Lehrstuhl für Informatik I
— Informationsstrukturen und
wissensbasierte Systeme —
Universität Würzburg

Prof. Dr. H. Noltemeier

Würzburg, 9. Juli 1998

Übungen zur Vorlesung
Praktische Informatik II (SS98)

Musterlösung zum 5. Übungsblatt

Lösung zu Aufgabe 17: (Aufspießanfragen)

Sei I eine Menge von n ganzzahligen Intervallen $[a_j, b_j]$ mit $0 < a_j < b_j \leq N$, $a_j, b_j \in \mathbb{N}$ für $j = 1, \ldots, n$.

Es waren die Vor- und Nachteile der Realisierung von Aufspießanfragen („Welche Intervalle aus I enthalten den Anfragepunkt $q \in \mathbb{R}$?")

- mittels Segmentbaum,
- mittels 3-seitiger Bereichssuche im geeignet gewählten Prioritätssuchbaum.

zu vergleichen und zu diskutieren, wobei auch der Fall $n > N$ zu betrachten war.

Zunächst erscheint unter der Annahme, daß N von vornherein unveränderlich festgelegt werden kann, ein Prioritätssuchbaum gegenüber einem natürlichen Prioritätssuchbaum als die bessere Wahl. So ergeben sich nämlich in jedem Fall und nicht nur in der amortisierten Betrachtung günstigere Laufzeiten, da die Form des entstehenden Skelettes nicht so extrem aus dem Rahmen fallen kann.

Vergleicht man dann Segmentbaum und Prioritätssuchbaum, so ist zunächst festzustellen, daß beide in $\mathcal{O}(N + n \log N)$ aufgebaut werden können:

- Der Segmentbaum besitzt $N-1$ Blätter für die $N-1$ möglichen Elementarintervalle $\{[i, i+1] \mid i \in \{1, \ldots, N-1\}\}$, und n Intervalle $[a_j, b_j]$ werden eingefügt (vgl. Satz der Vorlesung).
- Der Prioritätssuchbaum besitzt $N-1$ Blätter für die $N-1$ möglichen Anfangs- oder Endpunkte (abhängig von der gewählten Art der Repräsentation) der Intervalle $[a_j, b_j]$, und n Intervalle $[a_j, b_j]$ müssen eingefügt werden.

Hinsichtlich des Speicherbedarfs ist der Prioritätssuchbaum die günstigere Alternative:

- Der Prioritätssuchbaum benötigt Speicherplatz für die $(N-1)+(N-2) = 2N-3$ Knoten und für die n Punkte, die die Intervalle repräsentieren. Also liegt der Platzbedarf in $\mathcal{O}(N + n)$.
- Der Segmentbaum benötigt ebenfalls mindestens diesen Platz, man muß allerdings davon ausgehen, daß eingefügte Intervalle in mehrere Standardintervalle zerfallen. Deswegen ist der Speicherplatzbedarf etwas höher, und zwar aus $\mathcal{O}(n \log N + N)$ (vgl. Satz).

Der Unterschied im Speicherplatzbedarf kann sich in der Praxis für $n > N$ durchaus bemerkbar machen.

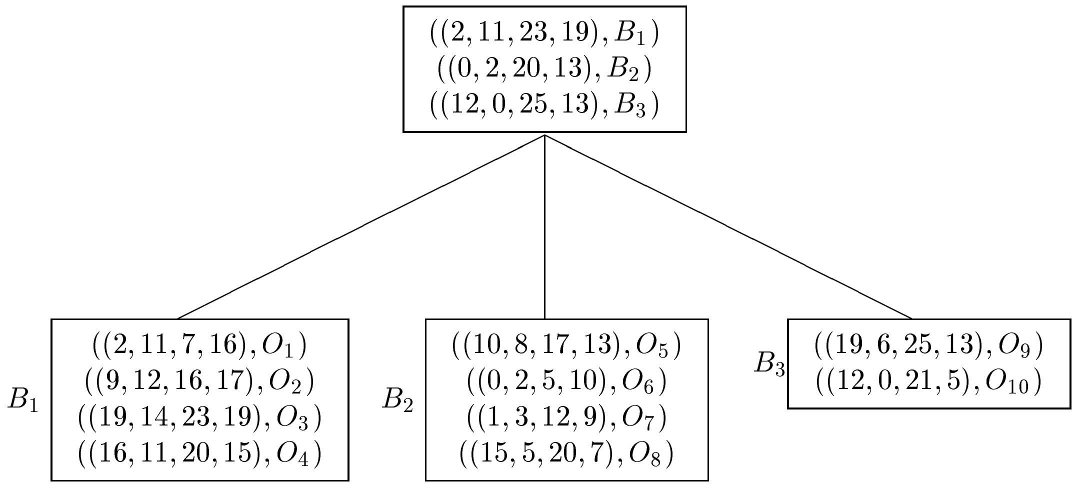
Im Segmentbaumes kann jedoch eine Aufspießanfrage stets unabhängig von der Anzahl der getroffenen Intervalle in Zeit $\mathcal{O}(\log N)$ durchgeführt werden. Beim Prioritätssuchbaum hingegen ist eine Abhängigkeit von der Anzahl k der getroffenen Intervalle festzustellen, es sind $\mathcal{O}(k + \log N)$ Schritte nötig. Für $n > N$ und wenn von sehr vielen Anfragen auszugehen ist, ist hier also der Segmentbaum die bessere Wahl.

