Gruppuppgift DA353A VT17
Grupp 6

Medlemmar
Filip Nilsson
Jesper Anderberg
Szilveszter Dezsi
Aron Polner
Ali Hassan

Innehållsförteckning

Innehållsförteckning 1
Arbetsbeskrivning 2
Instruktioner för programstart 2
Systembeskrivning 3
Klassdiagram 3
Sekvensdiagram 4
Start Program 4
Låna media 4
Logga in och visa tillgänglig media 5
Återlämning av media 6
Sök tillgänglig media 6
Källkod 7
BinarySearchTree.java 7
Book.java 12
BSTNode.java 13
Bucket.java 14
Controller.java 15
Dvd.java 20
HashtableCH.java 21
Lantagare.txt 26
LibraryPanel.java 27
LogInPanel.java 30
MainLauncher.java 31
Map.java 32
Media.txt 33
Media.java 34
MediaLibrary.java 36
MediaViewer.java 37
SearchTree.java 39
User.java 40

Arbetsbeskrivning

Innan vi satte oss vid en dator och började koda så hade vi ett möte där vi diskuterade gemensamma mål och riktlinjer för hur utförandet skulle planeras. Vi arbetade fram en gemensam bild över hur systemet skulle kunna se ut och arbetade ut efter den.
Mötet innebar tidsplanering, uppritning av det grafiska gränssnittet på White board, samt hur logiken i programmet skulle kunna fungera.

Under grupparbetets gång har vi inte haft någon specifik uppdelning över vem som ska göra vad, utan vi bestämde oss tidigt för att sitta tillsammans i ett grupprum med en monitor på skolan.

Därmed fick alla att sitta vid datorn och koda, samt att vi enkelt kunde diskutera fram olika lösningar under arbetets gång.

Av erfarenhet från tidigare grupparbeten fungerade detta system mycket bättre. Tidigare har vi haft uppdelningar, men då har kommunikationen i gruppen samt sammansättningen av programmet i slutändan varit ett stort problem. Nu upplevde vi att vi hade tydligare riktlinjer, alla var mycket mer involverade i både logiken och layouten samt i förståelsen för hur programmet fungerar.

Efter att vi lagt grunden för hur programmet fungerar så var det också mycket enklare för varje gruppmedlem att på egen hand implementera olika funktionsförbättringar man kanske kände behövdes. Vi använde oss av GitHub som gemensam plattform för att synkronisera ändringar i koden som gjorts hemma.

Instruktioner för programstart

För att starta vårt program, lägg alla java-filer i paketet ”gu1” och kör MainLauncher.java.
Se till att Lantagare.txt och Media.txt är placerade i mappen ”files” i projektet.

För att kunna logga in måste ni använda er av ett låntagar-id som ni hittar i Lantagare.txt.

- Tex **821223-6666**;Anna Ek;040-4528121

Systembeskrivning

Vårt program simulerar en biblioteksapplikation där man som låntagare, med ett specifikt id, kan logga in och låna/lämna tillbaka olika sorters media.

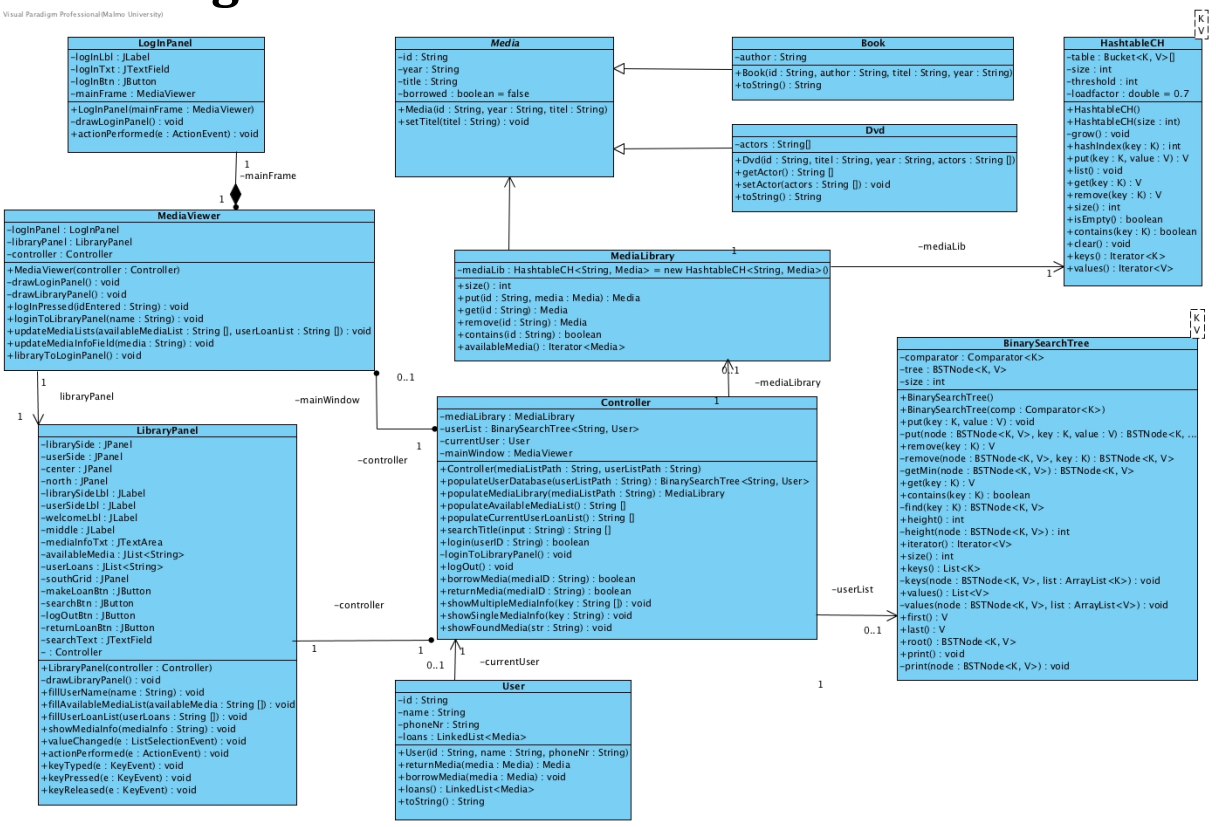
Vi har utöver detta en sökfunktion i programmet där användaren kan söka efter ett eller flera enskilda ord i en mediatitel. Sökfunktionen är begränsad till att bara kunna söka efter mediaobjekt som går att låna, dvs man kan **ej** söka efter mediaobjekt som redan är utlånade.
I programmet kan man, efter man loggat in och blivit godkänd, se en lista över medier man kan låna. Användaren kan markera ett mediaobjekt och välja att låna det genom att trycka på ”loan-knappen”.

Det nu lånade mediaobjektet läggs då till på användarens ”lånelista”. Användaren kan där tydligt se vilka olika mediaobjekt hen lånat.

Användaren kan även välja att lämna tillbaka ett mediaobjekt genom att på samma sätt markera det lånade mediaobjektet och trycka på ”return media-knappen”. Då återgår mediaobjektet igen till listan över möjliga mediaobjekt att låna.

När användaren är färdig så kan hen logga ut. Har användaren fortfarande lånade mediaobjekt kvar på sin lånelista efter utloggning, kommer nästa användare erfara att dessa inte går att låna.

