3.2 Binäre Suche

Beispiel 6.5.1: Intervallschachtelung (oder binäre Suche)
(Hier ist n die Anzahl der Elemente im Feld!)

Ein Feld A: array (1..n) of Integer sei gegeben. Das Feld sei sortiert, d.h.: $A(i) \leq A(i+1)$ für $i = 1, 2, ..., n-1$.

Aufgabe: Man schreibe einen Algorithmus, der zu einer Zahl S in möglichst kurzer Zeit feststellt, ob S im Feld A liegt oder nicht. Im Falle, dass S im Feld A enthalten ist, soll ein Index m mit $A(m) = S$ ausgegeben werden, anderenfalls sei $m = 0$.

Geht man das Feld von links nach rechts durch, so dauert es bis zu n Schritte, um das Ergebnis zu ermitteln.

Ein schnelleres Verfahren ist die Intervallschachtelung: Teste, ob S genau in der Mitte A(Mitte) von A liegt, falls nein und es ist $A(\text{Mitte}) < S$, suche rechts von der Mitte weiter, sonst links.

3.2 Binäre Suche (in Ada programmiert)

```
procedure Search1 is
Mitte, Links, Rechts, S: Integer; Gefunden: Boolean;
A: array (1..100_000) of Integer;
begin  ...;  -- "lies n, das Feld A und die zu suchende Zahl S ein"
      Links:=1; Rechts := n; Gefunden := false;
      while (Links <= Rechts) and (not Gefunden) loop
            Mitte := (Rechts+Links) / 2;
            if A(Mitte) = S then Gefunden := true;
            elsif A(Mitte) < S   then Links := Mitte+1;
                                 else Rechts := Mitte-1; end if;
      end loop;
      if not Gefunden then Mitte := 0; end if;
      ...;                           -- "drucke das Ergebnis Mitte aus"
end Search1;
```

Wie lange dauert es, bis die binäre Suche spätestens endet?

Hierzu zählen wir die Wertzuweisungen und Bedingungen, die im ungünstigsten Fall ausgerechnet werden müssen.
In diesem Sinne dauert die Durchführung der 3 Anweisungen

Links:=1; Rechts := n; Gefunden := false;

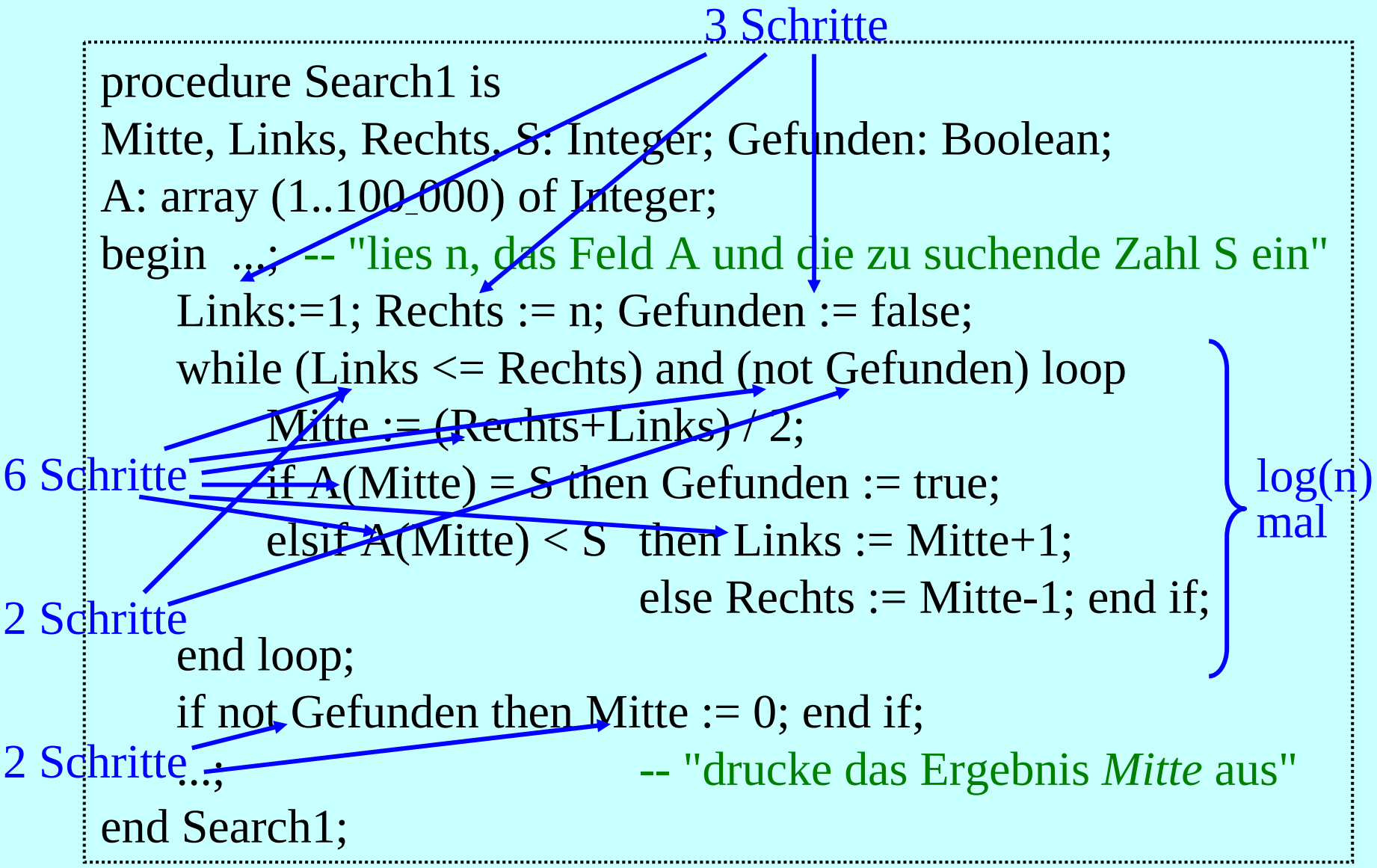
genau 3 Schritte.

