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
LinkedQueue........ 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.

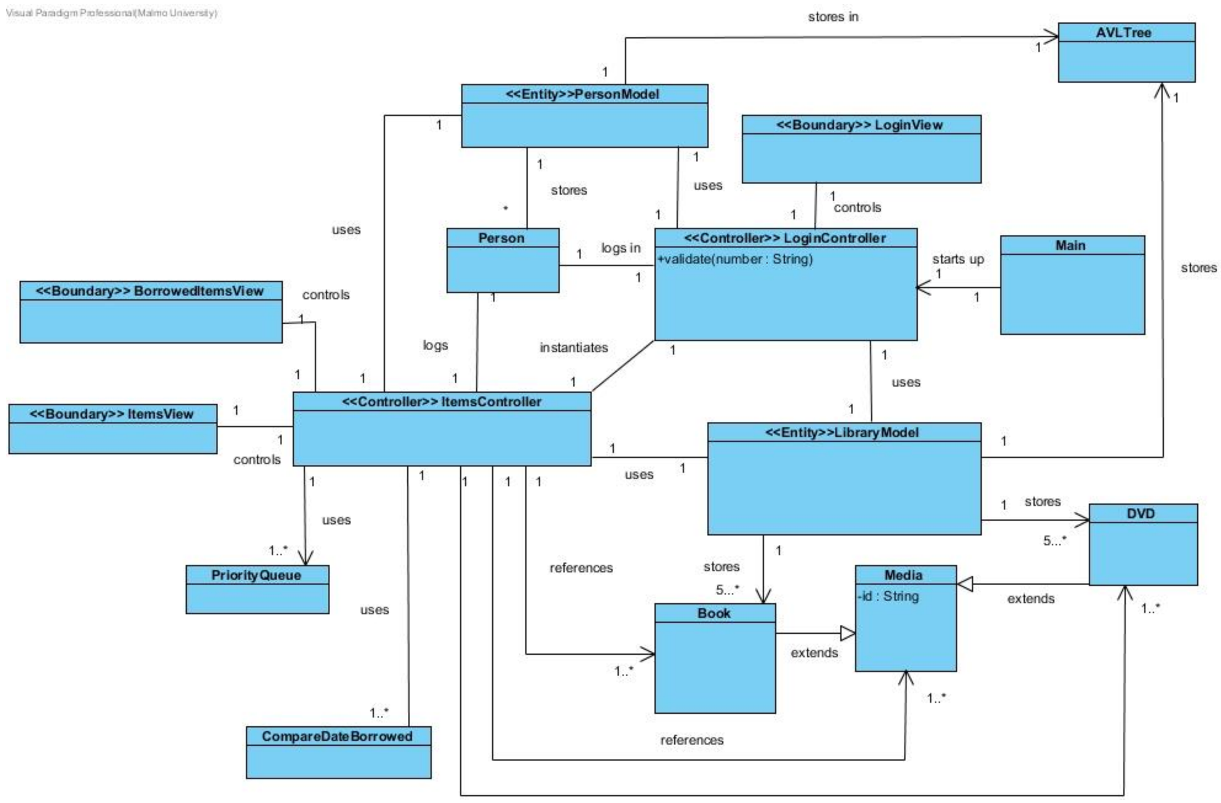
För lagringsstrukturer används allra mest AVL-träd då det tillåter snabb sökning såväl som sortering. Ett AVL-träd blir inte heller (till skillnad från standard BST) degenererat automatiskt utan hela tiden ombalanserar sig så fort nya element läggs till. Detta ger en viss prestandaförlust men vi finner 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 hashning-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

```
package collections;

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

/**
 *     ArrayList E
 *
 *      A class that extends the functionality of list using a simple
Array-data structure, implements the
```

```java
 *     interface List which states certain methods to be implemented. Supports generics to work with any homogeneous datatype.
 *
 *     @author Rolf Axelsson
 *     @author Philip Ekholm
 */

public class ArrayList<E> implements List<E> {
    /**
     *    One array elements to keep track of actual data
     *    size used to determine current length of list
     */

    private E[] elements;
    private int size;

    /**
     *    grow will double the size of the current list if
     *    number of elements exceed the number of cells available by
     *    creating a new array twice the size
     */

    private void grow() {
        E[] temp = (E[])new Object[2 * elements.length];
        for(int i = 0; i < this.elements.length; i++){
            temp[i] = this.elements[i];
        }

        this.elements = temp;
    }

    /**
     *    Pass default size of 20 to base constructor if no argument passed.
     */

    public ArrayList() {
        this(20);
    }

    /**
     *    Base constructor for this class, creates an initial capacity of
     *    at least 1, will be extended later if too small.
     *
     *     @param initialCapacity size of the initial list
     */

    public ArrayList(int initialCapacity) {
        initialCapacity = Math.max(1, initialCapacity);
        elements = (E[])new Object[initialCapacity];
    }

    /**
     *    Adds a new element to the list using target index.
     *
     *    @param index the target position to add the element to
     *    @param element the element to be added to the list
     *
     *    @throws IndexOutOfBoundsException if invalid index is passed
     */

    public void add(int index, E element) {
        if(index<0 || index>size)
            throw new IndexOutOfBoundsException();
        if(size==elements.length)
            grow();
        for(int i=size; i>index; i--) {
            elements[i]=elements[i-1];
        }
        elements[index] = element;
        size++;
    }

    /**
     *    Adds a new element to the list, will be passed to {@code add(int, <E>)}
     *    with size, meaning it will add the element to the end of the list.
     *
     *    @param element the element to be added to the list
     */

    public void add(E element) {
        add(size,element);
    }

    /**
     *    Adds a new element to the list, will be passed to {@code add(int, <E>)}
     *    with target index 0, meaning it will add the element to the 'top' of the list.
     *
     *    @param element the element to be added to the list
     */

    public void addFirst(E element) {
        add(0, element);
    }

    /**
     *    Same functionality as addLast.
     *
     *    @param element the element to be added to the list
     */

    public void addLast(E element) {
        add(size, element);
    }

    /**
     *    Will remove and return targeted object at passed index.
     *
     *     @param index target index where element can be found
     *     @return the object to be removed from the list
     *     @throws IndexOutOfBoundsException if invalid index is passed
     */

    public E remove(int index) {
        E targetObject;

        if(index < 0 || index > this.size){
            throw new IndexOutOfBoundsException("Invalid index passed to 
remove");
        }
        else{
            targetObject = this.get(index);

            for(int t = index; t < this.size; t++){
                this.elements[t] = this.elements[t + 1];
            }

            size--;
        }

        return targetObject;
    }

    /**
     *    Removes the first element on the list with index 0.
     *
     *    @return result from remove with argument index = 0
     */

    public E removeFirst() {
        return this.remove(0);
    }

    /**
     *    Removes the last element on the list with index size.
     *
     *    @return result from remove with argument index = size
     */

    public E removeLast() {
        return this.remove(this.size);
    }

    /**
     *    Removes all elements in the list by looping through and 
dereferencing object references.
     */

    public void clear() {
        for(int i = 0; i < this.size; i++){
            this.elements[i] = null;
        }

