ADS: Algorithmen und Datenstrukturen

Teil 6

Uwe Quasthoff

Institut für Informatik
Abteilung Automatische Sprachverarbeitung
Universität Leipzig

21.11.2017

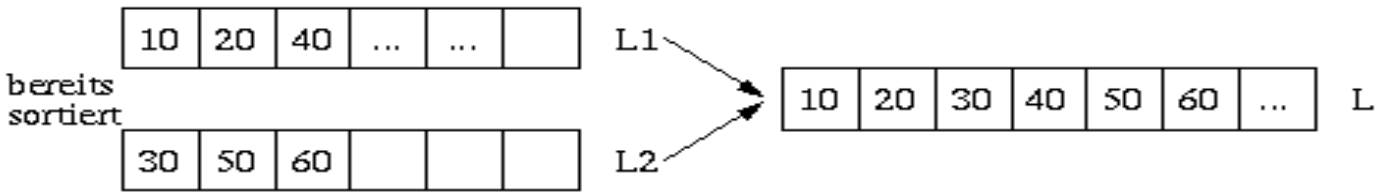
[Letzte Aktualisierung: 13/12/2017, 11:45]

Merge-Sort I (John v. Neumann, 1945)

Algorithmus Mergesort(L)

Falls L leer oder einelementig: fertig.
Sonst:

- Divide: Teile L in 2 möglichst gleichgroße Hälften L1, L2.
- Conquer: Sortiere L1 und L2 mittels Mergesort.
- Merge: Verschmelze die sortierten Teillisten zu sortierter Liste.



Merge ("Verschmelzen") in linearer Zeit.

! Anwendbar für internes und externes Sortieren

Merge-Sort II

Algorithmus (Rekursives Merge-Sort von A[l..r])

- Nach Aufruf `mergesort(A,l,r)` ist Teilarray A[l..r] sortiert
- Also: `mergesort(A,0,n-1)` sortiert gesamtes Array A[0..n-1]

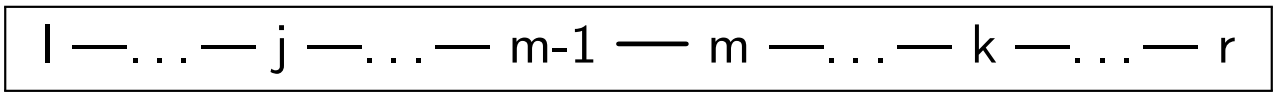
```
mergesort(A,l,r) {
  Falls r<=l { return }     // Teilliste mit Länge<=1; ok
  Sonst {                   // nichttrivialer Fall:
    int m = (l+r+1)/2       // Mitte der Liste bestimmen;
                            // beachte implizites Abrunden
    mergesort(A,l,m-1)      // linken Teil sortieren
    mergesort(A,m,r)        // rechten Teil sortieren
    merge(A,l,m,r)          // verschmelzen
  }
}
```

Merge-Operation des Merge-Sort Algorithmus

Verschmelze Teillisten $A[l..m-1]$ und $A[m..r]$

- Führe Indices j und k ein (jeweils nächstes Element in den Teillisten)
- Übernehme das jeweils kleinere Element A[j] oder A[k] in Ergebnisliste und erhöhe entsprechenden Index j oder k
- wenn eine Teilliste "erschöpft" ist, kopiere Rest der anderen Teilliste

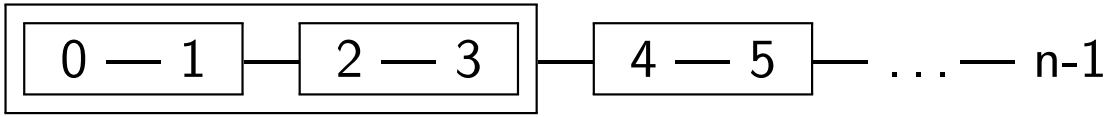
$\Rightarrow$ lineare Kosten




```
merge(A,l,m,r) {
  // 1) Verschmelze nach Ergebnisarray B
  j=l; k=m;
  for (i=0; i<=r-l; i++) {
    if (k>r or (j<m and A[j]<=A[k])) { B[i] = A[j]; j=j+1;}
    else                             { B[i] = A[k]; k=k+1;}
  }
  // 2) Kopiere B[0..r-l] nach A[l..r]
  for (i=0, i<= r-l; i++) { A[l+i] = B[i]; }
}
```

