

☐ Gr. 1, Dr. D. Auer☐ Gr. 2, Dr. G. Kronberger

Gr. 3, Dr. H. Gruber

Name Andreas Roither Aufwand in h 6 h

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 Wurzelknoten 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	$16 + 2 * \text{Anzahl der Parameter}$

- 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*)
    END;

    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?

Übung 10

Aufgabe 1

Lösungsidee

Es wird eine Prozedur erstellt, basierend auf der vorgeschlagenen Lösungsidee. Um die Größe des Arrays festlegen zu können wird die Anzahl der Nodes im Baum mithilfe von Count-Nodes ermittelt. Nachdem das Array initialisiert wurde und alle Nodes des Baumes in das Array eingefügt sind, werden die Mittelpositionen berechnet. Mit den berechneten Werten wird bestimmt welche Elemente des Arrays eingefügt werden. Elemente die bereits in den Baum gespeichert wurden, werden im Array auf NIL gesetzt. Alle fehlenden Elemente werden danach eingefügt.

```

1 PROGRAM balancedtree;

3 TYPE
  Node = ^NodeRec;
5  NodeRec = RECORD
    value: INTEGER;
    left, right: Node;
  END;
9  Tree = Node;

11 PROCEDURE InitTree (VAR t: Tree);
  BEGIN
13   t := NIL;
  END;

15 PROCEDURE InitArray (VAR arr: ARRAY OF Node);
17 VAR count : INTEGER;
  BEGIN
19   FOR count := 1 TO High(arr) DO
    arr[count] := NIL;
21  END;

23 FUNCTION NewNode (value: INTEGER): Node;
  VAR
25   n: Node;
  BEGIN
27   New(n);
    n^.value := value;
29   n^.left := NIL;
    n^.right := NIL;
31   NewNode := n;
  END;

33 (* Insert nodes recursive *)
35 PROCEDURE InsertRec (VAR t: Tree; n: Node);
  BEGIN
37   IF t = NIL THEN BEGIN
    t := n;
39   END
    ELSE BEGIN
41   IF n^.value < t^.value THEN
    InsertRec(t^.left, n)

```

```

43  ELSE
      InsertRec(t^.right, n)
45  END;
END;
47
(* Adds a node to the Array *)
49 PROCEDURE AddToArray(VAR arr : ARRAY OF Node; n : Node);
VAR count : INTEGER;
51 BEGIN
    count := 0;
53  WHILE (arr[count] <> NIL) AND (count <= High(arr)) DO
        count := count + 1;
55
    IF count <= High(arr) THEN
57        arr[count] := n
    ELSE
59        WriteLn('Array already full!');
END;
61
(* Adds Nodes to the Array, sorted *)
63 PROCEDURE SaveTreeInOrder (t: Tree; VAR b_array : ARRAY OF Node);
BEGIN
65  IF t <> NIL THEN
      BEGIN
67      SaveTreeInOrder(t^.left, b_array);
        AddToArray(b_array, t);
69      SaveTreeInOrder(t^.right, b_array);
      END;
71 END;

73 (* Sets the left & right pointer of all Elements in the array to NIL *)
PROCEDURE CleanArray(VAR b_array : ARRAY OF Node);
75 VAR count : INTEGER;
BEGIN
77  FOR count := 0 TO High(b_array) DO
      BEGIN
79      IF b_array[count] <> NIL THEN
          BEGIN
81          b_array[count]^left := NIL;
            b_array[count]^right := NIL;
83          END;
          END;
85 END;

87 (* Count the nodes in a tree *)
PROCEDURE CountNodes (t: Tree; VAR count : INTEGER);
89 BEGIN
    IF t <> NIL THEN
91        BEGIN
            CountNodes(t^.left, count);
93            count := count + 1;
            CountNodes(t^.right, count);
95        END;
    END;
97
(* Returns the height of the tree *)
99 FUNCTION Height(t: Tree) : INTEGER;
VAR