```

```java
            size = 0;
        }

        /**
         *    Returns the object found at passed index, will throw exception if invalid index is passed.
         *
         *    @param index target index where element can be found
         *    @return the object at given index
         *    @throws IndexOutOfBoundsException if invalid index is passed
         */

        public E get(int index) {
            if(index < 0 || index > this.size){
                throw new IndexOutOfBoundsException("Invalid index passed to get");
            }

            return this.elements[index];
        }

        /**
         *    Set a new object at a given index in list.
         *
         *    @param index target index where element can be found
         *    @param element the object to be set with
         *    @return the old object that was replaced
         *    @throws IndexOutOfBoundsException if invalid index is passed
         */

        public E set(int index, E element) {
            E previousObj;

            if(index < 0 || index > this.size){
                throw new IndexOutOfBoundsException("Invalid index passed to set");
            }
            else{
                previousObj = this.elements[index];
                this.elements[index] = element;
            }

            return previousObj;
        }

        /**
         *    Returns the index of a given object, will return -1 if not found.
         *
         *    @param element the element being searched for
         *    @return result from method indexOf(int, E)
         */

        public int indexOf(E element) {
            return this.indexOf(0, element);
```

```
    }

    /**
     *    Will return the index of a given object that has the same reference
     *    as passed object, give startIndex to improve search speed, will return -1 if not found.
     *
     *     @param startIndex to start searching at to improve search performance
     *     @param element the object whos reference will be compared
     *     @return the index of object, -1 if not found
     */

    public int indexOf(int startIndex, E element) {
        for(int i = startIndex; i < this.size; i++){
            if(elements[i].equals(element)){
                return i;
            }
        }

        return -1;
    }

    /**
     * Returns the size of the element.
     *
     * @return the current size of the list
     */

    public int size() {
        return this.size;
    }

    /**
     *    Overrides the toString from superclass and returns a listing of objects as a string.
     *    This is done via StringBuilder class.
     *
     *    @see StringBuilder
     *    @return the list as a string.
     */

    public String toString() {
        StringBuilder res = new StringBuilder("[ ");
        for(int i=0; i<size; i++) {
            res.append(elements[i]);
            if(i<size-1)
                res.append("; ");
        }
        res.append(" ]");
        return res.toString();
    }

    /**
```

```
     *    A simple method to return an iterator object in order to loop the list with
     *    other means than index arithmetics. Currently uses a private class, the lines
     *    commented do this by passing an anonymous class implementing the interface Iterator.
     *
     *     @return a new iterator object
     */

    public Iterator<E> iterator() {
        return new Iter();
//        return new Iterator<E>() {
//            private int index=0;
//
//            public boolean hasNext() {
//                return index<size;
//            }
//
//            public E next() {
//                if(index==size)
//                    throw new NoSuchElementException();
//                return elements[index++];
//            }
//
//            public void remove() {
//                throw new UnsupportedOperationException();
//            }
//        };
    }

    /**
     *    The private Iter class for passing Iterator objects, giving means to loop through the list
     */

    private class Iter implements Iterator<E> {
        private int index=0;

        public boolean hasNext() {
            return index<size;
        }

        public E next() {
            if(index==size)
                throw new NoSuchElementException();
            return elements[index++];
        }

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

AVLNode

```java
package collections;

/**
*   AVLNode
*
*   A wrapper class for storing data in a tree data-structure, works
*   by linking together nodes in an hierarchy, should be used together with
*   AVLTree.
*
*   @author Rolf Axelsson
*   @author Philip Ekholm
*/

class AVLNode<K,V> {
    K key;
    V value;
    AVLNode<K,V> left;
    AVLNode<K,V> right;
    int height=0;

    /**
    *   Basic constructor for the class, assigns data stored together with a key
    *   as well as left and right children (does not need to be set explicitly).
    *
    *   @param key key in which the node is identified with.
    *   @param value the data object to be stored in the node.
    *   @param left the left node to be assigned as left-child to this node.
    *   @param right the right node to be assigned as right-child to this node.
    */

    public AVLNode( K key, V value, AVLNode<K,V> left, AVLNode<K,V> right ) {
        this.key = key;
        this.value = value;
        this.left = left;
        this.right = right;
    }

    /**
    *   Get the height from this node all the way down to the leaves of
    *   the hierarchy, uses a recursive implementation to count the depth
    *   of the structure.
    *
    *   @return the height of the structure downwards.
    */

    public int height() {
        int leftD = -1, rightD = -1;
        if( left != null )
            leftD = left.height();
        if( right != null )
            rightD = right.height();
        return 1 + Math.max( leftD, rightD );
    }

    /**
    *   Get the number of nodes from this node all the way down to the
    *   leaves of the hierarchy, uses a recursive implementation to count
    *   the number of nodes in the structure.
    *
    *   @return the number of nodes in the structure downwards.
    */

    public int size() {
        int leftS = 0, rightS = 0;
        if( left != null )
            leftS = left.size();
        if( right != null )
            rightS = right.size();
        return 1 + leftS + rightS;
    }

    /**
    *   Get a console print of the current node (key and value) as well as
    *   for all the nodes downwards in the hierarchy. Algorithm uses a
recursive
    *   implementation to print all the nodes in the structure.
    */

    public void print() {
        if( left != null)
            left.print();
        System.out.println(key + ": " + value);
        if( right != null )
            right.print();
    }
}
```

AVLTree

```
package collections;
import java.util.Comparator;
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
*   AVLTree
*
*   AVLTree is a written datastructure class for storing data in a non-
linear
*   structure, while AVLTree is based on binary search trees it has the
*   ability to auto balance the tree whenever the structure degenerates
more
```

```
 *   than a balance factor of -1 or +1. Uses wrapper objects AVLNode for storing
 *   data in.
 *
 *   @author Philip Ekholm
 *   @created 2017-03-04
 */

public class AVLTree<K,V> implements SearchTree<K,V> {
    private Comparator<K> comparator;
    private AVLNode<K,V> tree;

    /**
    *   If no custom comparator implementation has been made
    *   a default comparator as inner class will be instantiated.
    */

    public AVLTree() {
        comparator = new Comp();
    }

    /**
    *   Custom comparator can be implemented depending
    *   on how keys should be compared against each other.
    *
    *   @param comp class that implements Comparator with overridden compare.
    */

    public AVLTree( Comparator<K> comp ) {
        comparator = comp;
    }

    /**
    *   Return the root of the whole AVL-structure.
    *
    *   @return root of the AVL-Structure.
    */

    public AVLNode<K,V> root() {
        return tree;
    }

    /**
    *   find a certain AVLNode using passed key, if found the value of the
    *   node will be returned, otherwise null will be returned.
    *
    *   @param key key of the target node to be obtained.
    *   @return the value of the node if found, otherwise null.
    */

    public V get(K key) {
        AVLNode<K,V> node = find( key );
        if(node!=null)
```

```java
            return node.value;
        return null;
    }

    /**
    *   put (insert) a new node into the AVL-structure using a unique key
    *   as well as a value (data). Make sure the key is unique to the
    *   AVL-structure to avoid errors.
    *
    *   @param key key identifier unique for the created node.
    *   @param value the data value to pass to the newly created node.
    */

    public void put(K key, V value) {
        tree = put(tree,key,value);
    }

    /**
    *   Remove a node from the AVL-structure by passing the key of the target node.
    *
    *   @param key key identifier unique for the created node.
    *   @return the value of the node that was removed if successful, otherwise
    *   returns null.
    */

    public V remove(K key) {
        V value = get( key );
        if(value!=null) {
            tree = remove(tree,key);
        }
        return value;
    }

    /**
     *    Check if a certain node with passed key can be found in the AVL-structure.
     *
     *     @param key key identifier unique for the created node.
     *     @return true if the node with specific key could be found, otherwise false.
     */

    public boolean contains( K key ) {
        return find( key ) != null;
    }

    /**
     *    Returns the height of the AVL-structure.
     *
     *    @return the height of the AVL-structure.
     */

    public int height() {
        return height( tree );
    }

    private int height( AVLNode<K,V> node ) {
        if( node == null )
            return -1;
        return 1 + Math.max( height( node.left ), height( node.right ));
    }

