Aufgabe 1 (AVL-Bäume) (5 Punkte)

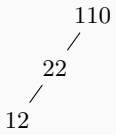
Fügen Sie die folgenden Zahlen nacheinander in einen *AVL-Baum* ein:

$$110 \quad 22 \quad 12 \quad 33 \quad 35 \quad 8$$

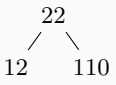
Zeichnen Sie den Baum vor und nach jeder durchgeführten Rotation. Geben Sie auch jeweils an, was für Rotationen Sie durchführen.

Solution:

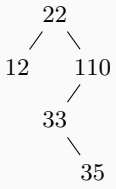
Einfügen von $110, 22, 12$



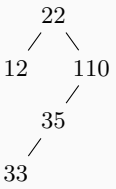
R-Rotation um 110



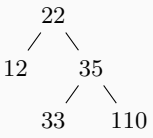
Einfügen von $33, 35$



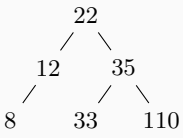
LR-Rotation um 110 (L)



LR-Rotation um 110 (R)



Einfügen von 8



Aufgabe 2 (Heaps) **(5 Punkte)**

Fügen Sie die folgenden Zahlen nacheinander in einen *Min-Heap* ein:

70 12 30 8 10 1 45

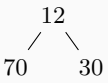
Zeichnen Sie den Baum vor und nach jedem vollständigen Einfügen.

Solution:

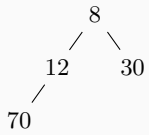
Einfügen von 70

Einfügen von 12

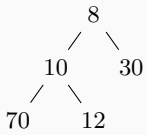
Einfügen von 30



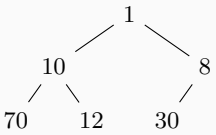
Einfügen von 8



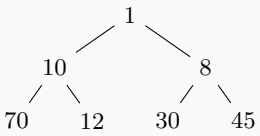
Einfügen von 10



Einfügen von 1



Einfügen von 45



Aufgabe 3 (Sortieralgorithmen) **(5 Punkte)**

Sortieren Sie die folgende Liste von Zahlen aufsteigend mit dem Verfahren *Bubble Sort*:
Geben Sie die Liste nach jedem Durchlauf der inneren Schleife an.

Solution:

$$17, 22, 13, 38, 35, 12$$
$$17, 13, 22, 35, 12, 38$$
$$13, 17, 22, 12, 35, 38$$
$$13, 17, 12, 22, 35, 38$$
$$13, 12, 17, 22, 35, 38$$
$$12, 13, 17, 22, 35, 38$$

Aufgabe 4 (Datentypen) **(10 Punkte)**

Erklären Sie Idee und Funktionsweise des folgenden Datentyps. Welche Vor- und Nachteile sehen Sie gegenüber einem binären Suchbaum?

```
4  type Table struct {
5    keys   []string
6    values []string
7  }
8
9  func targetIndex(key string) int {
10   firstchar := key[0]
11   return int((firstchar)-'a') * 10
12 }
13
14 func (h *Table) FirstEmptyIndexForKey(key string) int {
15   for i := 0; i < 260; i++ {
16     index := (targetIndex(key) + i) % 260
17     if h.keys[index] == "" {
18       return index
19     }
20   }
21   return -1
22 }
23
24 func (h *Table) FirstMatchingIndexForKey(key string) int {
25   for i := 0; i < 260; i++ {
26     index := (targetIndex(key) + i) % 260
27     if h.keys[index] == key {
28       return index
29     }
30   }
31   return -1
32 }
33
34 func (h *Table) Put(key string, value string) {
35   index := h.FirstEmptyIndexForKey(key)
36   if index == -1 {
37     panic("Table␣is␣full")
38   }
39   h.keys[index] = key
40   h.values[index] = value
41 }
```

Solution: Dieser Datentyp implementiert eine Tabelle, die Schlüssel auf Werte (beides Strings) abbildet. Damit erfüllt sie einen ähnlichen Zweck wie ein binärer Suchbaum.

