**Bitte geben Sie Heap.java, Aufgabe35 und Aufgabe36.java bis zum 31.01.2021 ab!**

## Aufgabe 33

**Heap**

Zeichnen Sie auf ein Blatt Papier den min-Heap aus der Zahlenfolge 8,4,7,3,9,1,5,2. Die Wurzel soll immer das kleinste Element sein.

Entfernen Sie zwei Elemente.

Geben sie für den verbleibenden Heap die Level- und In-Order an.

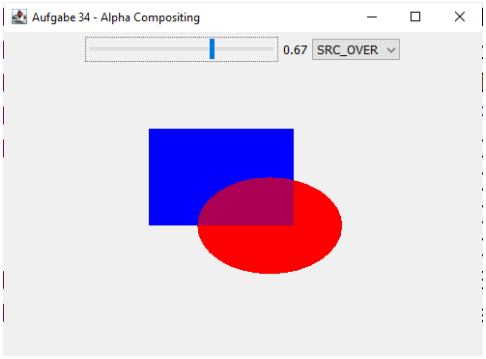
Wiederholen Sie die Aufgabe für verschiedene Zahlenfolgen bis Sie sicher sind. Überprüfen Sie Ihre Ergebnisse mit Hilfe der folgenden Webseite:

https://www.cs.usfca.edu/~galles/visualization/Heap.html

## Aufgabe 34

**AlphaComposite, Jslider, JcomboBox, Change- und ItemListener**

Programmieren Sie die Anwendung zum Alpha-Blending aus der Java-Vorlesung *8.3 Alpha Compositing*. Nennen Sie die Klasse ***Aufgabe34***.




Ergänzen Sie die fehlenden Operationen von Porter und Duff. Lesen Sie sich den Artikel „Composing Digital Images“ an und vergleichen Sie die Ergebnisse der Operationen mit Ihrem Programm (wobei im Programm $\alpha_d$ immer **1** ist).

Schauen Sie die Beschreibungen der Klasse *JSlider* und *JComboBox* sowie der Interfaces *ChangeListener* und *ItemListener* durch.

## Aufgabe 35

**Heap**

Als letzte Datenstruktur programmieren wir den **Heap**.

Bitt ergänzen Sie zunächst die Klasse ***Baum*** um die Methode **`getInhalt():T`**, welche *inhalt* zurückgibt.

Die Klasse ***Heap*** initialisiert den Array *knoten* mit der Länge 10.

Die Methode ***getWurzel*** liefert *null* wenn *laenge==0* ist, ansonsten *knoten[1]*.

In der Methoden ***offer*** verdoppeln Sie den Array falls nötig, so wie wir es z.B. in der Klasse *ArrayStack* gemacht haben. Dann inkrementieren Sie *laenge*, fügen das neue Element am der Position *laenge* ein und rufen *upHeap*.
**Am Ende rufen sie die Methode *knotenNeuVerbinden*.**

| baeume.Baum<T> |
|---|
| # inhalt: T<br># vater, links, rechts: Baum<T> |
| + «interface» Bearbeiter<T><br>+ Baum(inhalt:T)<br>+ getInhalt(): T<br>+ istNachkomme(k: Baum<T>): boolean<br>+ setLinks(k: Baum<T>): void<br>+ setRechts(k: Baum<T>): void<br>+ getVater(): Baum<T><br>+ tiefe(): int<br>+ toString(): String<br>+ inOrder(b: Bearbeiter<T>): void<br>+ preOrder(b: Bearbeiter<T>): void<br>+ postOrder(b: Bearbeiter<T>): void |

Die Methode ***poll*** merkt sich den Inhalt der Wurzel, dabei kann man das Ergebnis von *getInhalt* direkt in einer int-Variable speichern – Java macht eine automatische Typumwandlung. Dann wird das Element an die Position *der Wurzel* gesetzt und *laenge* dekrementiert. Am Ende wird *downHeap* und *knotenNeuVerbinden* gerufen.

| baeume.Heap |
|---|
| - knoten: Baum<Integer>[]<br>- laenge: int |
| + getWurzel():Baum<Integer><br>+ offer(prio: int, value: E): void<br>+ poll(): E<br>- upHeap(): void<br>- downHeap(): void<br>+ tausche(i, j: int): void<br>+ knotenNeuVerbinden(): void<br>+ toString():String |

Der Aufruf von *knotenNeuVerbinden* am Ende von *offer* und *poll* ist notwendig, damit wir zusätzlich zur Verwaltung der Knoten im Heap dafür sorgen, dass die Knoten untereinander im Sinne eines Binärbaumes richtig verbunden sind. Somit können wir den Baum auch grafisch mit Hilfe der Klasse *BinaerBaumPanel* darstellen.

