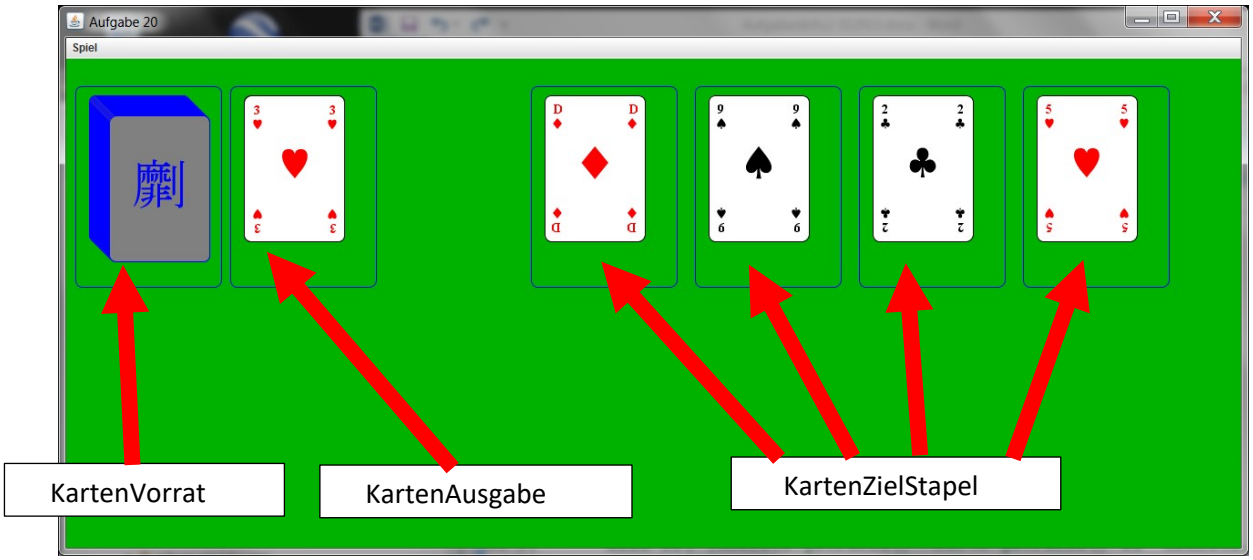
Bitte geben Sie Aufgabe26.java und Suchbaum.java bis zum 10.01.2021 ab!

Aufgabe 24

Solitär Kartenspiel fertig

In dieser Aufgabe stellen Sie das Kartenspiel fertig und können es zu Weihnachten verschenken.

Erstellen Sie zunächst eine Kopie des Pakets *aufgabe22* und nennen Sie es ***aufgabe24***. Die Klasse *Aufgabe22* nennen Sie in ***Aufgabe24*** um. Zur besseren Übersicht sind im Bild nochmal die wichtigsten Klassen gezeigt:



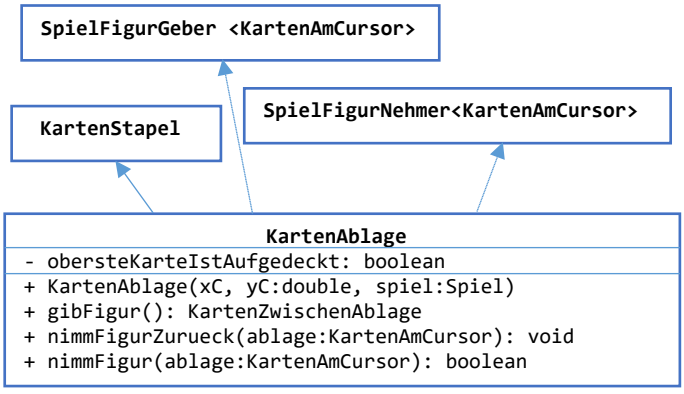
In der Klasse *SpielKarte* fügen Sie bitte zunächst folgende Methoden hinzu:

`+istRot():boolean,` welche *true* zurückgibt, wenn die Farbe der Spielkarte Karo oder Herz ist.
`+getZeigeVorderseite():boolean`, welche *zeigeVorderseite* zurückgibt.