In der Schleife werden bis zu 6 Schritte benötigt, nämlich je einer für die vier Bedingungen (Links <= Rechts), (not Gefunden), A(Mitte) = S und A(Mitte) < S
und je einer für zum Beispiel die Wertzuweisungen

Mitte := (Rechts+Links) / 2; und Links := Mitte+1;

[Zählt man das and noch zusätzlich, so sind es sogar bis zu 7 Schritte; das Folgende ist dann entstprechend anzupassen.]

Wie oft wird die Schleife durchlaufen? Das Intervall von Links bis Rechts halbiert sich mindestens in jedem Schritt, folglich muss nach log(n) Schleifendurchläufen Schluss sein.



Gesamt: 6·log(n) + 7 Schritte ∈ O(log(n))

Aufwand der binären Suche

Schlechtester Fall (der "worst case") tritt ein, wenn das gesuchte Element S nicht im Feld A enthalten ist. Wir sagen: Die *uniforme worst case Zeitkomplexität t(n)* des Programms Search1 lautet

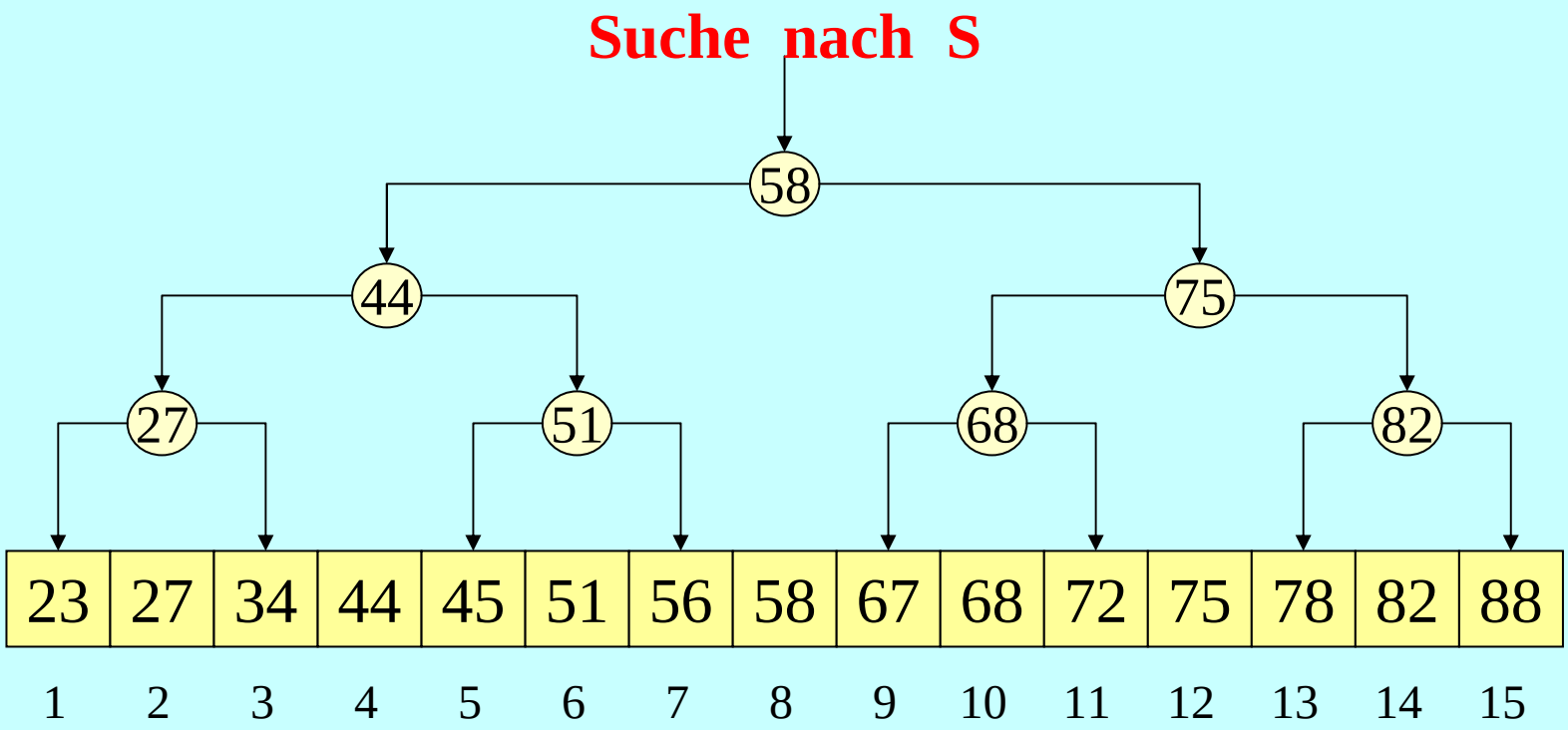
$$t(n) = 6 \cdot \log(n) + 7.$$