```

```

101  hl, hr: INTEGER;
BEGIN
103  IF t = NIL THEN
      Height := 0
105  ELSE BEGIN

107  hl := Height (t^.left);
      hr := Height (t^.right);

109

      IF hl > hr THEN
111  Height := 1 + hl
      ELSE
113  Height := 1 + hr;
      END;
115 END;

117 FUNCTION CalculateHalf(i : LONGINT): INTEGER;
BEGIN
119
      IF i/2 = i DIV 2 THEN
121  CalculateHalf := Round(i / 2 )
      ELSE
123  CalculateHalf := (i DIV 2) + 1;
      END;
125

      (* write values of the array to the console *)
127 PROCEDURE PrintArray(b_array: ARRAY OF Node);
      VAR count : INTEGER;
129 BEGIN
          WriteLn('Array:');
131  FOR count := 0 TO High(b_array) DO
          BEGIN
133      IF b_array[count] = NIL THEN
              Write('NIL ')
135      ELSE
              Write(b_array[count]^value, ' ');
137  END;
          WriteLn;
139 END;

141 (* Balance the tree so the height of the tree will be reduced*)
PROCEDURE Balance(VAR t: Tree);
143 VAR
    c, length : INTEGER;
145 pos : LONGINT;
    b_array : ARRAY OF Node;
147 t_temp : Tree;

149 BEGIN
    IF t <> NIL THEN
151  BEGIN
      WriteLn('Balancing...');

153

      c := 0;
155  CountNodes(t, c);
      SetLength(b_array, c);
157  InitArray(b_array);
      InitTree(t_temp);

```

```

159   SaveTreeInOrder(t, b_array);
      PrintArray(b_array);
161
      WriteLn('Old height: ', Height(t));
163
      (* Has to be done after height function because the pointer left & right
      are
165         set to NIL in cleanarray (the height after that will be 1) *)
      CleanArray(b_array);
167
      length := High(b_array)+1;
169      pos := length;

171      WHILE pos > 2 DO
      BEGIN
173          pos := CalculateHalf(pos);
          InsertRec(t_temp, b_array[pos-1]);
175          b_array[pos-1] := NIL;
      END;
177
      pos := CalculateHalf(length);
179

      WHILE pos < length-1 DO
      BEGIN
181          pos := pos + CalculateHalf(length - pos);
          InsertRec(t_temp, b_array[pos-1]);
183          b_array[pos-1] := NIL;
      END;
185

      FOR pos := 0 TO High(b_array) DO
      BEGIN
187          IF b_array[pos] <> NIL THEN
      BEGIN
189              InsertRec(t_temp, b_array[pos]);
              b_array[pos] := NIL;
191          END;
193      END;
195      t := t_temp;
      b_array := NIL;
197      t_temp := NIL;

199      WriteLn('New height: ', Height(t));
      END;
201 END;

203 PROCEDURE DisposeTree(VAR t: Tree);
      BEGIN
205     IF t <> NIL THEN BEGIN
          DisposeTree(t^.left);
207         DisposeTree(t^.right);
          Dispose(t);
209         t := NIL;
      END;
211 END;

213 VAR
      t : Tree;
215

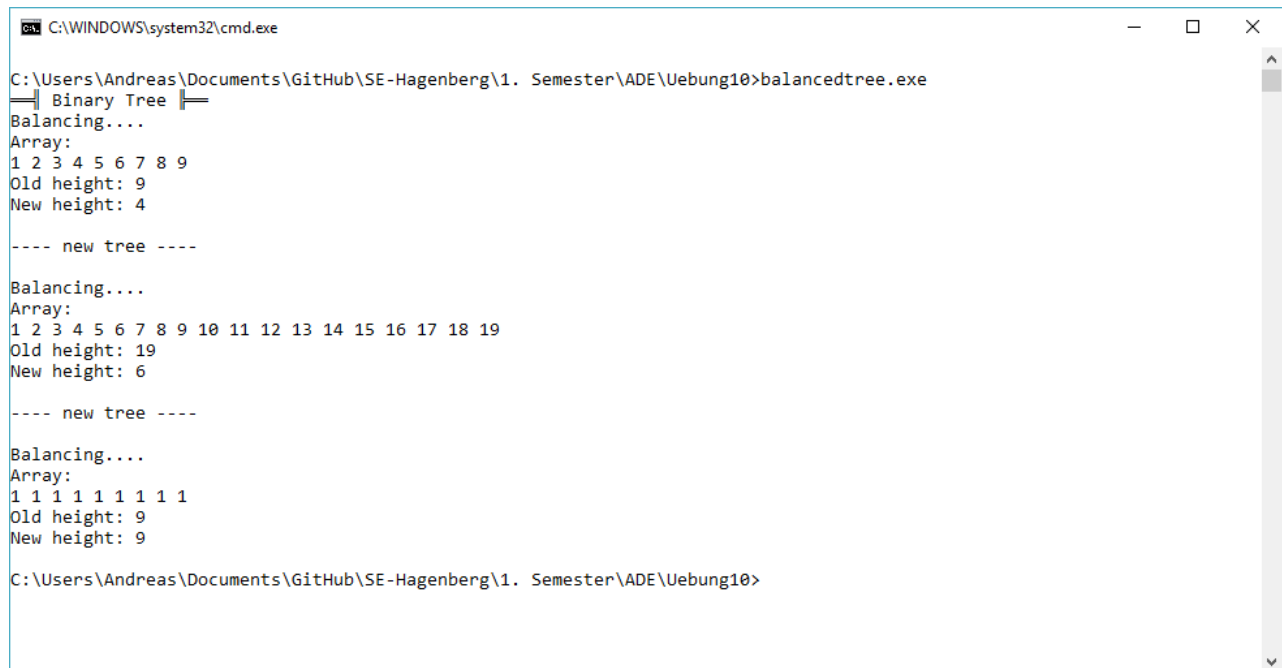
```