Und zur Klasse ***KartenStapel*** die Methode `+empty(): boolean,` die *stapel.empty()* zurückgibt

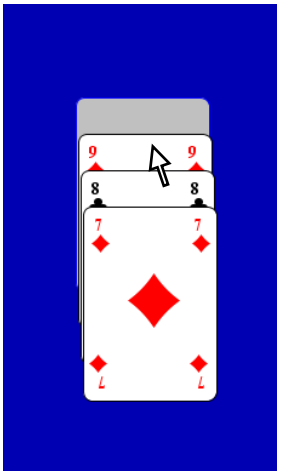
Die Kartenstapel für die Ablage

Für die sieben Kartenstapel im unteren Bereich des Spiels ist die neue Klasse ***KartenAblage*** zuständig. Diese Klasse ist von *KartenStapel* abgeleitet und implementiert sowohl *SpielFigurNehmer<KartenAmCursor>* als auch *SpielFigurGeber<KartenAmCursor>*.



Sie finden die **fertige Klasse** im persönlichen Stundenplan im Verzeichnis *aufgabe24*, damit Sie die Aufgabe bis Weihnachten fertigstellen können. In dieser Klasse stecken die meisten Spielregeln des Solitär-Spiels.

Der **Konstruktor** setzt *deltaX* auf 2, *deltaY* auf 30 sowie *height* auf 500 und *zeigeRand* auf false.



Die Methode ***gibFigur*** prüft, über welcher Karte der Cursor steht und hebt diese sowie alle darüber liegenden Karten ab. Dabei ist nur erlaubt, aufgedeckte Karten abzunehmen.

Dazu durchsucht sie den Stapel von oben nach der ersten aufgedeckten Karte unter dem Cursor. Im rechts gezeigten Bild ist das die Karo-9.

Alle Karten bis zu dieser Karte werden in eine neue *KartenAmCursorObjekt* eingefügt, welche die Methode zurückgibt. Es kann aber sein, dass der Cursor über keiner gültigen Karte steht, weil der Benutzer zwar innerhalb der Umrandung der Spielkartenablage geklickt hat, aber keine der aufgedeckten Karten erwischt hat. In diesem Fall wird *null* zurückgegeben.

Um dieses Verhalten zu implementieren, geht ***gibFigur*** wie folgt vor (Wichtig: die Kartenstapel erben von *SpielFigur* bzw. *Rectangle2D* eine Methode ***isEmpty().*** Bitte verwechseln Sie diese Methode nicht mit der Methode *empty()* zur Abfrage, ob der Kartenstapels leer ist!):