Merge-Sort, nicht-rekursiv

Algorithmus-Idee

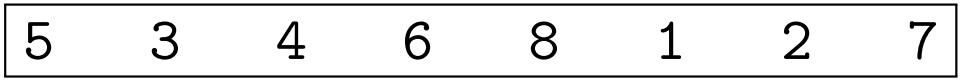


- beginne damit benachbarte Teillisten der Länge 1 zu mischen
- dann mische von links nach rechts jeweils die aufeinanderfolgenden Teillisten der doppelten Länge
- Iteriere bis noch zwei Teillisten übrig bleiben und verschmelze diese

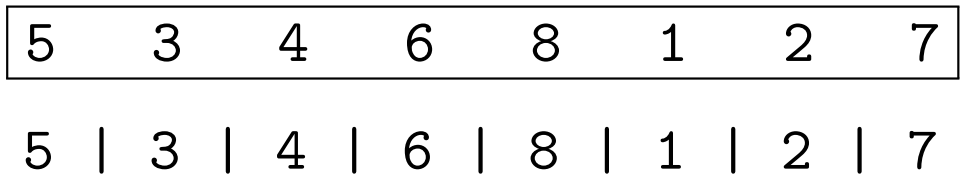
Algorithmus (Merge-Sort von A[0..n-1], nicht-rekursiv)

```
for (k=2; k < n; k*=2) {
  for (i=0; i+k <= n; i+=k) {
    merge(A, i, i+k/2, i+k-1)
  }
  if (i+k/2 < n-1) { //Spezialfall ungerade Elementmenge
    merge(A, i, i+k/2, n-1)
  }
}
merge(A, 0, k/2, n-1) // merge 2 letzte Teillisten
```

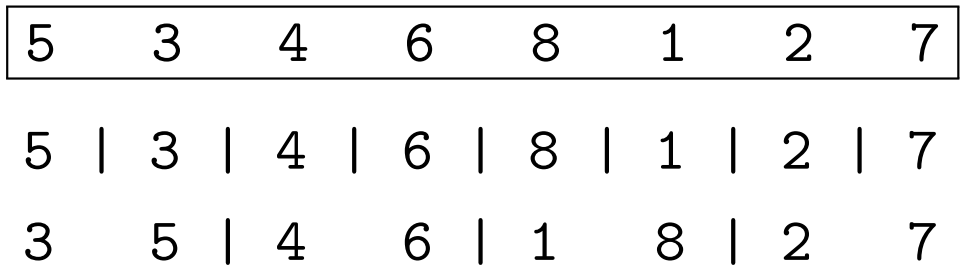
Beispiel: Merge-Sort



Beispiel: Merge-Sort



Beispiel: Merge-Sort



Beispiel: Merge-Sort

5	3	4	6	8	1	2	7

5	3	4	6	8	1	2	7
3	5	4	6	1	8	2	7
3	4	5	6	1	2	7	8

Beispiel: Merge-Sort

5	3	4	6	8	1	2	7

5	3	4	6	8	1	2	7
3	5	4	6	1	8	2	7
3	4	5	6	1	2	7	8
1	2	3	4	5	6	7	8

Beispiel: Merge-Sort

5	3	4	6	8	1	2	7

5	3	4	6	8	1	2	7
3	5	4	6	1	8	2	7
3	4	5	6	1	2	7	8
1	2	3	4	5	6	7	8

Beispiel: Merge-Sort

5	3	4	6	8	1	2	7

5	3	4	6	8	1	2	7
3	5	4	6	1	8	2	7
3	4	5	6	1	2	7	8
1	2	3	4	5	6	7	8

Wie behandelt der Algorithmus den Fall, dass n keine Potenz von 2 ist?

Komplexität von Merge-Sort

- Beim Mischen werden (auf jeder Rekursionsebene insgesamt) $\Theta(N)$ Schlüsselvergleiche gemacht. Da die Rekursionstiefe logarithmisch beschränkt ist, ergeben sich insgesamt $\Theta(N \log N)$ Schlüsselvergleiche.
- Gleiche Argumentation gilt für Moves, d.h. auch Anzahl der Bewegungen ist $\Theta(N \log N)$.

