Teil 3 der Grundvorlesung

3. Grundlegende Verfahren

~~3.1 Speicherverwaltung~~

3.2 Suchverfahren

3.3 Hashing

3.4 Sortieren

3.5 Algorithmen auf Graphen

3.6 Zufallszahlen

Gliederung des Kapitels

3.2 Suchverfahren

3.2.1 Suchen in sequentiellen Strukturen

3.2.2 Bäume und (binäre) Suchbäume

3.2.3 Optimale Suchbäume

3.2.4 Balancierte Bäume (insbesondere AVL-Bäume)

3.2.5 B-Bäume

3.2.6 Digitale Suchbäume (Tries)

Grundaufgabe des Suchens:

Gegeben: Menge $A = \{a_1, ..., a_n\} \subseteq B$ sowie ein Element $b \in B$.
Realisiere A in einer geeigneten Datenstruktur, so dass die folgenden drei Operationen "effizient" durchgeführt werden:

- Entscheide, ob b in A liegt (und gib ggf. an, wo). FIND
- Füge b in A ein. INSERT
- Entferne b aus A. DELETE

Statt des meist sehr umfangreichen Elements b betrachten wir nur eine Komponente s von b, durch die b eindeutig bestimmt ist. Dieses s nennen wir "Suchelement" oder meistens "**Schlüssel**" (englisch: "**key**").

Weitere Aufgaben: Wähle eine Datenstruktur so, dass alle oder einige der folgenden Tätigkeiten effizient durchführbar sind.

Gegeben seien zwei Mengen $A_1, A_2 \subseteq B$.

- Vereinige A_1 und A_2. UNION
- Schneide A_1 und A_2. INTERSECTION
- Bilde das Komplement $B \setminus A_1$. Complement
- Entscheide, ob A_1 leer ist. EMPTINESS
- Entscheide, ob $A_1 = A_2$ ist. EQUALITY
- Entscheide, ob $A_1 \subseteq A_2$ ist. SUBSET
- Entscheide, ob $A_1 \cap A_2$ leer ist. Empty Intersection

Annahme: Die Mengen haben keine Struktur (insbesondere sind sie nicht geordnet).

In diesem Fall muss man die die Elemente der Menge "wie sie kommen" in eine Liste einfügen.

Worst-Case-Aufwand der drei Operationen der Grundaufgabe, wenn die Menge A genau n Elemente enthält:

FIND: Durchlauf durch die Auflistung, also O(n).

INSERT: Füge das neue Element am Anfang ein: O(1). (Elemente treten dann mehrfach in der Struktur auf. Will man dies nicht, dann: O(n).)

DELETE: Zunächst das Element finden, dann aus der Auflistung entfernen: O(n).

Wir nehmen im Folgenden stets an, dass die zugrunde liegenden Mengen angeordnet sind.

Grund: Alle Elemente werden im Rechner durch eine Folge von Nullen und Einsen dargestellt. Dies impliziert eine (lexikografische) Anordnung, die wir stets ausnutzen können.

3.2.1 Suchen in "flachen" Strukturen

Flache oder sequentielle Strukturen sind üblicherweise
- (eindimensionale) Felder (siehe 1.3.2.3),
- Listen (siehe 1.3.3.1),
wobei egal ist, ob diese Strukturen linear oder zyklisch sind und ob die Listen einfach oder doppelt verkettet werden.

In eindimensionalen Feldern sucht man nach einem Element s, indem man das Feld linear durchläuft. Ist das Feld geordnet, so kann man eine binäre Suche (Intervallschachtelung) durchführen. Listen werden prinzipiell von vorne nach hinten oder von hinten nach vorne durchsucht. Im Falle zyklischer Strukturen muss man sich mit einem Index oder einem Zeiger merken, ab wo die Suche begonnen wurde.

Durchsuchen eines linearen Feldes:

```
s: element; i: integer; A: array (1..n) of element;
....; i:=1;
while i <= n and then A(i) /= s loop i := i+1; end loop;
if i <= n then < das gesuchte Element s steht an der Position i >
else  < s ist nicht im Feld A vorhanden > end if; ...
```

Mit einem Stopp-Element kann man eine Abfrage sparen:

```
s: element; i, n: positive; A: array (1..n+1) of element;
....; i:=1; A(n+1) := s;        -- Stopp-Element setzen!
while A(i) /= s loop i := i+1; end loop;
if i <= n then < das gesuchte Element s steht an der Position i >
else  < s ist nicht im Feld A vorhanden > end if; ...
```

Lineares Durchsuchen einer nicht-zyklischen Liste:

Wir erinnern an 1.3.3.1.(5): Suche s in einer Liste:

```
p := Anker;
while p /= null and then p.inhalt /= s loop
                    p := p.next;  end loop;
if p = null then < s ist nicht in der Liste enthalten >
else < p verweist auf das erste Element mit Inhalt s > end if;
```

Aufwand:
Zeit: Die lineare Suche erfordert O(n) Vergleichsschritte, dauert also relativ lange. Im Falle binärer Suche (geordnetes Feld) benötigt man maximal O(log(n)) Vergleiche. Eine genaue Analyse hierzu erfolgte bereits in Abschnitt 1.3.2.3.d, wobei dort zwei Programme im Detail untersucht wurden.
Platz: Es sind nur wenige Speicherplätze zusätzlich erforderlich.

