log out the user.
168      */
169
170     public void logOut() {
171         new LoginController(pm, lm);
172         this.itemsFrame.setVisible(false);
173         this.borrowedItemsFrame.setVisible(false);
174     }
175 }

```

ItemsView

```

1 package views;
2 import java.awt.BorderLayout;
3 import java.awt.GridLayout;
4 import java.awt.Dimension;
5 import java.awt.event.ActionEvent;
6 import java.awt.event.ActionListener;
7
8 import javax.swing.JButton;
9 import javax.swing.JLabel;
10 import javax.swing.JPanel;
11 import javax.swing.JTextArea;
12 import javax.swing.JTextField;
13
14 import controllers.ItemsController;
15
16 /**
17  * ItemsView
18  *
19  * Items from the collection will be listed here, from here
20  * you can also borrow a certain item by filling in the id of it.
21  */
22
23 public class ItemsView extends JPanel {
24     private static final long serialVersionUID = 1L;
25     private ItemsController controller;
26     private JPanel libraryPanel = new JPanel();
27     private JPanel borrowPanel = new JPanel();
28     private JLabel borrowLabel = new JLabel("Ange media id:");
29     private JTextField borrowField = new JTextField();
30     private JButton borrowButton = new JButton("Låna");
31     private JTextArea books = new JTextArea("Böcker"),
32         dvds = new JTextArea("DVD:er");
33
34     /**
35      * Controller is passed with since communication is
36      * necessary.
37      *
38      * @param controller a reference to the controller

```

```

39     * that instantiated this view.
40     */
41
42     public ItemsView(ItemsController controller){
43         this.setLayout(new BorderLayout());
44         this.controller = controller;
45         this.setDimensions();
46         this.setActionListeners();
47         this.addComponents();
48         libraryPanel.setLayout(new GridLayout(1, 2, 20, 20));
49         books.setEditable(false);
50         dvds.setEditable(false);
51     }
52
53     private void setDimensions(){
54         borrowLabel.setPreferredSize(new Dimension(120,25));
55         borrowField.setPreferredSize(new Dimension(150,25));
56         borrowButton.setPreferredSize(new Dimension(100,25));
57         books.setPreferredSize(new Dimension(500,500));
58         dvds.setPreferredSize(new Dimension(500, 500));
59     }
60
61     private void setActionListeners(){
62         borrowButton.addActionListener(new ActionListener(){
63             @Override
64             public void actionPerformed(ActionEvent e) {
65                 ItemsView ref = ItemsView.this;
66                 controller.borrow(ref.borrowField.getText());
67             }
68         });
69     }
70
71     private void addComponents(){
72         libraryPanel.add(books);
73         libraryPanel.add(dvds);
74         borrowPanel.add(borrowLabel);
75         borrowPanel.add(borrowField);
76         borrowPanel.add(borrowButton);
77
78         this.add(libraryPanel, BorderLayout.CENTER);
79         this.add(borrowPanel, BorderLayout.SOUTH);
80     }
81
82     /**
83     * Refresh the view with current items.
84     *
85     * @param books the books as strings with toString
86     * @param DVDs the DVDs passed as strings with toString
87     */
88
89     public void setItems(String books, String DVDs){
90         this.books.setText(books);
91         this.dvds.setText(DVDs);
92     }
93 }

```


Lantagare.txt

Notera att det är väldigt viktigt att få med den tomma raden vid slutet av filen!

891216-1111;Harald Svensson;040-471024

361025-2222;Rut Nilsson;040-1423142

730516-3333;Einar Andersson;040-2321112

931013-4444;Johanna Bok;040-2534423

900118-5555;Johan Nilsson;040-2454443

821223-6666;Anna Ek;040-452821

730421-7777;Carsten Panduro;040-2635222

700311-8888;Pernilla Johansson;040-2833652

681102-9999;Anders Boklund;040-2163542

LibraryModel

```
1 package models;
2
3 import java.io.BufferedReader;
4 import java.io.FileNotFoundException;
5 import java.io.FileReader;
6 import java.io.IOException;
7 import java.util.Iterator;
8
9 import collections.AVLTree;
10 import library.Book;
11 import library.DVD;
12 import library.Media;
13
14 /**
15  *   LibraryModel
16  *
17  *   A model for storing and accessing library-objects.
18  *   @author Philip Ekholm
19  *   @date 2017-04-01 12:13
20  */
21
22 public class LibraryModel {
```