Funktionsweise: Die Tabelle kann 260 Schlüssel-Wert-Paare speichern. Die Idee ist, dass für jeden Buchstaben im Alphabet 10 Plätze vorhanden sind. Beim Einfügen eines neuen Schlüssels wird anhand des ersten Buchstabens die Zielposition bestimmt. Falls diese Position bereits belegt ist, wird der nächste Platz in der Liste verwendet.

Vor- und Nachteile:

- Im Idealfall ist die Suche sehr schnell, da die Position immer direkt berechnet werden kann.
- Im Worst-Case muss allerdings die gesamte Liste durchlaufen werden, was sehr langsam ist.
- D.h. im Worst-Case ist die Suche mit $O(n)$ schlechter als bei einem binären Suchbaum. Solange die Liste nicht zu voll ist, ist die Suche aber im Durchschnitt schneller.

Anmerkung: Diese Art Datenstruktur nennt man eine *Hashtabelle*. Sie ist eine der wichtigsten Datenstrukturen in der Informatik. Sie erreicht im Durchschnitt eine bessere Laufzeit als ein binärer Suchbaum, allerdings erkauft man sich diesen Vorteil mit einem stark erhöhten Speicherbedarf.

Aufgabe 5 (Komplexität) **(10 Punkte)**

Bestimmen Sie die Komplexität des folgenden Algorithmus zur Bestimmung des Medians eine Liste der Länge n: Geben Sie auch einen Verbesserungsvorschlag an, wie die Komplexität reduziert werden kann.

```
6  func Median(list []int) int {
7    diffs := make([]int, len(list))
8
9    for i, el := range list {
10     d := largerCount(list, el) - smallerCount(list, el)
11     diffs[i] = abs(d)
12   }
13
14   return list[smallestIndex(diffs)]
15 }
```

Gehen Sie dabei davon aus, dass die Hilfsfunktionen tun, was ihre Namen vermuten lassen:

- largerCount und smallerCount zählen die Anzahl der Elemente, die größer bzw. kleiner als ein gegebenes Element sind.
- smallestIndex gibt den Index des kleinsten Elements zurück.
- abs berechnet den Betrag einer Zahl.

Gehen Sie weiter davon aus, dass diese Hilfsfunktionen jeweils optimal sind, d.h. dass sie die bestmögliche Laufzeit für ihre Aufgabe haben.

Solution: Analyse:

- Die Schleife wird n-mal durchlaufen.
- Dabei werden largerCount, smallerCount und abs jeweils einmal aufgerufen.
- Die Filfsfunktion smallestIndex wird einmal mit einer Liste der Länge n aufgerufen.
- Die Listen-Hilfsfunktionen können jeweils eine Laufzeit von $O(n)$haben.
- Die Funktion abs hat eine Laufzeit von $O(1)$ ist also vernachlässigbar.

Insgesamt ergibt sich eine Komplexität von $O(n)$im Schleifenrumpf und damit $O(n^2)$für den gesamten Algorithmus.

Verbesserungsvorschlag:

Wenn man die Liste zu Beginn der Median-Funktion sortiert, kann der Median anschließend in konstanter Zeit bestimmt werden, nämlich als das Element an der Position $\frac{n}{2}$.

Die Sortierung hat eine Komplexität von $O(n \log n)$, so dass die Bestimmung des Medians ebenfalls in $O(n \log n)$möglich ist.

Aufgabe 6 (Sortieralgorithmen) **(10 Punkte)**

Erläutern Sie die Funktionsweise des folgenden Sortierverfahrens:

```
4  func FooSort(a []int) {
5    pos := 0
6    for pos < len(a) {
7      if pos == 0 {
8        pos++
9      } else {
10       if a[pos-1] > a[pos] {
11         a[pos-1], a[pos] = a[pos], a[pos-1]
12         pos--
13       } else {
14         pos++
15       }
16     }
17   }
18 }
```

Solution: Bei diesem Verfahren handelt es sich um *Gnome Sort*.

Der Algorithmus läuft über das Array und vergleicht jeweils die beiden Elemente vor und an der aktuellen Position. Falls die beiden Elemente in der falschen Reihenfolge sind, werden sie vertauscht und die Position um eins verringert. Falls die beiden Elemente in der richtigen Reihenfolge sind, wird die Position um eins erhöht.

Wenn die Position 0 ist, wird sie um eins erhöht. Wenn die Position $n-1$ ist (also die letzte Position im Array), ist das Array sortiert.