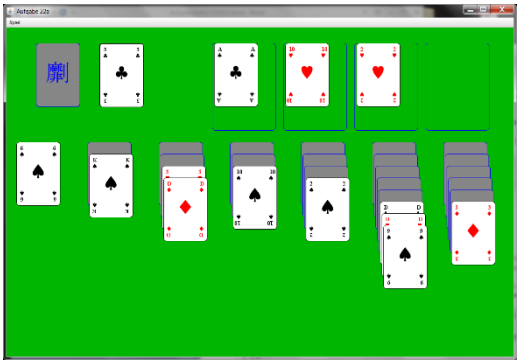
- Sie definiert die lokale boolean-Variable ***karteGefunden*** mit dem Anfangswert *false* und die lokale Variable ***ablage*** vom Typ *KartenAmCursor*, die mit dem Standardkonstruktor initialisiert wird.
- In der while-Schleife über alle aufgedeckten Karten:
  ```
  while(this.peek()!=null && this.peek().getZeigeVorderseite()){
    …
  ```
  wird das oberste Element vom eigenen Stapel genommen und auf *ablage* gelegt. Dann wird geprüft, ob der Cursor auf diese Karte zeigt:
  ```
  if(ablage.peek().contains(spiel.getMousePosition())) {
  ```
  Wenn ja, haben wir die Karte gefunden, welche abgehoben werden soll. Diese sowie alle Karten welche über dieser Karte lagen, befinden sich schon im Stapel *ablage*. Also wird *karteGefunden* auf *true* gesetzt und die *while*-Schleife mit *break* Verlassen.
- Nach der Schleife wird geprüft, ob *karteGefunden true* ist.
  Wenn **ja** müssen wir noch dafür sorgen, dass die oberste auf dem Stapel verbliebene Karte aufgedeckt wird, aber vorher merken wir uns, ob sie verdeckt war oder nicht, falls wir die Karten zurücknehmen müssen. Dazu prüfen wir, ob es noch Karten auf unserem Stapel gibt (`if(!stapel.empty()){…}`) und merken uns in der Instanzvariable *obersteKarteIstAufgedeckt*, ob die oberste verbleibende Karte aufgedeckt ist. Anschließend wird die oberste Karte mit *setZeigeVorderseite(true)* aufgedeckt. Nach der if-Anweisung wird *ablage* zurückgegeben.
  Wenn **nein**, legen wir in einer *while* Schleife alle in *ablage* vorhandene Karten wieder auf *this.stapel* und geben *null* zurück.

Die Methode ***nimmFigurZurueck*** prüft zunächst, ob auf dem eigenen Stapel eine Karte liegt und setzt deren Sichtbarkeit wieder auf den Wert *obersteKarteIstAufgedeckt*. Dann nimmt sie in einer *while*-Schleife alle angebotenen Karten aus *ablage* und legt sie auf den eigenen Stapel.

Die Methode ***nimmFigur*** führt eine Schleife über alle angebotenen Karten durch und legt sie auf den eigenen Stapel. Dabei wird hier die Regel implementieren, in welcher Reihenfolge Karten an die Ablagen angelegt werden dürfen:

- Entweder der Stapel ist leer,
- oder *istRot()* für die angebotene Karte ist ungleich *istRot()* für die oberste Karte und der Wert der angebotenen Karte ist um eins geringer als der Wert der Karte.

In ***Aufgabe24*** definieren wir als Instanzvariable einen Array vom Typ *KartenAblage*. In **initialisiere** erzeugen wir den Array der Länge 7 und in einer Schleife die *KartenAblage*-Objekte an der Position (100+180*i, 400) und registrieren sie sowohl als Nehmer als auch als Geber für den Drag-and-Drop Mechanismus. Dann holen wir für jede Ablage Karten vom Kartenvorrat, und zwar eine Karte für den ersten Stapel, zwei für den zweiten u.s.w. Die Oberste Karte jeder Ablage wird aufgedeckt.



In ***zeichneSpielstand*** werden die Ablagen dargestellt.

Unser Spiel sieht jetzt schon fertig aus, allerdings sind die Regeln noch nicht ganz implementiert und man kann von den Ablagestapeln immer nur die oberste Karte abheben.

Jetzt prüfen Sie am Ende von ***KartenVorrat.mousePressed***, ob der eigene Stapel leer ist. Wenn ja, holen Sie alle Karten von *kartenAusgabe* auf den Vorrat. So kann der Spieler mehrfach durch die noch vorhandenen Karten gehen.

Die restlichen Regeln für das Patience Spiel

Bevor Sie die restlichen Regeln implementieren, probieren Sie das Spiel eine Weile aus um sicherzugehen, dass die Mechanik funktioniert.

In ***KartenZielStapel.nimmFigur*** erweitern wir die Prüfung wie folgt:

- Wenn der Stapel leer ist, akzeptieren wir nur ein As (Wert 0), sonst geben wir *false* zurück.
- Wenn der Stapel Karten enthält, muss die angebotene Karte die gleiche Farbe und den nächst höheren Wert haben wie die oberste Karte.