Wie sieht es mit den Operationen der Grundaufgabe aus?

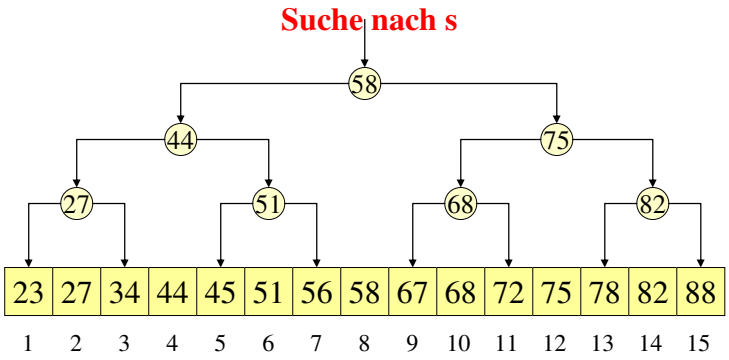
Wir wählen ein array als Datenstruktur, in das die Menge der Größe nach geordnet eingetragen wird. Dann

- dauert FIND nur O(log(n)) Schritte, sofern man ein geordnetes array verwendet und hierauf mit der Intervallschachtelung sucht,
- dauert INSERT aber O(n) Schritte, da beim Einfügen alle größeren Elemente um eine Komponente nach hinten verschoben werden müssen,
- dauert DELETE ebenfalls O(n) Schritte, da beim Löschen alle größeren Elemente um eine Komponente nach vorne verschoben werden müssen.

Darstellung als Liste: Klar (selbst ausführen).

Erinnerung an Abschnitt 1.3.2.3.d, Darstellung durch ein array:

Hier ist $n=15=2^4-1$:



Die Tiefe dieses Index-Baums, der über dem array liegt, beträgt hier 4, so dass man spätestens nach 4 Schritten (allgemein: nach log(n) Schritten) die Suche beenden kann.

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

Bei der Intervallschachtelung (=binäre Suche) halbieren wir jeweils das gesamte noch verbleibende Feld A(links..rechts). Als Teilungsindex t wählen wir stets die Mitte:
t = (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 folgenden Teilungsindex

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

So geht man beispielsweise beim Suchen in einem Lexikon vor.

Man kann zeigen, dass mit dieser "Interpolationssuche" die Operation FIND nur noch den uniformen Zeitaufwand O(log(log(n))) benötigt und, falls die Schlüssel gleichverteilt sind, so braucht man 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.)

Darstellung der Menge durch Bitvektoren

Da $A = \{a_1, ..., a_n\} \subseteq B = \{b_1, ..., b_r\}$ eine Teilmenge ist (mit $a_i \leq a_j$ und $b_i \leq b_j$ für $i < j$), kann man A auch durch einen Bitvektor $x = (x_1, ..., x_r)$, mit $x_i \in \{0, 1\}$, der Länge r darstellen, wobei für alle i gilt: $x_i = 1 \Leftrightarrow b_i \in A$.

Wird nach dem Element $s \in B$ gesucht und kennt man die Nummer, die s in der Anordnung von B trägt (also dasjenige i mit $s=b_i$), dann gilt für eine Teilmenge A mit Bitvektor x:

FIND: $s = b_i \in A \Leftrightarrow x_i = 1$.
INSERT: Setze $x_i := 1$.
DELETE: Setze $x_i := 0$.

Im uniformen Komplexitätsmaß läuft dann alles in O(1), also in konstanter Zeit ab!

Dennoch verwendet man diese Darstellung mit Bitvektoren nur selten, weil in der Praxis B meist sehr groß (wenn nicht sogar unendlich groß) ist. Auch benötigen alle Operationen der "weiteren Aufgaben" (UNION, INTERSECTION usw.) den Aufwand O(r), während man in der Praxis höchstens auf O(n) kommen darf (n = Größe der betrachteten Teilmengen von B).

Ist dagegen die Kardinalität $r = |B|$ relativ klein, dann sind Bitvektoren eine geeigente und vor allem eine leicht zu implementierende Darstellung für Mengen; auch die üblichen Mengenoperationen wie Vereinigung, Durchschnitt usw. lassen sich leicht implementieren.