Klassdiagram

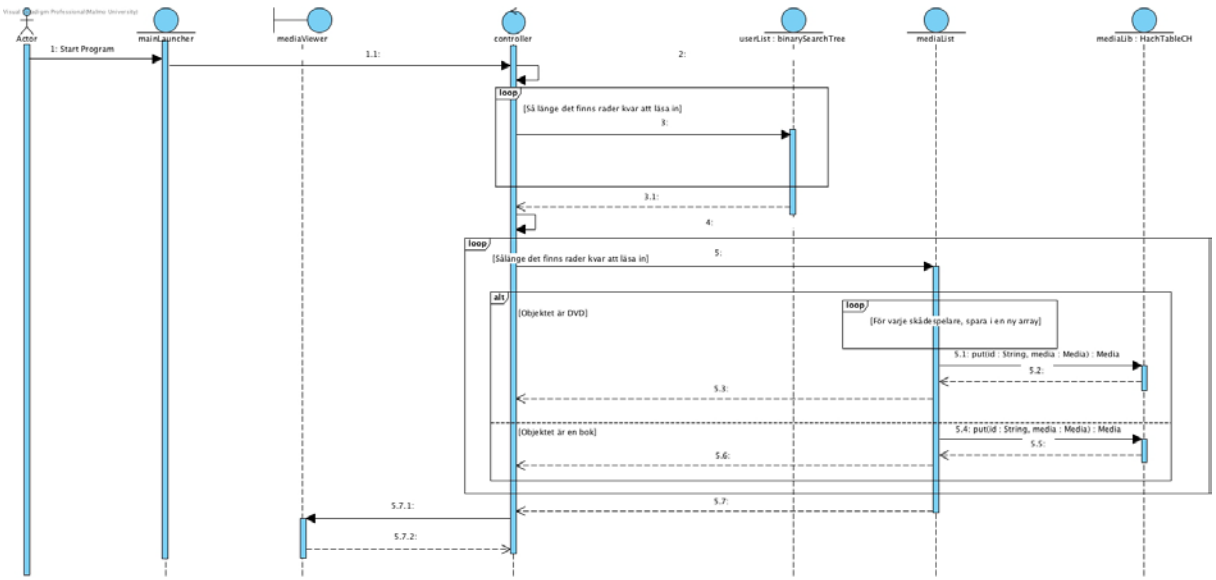


Figur 1

Sekvensdiagram

Start Program

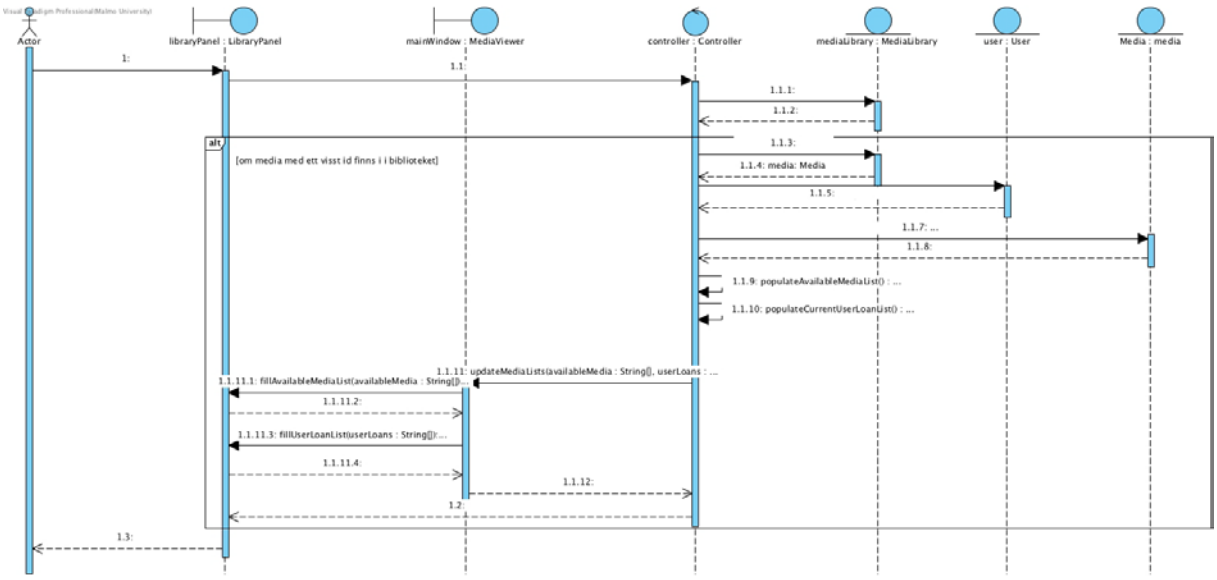
Ansvarig: Jesper Anderberg



Figur 2

Låna media

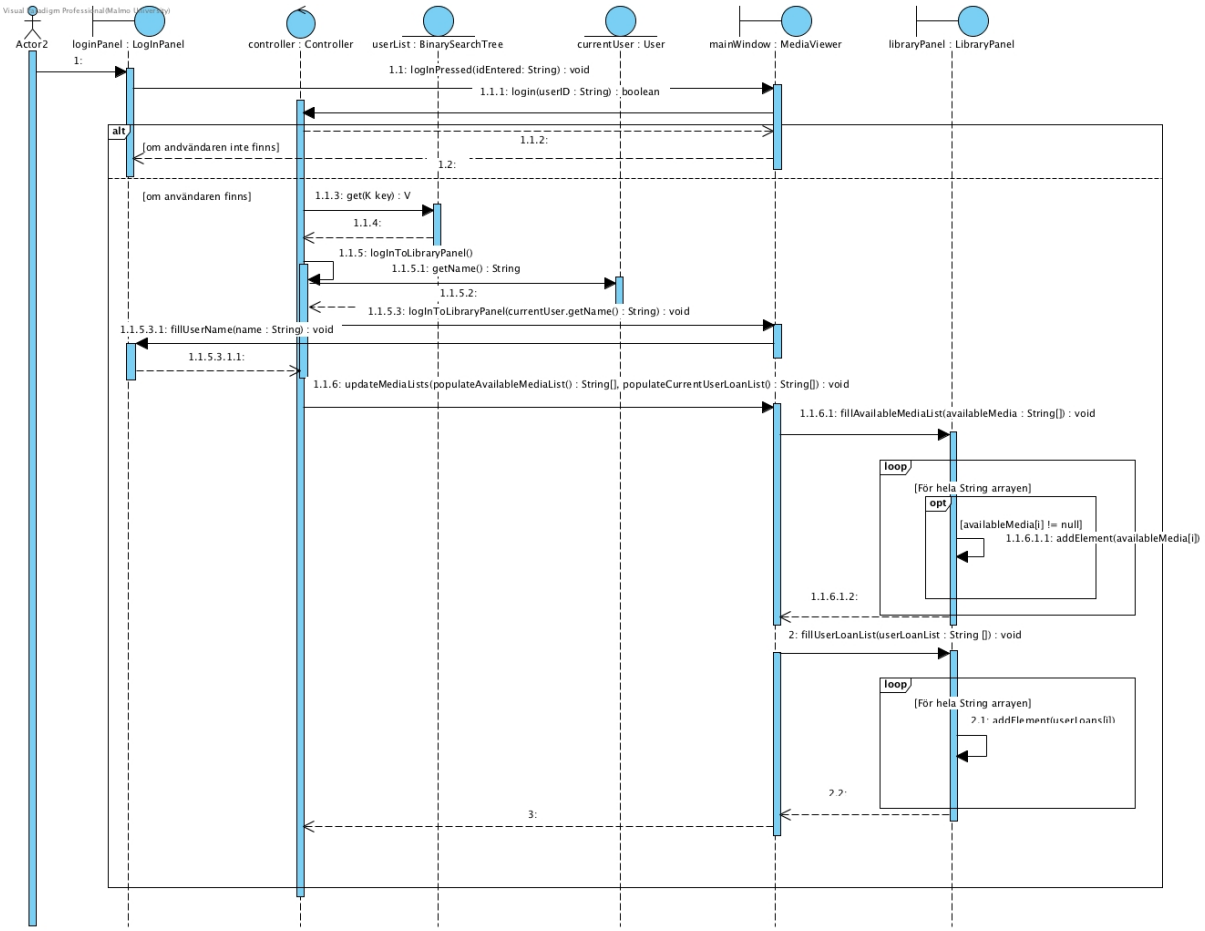
Ansvarig: Filip Nilsson



Figur 3

Logga in och visa tillgänglig media

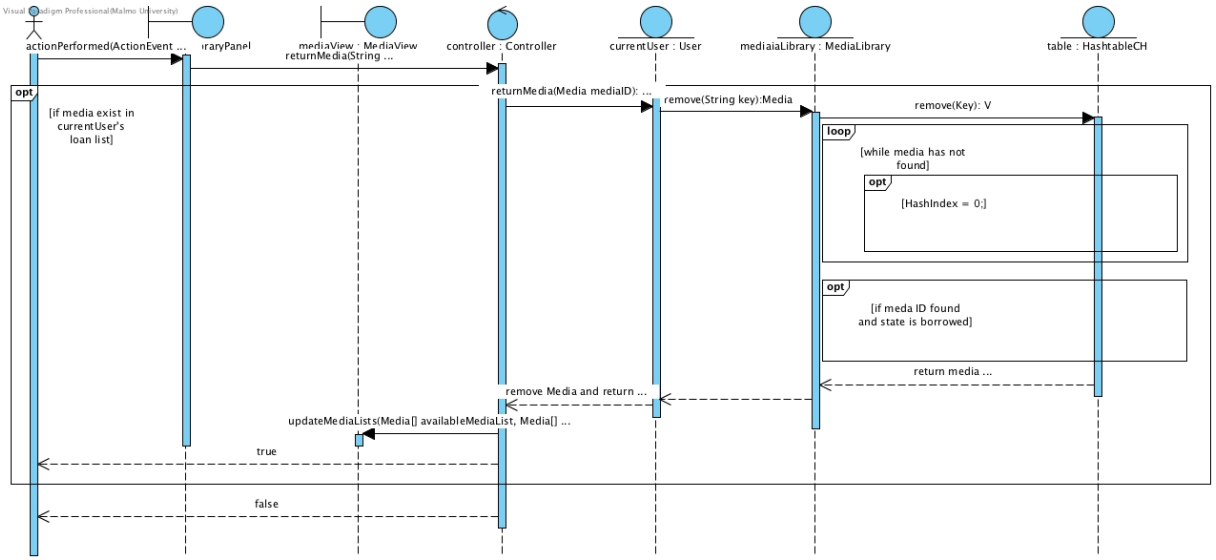
Ansvarig: Aron Polner



Figur 4

Återlämning av media

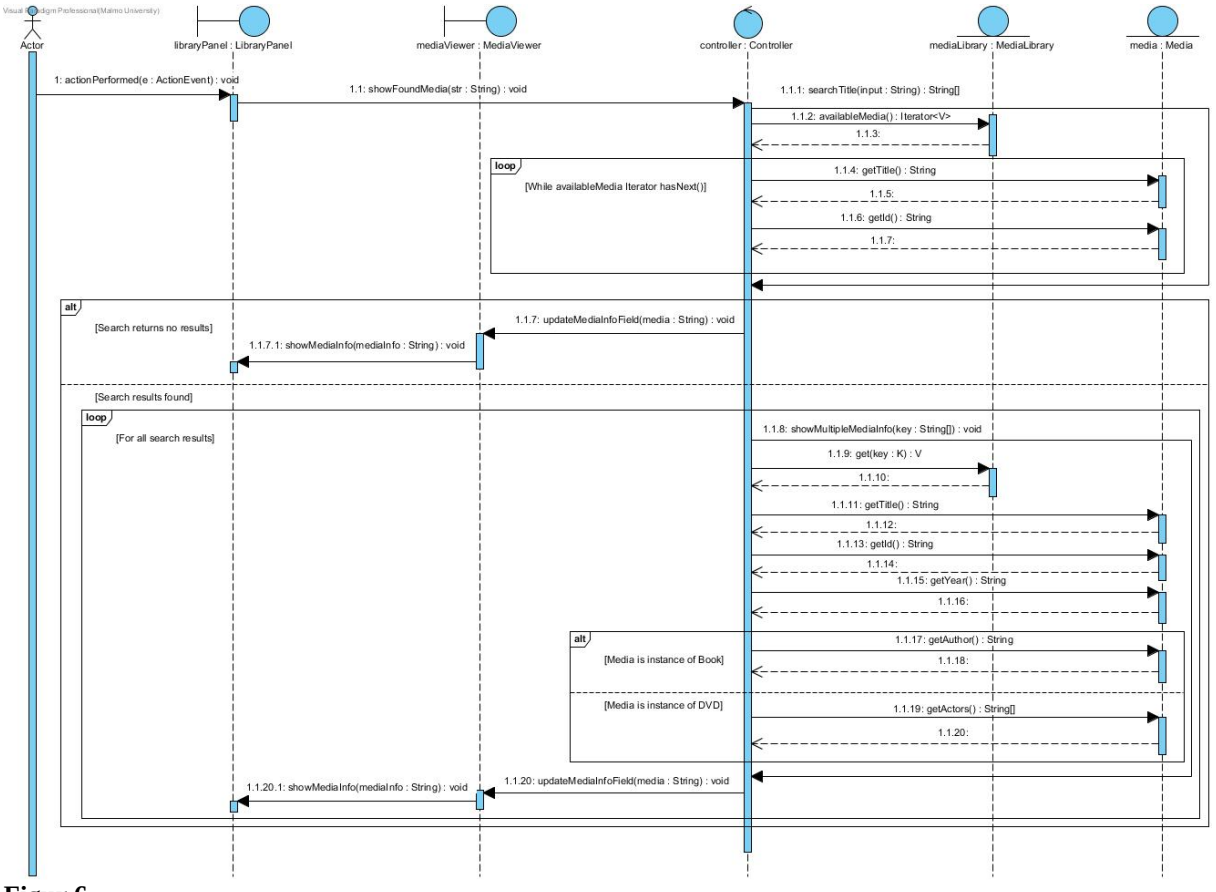
Ansvarig: Ali Hassan



Figur 5

Sök tillgänglig media

Ansvarig: Szilveszter Dezsi



Figur 6

Källkod

BinarySearchTree.java

```
package gu1;

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

/**
 * BinarySearchTree is a binary implemntation of the SearchTree interface.
 * BinarySearchTree uses an inner class implementation of the Comparator interface to sort elements.
 * BinarySearchTree uses an inner class implementation of the Iterable interface to traverse elements.
 * @author Jesper Anderberg, Filip Nilsson, Aron Polner, Ali Hassan, Szilveszter Dezsi
 * @param <K> specified key data reference type
 * @param <V> specified value data reference type
 */
public class BinarySearchTree<K,V> implements SearchTree<K,V> {
    private Comparator<K> comparator;
    private BSTNode<K,V> tree;
    private int size;

    /**
     * Constructs a new, empty tree set, sorted according to the natural ordering of its elements.
     */
    public BinarySearchTree() {
        comparator = new Comp();
    }

    /**
     * Constructs a new, empty tree set, sorted according to the specified comparator.
     * @param comp - Comparator
     */
    public BinarySearchTree( Comparator<K> comp ) {
        comparator = comp;
    }

    /**
     * Associates the specified value with the specified key in this tree.
     * @param key - key with which the specified value is to be associated
     * @param value - value to be associated with the specified key
     */
    public void put(K key, V value) {
        tree = put(tree,key,value);
    }

    /**
     * Associates the specified value with the specified key in this tree.
     * Tree sorts and balances according to the natural ordering of its elements.
     * @param node - node to be sorted and balanced
     * @param key - key with which the specified value is to be associated
     * @param value - value to be associated with the specified key
     * @return sorted and balanced tree
     */
    private BSTNode<K,V> put(BSTNode<K,V> node, K key, V value) {
        if( node == null ) {
            node = new BSTNode<K,V>( key, value, null, null );
            size++;
        } else {
            if(comparator.compare(key,node.key)<0) {
                node.left = put(node.left,key,value);
            } else if(comparator.compare(key,node.key)>0) {
                node.right = put(node.right,key,value);
            }
        }
        return node;
    }
```

```
    /**
     * Removes the value associated with the specified key in this tree, if present.
     * @param key - specified key
     * @return removed value, or <tt>null</tt> if not present in this tree
     */
    public V remove(K key) {
        V value = get( key );
        if(value!=null) {
            tree = remove(tree,key);
            size--;
        }
        return value;
    }

    /**
     * Removes the value associated with the specified key in this tree, if present.
     * Tree sorts and balances according to the natural ordering of its elements.
     * @param node - node to be sorted and balanced
     * @param key - specified key
     * @return sorted and balanced tree
     */
    private BSTNode<K,V> remove(BSTNode<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 {
                BSTNode<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);
        }
        return node;
    }

    /**
     * Returns the first (lowest) node contained in this tree.
     * @return the first (lowest) node contained in this tree
     */
    private BSTNode<K,V> getMin(BSTNode<K,V> node) {
        while(node.left!=null)
            node = node.left;
        return node;
    }

    /**
     * Returns the value associated with the specified key in this tree, if present.
     * @param key - specified key
     * @return value associated with the specified key, or <tt>null</tt> if not present in this tree
     */
    public V get(K key) {
        BSTNode<K,V> node = find( key );
        if(node!=null)
            return node.value;
        return null;
    }

/**
     * Returns <tt>true</tt> if this tree contains specified key, otherwise <tt>false</tt>.
     * @param key - specified key
     * @return <tt>true</tt> if this tree contains specified key, otherwise <tt>false</tt>
     */
    public boolean contains( K key ) {
        return find( key ) != null;
```

```
}

/**
 * Returns node containing specified key.
 * @param key - specified key