Die Methode ***upHeap*** beginnt beim Knoten mit dem Index *laenge* und folgt dem absoluten Pfad Richtung Wurzel, wobei wenn nötig Knoten vertauscht werden. Hier kann man wieder ausnutzen, dass der Compiler das Ergebnis von *getInhalt()* automatisch von *Integer* in *int* umwandet:

```
while(i>1 && knoten[i].getInhalt() < knoten[i/2].getInhalt){ …
```

deshalb können wir mit dem **<** Operator arbeiten statt die Methode *compareTo* zu verwenden.

In ***downHeap*** beginnen wir bei der Wurzel und vergleichen jeden Knoten mit dem kleinsten Kind, wobei wieder getauscht wird wenn nötig.

Die Methode ***tausche*** vertauscht die Knoten an den Indizes ***i*** und ***j*** im Array *knoten*.

Die Methode ***knotenNeuVerbinden*** führt folgende Schritte durch:

- In einer Schleife wird zunächst für **alle** Knoten *vater, links und rechts* auf *null gesetzt*.
- Eine zweite Schleife, die beginnend vom letzten Knoten rückwärts bis zum Knoten mit dem Index 2 (also ohne die Wurzel) läuft, werden für jeden Knoten die Verbindungen vom und zum Vater hergestellt. Für die Verbindung vom Vater müssen Sie abfragen, ob der aktuelle Knoten der linke oder der rechte Nachkomme ist.

Die ***toString*** Methode stellt den Heap in einer Zeile dar (Beispiel für einen Heap der Länge 5):

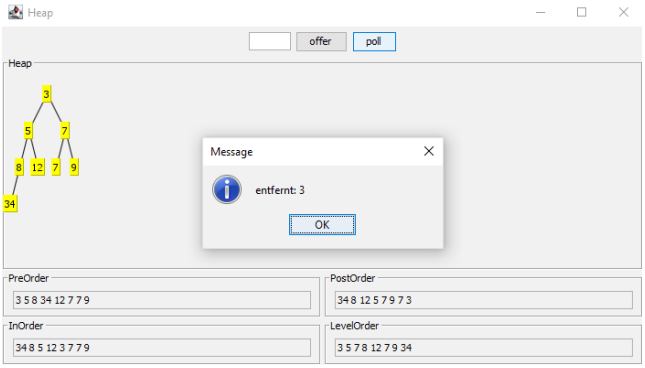
```
Heap[5]:  4 12 24 31 13
```

Jetzt kann man eine *main*-Methode einfügen, die eine Reihe von Elementen in den Heap einfügt und entfernt und jedes mal den Heap neu auf der Standardausgabe ausgibt, um die Funktionalität zu prüfen.

Wenn alles stimmt, schreiben Sie die im Bild gezeigte Klasse ***Aufgabe35*** um den Heap grafisch darzustellen.

Der *offer*-Buton liest den Wert aus dem Zahlenfeld und speichert ihn in einem Heap.

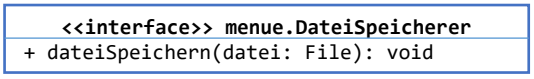
Der *poll*-Button entfernt einen Wert aus dem Heap und stellt ihn in einem *JoptionPane*-Dialog dar.




## Aufgabe 36

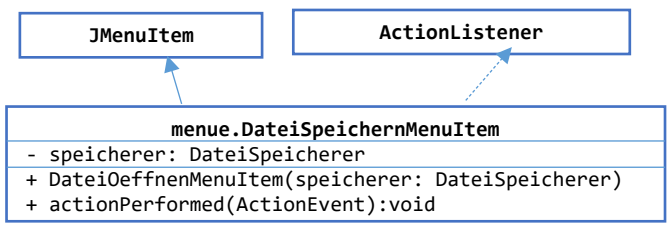
Wir erweitern unseren kleinen Editor aus Aufgabe 29 so, dass er Dateien auch speichern kann.