Damit gilt für best, worst und average case:

$$C_{\text{min}} = C_{\text{max}} = C_{\text{avg}} = \Theta(N \log N) \ .$$

$$M_{\text{min}} = M_{\text{max}} = M_{\text{avg}} = \Theta(N \log N) \ .$$

Wie argumentieren sie auf Basis des nicht-rekursiven Merge-Sort Algorithmus?
Weitere Eigenschaften (Stabilität, Speicherbedarf, Nutzen von Vorsortierung)?

Variante: Natürliches Merge-Sort

Algorithmus

- statt mit 1-elementigen Teillisten zu beginnen, werden bereits anfangs möglichst lange sortierte Teilfolgen ("runs") verwendet
- Nutzung einer bereits vorliegenden (natürlichen) Sortierung in der Eingabefolge

Natürliches 2-Wege-Merge-Sort

Verschmelzungsprozess wird nicht mit einelementigen Listen begonnen, sondern mit möglichst langen bereits sortierten Teilfolgen. Jeweils zwei benachbarte Teilfolgen werden verschmolzen.

Beispiel

3	6	|	5	7	9	|	1	8	|	0	2	4	
3	5	6	7	9	|	0	1	2	4	8				
0	1	2	3	4	5	6	7	8	9					

Algorithmus nutzt Vorsortierung aus: falls Liste bereits sortiert, so wird das in O(N) Schritten festgestellt.

Beispiel: Natürliches Merge-Sort

5 3 4 6 8 1 2 7

Beispiel: Natürliches Merge-Sort

5 3 4 6 8 1 2 7

5 | 3 4 6 8 | 1 2 7

Beispiel: Natürliches Merge-Sort

5 3 4 6 8 1 2 7

5 | 3 4 6 8 | 1 2 7

3 4 5 6 8 | 1 2 7

Beispiel: Natürliches Merge-Sort

```
5   3   4   6   8   1   2   7
5 | 3   4   6   8 | 1   2   7
3   4   5   6   8 | 1   2   7
1   2   3   4   5   6   7   8
```

Vorsortierung

Wie lässt sich Grad der Vorsortierung einer Folge $F = k_1, \ldots, k_n$ von Schlüsseln messen?

Vorschlag 1: Zahl der Inversionen (Vertauschungen) von F
$inv(F) = |\{(i,j)|1 \leq i < j \leq n, k_i > k_j\}|$ misst so etwas wie Entfernungen zur richtigen Position.

Vorschlag 2: Anzahl der runs, d. h. der vorsortierten Teillisten (siehe oben)
$runs(F) = |\{(i)|1 \leq i < n, k_{i+1} < k_i\}| + 1$

Vorschlag 3: Länge der längsten sortierten Teilliste, $las(F)$, bzw.
$rem(F) = n - las(F)$ (damit wie oben kleiner besser ist)

Vorsortierung Beispiele

F: 3 6 5 7 9 1 8 0 2 4

inv(F): $3+5+4+4+5+1+3+0+0+0=25$

runs(F): 4

3 6 — 5 7 9 — 1 8 — 0 2 4

rem(F): $10-3=7$

F: 1 0 3 2 5 4 7 6 9 8

inv(F): $1+0+1+0+1+0+1+0+1+0=5$

runs(F): 6

1 — 0 3 — 2 5 — 4 7 — 6 9 — 8

rem(F): $10-2=8$

Externes Sortieren

Anwendung, falls zu sortierende Daten nicht vollständig im Hauptspeicher gehalten werden können
Hauptziel: Minimierung von Externspeicherzugriffen
Ansatz

- Zerlegen der Eingabedatei in m Teildateien, die jeweils im Hauptspeicher sortiert werden können
- Internes Sortieren und Zwischenspeichern der Runs (extern)
- Mischen der m Runs (m-Wege-Mischen)

Bewertung

- gute E/A-Kosten (zweimal Lesen und Schreiben auf Externspeicher)
- setzt Dateispeicherung auf Direktzugriffsmedium (z.B. Magnetplatten) voraus, da ggf. sehr viele temporäre Dateien
- verfügbarer Hauptspeicher muss wenigstens m+1 Seiten (Blöcke) umfassen, um m Dateien gleichzeitig mischen zu können.