 * @return node containing specified key
 */
private BSTNode<K,V> find(K key) {
    int res;
    BSTNode<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;
}

/**
 * Returns the number of levels of this tree.
 * @return number of levels of this tree
 */
public int height() {
    return height( tree );
}

/**
 * Returns the number of levels of this node.
 * @param node - node to be measured
 * @return number of levels of this node
 */
private int height(BSTNode<K, V> node) {
   if (node == null)
      return -1;
   return 1 + Math.max(height(node.left), height(node.right));
}


/**
 * Returns an iterator over the values contained in this tree.
 * @return an iterator over the values contained in this tree
 */
public Iterator<V> iterator() {
    return new Iter();
}

/**
 * Inner class implementation of Iterator.
 */
private class Iter implements Iterator<V> {
    ArrayList<V> list = new ArrayList<V>();
    int index = -1;

    /**
     * Constructs new ordered iterator
     */
    public Iter() {
        inOrder(tree);
    }

    /**
     * Constructs new ordered iterator
     * @param node - target tree
     */
    private void inOrder(BSTNode<K,V> node) {
        if(node!=null) {
            inOrder(node.left);
            list.add(node.value);
            inOrder(node.right);
        }
    }

   /**
```

```
     * Returns true if the iteration has more elements.
     * @return boolean
     */
     public boolean hasNext() {
         return index<list.size()-1;
     }

    /**
     * Returns the next element in the iteration.
     * @return the next element in the iteration
     * @throws NoSuchElementException - if the iteration has no more elements
     */
     public V next() {
         if(!hasNext())
             throw new NoSuchElementException();
         index++;
         return list.get(index);
     }

    /**
     * Remove operation is not supported by this iterator.
     * @throws UnsupportedOperationException
     */
     public void remove() {
         throw new UnsupportedOperationException();
     }
}

/**
 * Returns the number of key-value nodes contained in this tree.
 * @return the number of key-value nodes contained in this tree
 */
public int size() {
   if( tree == null )
      return 0;
   return tree.size();
}

/**
 * Returns a list of the keys contained in this tree.
 * @return list of the keys contained in this tree
 */
public List<K> keys(){
   ArrayList<K> list = new ArrayList<K>();
   keys(tree, list);
    return list;
}

/**
 * Recursive algorithm that builds list.
 * @param node - target tree
 * @param list - target list
 */
private void keys(BSTNode<K,V> node, ArrayList<K> list){
   if( node != null ) {
      keys( node.left, list );
      list.add( list.size(), node.key );
      keys( node.right, list );
   }
}

/**
 * Returns a list of the values contained in this tree.
 * @return list of the values contained in this tree
 */
public List<V> values() {
   ArrayList<V> list = new ArrayList<V>();
   values(tree, list);
   return list;
}

/**
 * Recursive algorithm that builds list.
 * @param node - target tree
```

```
     * @param list - target list
     */
    private void values(BSTNode<K, V> node, ArrayList<V> list) {
       if (node != null) {
          values(node.left, list);
          list.add(list.size(), node.value);
          values(node.right, list);
       }
    }

     /**
      * Returns the first (lowest) value contained in this tree.
      * @return the first (lowest) value contained in this tree
      */
     public V first(){
       BSTNode<K, V> node = tree;
       if (node == null)
          return null;
       while (node.left != null)
          node = node.left;
         return node.value;
     }

     /**
      * Returns the last (greatest) value contained in this tree.
      * @return the last (greatest) value contained in this tree
      */
     public V last(){
       BSTNode<K, V> node = tree;
       if (node == null)
          return null;
       while (node.right != null)
          node = node.right;
         return node.value;
     }

     /**
      * Returns root node for this tree.
      * @return root node for this tree
      */
     public BSTNode<K,V> root() {
         return tree;
     }

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

     /**
      * Prints keys and values of the nodes in this node.
      * Format: <tt>key : value</tt>
      */
     public void print() {
       print(tree);
     }
     /**
      * Prints keys and values of the nodes in this tree in order.
      * Format: <tt>key:value</tt>
      * @param node - node to be printed
      */
     private void print(BSTNode<K,V> node) {
       if (node != null) {
          print(node.left);
          System.out.println(node.key + ":" + node.value);
          print(node.right);
       }
     }
}
```