BEGIN

```

217 WriteLn(chr(205),chr(205),chr(185), ' Binary Tree ',chr(204),chr(205),chr(205)
    );
219
    InitTree(t);
221 InsertRec(t,NewNode(1));
    InsertRec(t,NewNode(2));
223 InsertRec(t,NewNode(3));
    InsertRec(t,NewNode(4));
225 InsertRec(t,NewNode(5));
    InsertRec(t,NewNode(6));
227 InsertRec(t,NewNode(7));
    InsertRec(t,NewNode(8));
229 InsertRec(t,NewNode(9));
    Balance(t);
231
    DisposeTree(t);
233
    WriteLn(#13#10, '—— new tree ——', #13#10);
235
    InitTree(t);
237 InsertRec(t,NewNode(1));
    InsertRec(t,NewNode(2));
239 InsertRec(t,NewNode(3));
    InsertRec(t,NewNode(4));
241 InsertRec(t,NewNode(5));
    InsertRec(t,NewNode(6));
243 InsertRec(t,NewNode(7));
    InsertRec(t,NewNode(8));
245 InsertRec(t,NewNode(9));
    InsertRec(t,NewNode(10));
247 InsertRec(t,NewNode(11));
    InsertRec(t,NewNode(12));
249 InsertRec(t,NewNode(13));
    InsertRec(t,NewNode(14));
251 InsertRec(t,NewNode(15));
    InsertRec(t,NewNode(16));
253 InsertRec(t,NewNode(17));
    InsertRec(t,NewNode(18));
255 InsertRec(t,NewNode(19));
    Balance(t);
257
    DisposeTree(t);
259
    WriteLn(#13#10, '—— new tree ——', #13#10);
261
    InitTree(t);
263 InsertRec(t,NewNode(1));
    InsertRec(t,NewNode(1));
265 InsertRec(t,NewNode(1));
    InsertRec(t,NewNode(1));
267 InsertRec(t,NewNode(1));
    InsertRec(t,NewNode(1));
269 InsertRec(t,NewNode(1));
    InsertRec(t,NewNode(1));
271 InsertRec(t,NewNode(1));
    Balance(t);

```

273 **END.**`balancedtree.pas`

```
C:\WINDOWS\system32\cmd.exe

C:\Users\Andreas\Documents\GitHub\SE-Hagenberg\1. Semester\ADE\Uebung10>balancedtree.exe
  Binary Tree
Balancing....
Array:
1 2 3 4 5 6 7 8 9
Old height: 9
New height: 4

---- new tree ----

Balancing....
Array:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Old height: 19
New height: 6

---- new tree ----

Balancing....
Array:
1 1 1 1 1 1 1 1 1
Old height: 9
New height: 9

C:\Users\Andreas\Documents\GitHub\SE-Hagenberg\1. Semester\ADE\Uebung10>
```

Abbildung 1: Testfälle

Testfälle

Die Testfälle zeigen drei verschiedene Bäume mit unterschiedlicher Länge und Werten. Anhand des dritten Baumes wird ersichtlich das bei gleichen Werten die Höhe des Baumes nach dem Balancieren die gleiche ist wie davor.

Zusatzfrage

Da bei diesem Baum nicht extra spezifiziert wurde was mit gleichen Werten geschehen soll wird immer nur in eine Richtung eingefügt, die Zahl ist schließlich nicht größer als die Wurzel oder der jeweilige Knoten. Damit ergibt sich selbst nach dem ausführen des Algorithmus immer noch dieselbe Höhe.