Beachten Sie: n ist hier die Anzahl der Elemente im Feld A. "Uniform", weil wir annehmen, alle Wertzuweisungen und Bedingungen würden die gleiche Zeit kosten.

Bester Fall (best case): In diesem Fall wird S im ersten Durchgang der while-Schleife gefunden, also nach 13 Schritten.

Durchschnittlicher Fall (average case): Hierzu nehmen wir an, dass sich das gesuchte Element S tatsächlich im Feld A befindet (sonst kann man nur die obige worst case Abschätzung verwenden).

Wir skizzieren die Verhältnisse, wobei wir hier $n=15=2^4-1$ wählen:



In $2^3 = 8$ Fällen braucht man 4 Schleifendurchläufe,
in $2^2 = 4$ Fällen braucht man 3 Schleifendurchläufe,
in $2^1 = 2$ Fällen braucht man 2 Schleifendurchläufe,
in $2^0 = 1$ Fall braucht man 1 Schleifendurchlauf.

Allgemein gilt also, wenn $n = 2^k - 1$ ist:

In 2^{k-1} Fällen braucht man k Schleifendurchläufe,
in 2^{k-2} Fällen braucht man k-1 Schleifendurchläufe,
in 2^{k-3} Fällen braucht man k-2 Schleifendurchläufe,
....
in $2^0 = 1$ Fall braucht man 1 Schleifendurchlauf.