Book.java

```
package gu1;

/**
 * A class that holds the id, author, title and year of publication
 * of a given Book object. The Book class extends Media.
 * @author Jesper Anderberg, Filip Nilsson, Aron Polner, Ali Hassan, Szilveszter Dezsi
 *
 */

public class Book extends Media{
private String author;
   /**
    * A constructor that takes four arguments
    * three of these, id, year and title
    * are sent to, and later handled by, the Media class.
    * @param id
    * @param author
    * @param titel
    * @param year
    */
   public Book(String id, String author, String titel, String year) {
      super(id, year, titel);
      this.author = author;
   }
   /**
    * A method that returns the author or authors
    * connected to a specific Book object.
    * @return the authors full name.
    */
   public String getAuthor() {
      return author;
   }
   /**
    * A method that sets the author of a particular Book object.
    * @param author
    */
   public void setAuthor(String author) {
      this.author = author;
   }
   /**
    * A toString method that returns all of the available informations
    * surrounding a given Book object.
    */
   public String toString() {
return "Book [id=" + this.getId() + ", author=" + author + ", titel=" + this.getTitle() + ", year=" +
this.getYear() + "]";
   }
}
```

BSTNode.java

```
package gu1;

/**
 * UserNode is a container class used with classes User and BinarySearchTree.
 * User objects are mapped with key-value pair parameters within a BinarySearchTree map structure.
 *
 * @author Jesper Anderberg, Filip Nilsson, Aron Polner, Ali Hassan, Szilveszter Dezsi
 *
 * @param <K> specified key data reference type
 * @param <V> specified key value reference type
 */
public class BSTNode<K,V> {
    K key;
    V value;
    BSTNode<K,V> left;
    BSTNode<K,V> right;

    /**
     * Constructs new UserNode.
     * @param key - specified key
     * @param value - specified value
     * @param left - left child
     * @param right - right child
     */
    public BSTNode( K key, V value, BSTNode<K,V> left, BSTNode<K,V> right ) {
        this.key = key;
        this.value = value;
        this.left = left;
        this.right = right;
    }

    /**
     * Returns the number of levels of this node.
     * @return number of levels of this node
     */
    public int height() {
        int leftH = -1, rightH = -1;
        if( left != null )
            leftH = left.height();
        if( right != null )
            rightH = right.height();
        return 1 + Math.max( leftH, rightH );
    }

    /**
     * Returns the number of nodes contained in this node.
     * @return the number of nodes contained in this node
     */
    public int size() {
        int leftS = 0, rightS = 0;
        if( left != null )
            leftS = left.size();
        if( right != null )
            rightS = right.size();
        return 1 + leftS + rightS;
    }

    /**
     * Prints keys and values of the nodes in this node.
     * Format: <tt>key : value</tt>
     */
    public void print() {
        if( left != null)
            left.print();
        System.out.println(key + " : " + value);
        if( right != null )
            right.print();
    }
}
```

Bucket.java

```
package gu1;

/**
 * A Media object is kept in a Bucket object and labeled as
 * either EMPTY, OCCUPIED or REMOVED. Each object
 * also holds a unique key which is its ID and a value which
 * represents the entirety of the information inside the Media object.
 * @author Jesper Anderberg, Filip Nilsson, Aron Polner, Ali Hassan, Szilveszter Dezsi
 *
 * @param <K>
 * @param <V>
 */
class Bucket<K,V> {
    static final int EMPTY = 0, OCCUPIED = 1, REMOVED = 2;
    int state = EMPTY;
    K key;
    V value;
}
```

Controller.java

```
package gu1;

import java.io.*;
import java.util.*;
import javax.swing.*;

/**
 * The Controller class handles the core logic.
 * @author Jesper Anderberg, Filip Nilsson, Aron Polner, Ali Hassan, Szilveszter Dezsi
 */
public class Controller {
   private MediaLibrary mediaLibrary;
   private BinarySearchTree<String, User> userList;
   private User currentUser;
   private MediaViewer mainWindow;

   /**
    * This constructor takes two strings as its arguments. These strings in
    * turn correspond to names of text files. The constructor uses these
    * files to populate the UserDatabase and MediaLibrary. Also, an instance
    * of the MediaViewer is created. This enables the controller to draw the
    * appropriate panels later on during the execution of the code.
    * @param mediaListPath
    * @param userListPath
    */
   public Controller(String mediaListPath, String userListPath) {
      this.userList = populateUserDatabase(userListPath);
      this.mediaLibrary = populateMediaLibrary(mediaListPath);
      this.mainWindow = new MediaViewer(this);
   }

   /**
    * The method populates an instance of BinarySearchTree with the current users.
    * @param userListPath is a String containing the name of a userList file.
    * @return BinarySearchTree<String, User> an implementation of a search tree
    * populated with the user information found in the document specified by the userListPath.
    */
   public BinarySearchTree<String, User> populateUserDatabase(String userListPath) {
      BinarySearchTree<String, User> userList = new BinarySearchTree<String, User>();
      try {
         BufferedReader reader = new BufferedReader(
               new InputStreamReader(new FileInputStream(userListPath), "UTF-8"));
         String str;
         String[] values;
         while ((str = reader.readLine()) != null) {
            values = str.split(";");
            userList.put(values[0], new User(values[0], values[1], values[2]));
         }
         reader.close();
      } catch (IOException e) {
         System.out.println(e);
      }
      return userList;
   }

   /**
    * The method populates an instance of MediaLibrary with the current media objects.
    * @param mediaListPath
    * @return MediaLibrary containing the current media objects
    */
   public MediaLibrary populateMediaLibrary(String mediaListPath) {
      MediaLibrary mediaList = new MediaLibrary();
      try {
         BufferedReader reader = new BufferedReader(
               new InputStreamReader(new FileInputStream(mediaListPath), "UTF-8"));
         String str;
         String[] values;
         while ((str = reader.readLine()) != null) {
            values = str.split(";");
            if (values[0].equals("Dvd")) {
```

```
            String[] actors = new String[values.length - 4];
            for (int i = 0; i < actors.length; i++)
               actors[i] = values[i + 4];
            mediaList.put(values[1], new Dvd(values[1], values[2], values[3], actors));
         } else if (values[0].equals("Bok")) {
            mediaList.put(values[1], new Book(values[1], values[2], values[3], values[4]));
         }
      }
      reader.close();
   } catch (IOException e) {
      System.out.println(e);
   }
   return mediaList;
}

/**
 * The method populates a Media array with the available media.
 * @return a Media array with the found media.
 */
public String[] populateAvailableMediaList() {
   Iterator<Media> values = mediaLibrary.availableMedia();
   String[] mediaList = new String[mediaLibrary.size()];
   int index = 0;
   while (values.hasNext()) {
      Media temp = values.next();
      if(!temp.isBorrowed()){
         mediaList[index] = (temp.getId() + ", " + temp.getTitle());
         index++;
      }
   }
   return mediaList;
}

/**
 * The method populates a Media array with the current loaned media.
 * @return a Media array with the loaned list.
 */
public String[] populateCurrentUserLoanList() {
   Iterator<Media> values = currentUser.loans().iterator();
   String[] loanList = new String[currentUser.loans().size()];
   int index = 0;
   while (values.hasNext()) {
      Media temp = values.next();
      loanList[index] = (temp.getId() + ", " + temp.getTitle());
      index++;
   }
   return loanList;
}

/**
 * This method takes a string as its input and searches the available media objects
 * to find a match. If a match is found
 * @param input - the piece of information used to search through the media objects.
 * @return A string array containing the information about a certain type of media
 */
public String[] searchTitle(String input) {
   String[] inputs = input.toLowerCase().split("[^a-öA-Ö0-9]+");
   Iterator<Media> values = mediaLibrary.availableMedia();
   ArrayList<String> foundIdlist = new ArrayList<String>();
   int count = 0;
   while (values.hasNext()) {
      Media media = values.next();
      String ref = media.getTitle();
      String id = media.getId();
      String[] refs = ref.toLowerCase().split("[^a-öA-Ö0-9]+");
      for (int i = 0; i < inputs.length; i++) {
         for (int j = 0; j < refs.length; j++) {
            if (inputs[i].equals(refs[j])) {
               count++;
            }
         }
         if (count == inputs.length) {
            foundIdlist.add(id);
         }
```

```
            }
            count = 0;
        }
        return foundIdlist.toArray(new String[foundIdlist.size()]);
    }

    /**
     * This method handles the login.
     * @param userID is a string containing an userID
     * @return either true or false depending on
     * if the login was successful or not.
     */
    public boolean login(String userID) {
        if (userList.contains(userID)) {
            currentUser = userList.get(userID);
            loginToLibraryPanel();
            return true;
        }
        JOptionPane.showMessageDialog(null, "User not found");

        return false;
    }

    private void loginToLibraryPanel() {
        mainWindow.loginToLibraryPanel(currentUser.getName());
        mainWindow.updateMediaLists(populateAvailableMediaList(), populateCurrentUserLoanList());
    }

    /**
     * The logOut method, as the name implies, handles loging out from the system.
     */
    public void logOut() {
        currentUser = null;
        mainWindow.libraryToLoginPanel();
    }

    /**
     * This method borrows a media object if
     * the given mediaID exists in the current mediaLibrary
     * @param mediaID is a String containing an ID
     * @return a boolean depending on the the borrow was successful or not.
     */
    public boolean borrowMedia(String mediaID) {
        if (mediaLibrary.contains(mediaID)) {
            currentUser.borrowMedia(mediaLibrary.get(mediaID));
            mediaLibrary.get(mediaID).setBorrowed(true);
            mainWindow.updateMediaLists(populateAvailableMediaList(), populateCurrentUserLoanList());
            return true;
        }
        return false;
    }

    /**
     * This method returns a media object if
     * the given mediaID exists in the current users list of loans.
     * @param mediaID
     * @return a boolean depending on the the return was successful or not.
     */
    public boolean returnMedia(String mediaID) {
        if (currentUser.loans().contains(mediaLibrary.get(mediaID))) {
            currentUser.returnMedia(mediaLibrary.get(mediaID));
            mediaLibrary.get(mediaID).setBorrowed(false);
            mainWindow.updateMediaLists(populateAvailableMediaList(), populateCurrentUserLoanList());
            return true;
        }
        return false;
    }

    /**
     * This method shows the media objects found during
     * a search.
     * @param key is a String array containing the IDs
     * of the media objects to be displayed in the main window.
     */
```

```
public void showMultipleMediaInfo(String[] key) {
   StringBuilder sb = new StringBuilder();
   sb.append("Search Results: \n\n");
   for (int i = 0; i < key.length; i++) {
      Media theMedia = mediaLibrary.get(key[i]);
      if (theMedia instanceof Dvd) {
         Dvd dvd = (Dvd) theMedia;
         sb.append("Title: " + dvd.getTitle() + "\n");
         sb.append("Id: " + dvd.getId() + "\n");
         sb.append("Year: " + dvd.getYear() + "\n");
         String[] actors = dvd.getActor();
         sb.append("Actor(s): ");
         for (int j = 0; j < actors.length; j++) {
            sb.append(actors[j]);
            if (j < actors.length - 1) {
               sb.append(", ");
            }
         }
      } else if (theMedia instanceof Book) {
         Book book = (Book) theMedia;
         sb.append("Title: " + book.getTitle() + "\n");
         sb.append("Id: " + book.getId() + "\n");
         sb.append("Year: " + book.getYear() + "\n");
         sb.append("Author(s): " + book.getAuthor());
      }
      sb.append("\n" + "----------------------------------------" + "\n");
   }
   mainWindow.updateMediaInfoField(sb.toString());
}
/**
 * This method sends the appropriate information connected to a certain media object
 * to the main window. There it is later displayed to the user.
 */
public void showSingleMediaInfo(String key) {
   Media theMedia = mediaLibrary.get(key);
   if (theMedia instanceof Dvd) {
      Dvd dvd = (Dvd) theMedia;
      StringBuilder sb = new StringBuilder();
      sb.append("Title: " + dvd.getTitle() + "\n");
      sb.append("Id: " + dvd.getId() + "\n");
      sb.append("Year: " + dvd.getYear() + "\n");
      String[] actors = dvd.getActor();
      sb.append("Actor(s): ");
      for (int i = 0; i < actors.length; i++) {
         sb.append(actors[i]);
         if (i < actors.length - 1) {
            sb.append(", ");
         }
      }
      mainWindow.updateMediaInfoField(sb.toString());
   } else if (theMedia instanceof Book) {
      Book book = (Book) theMedia;
      StringBuilder sb = new StringBuilder();
      sb.append("Title: " + book.getTitle() + "\n");
      sb.append("Id: " + book.getId() + "\n");
      sb.append("Year: " + book.getYear() + "\n");
      sb.append("Author(s): " + book.getAuthor());
      mainWindow.updateMediaInfoField(sb.toString());
   }
}
/**
 * This method searches the unloaned media objects and
 * either returns the information connected to an object
 * or a message that the keyword has no result.
 * @param str represents a keyword that is used for searching the library.
 */
public void showFoundMedia(String str) {
   String[] keyword = searchTitle(str);
   if (keyword.length == 0) {
      mainWindow.updateMediaInfoField("Sorry, no results for the keyword: " + str);
   } else {
      showMultipleMediaInfo(keyword);
   }
}
```

```
}
```