```

23     private AVLTree<String, Media> mediaTree = new AVLTree<String,
Media>();
24
25     /**
26      *     Default constructor for libraryModel.
27      *
28      *     @param filePathMedia filePath to be entered for Media.txt
29      */
30
31     public LibraryModel(String filePathMedia){
32         try{
33             LibraryModel.readMediaEntries(mediaTree, filePathMedia);
34         }
35         catch(FileNotFoundException e1){
36             System.out.println("The file Media.txt could not be found at:
" + filePathMedia);
37             e1.printStackTrace();
38         }
39         catch(IOException e2){
40             e2.printStackTrace();
41         }
42     }
43
44     /**
45      *     Return an iterator instance containing media-objects
46      *
47      *     @return iterator instance with media-objects
48      */
49
50     public Iterator<Media> iterator(){
51         return mediaTree.iterator();
52     }
53
54     /**
55      *     Check if item can be found in the model.
56      *
57      *     @param key the id of the media.
58      *     @return true if found.
59      */
60
61     public boolean contains(String key){
62         return mediaTree.contains(key);
63     }
64
65     /**
66      *     Retrieve a media-item.
67      *
68      *     @param key the id of the media.
69      *     @return the media if found, otherwise null.
70      */
71
72     public Media get(String key){
73         return mediaTree.get(key);
74     }
75

```

```

76     /**
77     * Read all the Media-entries into one AVL-tree structure.
78     */
79
80     public static void readMediaEntries(AVLTree<String, Media> tree,
String filePath) throws FileNotFoundException, IOException{
81         try(BufferedReader br = new BufferedReader(new
FileReader(filePath))) {
82             String line = br.readLine();
83             while (line != null) {
84                 String[] details = line.split(";");
85
86                 if(details[0].equals("Dvd")){
87                     String[] actors = new String[details.length - 4];
88                     for(int i = 4; i < details.length; i++){
89                         actors[i - 4] = details[i];
90                     }
91                     DVD d = new DVD(details[1],
92 Integer.parseInt(details[3]), details[2], actors);
93                     tree.put(d.getId(), d);
94                 }
95                 else if(details[0].equals("Bok")){
96                     Book b = new Book(details[1],
Integer.parseInt(details[4]), details[2], details[3]);
97                     tree.put(b.getId(), b);
98                 }
99                 line = br.readLine();
100             }
101         }
102     }
103 }

```

LinkedList

```

1 package collections;
2
3 import java.util.Iterator;
4
5 /**
6  *   LinkedList E
7  *
8  *   A class that offers functionality to store data in a linked list
structure using chained object nodes (so called list nodes in this
9  *   implementation). Implements the interfaces List and Iterable,
where list is used for ensuring certain methods will be included. Iterable is
10  *   used for looping over a list with other means than index
arithmetics. Supports generics in order to work with homogeneous datatypes.
11  *
12  *   @author Rolf Axelsson
13  *   @author Philip Ekholm
14  */
15
16 public class LinkedList<E> implements List<E>, Iterable<E> {

```

```

17     //The starting point of the list, with a special list object to chain
other objects to.
18     private ListNode<E> list = null;
19
20     //Returns the node located at a certain index.
21     private ListNode<E> locate(int index) {
22         ListNode<E> node = list;
23         for( int i = 0; i < index; i++)
24             node = node.getNext();
25         return node;
26     }
27
28     /**
29     *     size will calculate the amount of elements currently in the
list, unlike the arrayList implementation this is not a variable
30     *     but must be counted manually by counting all chained elements.
31     *
32     *     @return the number of elements linked in the list
33     */
34
35     public int size() {
36         int n = 0;
37         ListNode<E> node = list;
38         while( node != null ) {
39             node = node.getNext();
40             n++;
41         }
42         return n;
43     }
44
45     /**
46     *     get the data of a certain object at a certain index. Method
will first check if index is valid, otherwise an exception will be thrown.
47     *
48     *     @throws IndexOutOfBoundsException if invalid index is passed
49     *     @return the data at a certain object in the list.
50     */
51
52     public E get( int index ) {
53         if( ( index < 0 ) || ( index > size() ) )
54             throw new IndexOutOfBoundsException( "size=" + size() + ",
index=" + index );
55
56         ListNode<E> node = locate( index );
57         return node.getData();
58     }
59
60     /**
61     *     Set the data of a certain object at given index. Method will
first check if index is valid, otherwise an exception will be thrown.
62     *
63     *     @param index of the object to be manipulated
64     *     @param data to be passed to object
65     *     @throws IndexOutOfBoundsException if invalid index is passed
66     *     @return the old data which was replaced

```

```

67     */
68
69     public E set( int index, E data ) {
70         if(index < 0 || index > this.size()){
71             throw new IndexOutOfBoundsException();
72         }
73         else{
74             E oldNode = this.get(index);
75             this.remove(index);
76             this.add(index, data);
77
78             return oldNode;
79         }
80     }
81
82     /**
83      * Add new data to the list, will be added to the end of the list
84      * if no index has been specified.
85      * @param data to be added to the list
86      */
87     public void add(E data) {
88         this.addLast(data);
89     }
90
91     /**
92      * Add new data to the "top" of the list, will be added to index
93      * 0.
94      * @param data the data to be added to the list
95      */
96     public void addFirst( E data ) {
97         this.add(0, data);
98     }
99
100    /**
101     * Same as add(E)
102     */
103
104    public void addLast( E data ) {
105        this.add(this.size(), data);
106    }
107
108    /**
109     * Adds the data and creates a new object node at given index.
110     * Method will first check if index is valid, otherwise an exception will be
111     * thrown.
112     * @param index the target position to add the element to
113     * @param data the data to be added to the list
114     * @throws IndexOutOfBoundsException if invalid index is passed
115     */
116    public void add( int index, E data ) {
117        if(index < 0 || index > this.size()){

```