Daher braucht man im Mittel:

$$\frac{1}{n}\left(k\cdot 2^{k-1} + (k-1)\cdot 2^{k-2} + (k-2)\cdot 2^{k-3} + \ldots + 2\cdot 2^1 + 1\right) \text{ Durchläufe.}$$

Berechne also die Summe $\sum_{j=1}^{k} j\cdot 2^{j-1} = \frac{1}{2}\sum_{j=1}^{k} j\cdot 2^j$

$$\sum_{j=1}^{k} j\cdot 2^{j-1} = \frac{1}{2}\sum_{j=1}^{k} j\cdot 2^{j} = \frac{1}{2}\sum_{j=1}^{k}(j-1)\cdot 2^{j} + \frac{1}{2}\sum_{j=1}^{k} 2^{j}$$

$$= \sum_{j=1}^{k}(j-1)\cdot 2^{j-1} + \frac{1}{2}(2^{k+1} - 2)$$

$$= \sum_{j=0}^{k-1} j\cdot 2^{j} + (2^{k} - 1) = \sum_{j=1}^{k} j\cdot 2^{j} - k\cdot 2^{k} + (2^{k} - 1), \quad \text{d.h.:}$$

$\frac{1}{2}\sum_{j=1}^{k} j\cdot 2^{j} = k\cdot 2^{k} - (2^{k} - 1)$. Folglich erhalten wir

$$\frac{1}{n}(k\cdot 2^{k-1}+(k-1)\cdot 2^{k-2}+\ldots+2\cdot 2^{1}+1) = \frac{k\cdot 2^{k} - (2^{k} - 1)}{2^{k} - 1} \approx k - 1$$

Somit beträgt die *average case Zeitkomplexität* der Intervall-schachtelung ziemlich genau $6\cdot\log(n) + 1$ Schritte, also nur einen Schleifendurchlauf weniger als im schlechtesten Fall.

Erkenntnis: Im Mittel spart man nur eine konstante Zahl an Operationen gegenüber dem schlechtesten Fall. Folglich lohnt sich zum Beispiel die Abfrage "A(Mitte) = S" nicht! Könnte man sie weglassen, so würde man log(n) viele Schritte sparen. Dies führt auf folgende bessere Version des Algorithmus für die Suche mittels Intervallschachtelung:

Man entscheide erst ganz am Ende, ob A(Mitte) = S ist; hierzu muss man im Falle A(Mitte) < S im rechten Teil des Feldes weitersuchen (Links:=Mitte+1), anderenfalls im linken Teil einschließlich des gerade betrachteten Feldes Mitte (Rechts:=Mitte, statt "Mitte+1", da A(Mitte) gleich S sein könnte).

Verbesserung (in Ada programmiert)

```
procedure Search2 is
Mitte, Links, Rechts, S: Integer; Gefunden: Boolean;
A: array (1..100_000) of Integer;
begin  ...;  -- "lies n, das Feld A und die zu suchende Zahl S ein"
      Links:=1; Rechts := n;
      while (Links < Rechts) loop
            Mitte := (Rechts+Links) / 2;
            if A(Mitte) < S   then Links := Mitte+1;
                              else Rechts := Mitte; end if;
      end loop;
      Gefunden := A(Mitte) = S;
      if not Gefunden then Mitte := 0; end if;
      ...;                          -- "drucke das Ergebnis Mitte aus"
end Search2;
```

Interpolationssuche

Lässt sich eine der Operationen noch beschleunigen? Unter der folgenden Bedingung, ja, für die Operation FIND.

Beachte: Bei der Intervallschachtelung (=binäre Suche) halbieren wir jeweils das gesamte noch verbleibende Feld A(links..rechts). Als Teilungsindex "Mitte" wählen wir:
Mitte = (rechts + links) / 2 = links + (rechts - links) / 2.

Verbesserung: Kann man mit den Schlüsseln "rechnen" und sind die Schlüssel recht gleichmäßig über den Indexbereich verteilt, so kann man den ungefähren Bereich, wo ein Schlüssel s im Feld A(links..rechts) liegen muss, genauer angeben durch den folgenden Teilungsindex

$$p = links + \frac{s - A(links)}{A(rechts) - A(links)} (rechts - links).$$

Interpolationssuche

So geht man beispielsweise beim Suchen in einem Lexikon vor.

Man kann zeigen, dass mit dieser "Interpolationssuche" die Operation FIND bei einem geordneten Feld nur noch den uniformen Zeitaufwand O(log(log(n))) benötigt. Falls die Schlüssel gleichverteilt sind, so braucht man für den gesamten Suchprozess nur mit $1 \cdot \log(\log(n)) + 1$ Schritten zu rechnen.

Da log(log(n)) eine sehr schwach wachsende Funktion ist, sollte man die Interpolationssuche einsetzen, wo immer es möglich ist.
(In der Praxis liegt log(log(n)) fast immer unterhalb von 10.)