Dvd.java

```
package gu1;

import java.util.Arrays;
/**
 * A class that holds the id, author, title and year of publication
 * of a given Dvd object. The Dvd class extends Media.
 * @author Jesper Anderberg, Filip Nilsson, Aron Polner, Ali Hassan, Szilveszter Dezsi
 *
 */
public class Dvd extends Media{
   private String[] actors;
   /**
    * A constructor that takes four arguments
    * three of these, id, year and title
    * are sent to, and later handled by, the Media class.
    * @param id
    * @param titel
    * @param year
    * @param actors
    */
   public Dvd(String id, String titel, String year, String[] actors) {
      super(id, year, titel);
      this.actors = actors;
   }
   /**
    * getActor returns the actors connected to the current Dvd object.
    * @return a String array of the actors connected to the current Dvd object.
    */
   public String[] getActor() {
      return actors;
   }
   /**
    * A method that sets the actors of a particular Dvd object.
    * @param actors
    */
   public void setActor(String[] actors) {
      this.actors = actors;
   }
   /**
    * A toString method that returns all of the available informations
    * surrounding a given Dvd object.
    */
   public String toString() {
return "Dvd [id=" + this.getId() + ", titel=" + this.getTitle() + ", year=" + this.getYear() + ",
actors=" + Arrays.toString(actors) + "]";
   }
}
```