```

118         throw new IndexOutOfBoundsException();
119     }
120     else if(index == 0){
121         list = new ListNode<E>(data, list);
122     }
123     else{
124         ListNode<E> n0 = locate(index - 1);
125         ListNode<E> n1 = new ListNode<E>(data, n0.getNext());
126         n0.setNext(n1);
127     }
128 }
129
130 /**
131  * Remove the very first element of the list, will call the
132  * remove(int) method with index = 0
133  * @return the data that was removed from the list
134  */
135
136 public E removeFirst() {
137     return this.remove(0);
138 }
139
140 /**
141  * Remove the very last element of the list, will call the
142  * remove(int) method with index = size() - 1
143  * @return the data that was removed from the list
144  */
145
146 public E removeLast() {
147     return this.remove(this.size() - 1);
148 }
149
150 /**
151  * Remove the element of the given index and return the data that
152  * was contained in the element.
153  * Method will first check if index is valid, otherwise an
154  * exception will be thrown.
155  * @param index the target position of the element to be removed
156  * @throws IndexOutOfBoundsException if invalid index is passed
157  * @return the old data stored at index
158  */
159
160 public E remove( int index ) {
161     if( ( index < 0 ) || ( index >= size() ) )
162         throw new IndexOutOfBoundsException( "size=" + size() + ",
163 index=" + index );
164
165     E res;
166     if( index == 0 ) {
167         res = list.getData();
168         list = setNull(list);
169     }
170     else {
171         list = list.getNext();
172     }
173 }

```

```

168         } else {
169             ListNode<E> node = locate( index - 1 );
170             res = node.getNext().getData();
171             node.setNext(setNull(node.getNext()));
172             // node.setNext( node.getNext().getNext() );
173         }
174         return res;
175     }
176
177     private ListNode<E> setNull(ListNode<E> toNull) {
178         ListNode<E> res = toNull.getNext();
179         toNull.setData(null);
180         toNull.setNext(null);
181         return res;
182     }
183
184     /**
185      * Removes all elements in the list by looping through every
186      * element and remove them
187      */
188     public void clear() {
189         while(this.size() > 0){
190             this.removeLast();
191         }
192     }
193
194     /**
195      * Returns the index of given data, will return -1 if not found.
196      *
197      * @param data the data being searched for
198      * @return result from method indexOf(int, E)
199      */
200
201     public int indexOf(E data) {
202         return indexOf(0, data);
203     }
204
205     /**
206      * Will return the index of a given object that has the same
207      * reference
208      * as passed object, give startIndex to improve search speed, will
209      * return -1 if not found.
210      *
211      * @param startIndex to start searching at to improve search
212      * performance
213      * @param data the object who's reference will be compared
214      * @return the index of object, -1 if not found
215      */
216
217     public int indexOf(int startIndex, E data) {
218         for(int i = startIndex; i < this.size(); i++){
219             if(data.equals(this.get(i))){
220                 return i;
221             }
222         }
223     }

```

```

219         }
220
221         return -1;
222     }
223
224     /**
225      * Returns an iterator object in order to loop the list with
226      * other means than index arithmetics. The method has been
227      * simplified using the iterator of the arraylist instead of having to develop a
228      * new algorithm to get all the elements in the linked list.
229      *
230      * @return a new iterator object
231      */
232
233     public Iterator<E> iterator() {
234         ArrayList<E> iterList = new ArrayList<E>(this.size());
235
236         for(int i = 0; i < this.size(); i++){
237             iterList.add(this.get(i));
238         }
239
240         return iterList.iterator();
241     }
242
243     /**
244      * Will return the toString from the ListNode class, which uses
245      * StringBuilder to manipulate strings.
246      * If the list is dereferenced it will return empty parenthesis
247      * [].
248      */
249
250     public String toString() {
251         if( list != null )
252             return list.toString();
253         else
254             return "[]";
255     }
256
257     /**
258      * Not implemented in this solution, can be ignored.
259      */
260
261     private class Iter implements Iterator<E> {
262
263         public boolean hasNext() {
264             return false;
265         }
266
267         public E next() {
268             return null;
269         }
270
271         public void remove() {

```



```

270         throw new UnsupportedOperationException();
271     }
272 }
273 }

```

LinkedList

