PRG1x & ADE1x **Einf. i d. Programmierung (int. LVA)**
Üb. zu Element. Alg. u. Datenstrukt. **WS 16/17, Übung 10**

Abgabetermin: Mi in der KW 3

☐ **Gr. 1,** Dr. D. Auer
☐ **Gr. 2,** Dr. G. Kronberger **Name** _____ **Aufwand in h** _____
☐ **Gr. 3,** Dr. H. Gruber

Punkte _____ **Kurzzeichen Tutor / Übungsleiter** _____ / _____

1. Balancieren von Binärbäumen (6 Punkte)

Operationen auf einem binären Suchbaum sind nur dann effizient, wenn dieser *balanciert* ist, wenn also die Wege von der Wurzel zu jedem Blatt (etwa) gleich lang sind. In der Praxis lässt sich aber nicht beeinflussen, in welcher Reihenfolge und Ausprägung die in den Baum einzufügenden Daten auftreten. Daher entstehen mit unserer einfachen *Insert*-Operation i. d. R. unbalancierte Bäume.

Um dieses Problem zu lösen, kann man einen binären Suchbaum, bevor er z. B. für viele Suchoperationen verwendet wird, balancieren. Ein einfacher Algorithmus zum Balancieren binärer Suchbäume, in denen keine Werte mehrfach vorkommen, kann so vorgehen:

1. Allokieren eines dynamischen Felds mit Platz für so viele Elemente (Zeiger), wie Knoten im Baum enthalten sind.
2. Eintragen aller Zeiger auf die Knoten des Baums in das Feld (sortiert, also mittels *In-Order*-Baumdurchlauf).
3. Aufbauen eines neuen, nun balancierten binären Suchbaum aus dem Feld: Das mittlere Element des Felds liefert den Wurzelknoten, das mittlere Element der linken Hälfte des Felds den Wurzelknoten des linken Teilbaums und das mittlere Element der rechten Hälfte des Felds den Wurzelkonten des rechten Teilbaums, usw.
4. Freigeben des Felds.

Implementieren Sie basierend auf dieser Idee eine Prozedur *Balance* und testen Sie diese ausführlich.

Zusatzfrage: Warum liefert dieser Algorithmus für binäre Suchbäume, in denen Werte mehrfach vorkommen, nicht immer korrekte Ergebnisse?

2. Laufzeitkomplexität (4 + 3 + 1 Punkte)

Gegeben ist folgender Algorithmus (in Form einer Pascal-Funktion) zur Umwandlung einer als Zeichenkette (*STRING*) gegebenen Binärzahl (z. B. '101') in einen *INTEGER*-Wert (z. B. 5).

```
FUNCTION IntOf(dual: STRING): INTEGER;
  VAR
    result, i: INTEGER;
BEGIN
  result := 0;
  i := 1;
  WHILE i <= Length(dual) DO BEGIN
    result := result * 2;
    IF dual[i] = '1' THEN
      result := result + 1;
    i := i + 1;
  END; (*WHILE*)
  IntOf := result;
END; (*IntOf*)
```

a) Entwickeln Sie unter Verwendung der nachstehenden Tabelle eine Formel, welche für die Länge einer Binärzahl und die Anzahl der darin enthaltenen Einsen die exakte Laufzeit des Algorithmus angibt. Berechnen Sie damit die Laufzeit für die Binärzahlen 100100, 100001, 110100, 1111, 0000, 1 und 0.

Operation	**Ausführungszeit**
Wertzuweisung	1
Vergleich	1
Indizierung	0,5
Addition, Subtraktion	0,5
Multiplikation	3
Prozeduraufruf	16 + 2 * Anzahl der Parameter

b) Entwickeln Sie nun unter der Annahme, dass Einsen und Nullen in einer Binärzahl etwa gleich oft vorkommen, eine neue Formel, die auf Basis der Länge einer beliebigen Binärzahl die (ungefähre) Laufzeit des Algorithmus angibt. Erstellen Sie damit eine Tabelle, welche die Laufzeiten für Binärzahlen der Länge 1 bis 20 sowie der Längen 50, 100 und 200 darstellt.

c) Bestimmen die asymptotische Laufzeitkomplexität für den gegebenen Algorithmus in Abhängigkeit von der Länge der Binärzahl. (Die Anzahl der Einsen und Nullen können Sie wieder als gleichverteilt annehmen.) Begründen Sie, wie Sie auf Ihre Lösung gekommen sind. Wie steht diese in Zusammenhang mit der in b) erstellten Tabelle?

3. Laufzeitkomplexität und Rekursion (4 + 3 + 1 + 2 Punkte)

Gegeben ist der folgende rekursive Algorithmus (in Form einer Pascal-Funktion), der wiederum für eine Binärzahl in Form einer Zeichenkette den *INTEGER*-Wert liefert.

```
FUNCTION IntOf2(dual: STRING): INTEGER;

   FUNCTION IORec(pos: INTEGER): INTEGER;
   BEGIN
     IF pos = 0 THEN
       IORec := 0
     ELSE IF dual[pos] = '1' THEN
       IORec:= IORec(pos - 1) * 2 + 1
     ELSE
       IORec:= IORec(pos - 1) * 2;
   END; (*IORec*)

BEGIN (*IntOf2*)
   IntOf2 := IORec(Length(dual));
END; (*IntOf2*)
```

a) Entwickeln Sie wieder eine Formel für die Laufzeit auf Basis von Länge und Anzahl der Einsen in der Binärzahl auf und bestimmen Sie „exakten“ Laufzeiten für die Beispiele aus 1.a). Vorgehensweise: Bestimmen Sie zunächst getrennt für jeden der (drei) möglichen Zweige der inneren Funktion *IORec* die Laufzeit eines einzelnen Durchlaufs und überlegen Sie dann (z. B. an Hand eines Beispiels), wie oft jeder der Zweige durchlaufen wird. Durch einfaches Multiplizieren und Addieren erhalten Sie dann die Gesamtlaufzeit.

b) Finden Sie auch hier wieder eine Formel, welche von einer gleichmäßigen Verteilung der Einsen und der Nullen ausgeht, und stellen Sie die gleiche Tabelle wie in 2.b) auf.

c) Bestimmen Sie für den rekursiven Algorithmus *IORec* die asymptotische Laufzeitkomplexität. Betrachten Sie dazu am besten wieder die Daten der unter b) erstellten Tabelle.

d) Vergleichen Sie Ihre Analysen des iterativen (Beispiel 2) und des rekursiven Algorithmus (Beispiel 3). Wie ist jeweils die asymptotische Laufzeitkomplexität? Was zeigen (im Gegensatz dazu) die Grob- bzw. Feinanalyse? Was schließen Sie daraus in Bezug auf die Güte der beiden Lösungen?