4.1: Rekursive Definition k-näre Bäume

Man kann k-näre Bäume leicht rekursiv definieren; sei $k \in \mathbb{N}$:

1) Die leere Menge ist ein Baum.

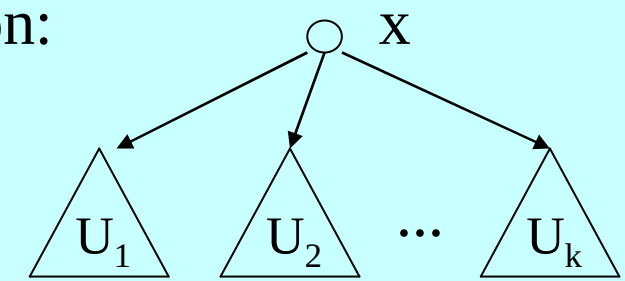
2) Wenn x ein Knoten und $U=\{U_1, U_2, ..., U_k\}$ eine geordnete Menge von k Bäumen ist, so ist auch $x(U)$ ein Baum.

x bildet die Wurzel des Baums $x(U)$, die Elemente von U sind Unterbäume oder Teilbäume im Baum $x(U)$.

Skizze: Leerer Baum: $\varnothing$ Tiefe dieses Baumes = 0

gerichtet oder ungerichtet

Rekursion:



Tiefe dieses Baumes = Max (Tiefe eines U_i) + 1.

Die k Nachfolger von x sind hier geordnet.

Definition Verwandtschaften

Es sei G=(V, E) ein Baum.

Ist G ein gerichteter Baum mit Wurzel w, so ist |Vor(x)| = 1 für alle Knoten x≠w und |Vor(w)| = 0. Der eindeutig bestimmte Knoten vorg(x) = y∈ Vor(x) heißt direkter Vorgänger oder Elternknoten oder Vaterknoten von x. Jeder Knoten, der auf dem Weg von der Wurzel w nach x liegt heißt Vorfahr von x (aber nicht x selbst). Verschiedene Knoten, die den gleichen direkten Vorgänger besitzen, heißen Geschwister oder Brüder. Jeder Knoten x∈ Nach(y) heißt direkter Nachfolger oder Kind oder Sohn von x.

Gerichteter Baum: Ein Knoten x mit x≠w und mit |Nach(x)| = 0 heißt Blatt des Baums.
Ungerichteter Baum: Ein Knoten x mit x≠w und mit |N(x)| = 1 heißt Blatt des Baums.

Folgerungen

(a) In einem ungerichteten Baum gibt es von jedem Knoten zu jedem Knoten *genau* einen doppelpunktfreien Weg.

(b) In jedem gerichteten Baum gibt es zu jedem Knoten $x \in V$ *genau* einen Weg von der Wurzel w nach x.

(c) Wenn es in einem gerichteten Baum einen Weg vom Knoten x zum Knoten y gibt, dann führt der Weg von der Wurzel w nach y über den Knoten x.

Definition Tiefe eines Baumes

Definition: Es sei G=(V, E) ein Baum mit Wurzel w.

(1) Die Anzahl der Knoten in einem längsten Weg von der Wurzel zu einem Blatt heißt die Tiefe des Baumes G. (Dies ist also die Länge des längsten Weges im Baum, der von w ausgeht, plus 1.)

(2) Zu jedem Baum mit Wurzel w gehört die Levelfunktion level: V $\rightarrow \mathbb{N}_0$, rekursiv definiert durch