```

1 package collections;
2
3 /**
4  *   LinkedList
5  *
6  *   A class that implements the interface Queue, which defined how
7  *   a queue datastructure should communicate with other objects. The
8  *   LinkedList is an implementation of a Queue using linking to other
9  *   objects (nodes) which can be added/removed. The LinkedList works by
10  *   other Queue implementations (specifically through the FiFo-
    structure).
11  *
12  *   @author Philip Ekholm
13  *   @created 2017-03-04
14  *
15  */
16
17 public class LinkedList<E> implements Queue<E>{
18     private LinkedList<E> elements;
19     private int size;
20
21     /**
22      *   Constructor without arguments, which will instantiate
23      *   a new LinkedList object. This implementation uses
24      *   a LinkedList to store nodes.
25      */
26
27     public LinkedList() {
28         elements = new LinkedList<E>();
29         size = 0;
30     }
31
32     /**
33      *   Enqueue (insert) new elements (data-objects) to the queue
34      *   by adding them to the end of the list.
35      *
36      *   @param elem data-object to insert into the queue
37      */
38
39     public void enqueue( E elem ) {
40         elements.addLast(elem);
41         size++;
42     }
43
44     /**
45      *   Dequeue (remove) the element (data-object) currently first up

```

```

46      *   ("first in line") on the list and return it wherever
47      *   the method was called.
48      *
49      *   If an attempt is made to dequeue an empty queue QueueException
50      *   will be thrown.
51      *
52      *   @return the element currently first in the queue
53      *   @throws QueueException if the queue is empty while attempting to
dequeue
54      */
55
56      public E dequeue() throws QueueException{
57          if(size==0) {
58              throw new QueueException("dequeue: Queue is empty");
59          }
60          E value = elements.removeFirst();
61          size--;
62          return value;
63      }
64
65      /**
66      *   Peek (get) the element (data-object) currently first up
67      *   ("first in line") on the list. If an attempt is made to peek at
68      *   an empty queue QueueException will be thrown.
69      *
70      *   @return the element currently first in the queue.
71      *   @throws QueueException if the queue is empty while attempting to
peek.
72      */
73
74      public E peek() throws QueueException{
75          if( size==0 ) {
76              throw new QueueException("peek: Queue is empty");
77          }
78          return elements.get(0);
79      }
80
81      /**
82      *   Check whether the queue is empty or not.
83      *
84      *   @return true if the queue is empty, otherwise false.
85      */
86
87      public boolean isEmpty() {
88          return (size<=0);
89      }
90
91      /**
92      *   Return the size (length) of the list.
93      *
94      *   @return the current size of the list.
95      */
96
97      public int size() {
98          return size;

```

```

99     }
100
101     /**
102     *   The toString-method will return a string-object containing a
print
103     *   of objects containing properties (among else). The current
104     *   implementation will use the toString of the LinkedList instead of
105     *   of a new implementation.
106     *
107     *   @return the toString return value of LinkedList
108     */
109
110     public String toString(){
111         return elements.toString();
112     }
113
114 }

```

List

```

1  package collections;
2  import java.util.Iterator;
3
4  public interface List<E> {
5
6      /**
7       * Appends the specified element to the end of this list
8       * @param element element to be appended to this list
9       */
10     public void add(E element);
11
12     /**
13     * Inserts the specified element at the specified position in this
list.
14     * Shifts the element currently at that position (if any) and any
15     * subsequent elements to the right (adds one to their indices).
16     * @param index index at which the specified element is to be
inserted
17     * @param element element to be inserted
18     */
19     public void add(int index, E element);
20
21     /**
22     * Inserts the specified element at the beginning of this list
23     * @param element element to be inserted at the beginning of this
list
24     */
25     public void addFirst(E element);
26
27     /**
28     * Appends the specified element to the end of this list
29     * @param element element to be appended at the end of this list
30     */

```

```

31 public void addLast(E element);
32
33 /**
34  * Removes the element at the specified position in this list. Shifts
35  * any subsequent elements to the left (subtracts one from their
36  * indices). Returns the element that was removed from the list.
37  * @param index the index of the element to be removed
38  * @return the element previously at the specified position
39  */
40 public E remove(int index);
41
42 /**
43  * Removes and returns the first element from this list.
44  * @return the first element from this list
45  */
46 public E removeFirst();
47
48 /**
49  * Removes and returns the last element from this list.
50  * @return the last element from this list
51  */
52 public E removeLast();
53
54 /**
55  * Removes all of the elements from this list. The list will be
56  * empty after this call returns.
57  */
58 public void clear();
59
60 /**
61  * Returns the element at the specified position in this list.
62  * @param index index of the element to return
63  * @return the element at the specified position in this list
64  */
65 public E get(int index);
66
67 /**
68  * Replaces the element at the specified position in this list with
the
69  * specified element
70  * @param index index of the element to replace
71  * @param element element to be stored at the specified position
72  * @return the element previously at the specified position
73  */
74 public E set(int index, E element);
75
76 /**
77  * Returns the index of the first occurrence of the specified element
78  * in this list, or -1 if this list does not contain the element.
79  * @param element element to search for
80  * @return the index of the first occurrence of the specified element
in
81  * this list, or -1 if this list does not contain the element
82  */
83 public int indexOf(E element);

```