Sortieren mit Bändern

Restriktiver, da hier kostengünstig nur sequenzielle Zugriffe möglich sind.

- Ausgeglichenes 2-Wege-Merge-Sort (4 Bänder)
- Ausgeglichenes k-Wege-Merge-Sort (2k Bänder)

Ausgeglichenes 2-Wege-Merge-Sort

- vier Bänder B1, B2, B3, B4; Eingabe sei auf B1; Hauptspeicher fasse r Datensätze
- Wiederholtes Lesen von r Sätzen von B1, interne Sortierung und abwechselndes Ausschreiben der Runs auf B3 und B4 bis B1 erschöpft ist
- Mischen der Runs von B3 und B4 (ergibt Runs der Länge 2r) und abwechselndes Schreiben auf B1 und B2
- Fortgesetztes Mischen und Verteilen bis nur noch ein Run übrig bleibt

Beispiel: Sortieren mit 4 Bändern, r=4

```
Band1:  14  4  3 17|22  5 25 13| 9 10  1 11|12  6  2 15
Band2:
Band3:   3  4 14 17| 1  9 10 11|
Band4:   5 13 22 25| 2  6 12 15|
Band1:   3  4  5 13 14 17 22 25|
Band2:   1  2  6  9 10 11 12 15|
```

Verbesserungen beim Merge-Sort

Ausgeglichenes k-Wege-Merge-Sort

- k-Wege-Aufteilen und k-Wege-Mischen mit 2k Bändern

Mischen bei Mehrwege-Merge-Sort durch Auswahlbaum (Tournament- oder Heap-Sort)

- Bei k Schlüsseln erfolgt Entfernen des Minimums und Hinzufügen eines neuen Schlüssels in O(log k) Schritten.

Replacement Selection Sort: Auswahlbaum wird als Priority Queue (beschränkter Grösse r) genutzt, um Länge der initialen Runs zu erhöhen. Damit erfordert das Sortieren weniger Durchgänge.

- gelesene Elemente kommen zunächst in die Priority Queue
- sobald die Queue voll ist wird jeweils Minimum x entfernt und geschrieben, falls es grösser ist als das letzte geschriebene Element.
- falls x kleiner ist, kann es erst im nächsten Run vorkommen; es werden dann erst die grösseren Elemente in der Queue geschrieben.
- man erreicht im Mittel $2 \cdot r$ Run-Länge.

Beispiel: Replacement Selection Sort (r=4)

```
Band1:  14  4  3 17|22  5 25 13| 9 10  1 11|12   6  2 15

                                PQ:  3  4 14 17
Band3:   3                      PQ:  4 17 14 22
Band3:   3  4                   PQ:  5 14 17 22
Band3:   3  4  5                PQ: 14 17 22 25
Band3:   3  4  5 14             PQ: 13 17 22 25
Band3:   3  4  5 14             PQ: 17 22 25 | 13   (!)
Band3:   3  4  5 14 17          PQ: 22 25 | 9 13
...
```

Einschub: Lexikographische Sortierung im Wörterbuch

1. Version: Einfach zu implementieren, aber nicht duden-korrekt.
Beispiel: ICD-10-Diagnosenthesaurus

- ä, ö, ü sind wie ae, oe, ue und ß wie ss einsortiert. ()
- Arabische Ziffern am Wortanfang sind in der Regel nur sekundär berücksichtigt (z. B. ist 2-Propanol wie Propanol eingeordnet), Ziffern innerhalb eines Wortes sind vor Buchstaben sortiert (z. B. E1-Trisomie vor Eales), die römischen Zahlzeichen sind wie Buchstaben behandelt.
- Buchstaben mit diakritischen Zeichen (ç, é, . . .) sind wie Grundbuchstaben eingeordnet.
- Bei der Sortierfolge stehen Leerzeichen vor Bindestrichen und Auslassungszeichen, danach folgen Ziffern und zuletzt Buchstaben