Testen ob das Spiel gewonnen ist

Programmieren Sie ***Aufgabe24*** *die überschriebene Methode* ***mouseReleased***, darin prüfen Sie, ob das Spiel gewonnen ist (Vorrat, Ausgabe und alle Ablagen sind leer, was Sie mit *peek()==null* überprüfen). Wenn ja, geben Sie eine entsprechende Meldung (*JOptionPane.showMessageDialog*), und rufen *gameOver.*

Damit sollten Sie ein funktionierendes Solitär Spiel haben! Diese Anwendung macht extensiv Gebrauch von der Datenstruktur Stack. Die einzelnen Klassen sind überschaubar und gut abgegrenzt.

Sie können jetzt wie in Aufgabe 2 beschrieben eine Jar-Datei erzeugen, in die Sie das Spiel packen um es an Freunde oder Familie weiterzugeben.

Aufgabe 25

Suchbäume

a Erzeugen Sie den Suchbaum aus den Buchstaben des Wortes GEISTERHAFT und geben Sie seine Pre-Order an. Erfüllt dieser Baum die AVL Bedingung?
b Erzeugen Sie den binären Suchbaum aus den Buchstaben des Wortes „KERNIG“. Prüfen Sie, ob die AVL Bedingung überall erfüllt ist. Geben Sie für den Baum die Pre-, Post- und Level-Order an.
c Erzeugen Sie den Suchbaum aus den Buchstaben des Wortes AUSGANG und geben Sie seine Level-Order an.
d Wiederholen Sie die Schritte a-c für verschiedene Zeichenfolgen.

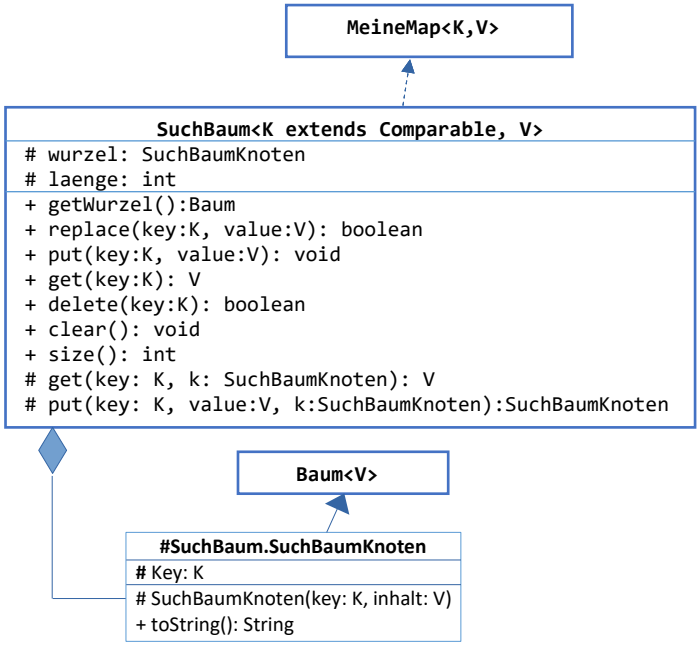
Aufgabe 26

Suchbaum

Wir schreiben die Klasse ***baeume.SuchBaum***, welche unser Interface *MeineMap* implementiert mit der Einschränkung, dass die Schlüssel *Comparable* sein müssen. Ähnlich wie unsere Klasse *MeineHashMap* hat auch *SuchBaum* eine innere Klasse für die Knoten des Baums.

Beginnen Sie mit der von *Baum* abgeleiteten inneren Klasse ***protected SuchbaumKnoten*****:**

- Der Konstruktor ruft *super(inhalt)* und speichert *key* in seiner Instanzvariable. Dann inkrementiert er den wert von ***laenge*** (auf diese Variable der äußeren Klasse hat die innere Klasse Zugriff).
- ***toString*** gibt *key.toString()* zurück.