```

84
85  /**
86   * Returns the index of the first occurrence of the specified element
87   * in this list, or -1 if this list does not contain the element. The
88   * search begins at startIndex in the list.
89   * @param startIndex the search starts at position startIndex in the
list
90   * @param element element to search for
91   * @return the index of the first occurrence of the specified element
in
92   *         this list, or -1 if this list does not contain the element
93   */
94   public int indexOf(int startIndex, E element);
95
96   /**
97   * Returns an iterator over the elements in this list in proper
sequence.
98   * @return an iterator over the elements in this list in proper
sequence
99   */
100   public Iterator<E> iterator();
101
102   /**
103   * Returns the number of elements in this list.
104   * @return the number of elements in this list
105   */
106   public int size();
107 }

```

ListNode

```

1 package collections;
2
3 /**
4  *   ListNode E
5  *
6  *   ListNode is what the linked list is built up on. It is able to
store data of homogeneous kind,
7  *   as well as the next listnode in order to continue the chain. It has
different getters and setters to these as well
8  *   as a toString method in order to be printed out as a string.
9  */
10
11 public class ListNode<E> {
12     private E data;
13     private ListNode<E> next;
14
15     /**
16      *   Base constructor, takes data as well as the next object in the
chain.
17      *
18      *   @param data the data to be stored
19      *   @param next the next object to be linked

```

```

20     */
21
22     public ListNode( E data, ListNode<E> next ) {
23         this.data = data;
24         this.next = next;
25     }
26
27     /**
28      *     Getter for extracting data.
29      *
30      *     @return the data from this object
31      */
32
33     public E getData() {
34         return this.data;
35     }
36
37     /**
38      *     Setter for setting new data.
39      *
40      *     @param data data to replace the current data with
41      */
42
43     public void setData( E data ) {
44         this.data = data;
45     }
46
47     /**
48      *     getter for next object chained. This method is applicable for
49      *     method-chaining, since
50      *     this is recursive.
51      *
52      *     @return the next object in line
53      */
54
55     public ListNode<E> getNext() {
56         return this.next;
57     }
58
59     /**
60      *     Setter for setting the next object in the line.
61      *
62      *     @param next the next object to be added after the current one
63      */
64
65     public void setNext( ListNode<E> next ) {
66         this.next = next;
67     }
68
69     /**
70      *     Overrides the toString from superclass and returns the toString
71      *     from data.
72      *     This is done via StringBuilder class.
73      *
74      *     @see StringBuilder

```

```

73      *      @return the list as a string.
74      */
75
76      public String toString() {
77          StringBuilder str = new StringBuilder("[ ");
78          str.append(data.toString());
79          ListNode<E> node = next;
80          while( node!=null ) {
81              str.append( "; ");
82              str.append( node.getData().toString() );
83              node = node.getNext();
84          }
85          str.append( " ]");
86          return str.toString();
87      }
88  }

```

LoginController

```

1  package controllers;
2
3  import javax.swing.JFrame;
4  import javax.swing.JOptionPane;
5
6  import models.LibraryModel;
7  import models.Person;
8  import models.PersonModel;
9  import views.LoginView;
10
11  /**
12   * LoginController
13   *
14   * The LoginController is responsible for starting up the LoginView,
15   * as well as validating the number used for logging in.
16   */
17
18  public class LoginController{
19      private LoginView view = new LoginView(this);
20      private PersonModel personModel;
21      private LibraryModel libraryModel;
22      private Person loggedInPerson;
23      private JFrame frame;
24
25      /**
26       * Default constructor will take arguments for loading
27       * in files required for the structure to work. These will
28       * be sent to the super class for processing.
29       *
30       * @param filePathPersons the String containing the file
31       * path for loading lantagare.txt
32       */
33
34      public LoginController(String filePathPersons, String filePathMedia){

```

```

35     personModel = new PersonModel(filePathPersons);
36     libraryModel = new LibraryModel(filePathMedia);
37
38     this.setupJFrame();
39 }
40
41 public LoginController(PersonModel pm, LibraryModel lm){
42     personModel = pm;
43     this.libraryModel = lm;
44
45     this.setupJFrame();
46 }
47
48 private void setupJFrame(){
49     this.frame= new JFrame();
50     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
51     frame.add(view);
52     frame.setLocation(50, 50);
53
54     frame.pack();
55     frame.setVisible(true);
56 }
57
58 /**
59  * Validates the number passed in, will allow access
60  * if number can be found as key. A confirmDialog
61  * will then be opened to confirm the login.
62  *
63  * If accepted, a LibraryController will be instantiated.
64  *
65  * @param number number entered in the loginView
66  */
67
68 public void validate(String number){
69     if(personModel.contains(number)){
70         int result = JOptionPane.showConfirmDialog(null, "Inloggning
gick, vill du gå vidare?");
71         if(result == JOptionPane.OK_OPTION){
72             view.setVisible(false);
73             loggedInPerson = personModel.get(number);
74             new ItemsController(personModel, libraryModel,
loggedInPerson);
75             frame.setVisible(false);
76         }
77     }
78     else{
79         JOptionPane.showMessageDialog(null, "Inloggning gick ej,
försök igen.");
80     }
81 }
82 }

```


LoginView