Dudenkonforme Sortierung

- Großbuchstaben werden wie Kleinbuchstaben eingeordnet.
- ä,ö,ü werden wie a,o,u eingeordnet.
- Bindestriche und Leerzeichen werden ignoriert.
- Bei zwei Wörtern, die so an der gleichen Stelle eingeordnet würden, folgt
  - das ohne Umlaut vor dem mit Umlaut. (d.h. Mutter vor Mütter)
  - das mit Kleinbuchstaben vor dem mit Großbuchstaben (d.h. los vor Los)

Beispiel: Korrekt ist die Reihenfolge

```
..., hatte, hätte, hatten, hattest, hättest, ...
..., Öl, olé, ölen, ...
```

Die Implementierung kann nicht einfach dadurch erfolgen, dass Buchstaben durch Zahlen kodiert werden.

Klassischer Sortierwettbewerb

Wettbewerb berücksichtigt

- Sortieralgorithmus
- Systemsoftware
- Hardware

und hat die Aufgabe, schnelle Sortiermethoden zu finden.
Erster Wettbewerb: Datamation: 1. April 1985

- Sortiere eine Million 100-Byte Records mit den ersten 10 Byte als Schlüssel.
- Bewertet auch das Dateisystem und Ein-/Ausgabe-Geschwindigkeit
- Dauerte 1985 eine Stunde, im Jahr 2000 nur noch eine Sekunde.

Danach mussten größere Sortieraufgaben gestellt werden.

Neue Sortierwettbewerbe

Wettbewerbe:

- Minute Sort: Sortiere soviel 100-Byte-Records wie möglich in einer Minute
- Penny Sort: Sortiere soviel wie möglich für einen Penny (Kosten für Hard- und Software, abgeschrieben über 3 Jahre =94.608.000 Sekunden). Inzwischen ersetzt durch Cloud Sort und Joule Sort.
- Terabyte Sort: Sortiere 1 Terabyte so schnell wie möglich.
- Gray Sort: Sortierrate für große Datenmange, momentan mindestens 100TB.

2 Kategorien:

- Daytona (stock car)
- Indy (formula 1)

Regeln

- Eingabe und Ausgabedateien auf Festplatte
- 100-Byte Records mit den ersetn 10 Byte als Schlüssel

Sortierergebnisse

Wettbewerb	Jahr	Ergebnis Daytona	Gewinner
Gray	2016	44.8 TB/min	Tencent Sort (512 Nodes)
Cloud	2016	\$1.44 / TB	NADSort (Alibaba Cloud)
Minute	2016	37 TB	Tencent Sort (512 Nodes)

Alle Ergbisse: http://sortbenchmark.org

Zusammenfassung: Sortieren I

	best case	avg case	worst case	zusätzl. Speicher
Auswahl	n	n^2	n^2	1
Einfügen	n	n^2	n^2	1
Bubblesort	n	n^2	n^2	1
Quicksort	n log n	n log n	n^2	$\log(n)$ [1]
Tuniersort	n log n	n log n	n log n	n
Heapsort	n log n	n log n	n log n	1
Bucketsort	n	n	n log n, n^2	n
Mergesort	n log n	n log n	n log n	n

[1] trotz in-place, durch Rekursionsstack; die Implementierung aus der Vorlesung benötigt im worst case sogar linearen Platz.

Zusammenfassung: Sortieren II

- Kosten allgemeiner Sortierverfahren wenigstens $O(n \log n)$
- Elementare Sortierverfahren: Kosten $O(n^2)$
  - einfache Implementierung; ausreichend bei kleinem n
  - gute Lösung: Insertion Sort
- $O(n \log n)$-Verfahren: Heap-Sort, Quick-Sort, Merge-Sort
  - Heap-Sort und Merge-Sort sind worst-case-optimal, $O(n \log n)$
  - in Messungen erzielte Quick-Sort in den meisten Fällen die besten Ergebnisse
- Begrenzung der Kosten für Umkopieren durch indirekte Sortierverfahren
- Generell vorteilhafte Eigenschaften von Sortierverfahren
  - Ausnutzen einer weitgehenden Vorsortierung
  - Stabilität

Bäume

Bäume als Graphen

- Ein *Graph* ist ein Paar (V, E). Hierbei ist V eine Menge (=Knotenmenge) und E eine Menge von ungeordneten Paaren (=Kanten) in V.