    /**
     *    Return a new instance of an iterator for iterating all the values of
     *    nodes in the AVL-structure.
     *
     *    @return new instance of iterator for values.
     */

    public Iterator<V> iterator() {
        return new IterValues();
    }

    /**
     *    Return a new instance of an iterator for iterating all the keys of
     *    nodes in the AVL-structure.
     *
     *    @return new instance of iterator for keys.
     */

    public Iterator<K> iteratorKeys(){
        return new IterKeys();
    }

    private AVLNode<K,V> find(K key) {
        int res;
        AVLNode<K,V> node=tree;
        while( ( node != null ) && ( ( res = comparator.compare( key, node.key ) ) != 0 ) ) {
            if( res < 0 )
                node = node.left;
            else
                node = node.right;
        }
        return node;
    }

    private AVLNode<K,V> put(AVLNode<K,V> node, K key, V value) {
        if( node == null ) {
            node = new AVLNode<K,V>( key, value, null, null );
        } else {
            if(comparator.compare(key,node.key)<0) {
                node.left = put(node.left,key,value);
                node = this.balanceLeft(node);
            } else if(comparator.compare(key,node.key)>0) {
                node.right = put(node.right,key,value);
                node = this.balanceRight(node);
            }
        }
        return node;
    }

    private AVLNode<K,V> remove(AVLNode<K,V> node, K key) {
        int compare = comparator.compare(key,node.key);
        if(compare==0) {
            if(node.left==null && node.right==null)
                node = null;
            else if(node.left!=null && node.right==null)
                node = node.left;
            else if(node.left==null && node.right!=null)
                node = node.right;
            else {
                AVLNode<K,V> min = getMin(node.right);
                min.right = remove(node.right,min.key);
                min.left = node.left;
                node = min;
            }
        } else if(compare<0) {
            node.left = remove(node.left,key);
        } else {
            node.right = remove(node.right,key);
        }
        node = this.balanceNode(node);
        return node;
    }

    private AVLNode<K,V> getMin(AVLNode<K,V> node) {
        while(node.left!=null)
            node = node.left;
        return node;
    }

    /**
     *    Return the size ("length") of the AVL-structure, will be
     *    calculated using recursion.
     *
     *    @return the size of the tree.
     */

    @Override
    public int size() {
        return this.size(tree);
    }

    // Laboration 13
    private int size(AVLNode<K, V> node) {
        int l = 0, r = 0;

        if(this.tree == null){
            return 0;
        }
```

```java
        if(node.left != null){
            l = this.size(node.left);
        }
        if(node.right != null){
            r = this.size(node.right);
        }

        return 1 + r + l;
    }


    public List<K> keys(){
        ArrayList<K> list = new ArrayList<K>();

        Iterator<K> iter = new IterKeys();

        while(iter.hasNext()){
            list.add(iter.next());
        }

        return list;
    }

    public List<V> values(){
        ArrayList<V> list = new ArrayList<V>();

        Iterator<V> iter = this.iterator();

        while(iter.hasNext()){
            list.add(iter.next());
        }

        return list;
    }

    /**
     *    Operation not supported in this implementation.
     *
     *    @throws UnsupportedOperationException since operation is
unsupported.
     */
    public V first() throws UnsupportedOperationException{
        throw new UnsupportedOperationException();
    }

    /**
     *    Operation not supported in this implementation.
     *
     *     @throws UnsupportedOperationException since operation is
unsupported.
     */

    public V last() throws UnsupportedOperationException{
        throw new UnsupportedOperationException();
    }

    /**
     *    When the tree is instantiated either a class that implements
     *    Comparator can be passed to constructor, otherwise this class
     *    will be passed which tries to typecast the passed arguments
     *    to Comparable.
     *
     *    If typecasting fails then ClassCastException will be thrown.
     */

    private class Comp implements Comparator<K> {
        public int compare( K key1, K key2 ) {
            Comparable<K> k1 = ( Comparable<K> )key1;
            return k1.compareTo( key2 );
        }
    }

    private AVLNode<K,V> balanceNode(AVLNode<K,V> node) {
        if(node!=null) {
            node = balanceLeft(node);
            node = balanceRight(node);
        }
        return node;
    }

    private AVLNode<K,V> balanceLeft(AVLNode<K,V> node) {
        if((this.height(node.left) - this.height(node.right) == 2)){
            if(this.height(node.left.left) - this.height(node.left.right) == -1){
                node.left = this.rotateLeft(node.left);
                node = this.rotateRight(node);
            }
            else{
                node = this.rotateRight(node);
            }
        }

        return node;
    }

    private AVLNode<K,V> balanceRight(AVLNode<K,V> node) {
        if(this.height(node.left) - this.height(node.right) == -2){
            if(this.height(node.right.left) - this.height(node.right.right) == 1){
                node.right = this.rotateRight(node.right);
                node = this.rotateLeft(node);
            }
            else{
                node = this.rotateLeft(node);
            }
        }

        return node;
    }

    private AVLNode<K,V> rotateLeft(AVLNode<K,V> node) {
        AVLNode<K, V> newRoot = node.right;
        AVLNode<K, V> leftChild = newRoot.left;

        newRoot.left = node;
        node.right = leftChild;

        return newRoot;
    }

    private AVLNode<K,V> rotateRight(AVLNode<K,V> node) {
        AVLNode<K, V> newRoot = node.left;
        AVLNode<K, V> rightChild = newRoot.right;
        newRoot.right = node;
        node.left = rightChild;

        return newRoot;
    }

    /**
     *    The iteration class for iterating values.
     */

    private class IterValues implements Iterator<V> {
        ArrayList<V> list = new ArrayList<V>();
        int index = -1;

        public IterValues() {
            inOrder(tree);
        }

        private void inOrder(AVLNode<K,V> node) {
            if(node!=null) {
                inOrder(node.left);
                list.add(node.value);
                inOrder(node.right);
            }
        }

        public boolean hasNext() {
            return index<list.size()-1;
        }

        public V next() {
            if(!hasNext())
                throw new NoSuchElementException();
            index++;
            return list.get(index);
        }

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

    /**
     *    The iteration class for iterating values.
     */

    private class IterKeys implements Iterator<K> {
        ArrayList<K> list = new ArrayList<K>();
        int index = -1;

        public IterKeys() {
            inOrder(tree);
        }

        private void inOrder(AVLNode<K,V> node) {
            if(node!=null) {
                inOrder(node.left);
                list.add(node.key);
                inOrder(node.right);
            }
        }

        public boolean hasNext() {
            return index<list.size()-1;
        }

        public K next() {
            if(!hasNext())
                throw new NoSuchElementException();
            index++;
            return list.get(index);
        }

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

Book

```
package library;

/**
*    The Book class for a book.
*/

public class Book extends Media{
    private String author, bookTitle;

    public Book(String id, int year, String author, String bookTitle){
        super(id, year);
        this.author = author;
        this.bookTitle = bookTitle;
    }
```

```java

    /**
     *     @return the author of the book.
     */

    public String getAuthor() {
        return author;
    }

    /**
     *     @return the title of the book.
     */

    public String getBookTitle() {
        return bookTitle;
    }

    /**
     *     @return the data about the book as a string.
     */

    public String toString(){
        return bookTitle + ", " + author + ", " + super.getYear() + ", ID: " + super.getId();
    }
}
```

BorrowedItemsView

```java
package views;

import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.ScrollPaneConstants;

import controllers.ItemsController;

/**
*    BorrowedItemsView
*
*     The view for currently borrowed media objects.
*     From here borrowed objects can be listed and returned.
*/