```
1 package views;
2
3 import java.awt.Color;
4 import java.awt.Dimension;
5 import java.awt.event.ActionEvent;
6 import java.awt.event.ActionListener;
7
8 import javax.swing.JButton;
9 import javax.swing.JLabel;
10 import javax.swing.JPanel;
11 import javax.swing.JTextField;
12
13 import controllers.LoginController;
14
15 /**
16  * The login view for entering the personnr.
17  */
18 public class LoginView extends JPanel{
19     private static final long serialVersionUID = 1L;
20     private JLabel label1 = new JLabel("Mata in personnr (10 siffor)
21 för att gå vidare: yymmdd-xxxx");
22     private JLabel label2 = new JLabel("Personnr: ");
23     private JTextField personField = new JTextField();
24     private JButton sendBtn = new JButton("Gå vidare");
25
26     /**
27      * Pass reference to the controller since communication
28      * back will be necessary.
29      *
30      * @param controller the controller that instantiated
31      * this view.
32      */
33     public LoginView(LoginController controller){
34         this.setPreferredSize(new Dimension(400, 120));
35         this.setBackground(Color.WHITE);
36         this.setLayout(null);
37
38         sendBtn.addActionListener(new ActionListener() {
39             @Override
40             public void actionPerformed(ActionEvent e) {
41                 controller.validate(personField.getText());
42             }
43         });
44
45         this.setDimensions();
46         this.addComponents();
47     }
48
49     /**
50      * Set the dimensions of objects.
51      */
52 }
```

```

53     private void setDimensions() {
54         label1.setBounds(10, 10, 400, 25);
55         label2.setBounds(10, 50, 100, 25);
56         personField.setBounds(80, 48, 160, 30);
57         sendBtn.setBounds(250, 48, 90, 30);
58     }
59
60     private void addComponents() {
61         this.add(label1);
62         this.add(label2);
63         this.add(personField);
64         this.add(sendBtn);
65     }
66 }

```

Main

```

1 package common;
2 import controllers.LoginController;
3
4 /**
5  * Start the program from here.
6  */
7
8 public class Main {
9     public static void main(String[] args) {
10         new LoginController("files/Lantagare.txt", "files/Media.txt");
11     }
12 }

```

Media

```

1 package library;
2 import java.util.Date;
3
4 import models.Person;
5
6 /**
7  * Media
8  *
9  * A generalized abstract object which is inherited by Book and DVD
10  */
11
12 public abstract class Media {
13     private String id;
14     private Person borrowedBy;
15     private Date dateBorrowed;
16     private int year;
17
18     public Media( String id, int year) {
19         this.id = id;
20         this.borrowedBy = null;

```

```

21         this.year = year;
22     }
23
24     public String getId() {
25         return id;
26     }
27
28     public int getYear() {
29         return year;
30     }
31
32     public Person getBorrowedBy() {
33         return borrowedBy;
34     }
35
36     public Date getDateBorrowed() {
37         return dateBorrowed;
38     }
39
40     public void setDateBorrowed(Date dateBorrowed) {
41         this.dateBorrowed = dateBorrowed;
42     }
43
44     public void setBorrowedBy(Person borrowedBy) {
45         this.borrowedBy = borrowedBy;
46     }
47
48     /**
49      * Equals is overridden and will return true if the id of the media
50      * matches the one passed.
51      *
52      * @param obj the object to be matched to see if it's equal.
53      * @return true if id matches, otherwise false.
54      */
55
56     @Override
57
58     public boolean equals( Object obj ) {
59         if(obj instanceof Media) {
60             Media media = (Media)obj;
61             return id.equals( media.getId() );
62         }
63         return false;
64     }
65 }

```

Media.txt

Notera att det är väldigt viktigt att få med den tomma raden vid slutet av filen!

Bok;427769;Deitel;Java how to program;2005

Dvd;635492;Nile City 105,6;1994;Robert Gustavsson;Johan Rheborg;Henrik

PHILIP EKHOLM, ANAS ABU AL-SOUD, HENRIK FREDLUND, VIKTOR TORKI

Schyffert

Bok;874591;Guillou;Vägen till Jerusalem;1999

Bok;456899;Lindblad, Westby;Ö-luffa i Grekland;2003

Bok;123938;Nilsson;Bock i örtagård;1933

Bok;775534;Thompson;Historiens matematik;1991

Dvd;722293;V för Vendetta;2006;Natalie Portman;Hugo Weaving;Stephen Rea;John Hurt

Dvd;237729;Time Bandits;1982;John Cleese;Sean Connery

Dvd;768841;Sin City;2005;Bruce Willis;Mickey Rourke;Josh Hartnett;Jessica Alba;Elijah Wood

Dvd;599223;I manegen med Glenn Killing;1992;Robert Gustafsson;Johan Rheborg;Henrik Schyffert;Jonas Inde

Dvd;398567;Hair;1979;John Savage;Treat Williams;Beverly D'Angelo;Annie Golden;Dorsey Wright;Don Dacus;Cheryl Barnes

Bok;899233;Alfredsson;Åtta glas;2004

Bok;993782;Fredriksson;Ondskans leende;2006

Dvd;366665;Finding Neverland;2004;Johnny Depp;Kate Winslet;Julie Christie;Dustin Hoffman

Dvd;283228;Donnie Darko;2002;Jake Gyllenhaal;Drew Barrymore;Jena Malone;Patrick Swayze;Noah Wyle

Dvd;834762;The office;2002;Ricky Gervais;Martin Freeman;Meckenzie Crook;Lucy Davis

Dvd;211185;Crash;2004;Sandra Bullock;Don Cheadle;Matt Dillon;Jennifer Esposito;Brendan Fraser; Terrance Howard

Bok;463390;Mankell;Brandvägg;1998
Bok;277877;Grisham;Agenten;2005
Bok;812621;Chevalier;Flicka med pärlörhänge;1999
Bok;712998;Smedberg;Försvinnanden;2001
Bok;399898;Lindell;Drömfångaren;1999
Bok;528739;Coelho;Birgitta och Katarina;2006
Bok;382231;Johansson;Nancy;2001
Dvd;498582;The boondock saints;1999;Willem Dafoe;Sean Patrick
Flanery;Norman Reedus;Billy Connolly
Bok;729384;Tönnis;Högre matematik för poeter och andra oskulder;1982
Bok;553245;Gustafsson;Tennisspelarna;1977

Person