HashtableCH.java

```
package gu1;
import java.util.ArrayList;
import java.util.Iterator;

/**
 * HashtableCH is an re-sizeable array implementation of the Map interface.
 * HashtableCH maps keys to values using Bucket-objects, that also keep track of
 * the following states: EMPTY, OCCUPIED or REMOVED.
 * This map uses closed hashing and cannot contain duplicate keys,
 * each key can map to at most one value.
 * This map also uses inner class implementations of the Iterable interface to traverse values and
keys.
 *
 * @author Jesper Anderberg, Filip Nilsson, Aron Polner, Ali Hassan, Szilveszter Dezsi
 *
 * @param <K> the type of keys maintained by this map
 * @param <V> the type of values maintained by this map
 */

public class HashtableCH<K,V> implements Map<K,V> {
    private Bucket<K,V>[] table;
    private int size;
    private int threshold;
    private double loadfactor = 0.7;

    /**
     * Constructs an empty map with an inital size of 11.
     */
    public HashtableCH( ) {
       this(11);
    }

    /**
     * Constructs an empty map with specified inital size.
     * @param size initial size
     */
    public HashtableCH( int size ) {
       table = (Bucket<K, V>[]) new Bucket[size];
       threshold = (int) (loadfactor * table.length);
       for (int i = 0; i < table.length; i++) {
          table[i] = new Bucket<K, V>();
       }
   }

   /**
    * Grows (doubles in size) if this map is at capacity.
    */
   private void grow() {
       Bucket<K, V>[] oldTable = table;
       table = (Bucket<K, V>[]) new Bucket[table.length * 2];
       size = 0;
       threshold = (int) (loadfactor * table.length);
       for (int i = 0; i < table.length; i++) {
          table[i] = new Bucket<K, V>();
       }
       for (int index = 0; index < oldTable.length; index++) {
          if (oldTable[index].state == Bucket.OCCUPIED)
             put(oldTable[index].key, oldTable[index].value);
       }
   }

    /**
     * Returns the hashcode (index) for the value associated with the specified key.
     * @param key specified key
     * @return hashcode (index) for value
     */
   public int hashIndex(K key) {
       int hashCode = key.hashCode();
       hashCode = hashCode % table.length;
       return (hashCode < 0) ? -hashCode : hashCode;
```

```
    }

     /**
      * Associates the specified value with the specified key in this map.
      * If the map previously contained a mapping for the key, the old value
      * is replaced by the specified value.
      *
      * @param key key with which the specified value is to be associated
      * @param value value to be associated with the specified key
      * @return the previous value associated with key, or
      *         {@code null} if there was no mapping for key.
      */
    public V put(K key, V value) {
       V res = null;
       if (size >= threshold) {
          grow();
       }
       int hashIndex = hashIndex(key), removed = -1;
       while (table[hashIndex].state != Bucket.EMPTY && !key.equals(table[hashIndex].key)) {
          if (removed == -1 && table[hashIndex].state == Bucket.REMOVED)
             removed = hashIndex;
          hashIndex++;
          if (hashIndex == table.length)
             hashIndex = 0;
       }
       if (table[hashIndex].state == Bucket.OCCUPIED) {
          res = table[hashIndex].value;
          table[hashIndex].value = value;
       } else {
          if (removed != -1)
             hashIndex = removed;
          table[hashIndex].key = key;
          table[hashIndex].value = value;
          table[hashIndex].state = Bucket.OCCUPIED;
          size++;
       }
       return res;
    }

    /**
     * Prints each key-value mapping in this map.
     */
     public void list() {
       System.out.println("Tabellen innehåller " + size() + " element:");
       for (int i = 0; i < table.length; i++)
          System.out.println(i + ": key=" + table[i].key + " value=" + table[i].value + " state=" +
table[i].state);
    }

     /**
      * Returns the value to which the specified key is mapped,
      * or {@code null} if this map contains no mapping for the key.
      *
      * @param key the key whose associated value is to be returned
      * @return the value to which the specified key is mapped, or
      *         {@code null} if this map contains no mapping for the key
      */
    public V get(K key) {
       int hashIndex = hashIndex(key);
       while ((table[hashIndex].state != Bucket.EMPTY) && (!key.equals(table[hashIndex].key))) {
          hashIndex++;
          if (hashIndex == table.length)
             hashIndex = 0;
       }
       if (table[hashIndex].state == Bucket.OCCUPIED) {
          return table[hashIndex].value;
       }
       return null;
    }

     /**
      * Removes the mapping for a key from this map if it is present.
      * Returns the value to which this map previously associated the key,
     * or {@code null} if the map contained no mapping for the key.
```

```
  *
  * @param key key whose mapping is to be removed from the map
  * @return the previous value associated with key, or
  *         {@code null} if there was no mapping for key.
  */
public V remove(K key) {
   int hashIndex = hashIndex(key);
   while ((table[hashIndex].state != Bucket.EMPTY) && (!key.equals(table[hashIndex].key))) {
      hashIndex++;
      if (hashIndex == table.length) {
         hashIndex = 0;
      }
   }
   if (table[hashIndex].state == Bucket.OCCUPIED) {
      V tempValue = table[hashIndex].value;
      table[hashIndex].key = null;
      table[hashIndex].value = null;
      table[hashIndex].state = Bucket.REMOVED;
      size--;
      return tempValue;
   }
   return null;
}

 /**
  * Returns the number of key-value mappings in this map.
  * @return the number of key-value mappings in this map
  */
 public int size() {
     return size;
 }

 /**
  * Returns {@code true} if this map contains no key-value mappings.
  * @return {@code true} if this map contains no key-value mappings
  */
public boolean isEmpty() {
   return size()==0;
}

 /**
  * Returns {@code true} if this map contains a mapping for the specified key.
  *
  * @param key key whose presence in this map is to be tested
  * @return {@code true} if this map contains a mapping for the specified key
  */
public boolean contains(K key) {
   return get(key)!=null;
}

 /**
  * Removes all of the mappings from this map. The map will be empty
  * after this call returns.
  */
public void clear() {
   for (int i = 0; i < table.length; i++) {
      if (table[i].state != Bucket.EMPTY) {
         table[i].key = null;
         table[i].value = null;
         table[i].state = Bucket.EMPTY;
      }
   }
   size = 0;
}

 /**
  * Returns an Iterator over all the keys contained in this map.
  * @return an Iterator over all the keys contained in this map
  */
public Iterator<K> keys() {
   return new KeyIterator();
}

 /**
```

```
  * Inner class implementation of Iterator over keys
  */
private class KeyIterator implements Iterator<K> {
   private ArrayList<K> keys = new ArrayList<K>();
   private int listIndex = 0;

   /**
    * Constructs a new KeyIterator
    */
   public KeyIterator() {
      for (int i = 0; i < table.length; i++)
         if (table[i].state == Bucket.OCCUPIED)
            keys.add(table[i].key);
   }

   /**
    * Returns true if the iteration has more keys.
    * @return boolean
    */
   public boolean hasNext() {
      return listIndex < keys.size();
   }

   /**
    * Returns the next key in the iteration.
    * @return the next key in the iteration
    */
   public K next() {
      return keys.get(listIndex++);
   }

   /**
    * Remove operation is not supported by this iterator.
    */
   public void remove() {
      throw new UnsupportedOperationException();
   }
}

 /**
  * Returns an Iterator over all the values contained in this map.
  * @return an Iterator over all the values contained in this map
  */
public Iterator<V> values() {
   return new ValueIterator();
}

 /**
  * Inner class implementation of Iterator over values
  */
private class ValueIterator implements Iterator<V> {
   private ArrayList<V> values = new ArrayList<V>();
   private int listIndex = 0;

   /**
    * Constructs a new ValueIterator
    */
   public ValueIterator() {
      for (int i = 0; i < table.length; i++)
         if (table[i].state == Bucket.OCCUPIED)
            values.add(table[i].value);
   }

   /**
    * Returns true if the iteration has more values.
    * @return boolean
    */
   public boolean hasNext() {
      return listIndex < values.size();
   }

   /**
    * Returns the next value in the iteration.
    * @return the next value in the iteration
```

```
         */
        public V next() {
           return values.get(listIndex++);
        }

        /**
         * Remove operation is not supported by this iterator.
         */
        public void remove() {
           throw new UnsupportedOperationException();
        }
    }
}
```

Lantagare.txt

```
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
```

LibraryPanel.java

```
package gu1;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * The class LibraryPanel represents graphical interface a user is faced with
 * after a successful login a library application. It extends JPanel giving it
 * attributes of a JPanel.
 * @author Jesper Anderberg, Filip Nilsson, Aron Polner, Ali Hassan, Szilveszter Dezsi
 */
public class LibraryPanel extends JPanel implements ListSelectionListener, ActionListener, KeyListener
{

   private JPanel librarySide;
   private JPanel userSide;
   private JPanel center;
   private JPanel north;
   private JLabel librarySideLbl;
   private JLabel userSideLbl;
   private JLabel welcomeLbl;
   private JLabel middle;
   private JTextArea mediaInfoTxt;
   private JList<String> availableMedia;
   private JList<String> userLoans;
   private JPanel southGrid;
   private JButton makeLoanBtn;
   private JButton searchBtn;
   private JButton logOutBtn;
   private JButton returnLoanBtn;
   private JTextField searchText;
   private Controller controller;

   /**
    * Constructor receives an instance of the class Controller.
    * Calls for drawLibraryPanel which adds components to the JPanel.
    * @param controller
    */
   public LibraryPanel(Controller controller) {
      this.controller = controller;
      drawLibraryPanel();
   }

   private void drawLibraryPanel() {
      setLayout(new BorderLayout());
      this.returnLoanBtn = new JButton("Return Media");
      this.searchBtn = new JButton("Seach Media");
      this.logOutBtn = new JButton("Log Out");
      this.searchText = new JTextField();
      this.southGrid = new JPanel(new GridLayout(1, 3));
      this.librarySide = new JPanel();
      this.center = new JPanel(new BorderLayout());
      this.north = new JPanel(new BorderLayout());
      this.middle = new JLabel();
      this.librarySide.setLayout(new BorderLayout());
      this.userSide = new JPanel();
      this.userSide.setLayout(new BorderLayout());
      this.librarySideLbl = new JLabel("Available Media:");
      this.librarySideLbl.setBackground(Color.CYAN);
      this.librarySideLbl.setOpaque(true);
      this.librarySideLbl.setPreferredSize(new Dimension(320, 40));
      this.welcomeLbl = new JLabel();
      this.welcomeLbl.setOpaque(true);
      this.welcomeLbl.setBackground(Color.PINK);
      this.userSideLbl = new JLabel("User loans:");
      this.userSideLbl.setBackground(Color.MAGENTA);
      this.userSideLbl.setOpaque(true);
      this.userSideLbl.setPreferredSize(new Dimension(320, 40));
```

```
    this.availableMedia = new JList<String>();
    this.userLoans = new JList<String>();
    this.makeLoanBtn = new JButton("Loan");
    add(center, BorderLayout.CENTER);
    this.mediaInfoTxt = new JTextArea();
    this.mediaInfoTxt.setEditable(false);
    searchText.setText("Search for available titles...");

    southGrid.setBackground(Color.BLUE);
    southGrid.setOpaque(true);
    southGrid.add(makeLoanBtn);
    southGrid.add(searchBtn);
    southGrid.add(returnLoanBtn);
    middle.setOpaque(true);
    middle.setBackground(Color.ORANGE);
    center.add(mediaInfoTxt, BorderLayout.NORTH);
    center.add(middle, BorderLayout.CENTER);
    center.add(searchText, BorderLayout.SOUTH);

    librarySide.add(librarySideLbl, BorderLayout.NORTH);
    librarySide.add(availableMedia, BorderLayout.CENTER);
    userSide.add(userSideLbl, BorderLayout.NORTH);
    userSide.add(userLoans, BorderLayout.CENTER);

    mediaInfoTxt.setBackground(Color.ORANGE);
    availableMedia.setBackground(Color.GREEN);
    userLoans.setBackground(Color.YELLOW);

    welcomeLbl.setHorizontalAlignment(SwingConstants.CENTER);
    welcomeLbl.setPreferredSize(new Dimension(getWidth(), 40));
    makeLoanBtn.setPreferredSize(new Dimension(getWidth(), 60));
    north.add(welcomeLbl, BorderLayout.CENTER);
    north.add(logOutBtn, BorderLayout.EAST);
    add(north, BorderLayout.NORTH);
    add(center, BorderLayout.CENTER);

    add(southGrid, BorderLayout.SOUTH);
    librarySide.setPreferredSize(new Dimension(320, getHeight()));
    add(librarySide, BorderLayout.WEST);
    userSide.setPreferredSize(new Dimension(320, getHeight()));
    add(userSide, BorderLayout.EAST);

    makeLoanBtn.addActionListener(this);
    searchBtn.addActionListener(this);
    returnLoanBtn.addActionListener(this);
    userLoans.addListSelectionListener(this);
    availableMedia.addListSelectionListener(this);
    logOutBtn.addActionListener(this);
    searchText.addKeyListener(this);
}

/**
 * Displays the name of the current user.
 * @param name , name to be displayed.
 */
public void fillUserName(String name) {
    welcomeLbl.setText("Welcome: " + name);
}

/**
 * Method receives an array of Media objects that are available for loan and displays them.
 * @param availableMedia , Media array
 */
public void fillAvailableMediaList(String[] availableMedia) {
    DefaultListModel<String> availableMediaListModel = new DefaultListModel<String>();
    for (int i = 0; i < availableMedia.length; i++) {
        if (availableMedia[i] != null) {
            availableMediaListModel.addElement(availableMedia[i]);
        }
    }
    this.availableMedia.setModel(availableMediaListModel);
}

/**
```

```
     * Method receives an array of Media objects corresponding to the loans of a certain user.
     * @param userLoans , Media array
     */
    public void fillUserLoanList(String[] userLoans) {
        DefaultListModel<String> userLoanListModel = new DefaultListModel<String>();
        for (int i = 0; i < userLoans.length; i++) {
            userLoanListModel.addElement(userLoans[i]);
        }
        this.userLoans.setModel(userLoanListModel);
    }

    /**
     * Receives the info of a particular media and displays it.
     * @param mediaInfo , info of a particular media to display.
     */
    public void showMediaInfo(String mediaInfo) {
        mediaInfoTxt.setText(mediaInfo);
    }

    /**
     * Called upon when the user selects something in either the media list or the userloan list.
     * Shows info of a certain media title that is selected in one of the lists.
     */
    public void valueChanged(ListSelectionEvent e) {
        if (e.getValueIsAdjusting()) {
            if (e.getSource() == availableMedia) {
                controller.showSingleMediaInfo(availableMedia.getSelectedValue().substring(0, 6));
                userLoans.clearSelection();
            } else if (e.getSource() == userLoans) {
                controller.showSingleMediaInfo(userLoans.getSelectedValue().substring(0, 6));
                availableMedia.clearSelection();
            }
        }
    }

    /**
     * Called upon when the user presses one of the buttons on the GUI.
     */
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == makeLoanBtn && !availableMedia.isSelectionEmpty()) {
            String mediaID = availableMedia.getSelectedValue().substring(0, 6);
            controller.borrowMedia(mediaID);
        } else if (e.getSource() == returnLoanBtn && !userLoans.isSelectionEmpty()) {
            String mediaID = userLoans.getSelectedValue().substring(0, 6);
            controller.returnMedia(mediaID);
        } else if (e.getSource() == searchBtn) {
            String input = searchText.getText();
            controller.showFoundMedia(input);
        } else if (e.getSource() == logOutBtn) {
            controller.logOut();
        }
    }
    /**
     * Not used.
     */
    public void keyTyped(KeyEvent e) {
    }

    /**
     * Called upon when the user presses Enter on the keyboard.
     * Does the same as when Search Media is pressed i.e searches for a media and prints the result.
     */
    public void keyPressed(KeyEvent e) {
        if ((e.getKeyCode() == KeyEvent.VK_ENTER)) {
            String input = searchText.getText();
            controller.showFoundMedia(input);
        }
    }
    /**
     * Not used.
     */
    public void keyReleased(KeyEvent e) {
    }
}
```