```

```java
public class BorrowedItemsView extends JPanel{
    private static final long serialVersionUID = 1L;
    private ItemsController controller;
    private JLabel     personnrLabel = new JLabel("Va¨lkommen!"),
                    returnLabel = new JLabel("A˚terla¨mna: ");
    private JTextArea currentItems = new JTextArea();
    private JTextField returnField = new JTextField();
    private JButton returnItemButton = new JButton("A˚terla¨mna!"),
                    changeUserButton = new JButton("Byt användare");
    private JScrollPane scrollPane = new JScrollPane(currentItems);

    /**
    *     controller will be passed since communication is necessary.
    *
    *     @param controller the controller controlling the
    *   current view (ItemsController)
    */

    public BorrowedItemsView(ItemsController controller){
        this.setLayout(new FlowLayout());
        this.setPreferredSize(new Dimension(500, 400));
        currentItems.setEditable(false);
        this.controller = controller;
scrollPane.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);

        this.setDimensions();
        this.setActionListeners();
        this.addComponents();
    }

    private void setDimensions(){
        personnrLabel.setPreferredSize(new Dimension(200, 25));
        returnLabel.setPreferredSize(new Dimension(100, 25));
        changeUserButton.setPreferredSize(new Dimension(130, 25));
        scrollPane.setPreferredSize(new Dimension(500, 300));
        currentItems.setPreferredSize(new Dimension(500, 300));
        returnField.setPreferredSize(new Dimension(100, 25));
    }


    private void setActionListeners(){
        AL buttonListener = new AL();

        returnItemButton.addActionListener(buttonListener);
        changeUserButton.addActionListener(buttonListener);
    }

    private class AL implements ActionListener{
        @Override
        public void actionPerformed(ActionEvent e) {
            BorrowedItemsView outerClass = BorrowedItemsView.this;

            if(e.getSource() == returnItemButton){
                controller.returnItem(outerClass.returnField.getText());
            }
            else if(e.getSource() == changeUserButton){
                //controller.changeUser(loginField.getText());
                controller.logOut();
            }
        }
    }

    public void setWelcomeText(String text){
        personnrLabel.setText(text);
    }

    private void addComponents(){
        this.add(personnrLabel);
        this.add(changeUserButton);
        this.add(scrollPane);
        this.add(returnLabel);
        this.add(returnField);
        this.add(returnItemButton);
    }

    public void setBorrowedItems(String items){
        this.currentItems.setText(items);
    }
}
```

CompareDateBorrowed

```
package library;

import java.util.Comparator;

/**
*     Used for PriorityQueue in order to sort media-objects
*     corresponding to time borrowed. Used for sorting borrowed objects
*     first.
*/

public class CompareDateBorrowed implements Comparator<Media>{

    @Override
    public int compare(Media o1, Media o2) {
        if(o1.getDateBorrowed() != null && o2.getDateBorrowed() != null){
            return o1.getDateBorrowed().compareTo(o2.getDateBorrowed());
        }

        return 0;
    }

}
```

DVD

```java
package library;

/**
*      DVD
*
*      Contains info to be associated with a DVD.
*/

public class DVD extends Media{
    private String name;
    private String[] actors;

    public DVD(String id, int year, String name, String[] actors) {
        super(id, year);
        this.name = name;
        this.actors = actors;
    }

    public String getName() {
        return name;
    }

    /**
    * Return the actors as a copy of the array.
    *
    * @return all the actors as an array
    */

    public String[] getActors() {
        String[] newArray = new String[actors.length];

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

        return newArray;
    }

    public String toString(){
        return name + ", " + super.getYear() + ", ID: " + super.getId();
    }
}
```

ItemsController

```java
package controllers;

import java.util.Date;
import java.util.Iterator;

import javax.swing.JFrame;
import javax.swing.JOptionPane;

import collections.PriorityQueue;
import models.LibraryModel;
import models.Person;
import models.PersonModel;
import views.BorrowedItemsView;
import views.ItemsView;
import library.Book;
import library.CompareDateBorrowed;
import library.DVD;
import library.Media;

/**
* ItemsController
*
* ItemsController is responsible for the exchange of items.
* It controls two views in this case. It also extends the
* LibraryController responsible for loading in media files.
*/

public class ItemsController{
    private ItemsView itemsView = new ItemsView(this);
    private BorrowedItemsView borrowedItemsView = new BorrowedItemsView(this);
    private Person currentlyLoggedOn;
    private PersonModel pm;
    private LibraryModel lm;
    private JFrame     itemsFrame = new JFrame(),
                        borrowedItemsFrame = new JFrame();

    /**
     *     File path is sent upwards to the GeneralController.
     *     The old frame is reused in the borrowed-window, and the
     *     person currently logged in will be passed upwards.
     *
     *     @param filePath filePath for the persons (Lantagare.txt)
     *     @param oldFrame the old frame used for the login.
     *     @param currentlyLoggedIn the person that is currently
     *    logged into the system.
     */

    public ItemsController(PersonModel personsModel, LibraryModel libraryModel, Person currentlyLoggedIn) {
        this.currentlyLoggedOn = currentlyLoggedIn;
        this.pm = personsModel;
        this.lm = libraryModel;

        this.listItems();
        this.openUpWindows();

        borrowedItemsView.setWelcomeText("Välkommen, " + currentlyLoggedOn.getName());
    }

    /**
    *     Refresh the views with new items,
    *     and decide which ones are borrowed
    *     and not borrowed.
    */

    public void listItems(){
        String     books = "",
                DVDs = "",
                borrowedObjects = "";

        Iterator<Media> iter = lm.iterator();
        PriorityQueue<Media> pq = new PriorityQueue<Media>(new
CompareDateBorrowed());

        while(iter.hasNext()){
            Media media = iter.next();

            if(media instanceof Book && media.getBorrowedBy() == null){
                books += media.toString() + "\n";
            }
            else if(media instanceof DVD && media.getBorrowedBy() ==
null){
                DVDs += media.toString() + "\n";
            }
            else if(media.getBorrowedBy().equals(currentlyLoggedOn)){
                pq.enqueue(media);
            }
        }

        while(pq.size() > 0){
            borrowedObjects += pq.dequeue().toString() + "\n";
        }

        borrowedItemsView.setBorrowedItems(borrowedObjects);
        itemsView.setItems(books, DVDs);
    }

    private void openUpWindows(){

        itemsFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
borrowedItemsFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        itemsFrame.add(itemsView);
        borrowedItemsFrame.add(borrowedItemsView);

        itemsFrame.pack();
        itemsFrame.setVisible(true);
        itemsFrame.setLocation(50, 50);
        borrowedItemsFrame.pack();
        borrowedItemsFrame.setVisible(true);

    }

```

```java
    /**
    * Borrow a certain media-object from the collection.
    *
    * @param id the id of the media object to be borrowed.
    */

    public void borrow(String id){
        if(lm.contains(id)){
            Media media = lm.get(id);

            media.setBorrowedBy(currentlyLoggedOn);
            media.setDateBorrowed(new Date());
        }

        this.listItems();
    }

    /**
    * Return an item currently borrowed to the collection.
    *
    * @param id the id of the media-object to be returned.
    */

    public void returnItem(String id){
        if(lm.contains(id)){
            Media media = lm.get(id);
            media.setBorrowedBy(null);
            media.setDateBorrowed(null);
        }

        this.listItems();
    }

    /**
     *    Change the user that is currently logged in.
     *
     *     @param personId personnr to be used for verification.
     */

    public void changeUser(String personId){
        if(pm.contains(personId)){
            int result = JOptionPane.showConfirmDialog(null, "Byte av användare gick, vill du fortsätta?");

            if(result == JOptionPane.OK_OPTION){
                itemsFrame.setVisible(false);
                borrowedItemsFrame.setVisible(false);
                Person loggedInPerson = pm.get(personId);
                new ItemsController(pm, lm, loggedInPerson);
            }
        }
        else{
            JOptionPane.showMessageDialog(null, "Byte av användare gick ej, försök igen.");
        }
    }

    /**
     *    Log out the user.
     */

    public void logOut(){
        new LoginController(pm, lm);
        this.itemsFrame.setVisible(false);
        this.borrowedItemsFrame.setVisible(false);
    }
}
```