Die Methoden von ***SuchBaum*** machen Folgendes:

getWurzel gibt *wurzel* zurück.

size gibt *laenge* zurück.

delete implementieren wir erst später.

get(K key) wirft eine *IllegalArgumentException* falls *key* null ist und ruft *get(key, wurzel)*.

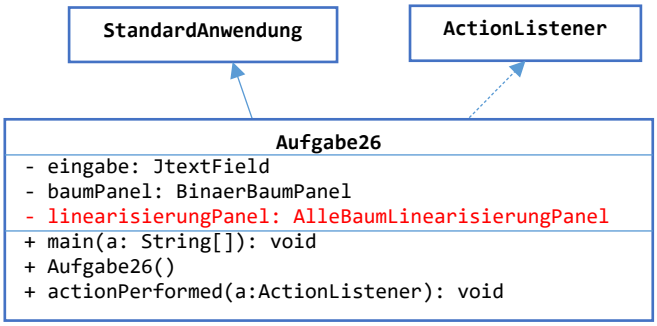
get(K key, SuchBaumKnoten k) führt folgende Schritte durch:

- Wenn der Parameter *k* null ist, wird null zurückgegeben,
- wenn der Parameter *key* gleich *k.key* ist (*compareTo* verwenden!), wird *k.inhalt* zurückgegeben,
- falls *key* kleiner als *k.key ist,* wird rekursiv im linken Teilbaum gesucht:
  return get(key, (SuchBaumKnoten)k,links);
  (der Cast ist notwendig, weil die von *Baum* geerbte Variable *links v*om Typ *Baum* ist),
- ansonsten wird rekursiv im rechten Teilbaum gesucht.

put(K key, V value) wirft eine *IllegalArgumentException, f*alls *key* oder *inhalt null* sind, *danach setzt sie wurzel = put(key, value, wurzel);*

put(K key, V value, SuchBaumKnoten k) führt folgende Schritte durch:

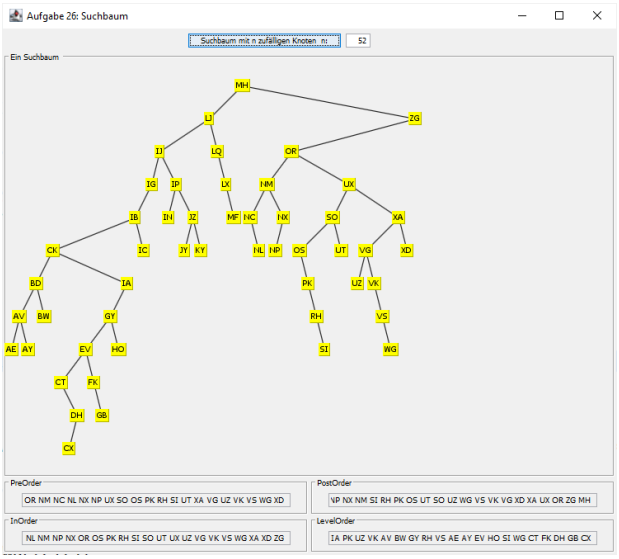
- *Wenn k null ist, wird ein neuer SuchBaumKnoten zurückgegeben,*
- wenn *key* gleich *k.key i*st (*compareTo* verwenden!), wird *k.inhalt* zu *value* gesetzt,
- wenn *key* kleiner als *k.key* ist, wird rekursiv der linke Teilbaum von *k* neu gesetzt:
  k.setLinks(put(key, value, SuchBaumKnoten)k.links));
  wobei der Cast wieder notwendig ist, weil *links* von *Baum* geerbt wurde.
- ansonsten wird analog der rechte Teilbaum gesetzt,
- am Ende wird *k* zurückgegeben.



replace gibt *false* zurück, wenn *get(key) null* liefert, ansonsten ruft sie *put* und gibt *true* zurück.