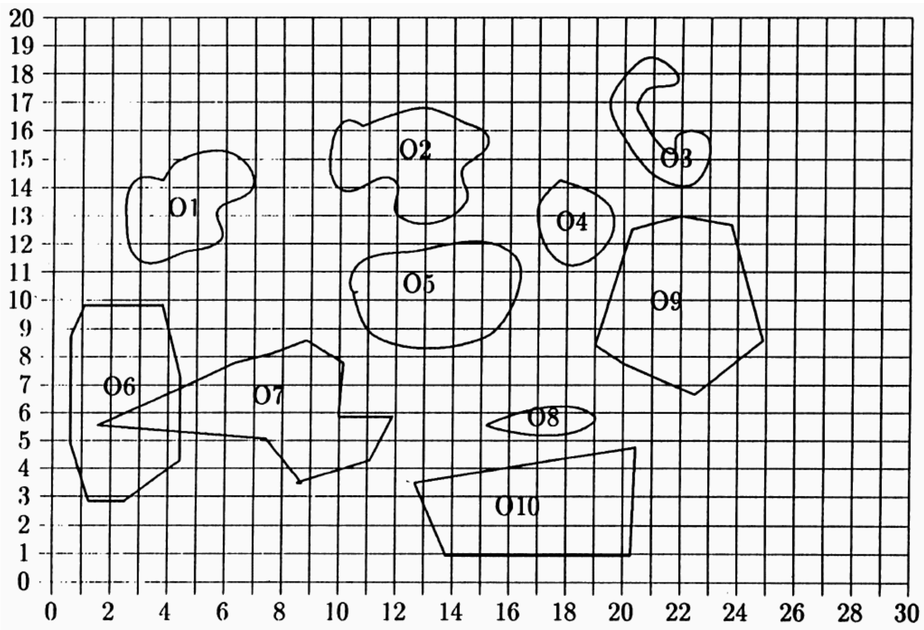
- Im Segmentbaum gibt es bei Anfrage mit Punkt q genau zwei Suchwege zu den Standardintervallen $[q-1, q]$ und $[q, q+1]$. Ein Durchlauf dieser Suchwege benötigt eine Laufzeit aus $\mathcal{O}(\log N)$ und liefert alle gesuchten Intervalle, indem für jeden besuchten Knoten die dort vermerkten Intervalle an die anfangs leere Ergebnismenge angehängt werden (dies kann durch eine Implementierung der Intervallmengen z.B. als doppelt verkettete Listen für jeden Knoten in konstanter Zeit erfolgen). Ist eine Kopie der getroffenen Intervalle anzulegen, so liegt die Laufzeit aber auch nur in $\mathcal{O}(k + \log N)$.
- Im Prioritätssuchbaum muß zunächst ein zu q passendes Blatt gesucht werden und dann der Baum links bzw. rechts des Pfades in Abhängigkeit von der Art der Intervallrepräsentation mit dem Anfangs- bzw. Endpunkt als x-Wert in LWR-Ordnung solange durchlaufen werden, bis die Priorität nicht mehr paßt.

Tritt schließlich der Fall ein, daß N auch vergrößert werden kann, so ist ohne Neuaufbau der Segmentbaum völlig ungeeignet, der Prioritätssuchbaum müßte durch einen natürlichen Prioritätssuchbaum ersetzt werden. Für diesen ergeben sich in der amortisierten Betrachtungweise ähnliche Ergebnisse.