```
1 package models;
2
3 /**
4  *      Person
5  *
6  *      Person is a class for storing common information about
7  *      people (a.k.a. Lantagare).
8  */
9
10 public class Person {
11     private String name,
12     personnr,
13     phoneNumber;
14
15     public Person(String personnr, String name, String phoneNumber){
16         this.name = name;
17         this.personnr = personnr;
18         this.phoneNumber = phoneNumber;
19     }
20
21     public String getName() {
22         return name;
23     }
24
25     public String getPersonnr() {
```

```

26         return personnr;
27     }
28
29     public String getPhoneNumber() {
30         return phoneNumber;
31     }
32
33     @Override
34     public String toString(){
35         return this.getName() + ", " + this.getPersonnr() + ", " +
this.getPhoneNumber();
36     }
37
38     //Persons equal if personnr matches
39     @Override
40     public boolean equals(Object obj){
41         if(obj instanceof Person){
42             Person p = (Person)obj;
43
44             return p.personnr.equals(this.personnr);
45         }
46
47         return false;
48     }
49 }

```

PersonModel

```

1 package models;
2
3 import java.io.BufferedReader;
4 import java.io.FileNotFoundException;
5 import java.io.FileReader;
6 import java.io.IOException;
7
8 import collections.AVLTree;
9
10 /**
11  *   PersonsModel
12  *
13  *   A model for storing and accessing persons-objects.
14  *   @author Philip Ekholm
15  *   @date 2017-04-01 12:13
16  */
17
18 public class PersonModel {
19     private AVLTree<String, Person> persons = new AVLTree<String,
Person>();
20
21     public PersonModel(String filePath){
22         try{
23             PersonModel.readPersons(persons, filePath);
24         }

```

```

25         catch(FileNotFoundException e1){
26             e1.printStackTrace();
27         }
28         catch(IOException e2){
29             e2.printStackTrace();
30         }
31     }
32
33     /**
34      *   Read in all persons from the lantagare file using a FileReader.
35      *
36      *   @param tree the AVL-structure to fill with found persons.
37      *   @param filePath the relative directory path to the file.
38      */
39
40     public static void readPersons(AVLTree<String, Person> tree, String
filePath)
41         throws FileNotFoundException, IOException{
42         try(BufferedReader br = new BufferedReader(new
FileReader(filePath))){
43             String line = br.readLine();
44             while (line != null) {
45                 String[] details = line.split(";");
46                 Person p = new Person(details[0], details[1], details[2]);
47                 tree.put(p.getPersonnr(), p);
48                 line = br.readLine();
49             }
50         }
51     }
52
53     public boolean contains(String key){
54         return persons.contains(key);
55     }
56
57     public Person get(String key){
58         return persons.get(key);
59     }
60 }

```

PriorityQueue

```

1 package collections;
2
3 import java.util.Comparator;
4
5 /**
6  *   PriorityQueue
7  *
8  *   PriorityQueue is another implementation of the interface
9  *   Queue. The priorityqueue implements the FiFo-structure of the Queue
10  *   datastructure, but also differs on objects by using priority.
11  */

```