ItemsView

```java
package views;
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import javax.swing.JTextField;

import controllers.ItemsController;

/**
* ItemsView
*
* Items from the collection will be listed here, from here
* you can also borrow a certain item by filling in the id of it.
*/

public class ItemsView extends JPanel{
    private static final long serialVersionUID = 1L;
    private ItemsController controller;
    private JPanel libraryPanel = new JPanel();
    private JPanel borrowPanel = new JPanel();
    private JLabel borrowLabel = new JLabel("Ange media id:");
    private JTextField borrowField = new JTextField();
    private JButton borrowButton = new JButton("Låna");
    private JTextArea books = new JTextArea("Böcker"),
            dvds = new JTextArea("DVD:er");

    /**
    * Controller is passed with since communication is
    * necessary.
    *
    * @param controller a reference to the controller
```

```
     * that instantiated this view.
     */

    public ItemsView(ItemsController controller){
        this.setLayout(new BorderLayout());
        this.controller = controller;
        this.setDimensions();
        this.setActionListeners();
        this.addComponents();
        libraryPanel.setLayout(new GridLayout(1, 2, 20, 20));
        books.setEditable(false);
        dvds.setEditable(false);
    }

    private void setDimensions(){
        borrowLabel.setPreferredSize(new Dimension(120,25));
        borrowField.setPreferredSize(new Dimension(150,25));
        borrowButton.setPreferredSize(new Dimension(100,25));
        books.setPreferredSize(new Dimension(500,500));
        dvds.setPreferredSize(new Dimension(500, 500));
    }

    private void setActionListeners(){
        borrowButton.addActionListener(new ActionListener(){
            @Override
            public void actionPerformed(ActionEvent e) {
                ItemsView ref = ItemsView.this;
                controller.borrow(ref.borrowField.getText());
            }
        });
    }

    private void addComponents(){
        libraryPanel.add(books);
        libraryPanel.add(dvds);
        borrowPanel.add(borrowLabel);
        borrowPanel.add(borrowField);
        borrowPanel.add(borrowButton);

        this.add(libraryPanel, BorderLayout.CENTER);
        this.add(borrowPanel, BorderLayout.SOUTH);
    }

    /**
     * Refresh the view with current items.
     *
     * @param books the books as strings with toString
     * @param DVDs the DVDs passed as strings with toString
     */

    public void setItems(String books, String DVDs){
        this.books.setText(books);
        this.dvds.setText(DVDs);
    }
}
```

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

```java
package models;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Iterator;

import collections.AVLTree;
import library.Book;
import library.DVD;
import library.Media;

/**
 *    LibraryModel
 *
 *     A model for storing and accessing library-objects.
 *     @author Philip Ekholm
 *     @date 2017-04-01 12:13
 */

public class LibraryModel {
    private AVLTree<String, Media> mediaTree = new AVLTree<String, Media>();

    /**
     *    Default constructor for libraryModel.
     *
     *    @param filePathMedia filePath to be entered for Media.txt
     */

    public LibraryModel(String filePathMedia){
        try{
            LibraryModel.readMediaEntries(mediaTree, filePathMedia);
        }
        catch(FileNotFoundException e1){
            System.out.println("The file Media.txt could not be found at: " + filePathMedia);
            e1.printStackTrace();
        }
        catch(IOException e2){
            e2.printStackTrace();
        }
    }

    /**
     *     Return an iterator instance containing media-objects
     *
     *     @return iterator instance with media-objects
     */

    public Iterator<Media> iterator(){
        return mediaTree.iterator();
    }

    /**
     *    Check if item can be found in the model.
     *
     *    @param key the id of the media.
     *    @return true if found.
     */

    public boolean contains(String key){
        return mediaTree.contains(key);
    }

    /**
     *    Retrieve a media-item.
     *
     *    @param key the id of the media.
     *    @return the media if found, otherwise null.
     */

    public Media get(String key){
        return mediaTree.get(key);
    }

    /**
    * Read all the Media-entries into one AVL-tree structure.
    */

    public static void readMediaEntries(AVLTree<String, Media> tree, String filePath) throws FileNotFoundException, IOException{
        try(BufferedReader br = new BufferedReader(new FileReader(filePath))) {
            String line = br.readLine();
            while (line != null) {
                String[] details = line.split(";");

                if(details[0].equals("Dvd")){
                    String[] actors = new String[details.length - 4];
                    for(int i = 4; i < details.length; i++){
                        actors[i - 4] = details[i];
                    }
                    DVD d = new DVD(details[1],
                    Integer.parseInt(details[3]), details[2], actors);
                    tree.put(d.getId(), d);
                }
                else if(details[0].equals("Bok")){
                    Book b = new Book(details[1], Integer.parseInt(details[4]), details[2], details[3]);
                    tree.put(b.getId(), b);
                }
                line = br.readLine();
            }
        }
    }
}
```