<u>Lösung zu Aufgabe 18:</u> (Rechteckbaum; R-Tree)

(a) Zur folgenden Objektszene $\subset \mathbb{R}^2$ wird ein R-Tree für $M = 4$ und $m = 2$ angegeben, wobei die bounding-boxes so klein wie möglich, aber mit ganzzahligen Koordinaten gewählt werden.





(b) Es war ein Algorithmus zu entwerfen, der zu gegebenem R-Tree T mit Wurzel W alle Einträge in T bestimmt, deren zugehörige Rechtecke sich mit einem gegebenen Suchrechteck S überlappen.

Sei v ein Knoten eines R-Trees. Dann bezeiche im folgenden $v[k]$ den k-ten Eintrag in v, d.h. $v[k] = (v[k].I, v[k].\textit{Tupel-Identifier})$ für ein Blatt v und $v[k] = (v[k].I, v[k].\textit{child-pointer})$ für einen inneren Knoten v. Existiert kein k-ter Eintrag in v, so sei $v[k] = \textit{nil}$. Dann leistet Algorithmus 1 (findIntersectionInTree) das Gewünschte.

Algorithmus 1 findIntersectionInTree(Rechteck S, R-Tree T)

$w \leftarrow$ Wurzel(T)
return findIntersection(S, w)

Algorithmus 2 findIntersection(Rechteck S, Baumknoten v)

$R \leftarrow \varnothing$ {R ist Ergebnismenge mit Referenzen auf geschnittene Objekte}
if v ist ein Blatt **then**
    for $k \leftarrow 1, \ldots, M$ **do**
        if ($v[k] \neq \textit{nil}$) **and** (intersects(S,$v[k].I$)) **then**
            $R \leftarrow R \cup \{v[k].\textit{Tupel-Identifier}\}$
else
    for $k \leftarrow 1, \ldots, M$ **do**
        if ($v[k] \neq \textit{nil}$) **and** (intersects(S,$v[k].I$)) **then**
            $R \leftarrow R \cup$ findIntersection(S, $v[k].\textit{child-pointer}$)
return R

Algorithmus 3 INTERSECTS(Rechteck S, Rechteck I)

```
Sei S = ([c_1, d_1], ..., [c_n, d_n])
Sei I = ([a_1, b_1], ..., [a_n, b_n])
intersecting ← true
for j ← 1, ..., n do
    if (c_j > b_j) or (d_j < a_j) then
        intersecting ← false
return intersecting
```

(c) In diesem Zusammenhang waren die Problematik der R-Trees bei der Bestimmung aller Objekte in T, die sich mit einem Suchrechteck S überlappen, zu erläutern und mögliche Strategien zur Verbesserung des R-Tree-Konzeptes anzugeben.

Da nicht garantiert ist, daß die Rechtecke in einer Page disjunkt sind (leeren Schnitt besitzen), kann das Suchrechteck mit mehreren Rechtecken aus einer Page überlappen, schlimmstenfalls mit allen, und es kann im allgemeinen kein logarithmischer Aufwand mehr garantiert werden.

Ein weiteres Problem ist, daß weit voneinander entfernte Objekte unter Umständen im selben Blatt gespeichert werden; dieses Blatt besitzt dann eine bounding-box, die großenteils leer ist. Die Wahrscheinlichkeit, daß bei einem Range-search das Anfragerechteck mit jedem Objekt in diesem Blatt auf Schnitt getestet wird, ist also auch dann hoch, wenn jeder dieser Schnitte leer ist.

Der Range-Search ist also mit R-Trees nicht sehr effizient möglich.

Mögliche Stategien zur Verbesserung:

- Man kann versuchen, die überlappenden Teile beim Aufbau des R-Trees so gering wie möglich zu halten, z.B. durch vorheriges Testen und Bewerten verschiedener Einfügepositionen.
- Fasse benachbarte Objekte möglichst nahe an den Blättern des R-Trees zusammen und entfernte höher im Baum.
- Erlaube komplexere Überdeckungsobjekte als Rechtecke (z.B. Quadratraster oder Polygone), um unnötig überdeckte Fläche zu reduzieren. Hier muß man abwägen zwischen dem Verlust durch höheren Speicherbedarf und des komplexeren Schnitttests und dem Gewinn durch die exaktere Gliederung des Baumes.

Lösung zu Aufgabe 19: (Dynamische Window-Verwaltung)

Eine Szene von Rechtecken $\subset \mathbb{R}^2$ sollte mittels einer partiell-persistenten linearen Liste verwaltet werden.