**verbesserter Editor**

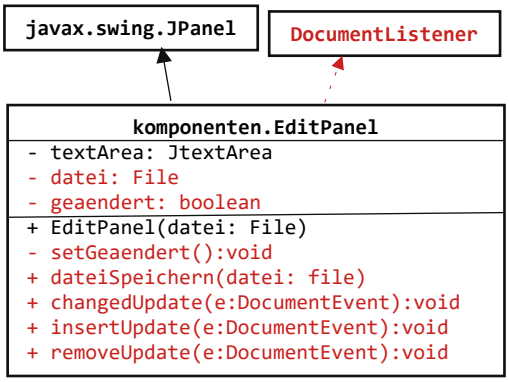



Schreiben Sie zunächst das Interface ***DateiSpeicherer*** anhand des UML Diagramms

Es folgt die Klasse ***DateiSpeichernMenuItem.*** Der **Konstruktor** speichert seinen Parameter in der Instanzvariable und registriert sich als ActionListener, ***actionPerformed*** ruft *dateiSpeichern* mit *null* als Parameter.



Jetzt lassen wir die Klasse ***EditPanel*** das Interface *javax.swing.event.DocumentListener* implementieren und erweitern sie wie folgt:




Der **Konstruktor** speichert seinen Parameter in der neuen Instanzvariable *datei* und registriert sich **nach dem Einlesen der Datei** als *DocumentListener:*

```
textArea.getDocument().addDocumentListener(this);
```

Die Methode ***setGeaendert*** setzt bei der ersten Änderung des Textes hinter den Titel des aktuellen Reiters der *JTabbedPane* ein Sternchen, um anzudeuten, dass die Datei sich geändert hat:

```
private void setGeaendert(boolean geaendert) {
  if(this.geaendert!=geaendert){
        JTabbedPane tabbedPane = (JTabbedPane)(this.getParent());
        int ix = tabbedPane.getSelectedIndex();
        String titel = datei.getName()+(geaendert?"*":"");
        tabbedPane.setTitleAt(ix, titel);
  }
  this.geaendert = geaendert;
}
```

Die Methode ***dateiSpeichern*** prüft zunächst, ob der Parameter *datei* **nicht** *null* ist, wenn ja, speichert sie ihn in *this.datei*. So können wir entweder in die aktuelle Datei speichern oder später „*Speichern unter*“ realisieren.

Dann speichert sie mit *`textArea.write(new FileWriter(datei));`* den Text in die Datei, wobei eventuelle Fehler mit try/catch abgefangen werden müssen. Danach ruft sie *setGeaendert(false)*.

Die drei Methoden von *DocumentListener* rufen jeweils *setGeaendert(true)*.

Erstellen Sie eine Kopie von *Aufgabe29* und nennen sie ***Aufgabe36***.

Fügen Sie ein *ZuletztBesuchteDateien* Menü hinzu wie bei Aufgabe 32 beschrieben.

Lassen Sie *Aufgabe36* zusätzlich das neue Interface *DateiSpeicherer* implementieren. In ***dateiSpeichern*** holen Sie sich das obere Panel mit *tabbedPane.getSelectedComponent* in eine Variable vom Typ EditPanel (Cast verwenden). Falls die Variable nicht *null* ist, rufen Sie *dateiSpeichern* für das Panel. Im **Konstruktor** legen Sie ein *DateiSpeichernMenuItem* auf das Datei-Menü – und schon haben wir einen funktionsfähigen Text-Editor.

Eine sinnvolle Erweiterung wäre noch ein Menüeintrag „SpeichernUnter“. Dazu schreiben Sie die neue Klasse ***DateiSpeichernUnterMenuItem***, die wie *DateiOeffnenMenuItem* einen *JFileChooser*-Dialog besitzt. Allerdings ruft sie statt *showOpenDialog* die Methode *showSaveDialog* und statt eines *DateiBearbeiters* hat sie einen *DateiSpeicherer* und ruft *speicherer.dateiSpeichern* mit der gewählten Datei.

Einen Schönheitsfehler hat unser Editor aber noch: wenn Änderungen vorgenommen wurden, sollte die Anwendung nicht sofort terminieren. Fügen Sie deshalb am Ende des Konstruktors folgende Zeilen ein:

```
this.fenster.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
this.fenster.addWindowListener(new WindowAdapter() {
   public void windowClosing(WindowEvent e) {
     boolean aenderungen = false;
     for(int i=0;i<tabbedPane.getTabCount();i++) {
        aenderungen = aenderungen || ((EditPanel)tabbedPane.getComponentAt(i)).isGeaendert();
     }
     if(!aenderungen)System.exit(0);
     int antwort = JOptionPane.showConfirmDialog(Aufgabe36.this, "Soll die Anwendung beendet werden?",
                                        "Es wurden Änderungen vorgenommen! ", JOptionPane.YES_NO_OPTION);
     if(antwort==JOptionPane.YES_OPTION) System.exit(0);
   }
});
```