LinkedList

```java
package collections;

import java.util.Iterator;

/**
 *    LinkedList E
 *
 *     A class that offers functionality to store data in a linked list structure using chained object nodes (so called list nodes in this
 *     implementation). Implements the interfaces List and Iterable, where list is used for ensuring certain methods will be included. Iterable is
 *     used for looping over a list with other means than index arithmetics. Supports generics in order to work with homogeneous datatypes.
 *
 *     @author Rolf Axelsson
 *     @author Philip Ekholm
 */

public class LinkedList<E> implements List<E>, Iterable<E> {
    //The starting point of the list, with a special list object to chain other objects to.
    private ListNode<E> list = null;

    //Returns the node located at a certain index.
    private ListNode<E> locate(int index) {
        ListNode<E> node = list;
        for( int i = 0; i < index; i++)
            node = node.getNext();
        return node;
    }

    /**
     *    size will calculate the amount of elements currently in the list, unlike the arrayList implementation this is not a variable
     *    but must be counted manually by counting all chained elements.
     *
     *     @return the number of elements linked in the list
     */

    public int size() {
        int n = 0;
        ListNode<E> node = list;
        while( node != null ) {
            node = node.getNext();
            n++;
        }
        return n;
    }

    /**
     *    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.
     *
     *    @throws IndexOutOfBoundsException if invalid index is passed
     *    @return the data at a certain object in the list.
     */

    public E get( int index ) {
        if( ( index < 0 ) || ( index > size() ) )
            throw new IndexOutOfBoundsException( "size=" + size() + ", index=" + index );

        ListNode<E> node = locate( index );
        return node.getData();
    }

    /**
     *    Set the data of a certain object at given index. Method will first check if index is valid, otherwise an exception will be thrown.
     *
     *    @param index of the object to be manipulated
     *    @param data to be passed to object
     *    @throws IndexOutOfBoundsException if invalid index is passed
     *    @return the old data which was replaced
     */

    public E set( int index, E data ) {
        if(index < 0 || index > this.size()){
            throw new IndexOutOfBoundsException();
        }
        else{
            E oldNode = this.get(index);
            this.remove(index);
            this.add(index, data);

            return oldNode;
        }
    }

    /**
     *    Add new data to the list, will be added to the end of the list if no index has been specified.
     *    @param data to be added to the list
     */

    public void add(E data) {
        this.addLast(data);
    }

    /**
     *    Add new data to the "top" of the list, will be added to index 0.
     *    @param data the data to be added to the list
     */

    public void addFirst( E data ) {
        this.add(0, data);
    }

    /**
     *    Same as add(E)
     */

    public void addLast( E data ) {
        this.add(this.size(), data);
    }

    /**
     *    Adds the data and creates a new object node at given index. Method will first check if index is valid, otherwise an exception will be thrown.
     *
     *    @param index the target position to add the element to
     *    @param data the data to be added to the list
     *    @throws IndexOutOfBoundsException if invalid index is passed
     */

    public void add( int index, E data ) {
        if(index < 0 || index > this.size()){
```

```java
            throw new IndexOutOfBoundsException();
        }
        else if(index == 0){
            list = new ListNode<E>(data, list);
        }
        else{
            ListNode<E> n0 = locate(index - 1);
            ListNode<E> n1 = new ListNode<E>(data, n0.getNext());
            n0.setNext(n1);
        }
    }

    /**
     *    Remove the very first element of the list, will call the remove(int) method with index = 0
     *
     *     @return the data that was removed from the list
     */

    public E removeFirst() {
        return this.remove(0);
    }

    /**
     *    Remove the very last element of the list, will call the remove(int) method with index = size() - 1
     *
     *    @return the data that was removed from the list
     */

    public E removeLast() {
        return this.remove(this.size() - 1);
    }

    /**
     *    Remove the element of the given index and return the data that was contained in the element.
     *    Method will first check if index is valid, otherwise an exception will be thrown.
     *
     *    @param index the target position of the element to be removed
     *    @throws IndexOutOfBoundsException if invalid index is passed
     *    @return the old data stored at index
     */

    public E remove( int index ) {
        if( ( index < 0 ) || ( index >= size() ) )
            throw new IndexOutOfBoundsException( "size=" + size() + ", index=" + index );

        E res;
        if( index == 0 ) {
            res = list.getData();
            list = setNull(list);
//            list = list.getNext();
```

```java
        } else {
            ListNode<E> node = locate( index - 1 );
            res = node.getNext().getData();
            node.setNext(setNull(node.getNext()));
//              node.setNext( node.getNext().getNext() );
        }
        return res;
    }

    private ListNode<E> setNull(ListNode<E> toNull) {
        ListNode<E> res = toNull.getNext();
        toNull.setData(null);
        toNull.setNext(null);
        return res;
    }

    /**
     *    Removes all elements in the list by looping through every element and remove them
     */

    public void clear() {
        while(this.size() > 0){
            this.removeLast();
        }
    }

    /**
     *    Returns the index of given data, will return -1 if not found.
     *
     *    @param data the data being searched for
     *    @return result from method indexOf(int, E)
     */

    public int indexOf(E data) {
        return indexOf(0, data);
    }

    /**
     *    Will return the index of a given object that has the same reference
     *    as passed object, give startIndex to improve search speed, will return -1 if not found.
     *
     *     @param startIndex to start searching at to improve search performance
     *     @param data the object who's reference will be compared
     *     @return the index of object, -1 if not found
     */

    public int indexOf(int startIndex, E data) {
        for(int i = startIndex; i < this.size(); i++){
            if(data.equals(this.get(i))){
                return i;
            }
        }

        return -1;
    }

    /**
     *    Returns an iterator object in order to loop the list with
     *    other means than index arithmetics. The method has been
simplified using
     *    the iterator of the arraylist instead of having to develop a
new algorithm to
     *    get all the elements in the linked list.
     *
     *     @return a new iterator object
     */

    public Iterator<E> iterator() {
        ArrayList<E> iterList = new ArrayList<E>(this.size());

        for(int i = 0; i < this.size(); i++){
            iterList.add(this.get(i));
        }

        return iterList.iterator();
    }

    /**
     *    Will return the toString from the ListNode class, which uses
StringBuilder to manipulate strings.
     *    If the list is dereferenced it will return empty parenthesis
[].
     */

    public String toString() {
        if( list != null )
            return list.toString();
        else
            return "[]";
    }

    /**
     *    Not implemented in this solution, can be ignored.
     */

    private class Iter implements Iterator<E> {

        public boolean hasNext() {
            return false;
        }

        public E next() {
            return null;
        }

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

LinkedQueue

```
package collections;

/**
*   LinkedQueue
*
*   A class that implements the interface Queue, which defined how
*   a queue datastructure should communicate with other objects. The
*   LinkedQueue is an implementation of a Queue using linkning to other
*   objects (nodes) which can be added/removed. The LinkedQueue works by
*   other Queue implementations (specifically through the FiFo-
structure).
*
*   @author Philip Ekholm
*   @created 2017-03-04
*
*/

public class LinkedQueue<E> implements Queue<E>{
    private LinkedList<E> elements;
    private int size;

    /**
    *   Constructor without arguments, which will instantiate
    *   a new LinkedQueue object. This implementation uses
    *   a LinkedList to store nodes.
    */

    public LinkedQueue() {
        elements = new LinkedList<E>();
        size = 0;
    }

    /**
    *   Enqueue (insert) new elements (data-objects) to the queue
    *   by adding them to the end of the list.
    *
    *   @param elem data-object to insert into the queue
    */

    public void enqueue( E elem ) {
        elements.addLast(elem);
        size++;
    }

    /**
    *   Dequeue (remove) the element (data-object) currently first up
    *   ("first in line") on the list and return it wherever
    *   the method was called.
    *
    *   If an attempt is made to dequeue an empty queue QueueException
    *   will be thrown.
    *
    *   @return the element currently first in the queue
    *   @throws QueueException if the queue is empty while attempting to
dequeue
    */

    public E dequeue() throws QueueException{
        if(size==0) {
            throw new QueueException("dequeue: Queue is empty");
        }
        E value = elements.removeFirst();
        size--;
        return value;
    }

    /**
    *   Peek (get) the element (data-object) currently first up
    *   ("first in line") on the list. If an attempt is made to peek at
    *   an empty queue QueueException will be thrown.
    *
    *   @return the element currently first in the queue.
    *   @throws QueueException if the queue is empty while attempting to
peek.
    */

    public E peek() throws QueueException{
        if( size==0 ) {
            throw new QueueException("peek: Queue is empty");
        }
        return elements.get(0);
    }

    /**
    *   Check whether the queue is empty or not.
    *
    *   @return true if the queue is empty, otherwise false.
    */

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

    /**
    *   Return the size (length) of the list.
    *
    *   @return the current size of the list.
    */

    public int size() {
        return size;
```

```java
    }

    /**
    *   The toString-method will return a string-object containing a
print
    *   of objects containing properties (among else). The current
    *   implementation will use the toString of the LinkedList instead of
    *   of a new implementation.
    *
    *   @return the toString return value of LinkedList
    */

    public String toString(){
        return elements.toString();
    }

}
```

List

```java
package collections;
import java.util.Iterator;

public interface List<E> {

    /**
     * Appends the specified element to the end of this list
     * @param element element to be appended to this list
     */
    public void add(E element);

    /**
     * Inserts the specified element at the specified position in this
list.
     * Shifts the element currently at that position (if any) and any
     * subsequent elements to the right (adds one to their indices).
     * @param index index at which the specified element is to be
inserted
     * @param element element to be inserted
     */
    public void add(int index, E element);

    /**
     * Inserts the specified element at the beginning of this list
     * @param element element to be inserted at the beginning of this
list
     */
    public void addFirst(E element);

    /**
     * Appends the specified element at the end of this list
     * @param element element to be appended at the end of this list
     */
    public void addLast(E element);

    /**
     * Removes the element at the specified position in this list. Shifts
     * any subsequent elements to the left (subtracts one from their
     * indices).  Returns the element that was removed from the list.
     * @param index the index of the element to be removed
     * @return the element previously at the specified position
     */
    public E remove(int index);

    /**
     * Removes and returns the first element from this list.
     * @return the first element from this list
     */
    public E removeFirst();

    /**
     * Removes and returns the last element from this list.
     * @return the last element from this list
     */
    public E removeLast();