Zum Testen des Suchbaums erzeugen Sie bitte zunächst eine Kopie von *Aufgabe23* , nennen sie ***Aufgabe26*** und passen die Texte an.

Die Methode ***actionPerformed*** wird erheblich einfacher: In einer Schleife über die Zahlen von 1 bis zur eingegebenen Zahl werden zufällige Zeichenketten als Schlüssel in einen Suchbaum eingetragen. Dem Binärbaum-Panel wird die Wurzel des Baumes übergeben, die man mit *getWurzel()* erhält.
Das Beispiel zeigt eine Suchbaum mit 52 zufälligen Zeichenfolgen.

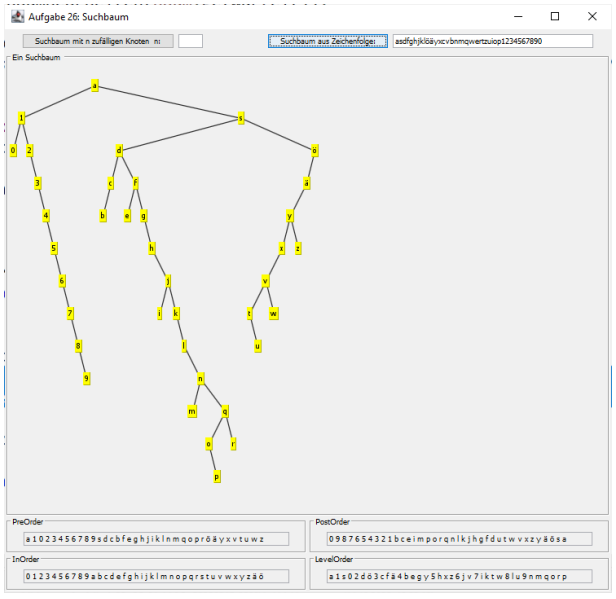


Eine zusätzliche Option ist ein Suchbaum aus eingegebenen Zeichen.

Programmieren Sie dazu die von *JButton* abgeleitete innere Klasse ***Aufgabe26.BaumAusStringButton*** mit der Aufschrift „*Suchbaum aus Zeichenfolge:*“, die bei Klick die Zeichen aus einem Eingabefeld (JTextField) liest und in einen Suchbaum speichert.

In ***actionPerformed*** holen Sie den Text aus dem Eingabefeld und wandeln ihn mit der String-Methode *toCharArray* in einen Character-Array um. In einer vereinfachten for-Schleife werden dann alle druckbaren Zeichen in einen *SuchBaum<Character,Character>* gespeichert:

```
for(char c:text.toCharArray()){
      if(!Character.isWhitespace(c)){
            baum.put(c, c);
      }
}
```



Diese App ist gut dazu geeignet, um das Erzeugen von Suchbäumen zu üben. Und mit minimalen Änderungen können wir sie später für AVL Bäume verwenden.

Aufgabe 27

Die Klasse File

Laden Sie sich die Klasse ***FileChooserBeispiel*** aus dem persönlichen Stundenplan herunter und lesen sie aufmerksam durch.

Erweitern Sie die Anzeige so, dass ein Verzeichnis rekursiv durchlaufen und angezeigt wird. Schreiben Sie dazu die Methode zeigeVerzeichnis(verzeichnis: File, ausgabe:JTextArea): void, die den Inhalt von verzeichnis darstellt und sich für jedes gefundene Verzeichnis selbst ruft.

Lassen Sie die Länge bei Verzeichnissen weg und fügen Sie noch weitere Informationen zu den Einträgen hinzu, z.B. wann eine Datei das letzte Mal geändert wurde.

Potentielle Prüfungsaufgaben

Programmieren Sie einfache Methoden zum Binärbaum, z.B.:

- int tiefe()
- bolean istBlatt()
- T getVaterInhalt()
- Int anzahlKinder()

Üben Sie das Erzeugen von Such- und AVL Bäumen.