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
☒ **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 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?

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 CountNodes 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.

```
PROGRAM balancedtree;

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

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

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

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

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

```
  ELSE
    InsertRec(t^.right, n)
  END;
END;

(* Adds a node to the Array *)
PROCEDURE AddToArray(VAR arr : ARRAY OF Node; n : Node);
VAR count : INTEGER;
BEGIN
  count := 0;
  WHILE (arr[count] <> NIL) AND (count <= High(arr)) DO
    count := count + 1;

  IF count <= High(arr) THEN
    arr[count] := n
  ELSE
    WriteLn('Array already full!');
END;

(* Adds Nodes to the Array, sorted *)
PROCEDURE SaveTreeInOrder (t: Tree; VAR b_array : ARRAY OF Node);
BEGIN
  IF t <> NIL THEN
  BEGIN
    SaveTreeInOrder(t^.left, b_array);
    AddToArray(b_array, t);
    SaveTreeInOrder(t^.right, b_array);
  END;
END;

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

(* Count the nodes in a tree *)
PROCEDURE CountNodes (t: Tree; VAR count : INTEGER);
BEGIN
  IF t <> NIL THEN
  BEGIN
    CountNodes(t^.left, count);
    count := count + 1;
    CountNodes(t^.right, count);
  END;
END;

(* Returns the height of the tree *)
FUNCTION Height(t: Tree) : INTEGER;
VAR
```

```
  hl, hr: INTEGER;
BEGIN
  IF t = NIL THEN
    Height := 0
  ELSE BEGIN

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

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

FUNCTION CalculateHalf(i : LONGINT): INTEGER;
BEGIN

  IF i/2 = i DIV 2 THEN
    CalculateHalf := Round(i / 2 )
  ELSE
    CalculateHalf := (i DIV 2) + 1;
END;

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

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

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

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

```
    SaveTreeInOrder(t, b_array);
    PrintArray(b_array);

    WriteLn('Old height: ', Height(t));

    (* Has to be done after height function because the pointer left & right
   are
        set to NIL in cleanarray (the height after that will be 1) *)
    CleanArray(b_array);

    length := High(b_array)+1;
    pos := length;

    WHILE pos > 2 DO
    BEGIN
      pos := CalculateHalf(pos);
      InsertRec(t_temp, b_array[pos-1]);
      b_array[pos-1] := NIL;
    END;

    pos := CalculateHalf(length);

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

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

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

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

VAR
  t : Tree;

```

```
BEGIN

  WriteLn(chr(205),chr(205),chr(185),' Binary Tree ',chr(204),chr(205),chr(205)
    );

  InitTree(t);
  InsertRec(t,NewNode(1));
  InsertRec(t,NewNode(2));
  InsertRec(t,NewNode(3));
  InsertRec(t,NewNode(4));
  InsertRec(t,NewNode(5));
  InsertRec(t,NewNode(6));
  InsertRec(t,NewNode(7));
  InsertRec(t,NewNode(8));
  InsertRec(t,NewNode(9));
  Balance(t);

  DisposeTree(t);

  WriteLn(#13#10, '------ new tree ------', #13#10);

  InitTree(t);
  InsertRec(t,NewNode(1));
  InsertRec(t,NewNode(2));
  InsertRec(t,NewNode(3));
  InsertRec(t,NewNode(4));
  InsertRec(t,NewNode(5));
  InsertRec(t,NewNode(6));
  InsertRec(t,NewNode(7));
  InsertRec(t,NewNode(8));
  InsertRec(t,NewNode(9));
  InsertRec(t,NewNode(10));
  InsertRec(t,NewNode(11));
  InsertRec(t,NewNode(12));
  InsertRec(t,NewNode(13));
  InsertRec(t,NewNode(14));
  InsertRec(t,NewNode(15));
  InsertRec(t,NewNode(16));
  InsertRec(t,NewNode(17));
  InsertRec(t,NewNode(18));
  InsertRec(t,NewNode(19));
  Balance(t);

  DisposeTree(t);

  WriteLn(#13#10, '------ new tree ------', #13#10);

  InitTree(t);
  InsertRec(t,NewNode(1));
  InsertRec(t,NewNode(1));
  InsertRec(t,NewNode(1));
  InsertRec(t,NewNode(1));
  InsertRec(t,NewNode(1));
  InsertRec(t,NewNode(1));
  InsertRec(t,NewNode(1));
  InsertRec(t,NewNode(1));
  InsertRec(t,NewNode(1));
  Balance(t);
```

```
END.
```