    /**
     * Removes all of the elements from this list. The list will be
     * empty after this call returns.
     */
    public void clear();

    /**
     * Returns the element at the specified position in this list.
     * @param index index of the element to return
     * @return the element at the specified position in this list
     */
    public E get(int index);

    /**
     * Replaces the element at the specified position in this list with
the
     * specified element
     * @param index index of the element to replace
     * @param element element to be stored at the specified position
     * @return the element previously at the specified position
     */
    public E set(int index, E element);

    /**
     * Returns the index of the first occurrence of the specified element
     * in this list, or -1 if this list does not contain the element.
     * @param element element to search for
     * @return the index of the first occurrence of the specified element
in
     *         this list, or -1 if this list does not contain the element
     */
    public int indexOf(E element);

    /**
     * Returns the index of the first occurrence of the specified element
     * in this list, or -1 if this list does not contain the element. The
     * search begins at startIndex in the list.
     * @param startIndex the search starts at position startIndex in the
list
     * @param element element to search for
     * @return the index of the first occurrence of the specified element
in
     *         this list, or -1 if this list does not contain the element
     */
    public int indexOf(int startIndex, E element);

    /**
     * Returns an iterator over the elements in this list in proper
sequence.
     * @return an iterator over the elements in this list in proper
sequence
     */
    public Iterator<E> iterator();

    /**
     * Returns the number of elements in this list.
     * @return the number of elements in this list
     */
    public int size();
}
```

ListNode

```java
package collections;

/**
 *    ListNode E
 *
 *     ListNode is what the linked list is built up on. It is able to
store data of homogeneous kind,
 *     as well as the next listnode in order to continue the chain. It has
different getters and setters to these as well
 *     as a toString method in order to be printed out as a string.
 */

public class ListNode<E> {
    private E data;
    private ListNode<E> next;

    /**
     *    Base constructor, takes data as well as the next object in the
chain.
     *
     *    @param data the data to be stored
     *    @param next the next object to be linked
     */

    public ListNode( E data, ListNode<E> next ) {
        this.data = data;
        this.next = next;
    }

    /**
     *    Getter for extracting data.
     *
     *    @return the data from this object
     */

    public E getData() {
        return this.data;
    }

    /**
     *    Setter for setting new data.
     *
     *     @param data data to replace the current data with
     */

    public void setData( E data ) {
        this.data = data;
    }

    /**
     *    getter for next object chained. This method is applicable for method-chaining, since
     *    this is recursive.
     *
     *    @return the next object in line
     */

    public ListNode<E> getNext() {
        return this.next;
    }

    /**
     *    Setter for setting the next object in the line.
     *
     *     @param next the next object to be added after the current one
     */

    public void setNext( ListNode<E> next ) {
        this.next = next;
    }

    /**
     *    Overrides the toString from superclass and returns the toString from data.
     *    This is done via StringBuilder class.
     *
     *    @see StringBuilder
     *    @return the list as a string.
     */

    public String toString() {
        StringBuilder str = new StringBuilder("[ ");
        str.append(data.toString());
        ListNode<E> node = next;
        while( node!=null ) {
            str.append( "; ");
            str.append( node.getData().toString() );
            node = node.getNext();
        }
        str.append( " ]");
        return str.toString();
    }
}
```

LoginController

```
package controllers;

import javax.swing.JFrame;
import javax.swing.JOptionPane;

import models.LibraryModel;
import models.Person;
import models.PersonModel;
import views.LoginView;

/**
* LoginController
*
* The LoginController is responsible for starting up the LoginView,
* as well as validating the number used for logging in.
*/

public class LoginController{
    private LoginView view = new LoginView(this);
    private PersonModel personModel;
    private LibraryModel libraryModel;
    private Person loggedInPerson;
    private JFrame frame;

    /**
    * Default constructor will take arguments for loading
    * in files required for the structure to work. These will
    * be sent to the super class for processing.
    *
    * @param filePathPersons the String containing the file
    * path for loading lantagare.txt
    */

    public LoginController(String filePathPersons, String filePathMedia){
```

```java
        personModel = new PersonModel(filePathPersons);
        libraryModel = new LibraryModel(filePathMedia);

        this.setupJFrame();
    }

    public LoginController(PersonModel pm, LibraryModel lm){
        personModel = pm;
        this.libraryModel = lm;

        this.setupJFrame();
    }

    private void setupJFrame(){
        this.frame= new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(view);
        frame.setLocation(50, 50);

        frame.pack();
        frame.setVisible(true);
    }

    /**
    * Validates the number passed in, will allow access
    * if number can be found as key. A confirmDialog
    * will then be opened to confirm the login.
    *
    * If accepted, a LibraryController will be instantiated.
    *
    * @param number number entered in the loginView
    */

    public void validate(String number){
        if(personModel.contains(number)){
            int result = JOptionPane.showConfirmDialog(null, "Inloggning gick, vill du gå vidare?");
            if(result == JOptionPane.OK_OPTION){
                view.setVisible(false);
                loggedInPerson = personModel.get(number);
                new ItemsController(personModel, libraryModel, loggedInPerson);
                frame.setVisible(false);
            }
        }
        else{
            JOptionPane.showMessageDialog(null, "Inloggning gick ej, försök igen.");
        }
    }
}
```

LoginView

```java
package views;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

import controllers.LoginController;

/**
* The login view for entering the personnr.
*/
public class LoginView extends JPanel{
    private static final long serialVersionUID = 1L;
    private JLabel     label1 = new JLabel("Mata in personnr (10 siffor)
för att gå vidare: yymmdd-xxxx"),
                       label2 = new JLabel("Personnr: ");
    private JTextField personField = new JTextField();
    private JButton sendBtn = new JButton("Gå vidare");

    /**
    * Pass reference to the controller since communication
    * back will be necessary.
    *
    * @param controller the controller that instantiated
    * this view.
    */

    public LoginView(LoginController controller){
    this.setPreferredSize(new Dimension(400, 120));
    this.setBackground(Color.WHITE);
    this.setLayout(null);

    sendBtn.addActionListener(new ActionListener(){
        @Override
        public void actionPerformed(ActionEvent e) {
            controller.validate(personField.getText());
        }
    });

    this.setDimensions();
    this.addComponents();
    }

    /**
    * Set the dimensions of objects.
    */

    private void setDimensions(){
        label1.setBounds(10, 10, 400, 25);
        label2.setBounds(10, 50, 100, 25);
        personField.setBounds(80, 48, 160, 30);
        sendBtn.setBounds(250, 48, 90, 30);
    }

    private void addComponents(){
        this.add(label1);
        this.add(label2);
        this.add(personField);
        this.add(sendBtn);
    }
}
```

Main

```java
package common;
import controllers.LoginController;

/**
*     Start the program from here.
*/

public class Main {
    public static void main(String[] args) {
        new LoginController("files/Lantagare.txt", "files/Media.txt");
    }
}
```

Media

```java
package library;
import java.util.Date;

import models.Person;

/**
* Media
*
* A generalized abstract object which is inherited by Book and DVD
*/

public abstract class Media {
    private String id;
    private Person borrowedBy;
    private Date dateBorrowed;
    private int year;

    public Media( String id, int year) {
        this.id = id;
        this.borrowedBy = null;
        this.year = year;
    }

    public String getId() {
        return id;
    }

    public int getYear(){
        return year;
    }

    public Person getBorrowedBy() {
        return borrowedBy;
    }

    public Date getDateBorrowed() {
        return dateBorrowed;
    }

    public void setDateBorrowed(Date dateBorrowed) {
        this.dateBorrowed = dateBorrowed;
    }

    public void setBorrowedBy(Person borrowedBy) {
        this.borrowedBy = borrowedBy;
    }