3.2.2 Bäume und (binäre) Suchbäume

Wiederholen Sie bitte die Abschnitte 1.1.5.6 bis 1.1.5.8, 1.3.3.3 und 1.6.4.3. Dort werden die Begriffe Baum, Wurzel, Vorgänger, direkter Vorgänger, Blatt, Höhe usw. erläutert.

Wir fassen diese Begriffe nochmals knapp in der folgenden Definition zusammen. Wir setzen voraus, dass aus den Abschnitten aus 1.3.3.4 und 1.6.4.3 die Begriffe ungerichteter und gerichteter Graph, Nachbar (im ungerichteten Fall) bzw. Vorgänger und Nachfolger (im gerichteten Fall), Weg in einem Graphen, Kreis (oder Zyklus), azyklischer Graph, Zusammenhang und Zusammenhangskomponente bekannt sind.

Definition 3.2.2.1: Es sei G=(V, E) ein Graph, $|V| = n \geq 0$.

(1) Ein Knoten $w \in V$ heißt Wurzel von G, wenn es von w zu *jedem* Knoten des Graphen einen Weg gibt (im gerichteten Fall muss der Weg natürlich auch gerichtet sein).

(2) Der ungerichtete Graph G = (V, E) heißt Baum, wenn er azyklisch und zusammenhängend ist (insbesondere ist dann jeder Knoten des Baums auch Wurzel).

(3) Der gerichtete Graph G = (V, E) heißt Baum, wenn er eine Wurzel w besitzt, die keinen Vorgänger hat, und jeder Knoten ungleich der Wurzel genau einen Vorgänger besitzt, d.h., zu jedem $x \in V$, $x \neq w$ gibt es genau einen Knoten y mit $(y, x) \in E$, und es gibt kein $u \in V$ mit $(u,w) \in E$.

(4) Ein Graph heißt Wald, wenn alle seine (schwachen) Zusammenhangskomponenten Bäume sind.

Folgerung: Überzeugen Sie sich von folgenden Aussagen:

(a) Es sei G=(V, E) ein ungerichteter Baum. Wähle irgendeinen Knoten w als Wurzel aus und ersetze jede ungerichtete Kante {x,y} durch die gerichtete Kante (x,y), wobei x näher an der Wurzel liegt als y, d.h., die Richtung zeigt stets von der Wurzel weg. So erhält man aus G in eindeutiger Weise den gerichteten Baum G'=(V, E') mit Wurzel w.
Anwendung für die Programmierung: Man kann einen Baum stets als gerichteten Baum auffassen/implementieren.

(b) Es sei G'=(V, E') ein gerichteter Baum mit Wurzel w. Ersetze jede gerichtete Kante (x,y) durch die ungerichtete Kante {x,y}, so erhält man aus G' in eindeutiger Weise den ungerichteten Baum G=(V, E).

Folgerung (a) begründet, warum wir Bäume mit Hilfe von Zeigern darstellen. Fügt man noch für jeden Knoten einen Inhalt hinzu, so erhält man die Ada-Darstellung, die im wesentlichen in Abschnitt 1.3.3.3 vorgestellt wurde. Hierbei ist MaxG der maximale Ausgangsgrad des Baums:

```
MaxG: constant Positive := ...; type Grad is 1..MaxG;
type Element is ...;
type Baum;  type Ref_Baum is access Baum;
type Baum (ausgangsgrad: Grad) is record
        Inhalt: Element;
        Nachf: array (1..ausgangsgrad) of Ref_Baum;
end record;
```

Hier hat jeder Knoten mindestens einen Nachfolger. Knoten ohne Nachfolger müssen daher einen null-Zeiger erhalten.

Folgerung: Überzeugen Sie sich von folgenden Aussagen:

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

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

(e) 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 3.2.2.1 (Fortsetzung): Es sei G=(V, E) ein Graph.

(5) Zu jedem Knoten x eines ungerichteten Graphen G=(V, E) heißt N(x) = {y | {x,y} ∈ E } die Menge der *Nachbarn* von x. Ihre Anzahl |N(x)| heißt *Grad des Knotens* x. Der maximale Knotengrad heißt *Grad des Graphen* G.

(6) Zu jedem Knoten x eines gerichteten Graphen G=(V, E) heißt Vor(x) = {y | (y,x) ∈ E } die Menge der *Vorgänger* von x und Nach(x) = {y | (x,y) ∈ E } die Menge der *Nachfolger* von x. Ihre Anzahlen |Vor(x)| bzw. |Nach(x)| heißen *Eingangsgrad* bzw. *Ausgangsgrad des Knotens* x. Der jeweils maximale Grad heißt *Eingangsgrad* bzw. *Ausgangsgrad des Graphen* G.