balancedtree.pas

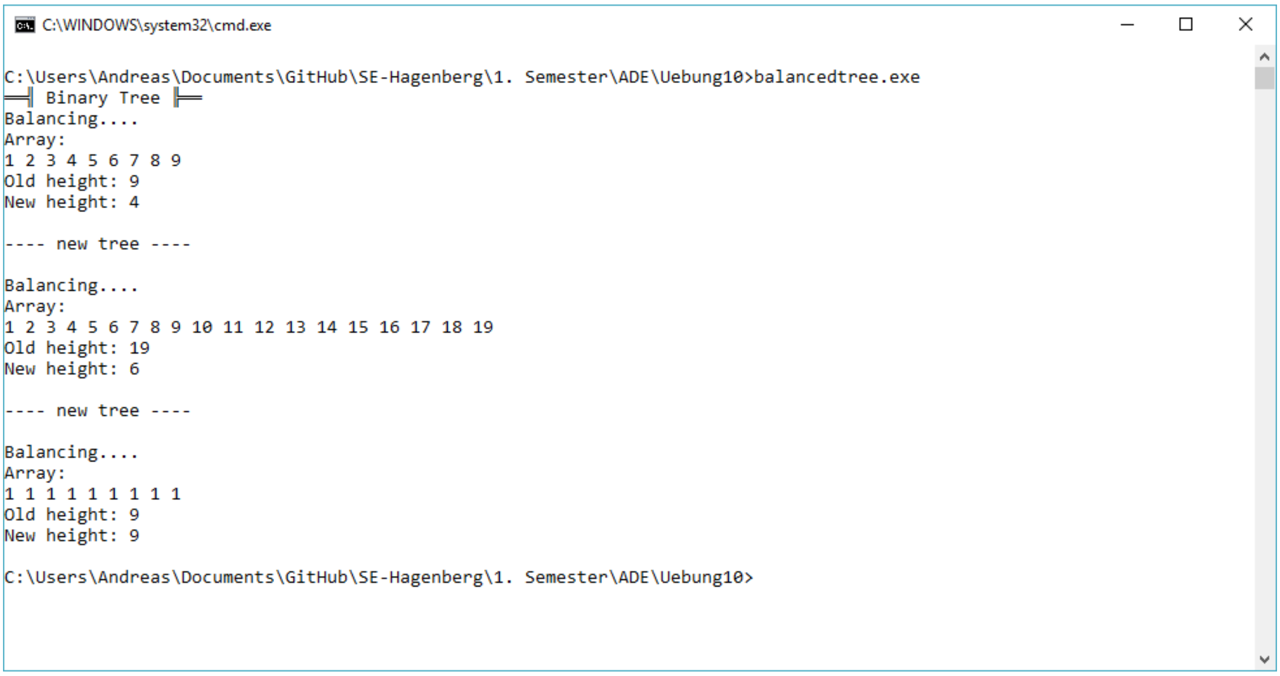

Abbildung 1: Testfälle

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.

Aufgabe 2

Anhand dieser Funktion und mit der nachfolgenden Tabelle soll eine Formel erstellt werden mit der die Laufzeit der Funktion berechnet werden kann.

```
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*)
```

code_2.pas

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

Tabelle 1: Laufzeiten

Damit ergibt sich folgende Formel: $26 * length(input) + 1,5 * Count1(input) + 22$

Length(input) bestimmt die Länge bestimmt die Länge der Eingabe, Count1 steht für die Anzahl der vorkommenden '1'.

Für die Binärzahlen 100100, 100001, 110100, 1111, 0000, 1 und 0 ergeben sich folgende Laufzeiten:

Input	Laufzeitlänge
100100	181
100001	181
110100	182,5
1111	132
0000	126
1	49,5
0	48

Tabelle 2: Laufzeiten Formel

Vereinfachung

Unter der Annahme das die Anzahl der vorkommenden '1' etwa der Anzahl der vorkommenden '0' entspricht, kann die Formel vereinfacht werden: $26,75 * length(input) + 22$

Damit ergeben sich bei den vorgegebenen Längen folgende Laufzeiten:

Länge	Laufzeitlänge
1	48,75
2	75,5
3	102,25
4	129
5	155,75
6	182,5
7	209,25
8	236
9	262,75
10	289,5
11	316,25
12	343
13	369,75
14	396,5
15	423,25
16	450
17	476.75
18	503.5
19	530.25
20	557
50	1.359,5
100	2697
200	5.372

Tabelle 3: Laufzeiten vereinfachte Formel

Asymptotische Laufzeitkomplexität

Da die Formel eine lineare Funktion beschreibt(konstante Steigung), ist die asymptotische Laufzeitkomplexität linear. In der Tabelle sieht man das bei einer Änderung der Wortlänge um 1 jedes mal ein konstanter Wert hinzugefügt wird.

Aufgabe 3

Es wird wieder eine Tabelle mit nachfolgendem Code und mit der oben erwähnten Tabelle Laufzeiten. Dabei ergibt sich folgende Formel: $2,5*length(input)+23*Count1(input)+22,5*length(input)-count1(input)+39$

```
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*)
```

code_3.pas

Für die Binärzahlen 100100, 100001, 110100, 1111, 0000, 1 und 0 ergeben sich folgende Laufzeiten:

Input	Laufzeitlänge
100100	190
100001	190
110100	190,5
1111	141
0000	139
1	64,5
0	64

Tabelle 4: Laufzeiten Formel

Vereinfachung

Unter der Annahme das die Anzahl der vorkommenden '1' etwa der Anzahl der vorkommenden '0' entspricht, kann die Formel vereinfacht werden: $48 * length(input) + 39$

Damit ergeben sich bei den vorgegebenen Längen folgende Laufzeiten:

Länge	Laufzeitlänge
1	87
2	135
3	183
4	231
5	279
6	327
7	375
8	423
9	471
10	519
11	567
12	615
13	663
14	711
15	759
16	807
17	855
18	903
19	951
20	999
50	2.439
100	4.839
200	9.639

Tabelle 5: Laufzeiten vereinfachte Formel

Die asymptotische Laufzeitkomplexität ist wieder linear. Man kann beobachten das der rekursive Algorithmus eine höhere Laufzeit hat als der iterative Algorithmus. Das liegt vor allem daran das bei der rekursiven Lösung immer wieder ein Funktionsaufruf stattfindet.