LogInPanel.java

```
package gu1;

import java.awt.*;
import java.awt.event.*;

import javax.swing.*;

/**
 * The class LogInPanel represents the login-screen of a library application.
 * It extends JPanel giving it attributes of a JPanel.
 *
 * @author Jesper Anderberg, Filip Nilsson, Aron Polner, Ali Hassan, Szilveszter Dezsi
 *
 */
public class LogInPanel extends JPanel implements ActionListener {
   private MediaViewer mainFrame;
   private JLabel logInLbl;
   private JTextField logInTxt;
   private JButton logInBtn;

   /**
    * Constructor receives an instance of MediaViewer.
    * Calls for drawLoginPanel which adds components to the JPanel.
    * @param mainFrame , MediaViewer extends JFrame
    */
   public LogInPanel(MediaViewer mainFrame){
      this.mainFrame = mainFrame;
      drawLoginPanel();
   }

   private void drawLoginPanel(){
      setBackground(Color.PINK);
      logInLbl = new JLabel("Enter login");
      logInTxt = new JTextField();
      logInTxt.setPreferredSize(new Dimension(100, 20));
      logInBtn = new JButton("Press to login");
      setLayout(new GridBagLayout());
      add(logInLbl);
      add(logInTxt);
      add(logInBtn);
      logInBtn.addActionListener(this);
   }

   /**
    * Called upon when the JButton logInBtn is pressed.
    * When the button is pressed the logInPressed method within the MediaViewer is called upon.
    */
   public void actionPerformed(ActionEvent e) {
      if(e.getSource()==logInBtn){
         mainFrame.logInPressed(logInTxt.getText());
      }
   }
}
```

MainLauncher.java

```
package gu1;

import javax.swing.SwingUtilities;

public class MainLauncher {

   public static void main(String[] args) {
      SwingUtilities.invokeLater(new Runnable() {
         public void run() {
            Controller controller = new Controller("files/Media.txt", "files/Lantagare.txt");
         }
      });
   }
}
```

Map.java

```
package gu1;
import java.util.Iterator;
/**
 * An object that maps keys to values.  A map cannot contain duplicate keys;
 * each key can map to at most one value.
 * @param <K> the type of keys maintained by this map
 * @param <V> the type of mapped values
 */
public interface Map<K,V> {
    /**
     * Returns the number of key-value mappings in this map.
     * @return the number of key-value mappings in this map
     */
    int size();
    /**
     * Returns <tt>true</tt> if this map contains no key-value mappings.
     * @return <tt>true</tt> if this map contains no key-value mappings
     */
    boolean isEmpty();
    /**
     * Returns <tt>true</tt> if this map contains a mapping for the specified
     * key.
     * @param key key whose presence in this map is to be tested
     * @return <tt>true</tt> if this map contains a mapping for the specified key
     */
    boolean contains(K key);
    /**
     * Returns the value to which the specified key is mapped,
     * or {@code null} if this map contains no mapping for the key.
     * @param key the key whose associated value is to be returned
     * @return the value to which the specified key is mapped, or
     *         {@code null} if this map contains no mapping for the key
     */
    V get(K key);
    /**
     * Associates the specified value with the specified key in this map
     * (optional operation).  If the map previously contained a mapping for
     * the key, the old value is replaced by the specified value.
     * @param key key with which the specified value is to be associated
     * @param value value to be associated with the specified key
     * @return the previous value associated with <tt>key</tt>, or
     *         <tt>null</tt> if there was no mapping for <tt>key</tt>.
     */
    V put(K key, V value);
    /**
     * Removes the mapping for a key from this map if it is present.
     * Returns the value to which this map previously associated the key,
* or <tt>null</tt> if the map contained no mapping for the key.
     * @param key key whose mapping is to be removed from the map
     * @return the previous value associated with <tt>key</tt>, or
     *         <tt>null</tt> if there was no mapping for <tt>key</tt>.
     */
    V remove(K key);
    /**
     * Removes all of the mappings from this map. The map will be empty
     * after this call returns.
     */
    void clear();
    /**
     * Returns an Iterator<K> view of the keys contained in this map.
     * @return an Iterator<K>-view of the keys contained in this map
     */
    Iterator<K> keys();

    /**
     * Returns an Iterator<V> view of the values contained in this map.
     * @return an Iterator<V>-view of the values contained in this map
     */
    Iterator<V> values();
}
```

Media.txt

```
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
```

Media.java

```
package gu1;

/**
 * Media is an abstract superclass for media type classes such as Book and Dvd.
 * Media stores id-number, year of publication, title for subclasses and boolean for if an object is
borrowed.
 *
 * @author Jesper Anderberg, Filip Nilsson, Aron Polner, Ali Hassan, Szilveszter Dezsi
 */
public abstract class Media {
   private String id;
   private String year;
   private String title;
   private boolean borrowed = false;

   /**
    * Constructor
    * @param id id-number
    * @param year publication year
    * @param title title
    */
   public Media(String id, String year, String titel) {
      this.id = id;
      this.year = year;
      this.title = titel;
   }

   /**
    * Returns borrowed status
    * @return borrowed boolean
    */
   public boolean isBorrowed() {
      return borrowed;
   }

   /**
    * Sets borrowed status
    * @param borrowed boolean
    */
   public void setBorrowed(boolean borrowed) {
      this.borrowed = borrowed;
   }

   /**
    * Returns the year of publication
    * @return publication year
    */
   public String getYear() {
      return year;
   }

   /**
    * Sets the year of publication
    * @param year publication year
    */
   public void setYear(String year) {
      this.year = year;
   }

   /**
    * Returns the title
    * @return title title
    */
   public String getTitle() {
      return title;
   }

   /**
    * Sets the title
    * @param year publication year
```

```
     */
    public void setTitel(String titel) {
        this.title = titel;
    }

    /**
     * Returns the id-number
     * @return id-number
     */
    public String getId() {
        return id;
    }

    /**
     * Sets the id-number
     * @param id id-number
     */
    public void setId(String id) {
        this.id = id;
    }
}
```