Definition 3.2.2.1 (Fortsetzung): Es sei G=(V, E) ein Baum.

(7) 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 vater(x) = y ∈ Vor(x) heißt direkter Vorgänger oder Vater 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 Brüder. Jeder Knoten x ∈ Nach(y) heißt direkter Nachfolger oder Sohn von x.

(8) 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.

Folgerung: Überzeugen Sie sich von folgenden Aussagen:

(f) Es sei G=(V, E) ein Baum mit Wurzel w. Der von einem Knoten x in G erzeugte Unterbaum (oder Teilbaum) ist $G_x = (V_x, E_x)$ mit
$V_x = \{y \mid$ jeder doppelpunktfreie Weg von w nach y führt über x$\}$,
$E_x = E / V_x$ (= alle Kanten zwischen Knoten aus V_x).
Offenbar ist G_x ein Baum mit Wurzel x. Beachte, dass G_x nicht leer ist, da stets $x \in G_x$ gilt. Speziell ist G_w=G.

(g) Wenn es in einem gerichteten Baum einen Weg vom Knoten x zum Knoten y gibt, so liegt y in dem von x erzeugten Unterbaum.

Folgerung: Überzeugen Sie sich von folgenden Aussagen:

(h) Wenn G ein Baum mit n Knoten ist, so besitzt G genau n-1 Kanten (für n>0).

(i) Jeder Baum lässt sich mit zwei Farben färben, d.h., es gibt eine Abbildung $f: V \to \{1,2\}$ mit $f(x) \neq f(y)$ für alle Kanten $\{x,y\}$ bzw. (x,y).

Bäume sind 2-färbbar. Definition hierzu: Sei $k \in \mathbb{N}$.
Ein beliebiger Graph G=(V, E) lässt sich mit k Farben färben (man sagt auch, G ist k-färbbar), wenn eine Abbildung
$f: V \to \{1,2,...,k\}$
existiert mit $f(x) \neq f(y)$ für alle Kanten $\{x,y\}$ bzw. (x,y). Die minimale Zahl k, so dass sich G mit k Farben färben lässt, heißt Färbungszahl von G; sie zu bestimmen, heißt "Färbungsproblem". Diese Zahl lässt sich nach heutiger Kenntnis für beliebige Graphen nur mit großem Zeitaufwand berechnen.

Definition 3.2.2.1 (Fortsetzung): Es sei G=(V,E) ein Baum.

(9) Ist für jeden Knoten x die Menge N(x) bzw. im gerichteten Fall die Menge Nach(x) geordnet (d.h., die Knoten y_i der Menge N(x) bzw. Nach(x) sind angeordnet: $y_1 < y_2 < ... < y_k$), dann heißt G ein geordneter Baum.

(10) Es sei $k \in \mathbb{N}$ eine positive Zahl. Sei G=(V,E) ein Baum mit $0 \notin V$. Dieser Baum zusammen mit einer Abbildung
$\nu: V \times \{1,...,k\} \to V \cup \{0\}$, so dass für alle $x \in V$ gilt:
$\text{Nach}(x) \subseteq \{\nu(x,i) \mid i = 1,...,k\}$,
aus $\nu(x,i) \neq 0$, $\nu(x,j) \neq 0$ und $i \neq j$ folgt $\nu(x,i) \neq \nu(x,j)$,
heißt k-närer Baum.
(Die direkten Nachfolger stehen also in einem k-stelligen Vektor, wobei Lücken - durch 0 bezeichnet - auftreten, sofern der Ausgangsgrad von x kleiner als k ist.)
Speziell: Im Fall k=2 heißt der Baum binär oder Binärbaum.

Definition 3.2.2.1 (Fortsetzung): Es sei G=(V,E) ein Baum.

(11) 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 plus 1.)

(12) Zu jedem Baum mit Wurzel w gehört die Levelfunktion
level: $V \to \mathbb{N}_0$, rekursiv definiert durch
$\text{level}(w) = 1$ und
$\text{level}(x) = \text{level}(\text{vater}(x)) + 1$ 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 $G = \text{Max}\{\text{level}(x) \mid x \in V\}$.

3.2.2.2: Rekursive Definition *(Erinnerung an 1.6.4.3)*

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

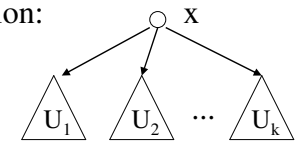
1) Die leere Menge ist ein Baum.
2) Wenn x ein Knoten und $U=\{U_1, U_2, ..., U_k\}$ eine geordnete Menge von k Bäumen ist, so ist auch $x(U)$ ein Baum.