Für die Knoten der kurzlebigen Struktur wählen wir folgenden Aufbau:

t	S	$R.x_1$	$R.y_1$	$R.x_2$	$R.y_2$	P

wobei t den Zeitstempel, S den Schlüssel (Index) des Rechteckes, $R.x_1$, $R.y_1$, $R.x_2$ und $R.y_2$ die Koordinaten der Ecken des Rechteckes und P den Zeiger auf den nächsten Knoten bezeichnen.

Es war folgende Sequenz von Operationen auszuführen:

1. Einfügen des Rechtecks $R_1 = (1, 1, 5, 4)$.

v_1	1	1	1	1	5	4	nil

2. Einfügen des Rechtecks $R_2 = (7, 6, 12, 9)$.

v_1	1	1	1	1	5	4	nil	2	P	v_2
v_2	2	2	7	6	12	9	nil			

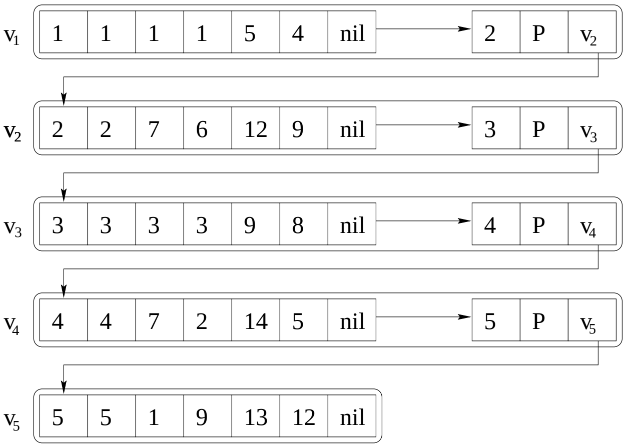
3. Einfügen des Rechtecks $R_3 = (3, 3, 9, 8)$.

v_1	1	1	1	1	5	4	nil	2	P	v_2
v_2	2	2	7	6	12	9	nil	3	P	v_3
v_3	3	3	3	3	9	8	nil			

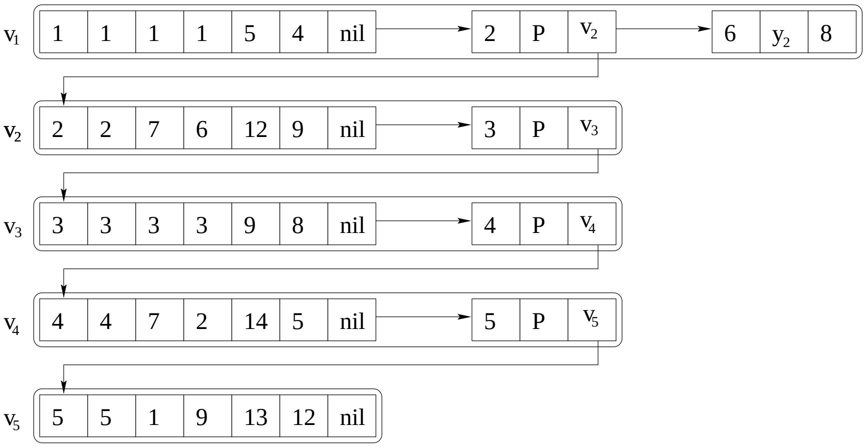
4. Einfügen des Rechtecks $R_4 = (7, 2, 14, 5)$.

v_1	1	1	1	1	5	4	nil	2	P	v_2
v_2	2	2	7	6	12	9	nil	3	P	v_3
v_3	3	3	3	3	9	8	nil	4	P	v_4
v_4	4	4	7	2	14	5	nil			

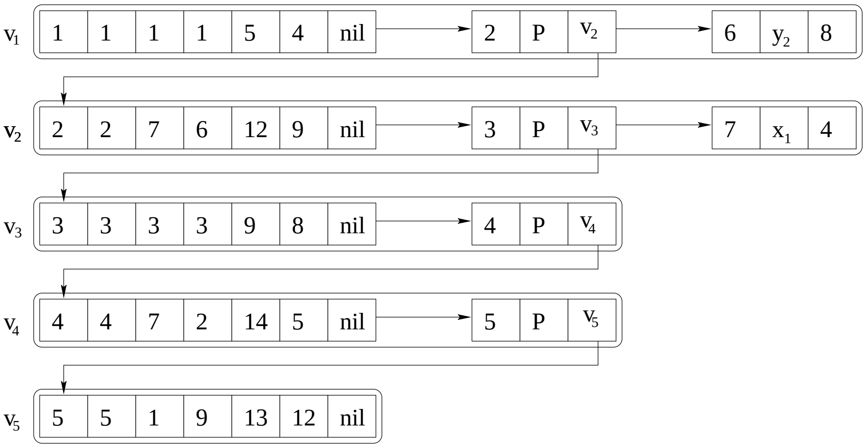
5. Einfügen des Rechtecks $R_5 = (1, 9, 13, 12)$.