MediaLibrary.java

```
package gu1;
import java.util.Iterator;

/**
 * MediaLibrary is a library class that handles Media objects, mapping them with ids using a hastable.
 * @author Jesper Anderberg, Filip Nilsson, Aron Polner, Ali Hassan, Szilveszter Dezsi
 */
public class MediaLibrary {
   private HashtableCH<String, Media> mediaLib = new HashtableCH<String, Media>();

   /**
    * Returns the number of Media objects in this library.
    * @return the number of Media objects in this library
    */
   public int size(){
      return mediaLib.size();
   }

   /**
    * Associates the specified Media object with a specified id in this library.
    * If the library previously contained a mapping for the id, the old Media object
    * is replaced by the specified Media object.
    * @param id String ID to be associated with Media object
    * @param media Media object
    * @return the previous Media object associated with id, or
    *         {@code null} if there was no mapping for id.
    */
   public Media put(String id, Media media){
      return mediaLib.put(id, media);
   }

   /**
    * Returns the Media object to which the specified id is mapped,
    * or {@code null} if this library contains no mapping for the id.
    * @param id String ID for Media object to be returned
    * @return the Media object to which the specified id is mapped, or
    *         {@code null} if this library contains no mapping for the id
    */
   public Media get(String id){
      return mediaLib.get(id);
   }

   /**
    * Removes the mapping for an id from this library if it is present.
    * Returns the Media object to which this library previously associated the id,
    * or {@code null} if the library contained no mapping for the id.
    * @param id String ID for Media object to be removed
    * @return the previous Media object associated with id, or
    *         {@code null} if there was no mapping for id.
    */
   public Media remove(String id){
      return mediaLib.remove(id);
   }

   /**
    * Returns {@code true} if library contains a mapping for the specified id.
    * @param id String ID for Media object whose presence in this library is to be tested
    * @return {@code true} if this library contains a mapping for the specified id
    */
   public boolean contains(String id){
      return mediaLib.contains(id);
   }
   /**
    * Returns an Iterator over all the Media objects contained in this library.
    * @return an Iterator over all the Media objects contained in this library
    */
   public Iterator<Media> availableMedia(){
      return mediaLib.values();
   }
}
```

MediaViewer.java

```
package gu1;

import java.awt.*;
import javax.swing.*;

/**
 * The class MediaViewer represents the graphical interface of the Library
 * program. The class uses two instances of JPanel(LogInPanel & LibraryPanel) to
 * display different kinds of content. A Controller instance decides which of
 * these two panels to show.
 *
 *
 * @author Jesper Anderberg, Filip Nilsson, Aron Polner, Ali Hassan, Szilveszter
 *         Dezsi
 *
 */
public class MediaViewer extends JFrame {
   private Controller controller;
   private LogInPanel logInPanel;
   private LibraryPanel libraryPanel;

   /**
    * The constructor draws the frame together with adding an instance of
    * LogInPanel and an instance of LibraryPanel. When constructed, only the
    * LogInPanel is set to visible and not the LibraryPanel.
    *
    * @param controller
    */
   public MediaViewer(Controller controller) {
      this.controller = controller;
      setSize(1200, 700);
      setVisible(true);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setLayout(new CardLayout());
      drawLoginPanel();
      drawLibraryPanel();
   }

   private void drawLoginPanel() {
      logInPanel = new LogInPanel(this);
      add(logInPanel);
   }

   private void drawLibraryPanel() {
      libraryPanel = new LibraryPanel(controller);
      add(libraryPanel);
      libraryPanel.setVisible(false);
   }

   /**
    * Calls for the method login in the class controller.
    *
    * @param idEntered
    */
   public void logInPressed(String idEntered) {
      controller.login(idEntered);
   }

   /**
    * Sets the LoginPanel to invisible and the LibraryPanel to visible. Also
    * passes a userName to the LibraryPanel.
    *
    * @param name
    */
   public void loginToLibraryPanel(String name) {
      logInPanel.setVisible(false);
      libraryPanel.fillUserName(name);
      libraryPanel.setVisible(true);
   }
```

```
    /**
     * Used to display the available media and the loans of a specific user.
     *
     * @param availableMediaList
     *            , media-array to display as available.
     * @param userLoanList
     *            , media-array to show the logged in users loans.
     */
    public void updateMediaLists(String[] availableMediaList, String[] userLoanList) {
        libraryPanel.fillAvailableMediaList(availableMediaList);
        libraryPanel.fillUserLoanList(userLoanList);
    }

    /**
     * Recieves the info of a specific media as a String and passes it to the
     * method showMediaInfo in the class LibraryPanel.
     *
     * @param media
     *            , information of a certain media within the library.
     */
    public void updateMediaInfoField(String media) {
        libraryPanel.showMediaInfo(media);
    }

    /**
     * Sets the logInPanel to visible and the libraryPanel to invisible.
     */
    public void libraryToLoginPanel() {
        logInPanel.setVisible(true);
        libraryPanel.setVisible(false);
    }
}
```

SearchTree.java

```
package gu1;
import java.util.Iterator;
import java.util.List;
/**
 * An object that maps keys to values in a tree structure.
 *
 * @param <K> specified key data reference type
 * @param <V> specified value data reference type
 */
public interface SearchTree<K,V> {
    /**
     * Associates the specified value with the specified key in this tree.
     * @param key - key with which the specified value is to be associated
     * @param value - value to be associated with the specified key
     */
    public void put(K key, V value);
    /**
     * Removes the value associated with the specified key in this tree, if present.
     * @param key - specified key
     * @return removed value, or <tt>null</tt> if not present in this tree
     */
    public V remove(K key);
    /**
     * Returns the value associated with the specified key in this tree, if present.
     * @param key - specified key
     * @return value associated with the specified key, or <tt>null</tt> if not present in this tree
     */
    public V get(K key);
    /**
     * Returns <tt>true</tt> if this tree contains specified key, otherwise <tt>false</tt>.
     * @param key - specified key
     * @return <tt>true</tt> if this tree contains specified key, otherwise <tt>false</tt>
     */
    public boolean contains(K key);
    /**
     * Returns the number of levels of this tree.
     * @return number of levels of this tree
     */
    public int height();
    /**
     * Returns an iterator over the values contained in this tree.
     * @return an iterator over the values contained in this tree
     */
    public Iterator<V> iterator();
    /**
     * Returns the number of key-value nodes contained in this tree.
     * @return the number of key-value nodes contained in this tree
     */
    public int size();
    /**
     * Returns a list of the keys contained in this tree.
     * @return list of the keys contained in this tree
     */
    public List<K> keys();
    /**
     * Returns a list of the values contained in this tree.
     * @return list of the values contained in this tree
     */
    public List<V> values();
    /**
     * Returns the first (lowest) value contained in this tree.
     * @return the first (lowest) value contained in this tree
     */
    public V first();
    /**
     * Returns the last (highest) value contained in this tree.
     * @return the last (highest) value contained in this tree
     */
    public V last();
}
```

User.java

```
package gu1;

import java.util.LinkedList;
/**
 * A class that holds the the infomation of a library borrower such a
 * name, Id, phone number and the loans of a boroower.
 * @author Jesper Anderberg, Filip Nilsson, Aron Polner, Ali Hassan, Szilveszter Dezsi
 *
 */
public class User {
   private String id;
   private String name;
   private String phoneNr;
   private LinkedList<Media> loans;

   /**
    * A constructor that takes three arguments
    *the Id , name and the phone number of the library borrower.
    * constructor initietes the loans of the borrower.
    * @param id
    * @param name
    * @param phoneNr
    * @param loans
    */
   public User(String id, String name, String phoneNr) {
      this.id = id;
      this.name = name;
      this.phoneNr = phoneNr;
      this.loans = new LinkedList<Media>();
   }
   /**
    * A method that returns the media object
    * and removes the object from the borrowers loan list.
    * @return the index of the removed media.
    */
   public Media returnMedia(Media media){
      return loans.remove(loans.indexOf(media));
   }
   /**
    * A method that allows the borrower to borrow a media object
    * and adds the object to the borrowers loan list .
    * @param media the object to be borrowed.
    */
   public void borrowMedia(Media media){
      loans.add(media);
   }
   /**
    * A method that returns the borrowers loan list
    * @return the loans.
    */
   public LinkedList<Media> loans() {
      return loans;
   }
   /**
    * A method that returns the ID of borrower.
    * @return Id of a borrower.
    */
   public String getId() {
      return id;
   }
   /**
    * A method that changes the ID of borrower.
    * @param id the id of a specified
    * library borrower.
    */
   public void setId(String id) {
      this.id = id;
   }
```

```
   /**
    * A method that returns the name of borrower.
    * @return a name of a borrower.
    */
   public String getName() {
      return name;
   }
   /**
    * A method that changes the name of borrower.
    * @param name the name of a specified
    * library borrower.
    */
   public void setName(String name) {
      this.name = name;
   }
   /**
    * A method that returns the phone number of borrower.
    * @return a phone number of a borrower.
    */
   public String getPhoneNr() {
      return phoneNr;
   }
   /**
    * A method that changes the phone number of borrower.
    * @param phoneNr the phone nember
    *  of a specified library borrower.
    */
   public void setPhoneNr(String phoneNr) {
      this.phoneNr = phoneNr;
   }
   /**
    * A toString method that returns all of the available informations
    *  a libray borrower.
    */
   public String toString() {
return "User [id=" + id + ", name=" + name + ", phoneNr=" + phoneNr + ", loans=" + loans.toString() +
"]";
   }
}
```