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

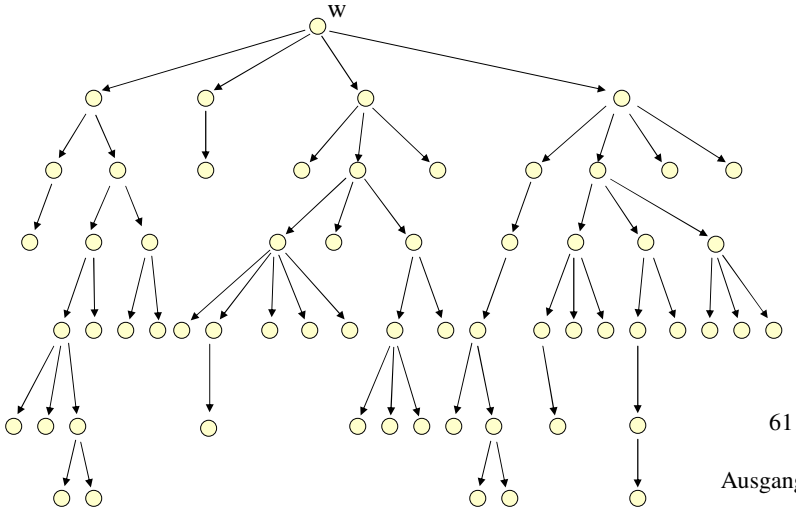
Rekursion:



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

Die k Nachfolger von x sind hier geordnet.

3.2.2.3: Beispiel für einen "beliebigen geordneten Baum":



61 Knoten,
Tiefe 7,
Ausgangsgrad 5

3.2.2.4 Binäre Bäume

Binäre Bäume sind also 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).

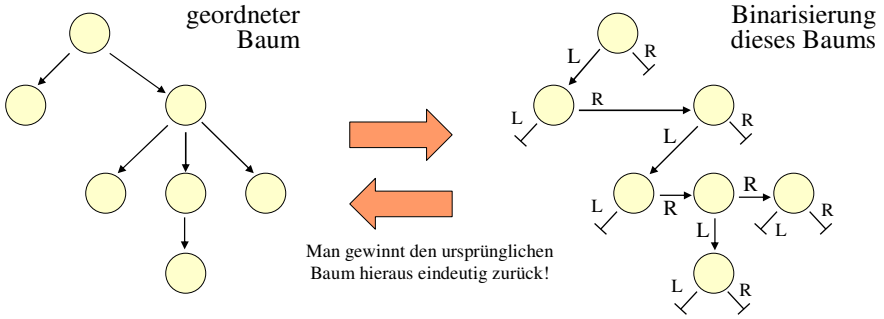
In 1.6.4.4 wurde die Darstellung binärer Bäume in Ada angegeben (hier: mit Inhalt vom Typ Integer):

```
type BinBaum;
type Ref_BinBaum is access BinBaum;
type BinBaum is record
        Inhalt: Integer;
        L, R: Ref_BinBaum;
end record;
```


3.2.2.5 Binarisierung von beliebigen geordneten Bäumen

Jeder geordnete Baum lässt sich eindeutig in einen binären Baum umwandeln, indem

- der linke Zeiger L stets auf den ersten Sohn und
- der rechte Zeiger R stets auf den nächsten Bruder zeigt.



3.2.2.6 Folgerung aus dieser eindeutigen Umwandlung:

Es sei C_n die Anzahl aller binärer Bäume mit n Knoten.

Es sei B_n die Anzahl aller geordneter Bäume mit n Knoten.

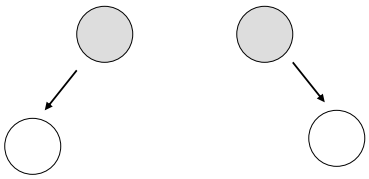
Dann gilt: $B_n = C_{n-1}$ für alle $n > 0$.

Wie viele binäre Bäume mit n Knoten gibt es?

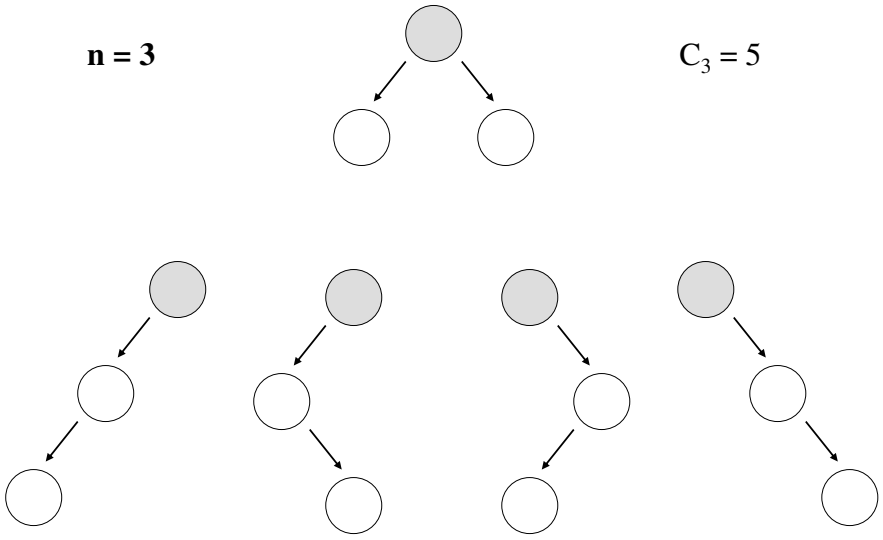
Das heißt: Man berechne C_n.

