Aufgabe 1: Sortierverfahren (5 Punkte)

Sortieren Sie die folgende Liste von Zahlen aufsteigend mit dem Verfahren *Insertion Sort*:

38 18 5 21 29 14 35

Geben Sie die Liste nach jedem Durchlauf der inneren Schleife an, d.h. nach jedem vollständigen Einsortieren eines Elements.

Aufgabe 2: AVL-Bäume **(5 Punkte)**

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

42 16 12 19 38 1

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

Aufgabe 3: Heaps **(5 Punkte)**

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

45 18 19 7 13 2 22

Zeichnen Sie den Baum vor und nach jedem vollständigen Einfügen.

Aufgabe 4: Sortierverfahren (10 Punkte)

Erläutern Sie die Funktionsweise des folgenden Sortierverfahrens. Erläutern Sie auch, welche Einschränkungen bzw. Anforderungen an die Liste gelten müssen, damit das Verfahren korrekt funktioniert, und was daran ggf. nicht optimal oder sinnvoll ist.

```
1  func FooSort(list []int) {
2    for !Bar(list) {
3      i, j := rand.Intn(len(list)), rand.Intn(len(list))
4      list[i], list[j] = list[j], list[i]
5    }
6  }
7
8  func Bar(list []int) bool {
9    if len(list) <= 1 {
10     return true
11   }
12   a, b := list[0], list[1:]
13   if a > b[0] {
14     return false
15   }
16   return Bar(b)
17 }
```

Aufgabe 5: Datenstrukturen (10 Punkte)

Erläutern Sie die Idee und Funktionsweise des folgenden Datentyps. Diskutieren Sie kurz die Vor- und Nachteile gegenüber ähnlichen Datentypen aus der Vorlesung.

```
type FooType struct {
  values   []int
  children []*FooType
}

func (f *FooType) Add(value int) {
  if len(f.values) < 2 {
    f.values = append(f.values, value)
    return
  }
  if value < f.values[0] {
    f.AddToChild(0, value)
    return
  }
  if value < f.values[1] {
    f.AddToChild(1, value)
    return
  }
  f.AddToChild(2, value)
}

func (f *FooType) AddToChild(i, value int) {
  for len(f.children) <= i {
    f.children = append(f.children, &FooType{})
  }
  f.children[i].Add(value)
}
```

Aufgabe 6: Komplexitaet (10 Punkte)

Betrachten Sie die folgende Funktion `SameElements()`. Die Funktion erwartet zwei `int`-Listen und prüft, ob diese beiden Listen die gleiche Menge an Elementen enthalten. D.h. ob jedes Element aus der einen Liste auch in der anderen enthalten ist. Dabei spielt es keine Rolle, ob die Länge der Listen gleich ist bzw. ob die Elemente in der gleichen Anzahl vorkommen.

```
1  func SameElements(list1, list2 []int) bool {
2    for _, v1 := range list1 {
3      contained := false
4      for _, v2 := range list2 {
5        if v1 == v2 {
6          contained = true
7        }
8      }
9      if !contained {
10       return false
11     }
12   }
13   for _, v2 := range list2 {
14     contained := false
15     for _, v1 := range list1 {
16       if v1 == v2 {
17         contained = true
18       }
19     }
20     if !contained {
21       return false
22     }
23   }
24   return true
25 }
```

a) Bestimmen Sie die Komplexität dieser Funktion. Geben Sie in O-Notation an, wie oft die Vergleiche `if v1 == v2` durchgeführt werden. Begründen Sie Ihr Ergebnis.

b) Machen Sie einen Vorschlag, wie diese Funktion optimiert werden kann. Erläutern Sie, inwiefern dieser eine bessere Komplexität hat.

Hinweis: Sie müssen keinen konkreten, vollständigen Algorithmus angeben.