Ein Baum ist

- ein azyklischer einfacher, zusammenhängender Graph
- d. h. er enthält keine Schleifen und Zyklen: zwischen jedem Paar von Knoten besteht höchstens eine Kante

Bäume

- Verallgemeinerung von Listen: Element (Knoten) hat möglicherweise mehrere Nachfolger (Kinder).
  - Genau 1 Knoten ohne Vorgänger: Wurzel
  - Knoten *mit* Nachfolger: innere Knoten
  - Knoten *ohne* Nachfolger: Blätter
- Häufig verwendete Datenstruktur: Entscheidungsbäume, Syntaxbäume, Ableitungsbäume, Suchbäume, ...
- Hier besonders interessant: Verwendung von Bäumen zur Speicherung von Schlüsseln und Realisierung der Wörterbuchoperationen (Suchen, Einfügen, Entfernen) in Binärbäumen.

Orientierte Bäume

Eine Menge B ist ein orientierter (Wurzel-) Baum, falls

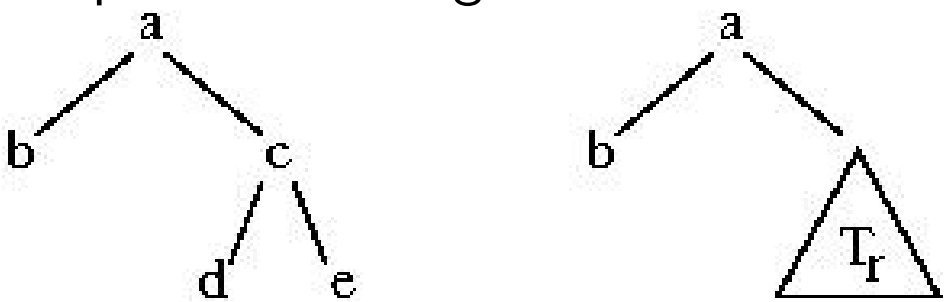
1. in B ein ausgezeichnetes Element w – die Wurzel von B – existiert
2. die Elemente in $B \setminus \{w\}$ disjunkt zerlegt werden können in $B_1, B_2, \ldots, B_m$, wobei jedes B_i ebenfalls ein orientierter Baum ist.

Anschaulicher:

- Die Kanten des Graphen sind gerichtet.
- Von der Wurzel gehen nur Kanten aus, keine treffen ein.
- Jeder Knoten ist von der Wurzel aus auf genau einem Weg entlang der gerichteten Kanten erreichbar.
- Für jeden Nicht-Wurzelknoten gibt es Knoten, die nicht entlang der gerichteten Kanten erreichbar sind.

Darstellungsarten für orientierte Bäume

1. Graphendarstellung



2. Mengendarstellung
   $\{\{a, b, c, d, e\}, \{b\}, \{c, d, e\}, \{d\}, \{e\}\}$
3. Klammerdarstellung
   $(a, (b), (c, (d), (e)))$
4. Rekursives Einrücken

   a
   b
   c
   d
   e

Eigenschaften I

Baum B **heißt geordnet**, wenn Nachfolger jedes Knotens geordnet sind (1., 2., 3. etc.; linker, rechter). Bei einem geordneten Baum bilden die Unterbäume B_i jedes Knotens eine geordnete Menge. (Beispiel: Arithmetischer Ausdruck)

Eine geordnete Menge von geordneten Bäumen heißt geordneter Wald.

Ordnung von B: maximale Anzahl von Nachfolgern eines Knotens

Eigenschaften II

Tiefe eines Knotens: Abstand zur Wurzel, d.h. Anzahl der Kanten auf dem Pfad von diesem Knoten zur Wurzel. Die Knoten auf der **Stufe** i sind alle Knoten mit Tiefe i.

Höhe eines Baums: maximale Tiefe eines Knotes $\underline{+\ 1}$. Der leere Baum hat Höhe 0.

Ein Baum der Ordnung n heißt **vollständig**, wenn alle Blätter dieselbe Tiefe haben und auf jeder Stufe die maximale Anzahl von Knoten vorhanden ist.