Wir werden für C_n eine geschlossene Formel angeben. Zunächst berechnen wir $C_0, C_1, ..., C_4$ durch Aufzählen aller zugehöriger Binärbäume.

Wir listen alle binären Bäume mit höchstens 4 Knoten auf. Die Wurzel des binären Baums ist hier grau dargestellt. Die leeren Zeiger wurden weggelassen.

n = 0	<leerer Baum>	$C_0 = 1$
n = 1		$C_1 = 1$
n = 2		$C_2 = 2$

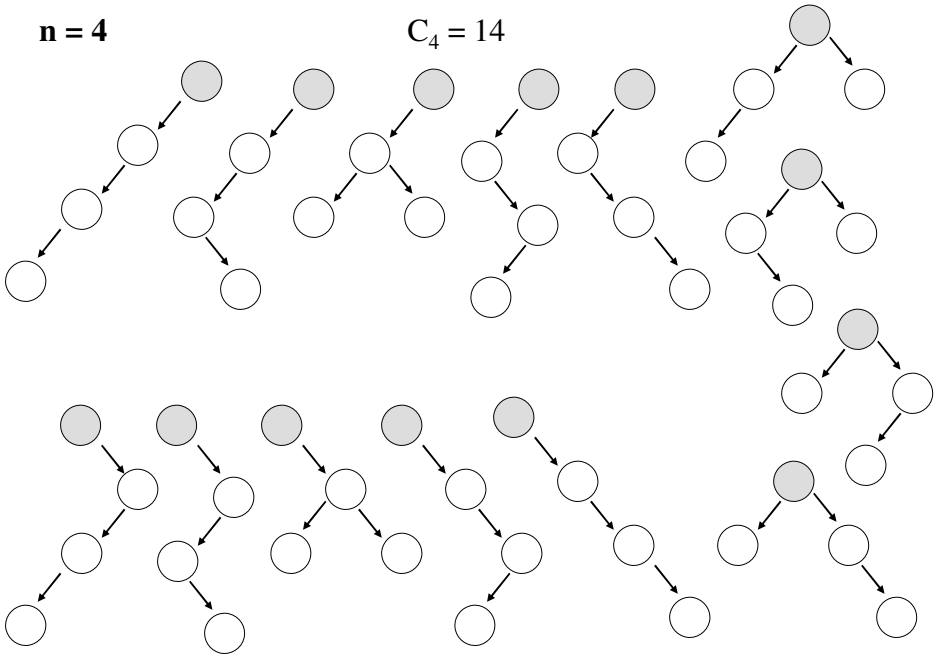
n = 3



$C_3 = 5$

n = 4

$C_4 = 14$



Durch Ausprobieren erhält man die Werte:

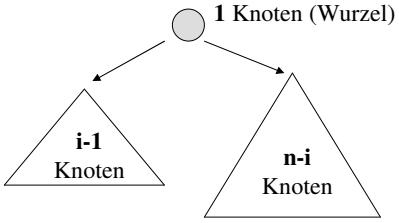
n	Anzahl
1	1
2	2
3	5
4	14
5	42
6	132
7	429
8	1430

Für C_n, die Anzahl der Binärbäume mit n Knoten, gilt die Rekursionsformel:

$C_0 = 1$, und für alle $n \geq 1$:

$$C_n = \sum_{i=1}^{n} C_{i-1} \cdot C_{n-i}$$

wegen:



3.2.2.7 Satz "Catalansche Zahlen"

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

Näherung: $C_n \approx \frac{4^n}{(n+1)\sqrt{\pi \cdot n}}$ für n>0

Wir beweisen nur den Satz. Die Näherung ergibt sich direkt aus dem Satz unter Verwendung der Stirlingschen Formel für die Fakultät.

Wir zeigen zunächst: Die Anzahl C_n der Binärbäume ist gleich der Anzahl der Möglichkeiten, um n „Klammer auf“ und n „Klammer zu“ wie in korrekt geklammerten Ausdrücken aneinander zu reihen. Z.B. gibt es genau 5 korrekte Klammerungmöglichkeiten für n = 3:
((())), (()()), (())(), ()(()), ()()().

