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.

Lösung

Die folgenden Zwischenergebnisse entstehen bei einem In-Place durchgeführten Insertion Sort. Sortierter und unsortierter Teil der Liste sind jeweils durch einen senkrechten Strich (|) getrennt:

$$
\begin{array}{cccccccc}
38 & | & 18 & 5 & 21 & 29 & 14 & 35 \\
18 & 38 & | & 5 & 21 & 29 & 14 & 35 \\
5 & 18 & 38 & | & 21 & 29 & 14 & 35 \\
5 & 18 & 21 & 38 & | & 29 & 14 & 35 \\
5 & 18 & 21 & 29 & 38 & | & 14 & 35 \\
5 & 14 & 18 & 21 & 29 & 38 & | & 35 \\
5 & 14 & 18 & 21 & 29 & 35 & 38 & |
\end{array}
$$

Aufgabe 2: AVL-Bäume (5 Punkte)

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

$$42 \quad 16 \quad 12 \quad 19 \quad 38 \quad 1$$

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

Lösung

Einfügen von $42, 16, 12$

```
      42
     /
   16
  /
12
```

R-Rotation um 42

```
   16
  /  \
12    42
```

Einfügen von $19, 38$

```
   16
  /  \
12    42
     /
   19
     \
      38
```

RL-Rotation um 42 (1)

```
   16
  /  \
12    19
        \
         42
        /
      38
```

RL-Rotation um 42 (2)

```
      19
     /  \
   16    42
  /     /
12    38
```

Einfügen von 1

```
         19
        /  \
      16    42
     /     /
   12    38
  /
1
```

R-Rotation um 16

```
          19
         /   \
      12       42
     /  \     /
    1    16  38
```

Aufgabe 3: Heaps (5 Punkte)

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

45 18 19 7 13 2 22

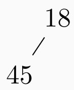
Zeichnen Sie den Baum vor und nach jedem vollständigen Einfügen.

Lösung

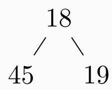
Einfügen von 45

45

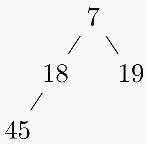
Einfügen von 18



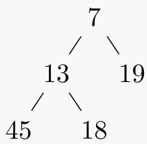
Einfügen von 19



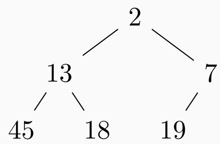
Einfügen von 7



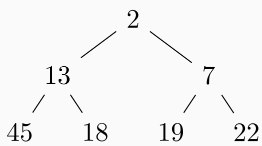
Einfügen von 13



Einfügen von 2



Einfügen von 22



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.

```go
func FooSort(list []int) {
  for !Bar(list) {
    i, j := rand.Intn(len(list)), rand.Intn(len(list))
    list[i], list[j] = list[j], list[i]
  }
}

func Bar(list []int) bool {
  if len(list) <= 1 {
    return true
  }
  a, b := list[0], list[1:]
  if a > b[0] {
    return false
  }
  return Bar(b)
}
```

Lösung

Das Verfahren bestimmt immer wieder zwei zufällige Positionen in der Liste und vertauscht die Elemente an diesen Positionen. Dies wird so lange wiederholt, bis die Liste sortiert ist.

Die Funktion `Bar` ist eine Hilfsfunktion, die prüft, ob die Liste sortiert ist. Falls die Länge der Liste kleiner als 2 ist, ist die Liste sortiert. Ansonsten werden die ersten beiden Elemente der Liste verglichen. Falls diese falsch sortiert sind, ist die Liste nicht sortiert. Ansonsten wird die Funktion rekursiv auf den Rest der Liste angewendet.

Es gibt keine Einschränkungen an die Liste, der Wertebereich der Elemente ist ganz $\mathbb{Z}$. Das Verfahren ist nicht optimal, da es im Worst-Case beliebig lange dauern kann, da die zufälligen Positionen immer wieder die gleichen sein können. Es ist sehr viel wahrscheinlicher, dass das Verfahren sehr lange dauert, als dass es schnell ist.

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.

```go
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)
}
```

Lösung

Der Datentyp `Footype` ist ein *ternärer Suchbaum*. In jedem Knoten des Baums werden bis zu zwei Werte gespeichert und es gibt bis zu drei Kindknoten. Die Werte im linken Kindknoten sind kleiner als der erste Wert im Elternknoten, die Werte im mittleren Kindknoten sind größer als der erste Wert im Elternknoten und kleiner als der zweite Wert im Elternknoten, und die Werte im rechten Kindknoten sind größer als der zweite Wert im Elternknoten.

Die Funktion `Add` fügt einen Wert in den Baum ein. Dazu benutzt sie die Funktion `AddToChild`, die einen Wert in einen Kindknoten einfügt und diesen ggf. vorher erzeugt. Die Funktion `AddToChild` ruft dann wieder `Add` auf, um den Wert in den Kindknoten einzufügen.

Die Datenstruktur ist sehr ähnlich zu einem binären Suchbaum, allerdings wird der Baum breiter und flacher, da jeder Knoten bis zu zwei Werte speichert. Dadurch ergibt sich ein minimal besseres Laufzeitverhalten.

Die Komplexitätsklasse der Operationen ist die gleiche wie bei einem binären Suchbaum. Der Nachteil ist, dass die Datenstruktur wesentlich unübersichtlicher ist. Sobald bspw. eine Funktion zum Löschen hinzu kommt, kann es passieren, dass ein Knoten nur noch einen Wert enthält, dieser aber nicht der linke der beiden Werte ist. Dann muss der Baum umorganisiert werden oder beim Einfügen muss darauf geachtet werden, dass auch Werte ungültig sein können.

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.

```
func SameElements(list1, list2 []int) bool {
  for _, v1 := range list1 {
    contained := false
    for _, v2 := range list2 {
      if v1 == v2 {
        contained = true
      }
    }
    if !contained {
      return false
    }
  }
  for _, v2 := range list2 {
    contained := false
    for _, v1 := range list1 {
      if v1 == v2 {
        contained = true
      }
    }
    if !contained {
      return false
    }
  }
  return true
}
```

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.

Lösung

a) Die Funktion liegt in $O(n \cdot m)$, wobei n und m die Längen der beiden Listen sind. Es gibt zwei geschachtelte Schleifen, wo jeweils innerhalb eines Durchlaufs durch eine Liste für jedes Element ein Durchlauf durch die andere Liste passiert. In jeder dieser Schleifen werden daher $n \cdot m$ Vergleiche durchgeführt.

b) Eine Optimierungsmöglichkeit wäre, die beiden Listen zu sortieren, Duplikate zu entfernen und die Listen dann auf Gleichheit zu testen. Das Sortieren liegt in $O(n \cdot \log n + m \cdot \log m)$, Duplikate entfernen und der exakte Vergleich von Listen haben jeweils lineare Laufzeit. Da diese Aktionen hintereinander geschehen und nicht geschachtelt werden, bleibt es bei der Komplexität des Sortierens, also insgesamt bei $O(n \cdot \log n)$, wobei n die Länge der größeren Liste ist.