level(w) = 1 und

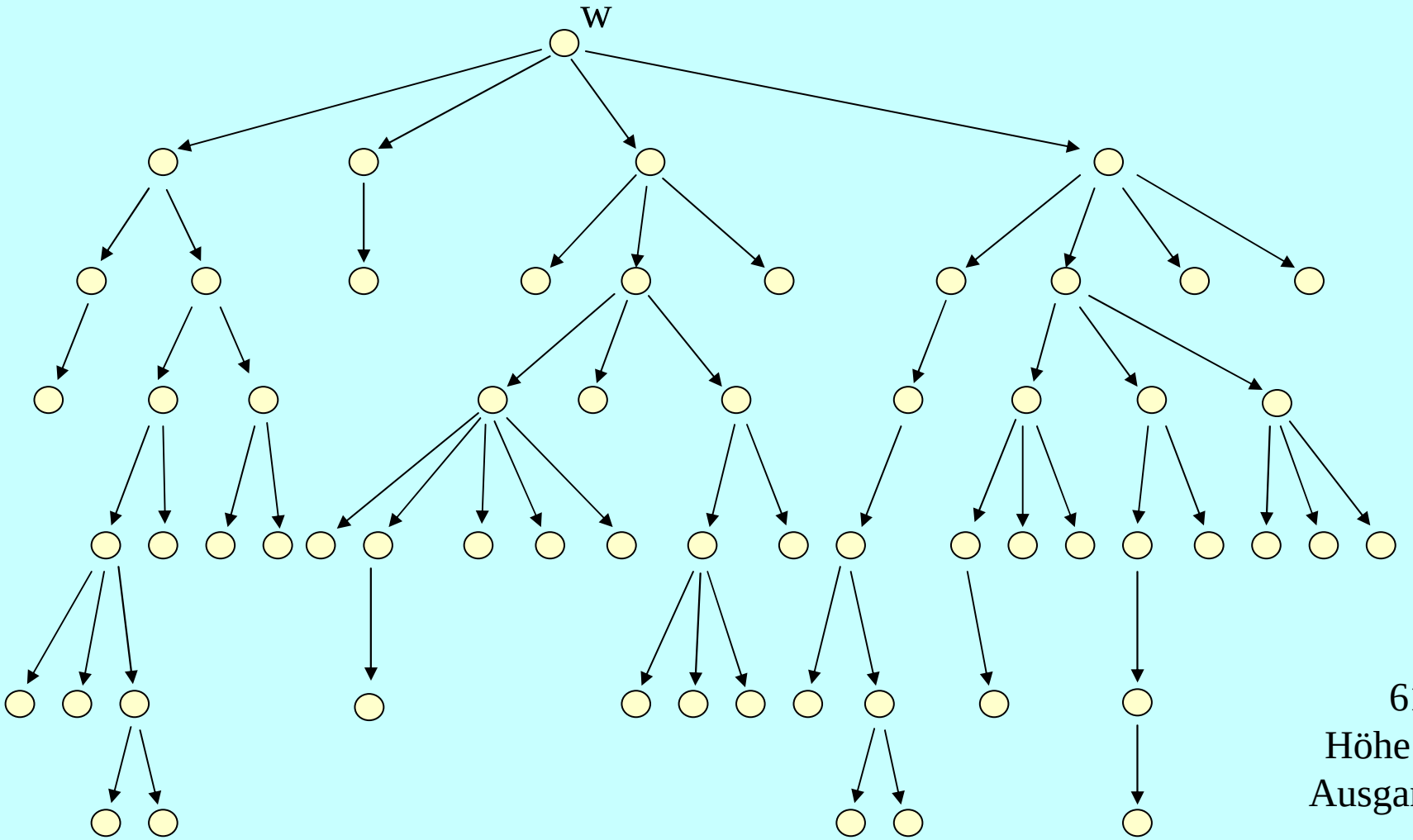
level(x) = level(vorg(x)) + 1 für $x \neq w$.

Hinweis: Das maximale Level kann offenbar nur von einem Blatt angenommen werden. Die Tiefe des Baumes ist das maximale Level eines Knotens x im Baum:

Tiefe von G = Max{level(x) | $x \in V$}.

Weiterhin wird auch der leere Baum mit Tiefe 0 zugelassen.

Beispiel für einen "beliebigen geordneten Baum":



61 Knoten,
Höhe = Tiefe 7,
Ausgangsgrad 5

Definition: (binäre) Suchbäume

Binäre Bäume sind gerichtete und geordnete Bäume, bei denen jeder Knoten genau einen linken und einen rechten Nachfolger besitzt (diese Nachfolger können auch leer sein; *wichtig ist, dass der linke und der rechte Nachfolger stets unterschieden werden*).

```
typedef struct BinBaum {
        Int Inhalt;
         BinBaum * L, R;
}
```

Ein binärer Baum, dessen Inhalts-Datentyp geordnet ist (z.B. ganze Zahlen), heißt (binärer) Suchbaum, wenn für jeden Knoten u gilt:

Alle Inhalte von Knoten im linken Unterbaum von u sind echt kleiner als der Inhalt von u und alle Inhalte von Knoten im rechten Unterbaum von u sind größer oder gleich dem Inhalt von u.