    /**
     *    Equals is overriden and will return true if the id of the media
     *    matches the one passed.
     *
     *    @param obj the object to be matched to see if it's equal.
     *    @return true if id matches, otherwise false.
     */

    @Override

    public boolean equals( Object obj ) {
        if(obj instanceof Media) {
            Media media = (Media)obj;
            return id.equals( media.getId() );
        }
        return false;
    }
}
```

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

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;Tönisson;Högre matematik för poeter och andra oskulder;1982

Bok;553245;Gustafsson;Tennisspelarna;1977

Person

```java
package models;

/**
 *      Person
 *
 *      Person is a class for storing common information about
 *      people (a.k.a. Lantagare).
 */

public class Person {
    private String name,
    personnr,
    phoneNumber;

    public Person(String personnr, String name, String phoneNumber){
        this.name = name;
        this.personnr = personnr;
        this.phoneNumber = phoneNumber;
    }

    public String getName() {
        return name;
    }

    public String getPersonnr() {
        return personnr;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }

    @Override
    public String toString(){
        return this.getName() + ", " + this.getPersonnr() + ", " + this.getPhoneNumber();
    }

    //Persons equal if personnr matches
    @Override
    public boolean equals(Object obj){
        if(obj instanceof Person){
            Person p = (Person)obj;

            return p.personnr.equals(this.personnr);
        }

        return false;
    }
}
```

PersonModel

```java
package models;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

import collections.AVLTree;

/**
 *    PersonsModel
 *
 *     A model for storing and accessing persons-objects.
 *     @author Philip Ekholm
 *     @date 2017-04-01 12:13
 */

public class PersonModel {
    private AVLTree<String, Person> persons = new AVLTree<String, Person>();

    public PersonModel(String filePath){
        try{
            PersonModel.readPersons(persons, filePath);
        }
        catch(FileNotFoundException e1){
            e1.printStackTrace();
        }
        catch(IOException e2){
            e2.printStackTrace();
        }
    }

    /**
     *    Read in all persons from the lantagare file using a FileReader.
     *
     *     @param tree the AVL-structure to fill with found persons.
     *     @param filePath the relative directory path to the file.
     */

    public static void readPersons(AVLTree<String, Person> tree, String filePath)
        throws FileNotFoundException, IOException{
        try(BufferedReader br = new BufferedReader(new FileReader(filePath))) {
            String line = br.readLine();
            while (line != null) {
                String[] details = line.split(";");
                Person p = new Person(details[0], details[1], details[2]);
                tree.put(p.getPersonnr(), p);
                line = br.readLine();
            }
        }
    }

    public boolean contains(String key){
        return persons.contains(key);
    }

    public Person get(String key){
        return persons.get(key);
    }
}
```

PriorityQueue

```
package collections;

import java.util.Comparator;

/**
 *    PriorityQueue
 *
 *    PriorityQueue is another implementation of the interface
 *    Queue. The priorityqueue implements the FiFo-structure of the Queue
 *    datastructure, but also differs on objects by using priority.
 *
 *    Objects to be prioritized will be compared to other objects through a
 *    class that implements comparator which can either be passed into constructor.
 *    If no class that implements Comparator has been passed the objects are
 *    assumed to implement the Comparable interface.
 *
 *    @author Philip Ekholm
 *    @crated 2017-03-04
 */

public class PriorityQueue<E> implements Queue<E>{
    private LinkedList<E> elements = new LinkedList<E>();
    private Comparator<E> comp;

    /**
    *    Constructor without arguments, which will instantiate
    *    a new PriorityQueue object. This implementation will
    *    uses a LinkedList to store nodes.
    *
    *    Classes instantiating a PriorityQueue without arguments are
    *    assumed to only store objects that implements Comparable.
    */

    public PriorityQueue(){
        this.comp = new Comp();
    }

    /**
    *    Constructor that takes a class that implements Comparator to
    *    compare objects by.
    */

    public PriorityQueue(Comparator<E> comp){
        this.comp = comp;
    }

    /**
    *    Enqueue (insert) new elements (data-objects) to the queue
    *   by adding them to the end of the list. If the object passed
    *    is prioritized by Comparator it will be moved further into line
    *    after another object with the same priority.
    *
    *   @param data data-object to insert into the queue
    */

    @Override
    public void enqueue( E data ) {
         int     index = 0,
                size = size();

         while (index<size && comp.compare(elements.get(index), data) <= 0) {
             index++;
         }

         elements.add(index, data);
    }

    /**
    *    Dequeue (remove) the element (data-object) currently first up
    *   ("first in line") on the list and return it wherever
    *   the method was called.
    *
    *   If an attempt is made to dequeue an empty queue QueueException
    *   will be thrown.
    *
    *   @return the element currently first in the queue.
    *   @throws QueueException if the queue is empty while attempting to dequeue.
    */

    @Override
    public E dequeue() throws QueueException{
        if(isEmpty()) {
            throw new QueueException("dequeue: Queue is empty");
        }

        return elements.removeFirst();
    }

    /**
    *   Peek (get) the element (data-object) currently first up
    *   ("first in line") on the list. If an attempt is made to peek at
    *   an empty queue QueueException will be thrown.
    *
    *   @return the element currently first in the queue.
    *   @throws QueueException if the queue is empty while attempting to peek.
    */

    @Override
    public E peek() throws QueueException{
        if( size()==0 ) {
            throw new QueueException("peek: Queue is empty");
        }
        return elements.get(0);
    }

    /**
    *   Check whether the queue is empty or not.
    *
    *   @return true if the queue is empty, otherwise false.
    */

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

```java

    /**
    *   Return the size (length) of the list.
    *
    *   @return the current size of the list.
    */

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

    /**
    *   The toString-method will return a string-object containing a print
    *   of objects containing properties (among else). The current
    *   implementation will use the toString of the LinkedList instead of
    *   of a new implementation.
    *
    *   @return the toString return value of LinkedList
    */

    @Override
    public String toString(){
        return elements.toString();
    }

    /**
    *    Comp is the default class that implements Comparator
    *    if no other class has been passed to constructor.
    *    Misuse of the class will result in a ClassCastException
    */

    private class Comp implements Comparator<E>{
        @Override
        public int compare(E obj1, E obj2) {
            Comparable<E> com1 = (Comparable<E>)obj1;

            return com1.compareTo(obj2);
        }
    }

}
```

Queue

```java
package collections;

public interface Queue<E> {

    /**
     * Inserts the specified element into this queue.
     * @param data the object to add
     * @throws QueueException if the element cannot be added at this
     *         time due to capacity restrictions
     */
    public void enqueue(E data);

    /**
     * Retrieves and removes the head of this queue.
     * @return the head of this queue
     * @throws QueueException if this queue is empty
     */
    public E dequeue();

    /**
     * Retrieves, but does not remove, the head of this queue.
     * @return the head of this queue
     * @throws QueueException if this queue is empty
     */
    public E peek();


    /**
     * Returns true if this stack contains no elements.
     * @return true if this stack contains no elements
     */
    public boolean isEmpty();

    /**
     * Returns the number of elements in this stack.
     * @return the number of elements in this stack
     */
    public int size();
}
```

QueueException

```java
package collections;

/**
*    QueueException
*
*    An exception written for handling different runtime-exceptions
*    that can occur in Queue implementations.
*/

public class QueueException extends RuntimeException {
    public QueueException() {}
    public QueueException( String message ) {
        super( message );
    }
}
```

SearchTree

```
package collections;
import java.util.Iterator;
import collections.List;

public interface SearchTree<K,V> {
    public void put(K key, V value);
    public V remove(K key);
    public V get(K key);
    public boolean contains(K key);
    public int height();
    public Iterator<V> iterator();
    public int size();
    public List<K> keys();
    public List<V> values();
    public V first();
    public V last();
}
```