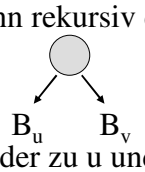
Behauptung: Die Anzahl C_n der Binärbäume ist gleich der Anzahl der Möglichkeiten, um n „Klammer auf“ und n „Klammer zu“ wie in korrekt geklammerten Ausdrücken aneinander zu reihen:

((())), **(()())**, **(())()**, **()(())**, **()()()**.

Wir geben diesen Zusammenhang präzise an:

1. Jeder korrekt geklammerte Ausdruck fängt mit "(" an.
2. Es gibt zu dieser "(" genau eine zugehörige ")", nämlich die erste "Klammer zu", bei der die Anzahl der "Klammer auf" und "Klammer zu" gleich sind (von links nach rechts gezählt).
3. Also hat jeder Klammerausdruck die Form (u)v, wobei sowohl u als auch v korrekt geklammerte Ausdrücke sind. u und v sind eindeutig bestimmt. (u und v können leer sein.)

4. Ordne dem leeren Wort den leeren Binärbaum zu und ordne **()** den einknotigen Baum zu:
5. Ordne dann rekursiv dem Ausdruck **(u)**v folgenden Baum zu:



wobei B_u der zu u und B_v der zu v gehörige Baum ist.

Umgekehrt gewinnt man aus diesem Baum den Ausdruck **(u)**v.

Auf diese Weise lässt sich jedem korrekt geklammerten Ausdruck umkehrbar eindeutig ein binärer Baum zuordnen.

Folglich ist auch die Anzahl der korrekten Klammerungen aus n Klammerpaaren gleich C_n.

Damit ist die Behauptung bewiesen.

Wie viele korrekt geklammerte Ausdrücke gibt es?

Man muss n "Klammer auf" auf 2n Positionen verteilen.

Hiervon gibt es genau $\binom{2n}{n}$ Möglichkeiten.

Von dieser Anzahl muss man die abziehen, die zu keinen korrekten Klammerungen führen. Diese besitzen eine erste Position, bis zu der mehr "Klammer zu" als "Klammer auf" stehen; sie haben also die Form:

x) y

wobei x korrekt geklammert ist und y genau eine "Klammer auf" mehr besitzt als "Klammer zu".

Ersetze nun in y jede "(" durch ")" und umgekehrt. So möge die Klammerfolge y' entstehen. Für x) y' gilt dann:

x möge k Klammerpaare besitzen ($0 \le k \le n$).
Dann besitzt y n-k "Klammer auf" und n-k-1 "Klammer zu".
Dann besitzt y' n-k-1 "Klammer auf" und n-k "Klammer zu".
Also besitzt x)y' n-1 "Klammer auf" und n+1 "Klammer zu".

Weiterhin gilt: Geht man von zwei verschiedenen unkorrekten Klammerungen aus, so erhält man auch zwei verschiedene Ausdrücke der Form x)y'. Wäre nämlich $x)y' = x_1)y_1'$, dann muss $x=x_1$ sein, da x und x_1 beide korrekt geklammert sind und ")" an der ersten Position steht, an der die Anzahl der "Klammer zu" die Anzahl der "Klammer auf" übersteigt. Ebenso muss dann $y' = y_1'$ sein, da die Längen der beiden Ausdrücke gleich lang, nämlich 2n, sind, und folglich gilt auch $y = y_1$.

Wir haben also gezeigt: Jeder unkorrekten Klammerung von n Klammerpaaren lässt sich eindeutig eine Folge von n-1 "Klammer auf" und n+1 "Klammer zu" zuordnen.

Die Umkehrung gilt aber auch.

Wenn eine Folge aus n-1 "Klammer auf" und n+1 "Klammer zu" gegeben ist, so muss es genau eine erste Stelle geben, an der die Zahl der "Klammer zu" die Zahl der "Klammer auf" erstmals übersteigt. Die Folge hat also die Form x)y' , wobei in x überall die Anzahl der "Klammer auf" größer oder gleich der Anzahl der "Klammer zu" bis zu dieser Stelle ist. x ist also ein korrekt geklammerter Ausdruck. In y' gibt es dann eine "Klammer auf" weniger, als es "Klammer zu" gibt. Wandle nun y' in ein y um, indem jede "(" durch ")" ersetzt wird und umgekehrt. So erhält man eine Folge x)y, die gleich viele "Klammer auf" und "Klammer zu" besitzt und die nicht korrekt geklammert ist.

Diese Zuordnung ist offenbar ebenfalls eindeutig.