```

12 *   Objects to be prioritized will be compared to other objects through
a
13 *   class that implements comparator which can either be passed into
constructor.
14 *   If no class that implements Comparator has been passed the objects
are
15 *   assumed to implement the Comparable interface.
16 *
17 *   @author Philip Ekholm
18 *   @crated 2017-03-04
19 */
20
21 public class PriorityQueue<E> implements Queue<E>{
22     private LinkedList<E> elements = new LinkedList<E>();
23     private Comparator<E> comp;
24
25     /**
26      *   Constructor without arguments, which will instantiate
27      *   a new PriorityQueue object. This implementation will
28      *   uses a LinkedList to store nodes.
29      *
30      *   Classes instantiating a PriorityQueue without arguments are
31      *   assumed to only store objects that implements Comparable.
32      */
33
34     public PriorityQueue() {
35         this.comp = new Comp();
36     }
37
38     /**
39      *   Constructor that takes a class that implements Comparator to
40      *   compare objects by.
41      */
42
43     public PriorityQueue(Comparator<E> comp) {
44         this.comp = comp;
45     }
46
47     /**
48      *   Enqueue (insert) new elements (data-objects) to the queue
49      *   by adding them to the end of the list. If the object passed
50      *   is prioritized by Comparator it will be moved further into line
51      *   after another object with the same priority.
52      *
53      *   @param data data-object to insert into the queue
54      */
55
56     @Override
57     public void enqueue( E data ) {
58         int    index = 0,
59             size = size();
60
61         while (index<size && comp.compare(elements.get(index), data) <=
0) {
62             index++;

```



```

63         }
64
65         elements.add(index, data);
66     }
67
68     /**
69     *   Dequeue (remove) the element (data-object) currently first up
70     *   ("first in line") on the list and return it wherever
71     *   the method was called.
72     *
73     *   If an attempt is made to dequeue an empty queue QueueException
74     *   will be thrown.
75     *
76     *   @return the element currently first in the queue.
77     *   @throws QueueException if the queue is empty while attempting to
dequeue.
78     */
79
80     @Override
81     public E dequeue() throws QueueException{
82         if(isEmpty()) {
83             throw new QueueException("dequeue: Queue is empty");
84         }
85
86         return elements.removeFirst();
87     }
88
89     /**
90     *   Peek (get) the element (data-object) currently first up
91     *   ("first in line") on the list. If an attempt is made to peek at
92     *   an empty queue QueueException will be thrown.
93     *
94     *   @return the element currently first in the queue.
95     *   @throws QueueException if the queue is empty while attempting to
peek.
96     */
97
98     @Override
99     public E peek() throws QueueException{
100         if( size()==0 ) {
101             throw new QueueException("peek: Queue is empty");
102         }
103         return elements.get(0);
104     }
105
106     /**
107     *   Check whether the queue is empty or not.
108     *
109     *   @return true if the queue is empty, otherwise false.
110     */
111
112     @Override
113     public boolean isEmpty() {
114         return (size()<=0);
115     }

```

```

116
117  /**
118   *   Return the size (length) of the list.
119   *
120   *   @return the current size of the list.
121   */
122
123  @Override
124  public int size() {
125      return elements.size();
126  }
127
128  /**
129   *   The toString-method will return a string-object containing a
print
130   *   of objects containing properties (among else). The current
131   *   implementation will use the toString of the LinkedList instead of
132   *   of a new implementation.
133   *
134   *   @return the toString return value of LinkedList
135   */
136
137  @Override
138  public String toString(){
139      return elements.toString();
140  }
141
142  /**
143   *   Comp is the default class that implements Comparator
144   *   if no other class has been passed to constructor.
145   *   Misuse of the class will result in a ClassCastException
146   */
147
148  private class Comp implements Comparator<E>{
149      @Override
150      public int compare(E obj1, E obj2) {
151          Comparable<E> com1 = (Comparable<E>)obj1;
152
153          return com1.compareTo(obj2);
154      }
155  }
156
157 }

```

Queue

```

1 package collections;
2
3 public interface Queue<E> {
4
5     /**
6      * Inserts the specified element into this queue.
7      * @param data the object to add

```

```

8      * @throws QueueException if the element cannot be added at this
9      *      time due to capacity restrictions
10     */
11     public void enqueue(E data);
12
13     /**
14      * Retrieves and removes the head of this queue.
15      * @return the head of this queue
16      * @throws QueueException if this queue is empty
17      */
18     public E dequeue();
19
20     /**
21      * Retrieves, but does not remove, the head of this queue.
22      * @return the head of this queue
23      * @throws QueueException if this queue is empty
24      */
25     public E peek();
26
27
28     /**
29      * Returns true if this stack contains no elements.
30      * @return true if this stack contains no elements
31      */
32     public boolean isEmpty();
33
34     /**
35      * Returns the number of elements in this stack.
36      * @return the number of elements in this stack
37      */
38     public int size();
39 }

```

QueueException

```

1 package collections;
2
3 /**
4  * QueueException
5  *
6  * An exception written for handling different runtime-exceptions
7  * that can occur in Queue implementations.
8  */
9
10 public class QueueException extends RuntimeException {
11     public QueueException() {}
12     public QueueException( String message ) {
13         super( message );
14     }
15 }

```

SearchTree

```
1 package collections;
2 import java.util.Iterator;
3 import collections.List;
4
5 public interface SearchTree<K,V> {
6     public void put(K key, V value);
7     public V remove(K key);
8     public V get(K key);
9     public boolean contains(K key);
10    public int height();
11    public Iterator<V> iterator();
12    public int size();
13    public List<K> keys();
14    public List<V> values();
15    public V first();
16    public V last();
17 }
```