Binärbäume

Ein Binärbaum ist eine endliche Menge von Elementen, die entweder leer ist oder ein ausgezeichnetes Element - die Wurzel des Baumes - besitzt und folgende Eigenschaften aufweist:

- Die verbleibenden Elemente sind in zwei disjunkte Untermengen zerlegt.
- Jede Untermenge ist selbst wieder ein Binärbaum und heißt linker bzw. rechter Unterbaum des ursprünglichen Baumes

Ein Binärbäum ist also ein geordneter Baum, in dem jeder Knoten höchstens zwei Kinder besitzt (Ordnung 2).
Üblicherweise wird verlangt, dass jeder Knoten 2 oder 0 Nachfolger hat.

Formale ADT-Spezifikation: BINTREE

Datentyp BINTREE, Basistyp ELEM

Operationen:

CREATE:		$\rightarrow$ BINTREE
EMPTY:	BINTREE	$\rightarrow \{\text{TRUE, FALSE}\}$
BUILD:	BINTREE$\times$ELEM$\times$BINTREE	$\rightarrow$ BINTREE
LEFT:	BINTREE$\setminus\{0\}$	$\rightarrow$ BINTREE
ROOT:	BINTREE$\setminus\{0\}$	$\rightarrow$ ELEM
RIGHT:	BINTREE$\setminus\{0\}$	$\rightarrow$ BINTREE

Axiome

1. CREATE() $= 0$;
2. EMPTY (CREATE()) $=$ TRUE;
3. $\forall l, r \in$ BINTREE, $\forall d \in$ ELEM:
   EMPTY (BUILD (l, d, r)) $=$ FALSE
   LEFT (BUILD (l, d, r)) $=$ l;
   ROOT (BUILD (l, d, r)) $=$ d;
   RIGHT(BUILD (l, d, r)) $=$ r;

Beispiele für Konstruktion

Welche Binärbäume entstehen durch

- BUILD (BUILD (0, b, BUILD (0, d, 0)), a, BUILD (0, c, 0))
- BUILD (BUILD (BUILD (0, d, 0), b, 0), a, BUILD (0, c, 0))

Eigenschaften von Binärbäumen I

Satz: Die maximale Anzahl von Knoten eines Binärbaumes

1. auf Stufe i ist 2^i, $i \geq 0$
2. der Höhe h ist 2^h-1, $h \geq 0$ (Der leere Baum hat Höhe 0)

Definition: Ein *vollständiger* Binärbaum der Höhe $k+1$ hat folgende Eigenschaften:

- Jeder Knoten der Stufe k ist ein Blatt.
- Jeder Knoten auf einer Stufe $< k$ hat nicht-leere linke und rechte Unterbäume.

Definition: In einem *strikten* Binärbaum besitzt jeder innere Knoten nicht-leere linke und rechte Unterbäume

Eigenschaften von Binärbäumen II

Definition: Ein *ausgeglichener* Binärbaum der Höhe $k+1$ ist ein Binärbaum, so dass gilt:

1. Jedes Blatt im Baum ist auf Stufe k oder $k-1$ (falls $k \geq 1$).
2. Jeder Knoten auf Stufe $< k-1$ hat nicht-leere linke und rechte Teilbäume

Definition: Ein *fast vollständiger* Binärbaum ist ein ausgeglichener Binärbaum, bei dem die Blätter auf Stufe k möglichst weit links stehen. Letzteres heisst formal: für jeden inneren Knoten dessen rechter Teilbaum mindestens ein Blatt auf Stufe k enthält, ist sein linker Teilbaum nicht leer und hat alle Blätter auf Stufe k.

Eigenschaften von Binärbäumen III

- **Definition**: Zwei Binärbäume werden als *ähnlich* bezeichnet, wenn sie dieselbe Struktur besitzen.
- **Definition**: Sie heißen *äquivalent*, wenn sie ähnlich sind und dieselbe Information enthalten.

Abgeleitete Eigenschaften von Binärbäumen

1. Für zwei beliebige Knoten in einem Baum existiert genau ein Pfad, der sie verbindet.
2. Ein Binärbaum mit N Knoten hat $N - 1$ Kanten.
3. Ein strikter binärer Baum mit N inneren Knoten hat $N + 1$ äußere Knoten.
4. Die Höhe eines vollständigen binären Baumes mit N Knoten beträgt $\log_2(N + 1)$.