Daher gilt:

Jeder unkorrekten Klammerung von n Klammerpaaren lässt sich umkehrbar eindeutig eine Folge von n-1 "Klammer auf" und n+1 "Klammer zu" zuordnen. Hieraus folgt:

Die Anzahl der unkorrekten Klammerungen mit n Klammerpaaren ist gleich der Anzahl der Folgen von n-1 "Klammer auf" und n+1 "Klammer zu".

Deren Anzahl ist aber $\binom{2n}{n-1}$

Wir haben also gezeigt:

$$C_n = \binom{2n}{n} - \binom{2n}{n-1} = \frac{1}{n+1}\binom{2n}{n} .$$

Damit ist Satz 3.2.2.7 bewiesen. ■

Folgerung:

$$C_{n+1} = C_n \cdot \frac{4n+2}{n+2}$$

Dieser Formel kann man zum einen das exponentielle Wachstum entnehmen, das in der Näherungsformel ausgedrückt wird. Zum anderen lassen sich hiermit die Catalanschen Zahlen, ausgehend von $C_0 = 1$, leicht iterativ berechnen.

Hinweis: Aus dem Satz lässt sich schließen:

Der Binomialkoeffizient $\binom{2n}{n}$ ist stets durch n+1 teilbar.

Unter welchen Bedingungen auch durch n+2?

Man erhält oft solche "nebensächlichen" Resultate.

Erinnerung an 1.6.4.4:

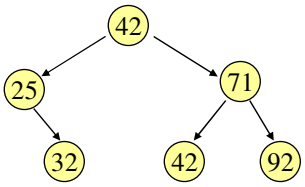
Ein binärer Baum kann preorder, inorder oder postorder in linearer Zeit durchlaufen werden.

Man kann eine Folge sortieren, indem man ihre Elemente nacheinander in einen Binärbaum einfügt und diesen am Ende inorder ausgibt.

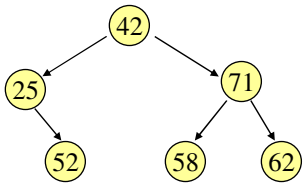
Hierzu muss jeder Knoten einen "Inhalt" erhalten und die Elemente müssen in die Knoten "richtig" eingeordnet werden. Solch einen Baum werden wir einen Suchbaum nennen.

3.2.2.8 Definition "Suchbaum"

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.



Dies ist ein Suchbaum



Dies ist kein Suchbaum

3.2.2.9 Hinweis

Es sei $a_1, a_2, ..., a_n$ eine sortierte Folge von n ganzen Zahlen und es sei B ein Suchbaum mit dem Inhalts-Datentyp Integer. Dann kann man die Zahlen a_i auf genau eine Art so in die Knoten von B legen, dass der inorder-Durchlauf von B genau die sortierte Folge $a_1, a_2, ..., a_n$ ergibt.

Diese Aussage ist klar: Man durchlaufe B inorder und ordne dem i-ten Knoten bei der Besuchsreihenfolge die Zahl a_i zu.

Da es exponentiell viele (genauer: C_n) Bäume mit n Knoten gibt, kann man den jeweils "besten Baum" nicht durch Ausprobieren ermitteln. Dies ist der Grund, warum "geeignete" Algorithmen gesucht und "geeignete" Bäume definiert werden müssen. Wir betrachten zunächst Suchbäume und gehen im nächsten Kapitel dann zu speziellen Bäumen über.

3.2.2.10 Binärbaum und die Grundaufgaben

Gegeben sei eine geordnete Menge. Als Beispiel werden wir die ganzen Zahlen verwenden.
Eine Teilmenge oder eine Folge solcher Elemente soll in einem Binärbaum verwaltet werden. Um die Effizienz beureilen zu können, werden wir konkrete Verfahren für das Suchen (FIND), das Einfügen (INSERT) und das Löschen (DELETE) angeben und deren Aufwand im schlechtesten Fall und im Durchschnitt (im worst case und im average case) berechnen. Die Datenstruktur für Binärbäume sei:

```
type BinBaum;
type Ref_BinBaum is access BinBaum;
type BinBaum is record
        Inhalt: Integer;
        L, R: Ref_BinBaum;
end record;
```


3.2.2.10.a Suchen in einem Binärbaum

Der Binärbaum ist durch den Zeiger "Anker" auf seine Wurzel gegeben. Wir formulieren die Prozedur *Suche*, die zu dem Zeiger Anker und dem zu suchenden Element s einen Verweis auf den Knoten zurückgibt, dessen Inhalt s ist.

```
procedure Suche (Anker: in Ref_BinBaum; s: Integer;
                                        q: out Ref_BinBaum) is
p: Ref_BinBaum := Anker;
begin  q := Anker;
    while q /= null loop
    if q.Inhalt = s then return;
    else if q.Inhalt > s then q := q.L; else q := q.R; end if;
    return;
end Suche;
```