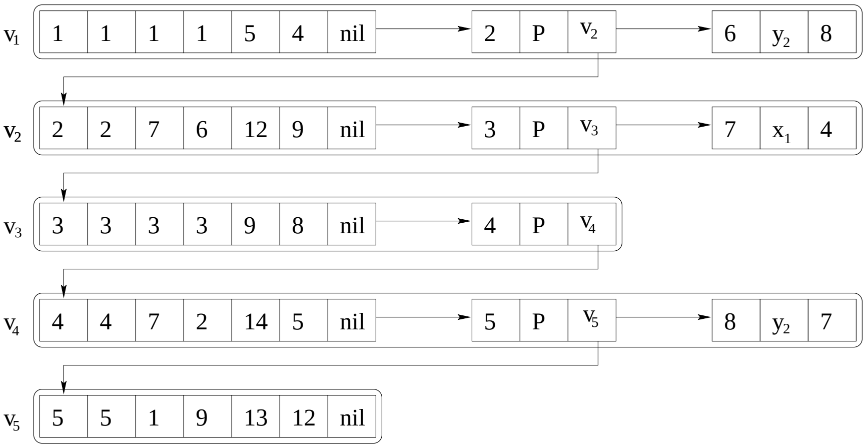
6. Ändern der Koordinate y_2 von R_1 auf 8.



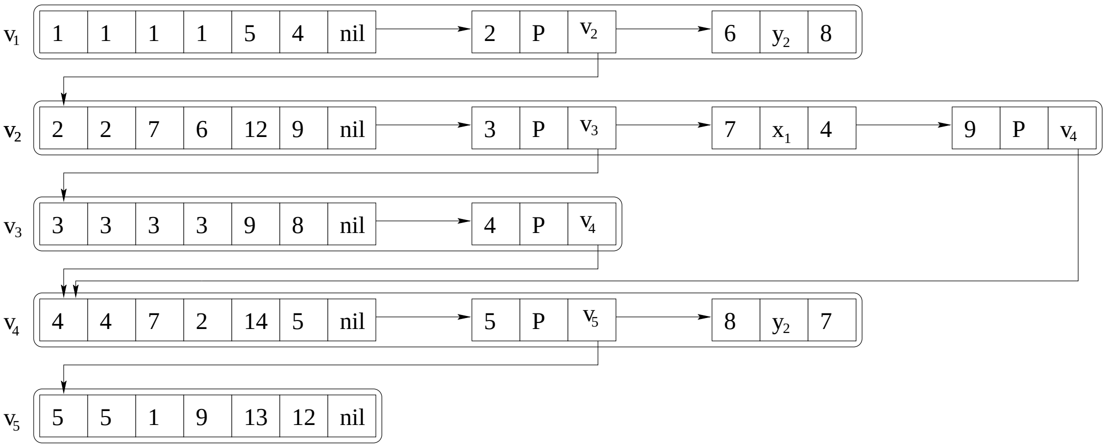
7. Ändern der Koordinate x_1 von R_2 auf 4.



8. Ändern der Koordinate y_2 von R_4 auf 7.



9. Löschen des Rechtecks R_3.



10. $\textsc{Access}(R_1.y_2, 7)$ liefert den Wert 8.

11. $\textsc{Access}(R_3.x_1, 9)$ liefert einen Fehler, da R_3 zum Zeitpunkt 9 gelöscht wurde.

Zusätzlich muß man in einem *Access-Array* für jeden Zeitpunkt eine Referenz auf denjenigen Knoten speichern, der zu diesem Zeitpunkt die Wurzel war. Da wir aber am Kopf der Liste nichts gelöscht oder eingefügt haben, verweisen alle Einträge ab Zeitpunkt 1 auf den Knoten v_1. Für Zeitpunkt 0 wird *nil* vermerkt, da sie Liste anfangs leer war.