6. Suchbaumstrukturen für Intervalle

- *abzuspeichern:* Mengen geschlossener Intervalle als Elemente
- *Suchoperationen:* suche zu x ein/alle Intervalle J mit $x \in J$ (auch "Aufspieß"-Anfragen, engl. *stabbing queries*: alle Intervalle, die von x "aufgespießt" werden);

  oder allgemeiner: suche zu gegebenem Intervall I ein überlappendes/alle überlappenden Intervall[e] J, also mit I overlaps J

Erweiterung klassischer Suchbäume: Intervallbäume v1 [1]

- *Basis:* ausgeglichene Suchbäume, z.B. AVL- oder Rot-Schwarz-Bäume; Ideen auch auf B-Bäume übertragbar
- *Suchoperation:* suche zu Intervall I ein überlappendes Intervall J
- *Schlüssel:* linker Endpunkt begin(I)
- *Nachteil:* u.U. muss gesamter Baum durchlaufen werden

---

[1] Version aus Datenstrukturen-Buch von Cormen et al.

Erweiterung klassischer Suchbäume (Forts.)

- *nötige Erweiterung:* Jeder Knoten[2] v erhält zusätzlich die Information

  $\mathsf{max}(v) =$ maximaler Endpunkt aller Intervalle im Teilbaum mit Wurzel v

- *Aktualisierung:* Wegen

  $\mathsf{max}(v) = \mathsf{max}(\mathsf{end}(\mathsf{cont}(v)), \mathsf{max}(\mathsf{leftchild}(v)), \mathsf{max}(\mathsf{rightchild}(v)))$

  bleiben Einfügen und Löschen logarithmisch.

- *Suche* nach einem mit I überlappenden Intervall im Baum T :

  $v \leftarrow \mathsf{root}(T)$;
  while not isExternal(v) **and not** I overlaps $\mathsf{cont}(v)$ **do**
    if not isExternal$(\mathsf{leftchild}(v))$
      and $\mathsf{begin}(I) \leq \mathsf{max}(\mathsf{leftchild}(v))$
        then $v \leftarrow \mathsf{leftchild}(v)$ **else** $v \leftarrow \mathsf{rightchild}(v)$;
  return v.

  arbeitet logarithmisch und korrekt !

- *Beispiele:* Baum s. Folgeseite; teste $I := [22, 25]$, $I := [11, 14]$

[2] Jeder interne Knoten v hat normalerweise einen Inhalt $\mathsf{cont}(v)$, hier ein Intervall, und zwei Nachfolger $\mathsf{leftchild}/\mathsf{rightchild}(v)$; externe Knoten sind leer.

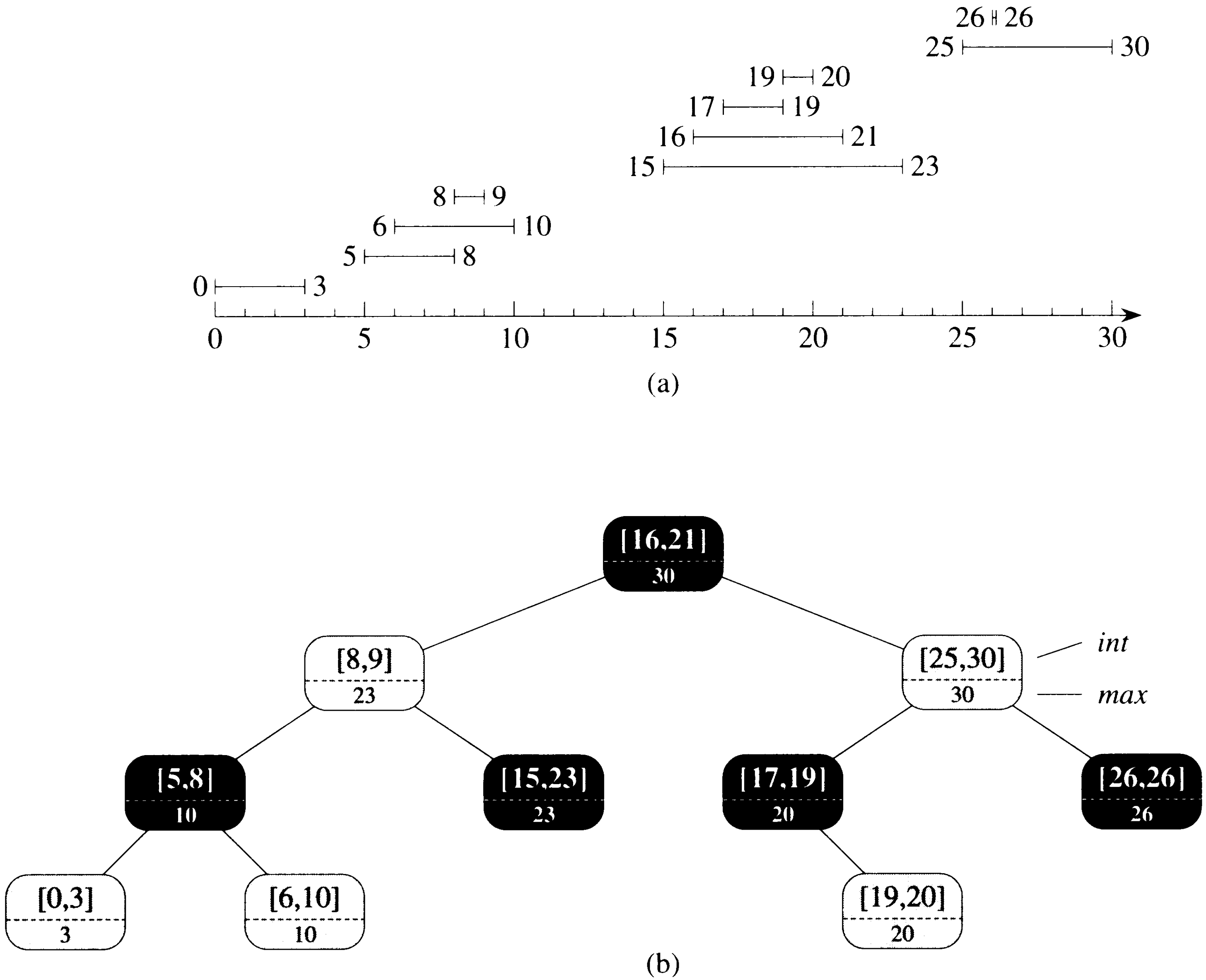


Abbildung 14.4: *Ein Intervallbaum.* ***(a)*** *Eine Menge von 10 Intervallen, sortiert in aufsteigender Reihenfolge nach dem linken Endpunkt.* ***(b)*** *Der Intervallbaum, der diese Intervalle darstellt. Eine Inorder-Traversierung des Baumes listet die Knoten nach dem linken Endpunkt sortiert auf.*

Ursache dafür, dass an jedem Knoten v mit $\mathsf{cont}(v) = J$ die richtige Entscheidung getroffen wird, ist die sogenannte "Intervalltrichotomie":

$$\text{Für je zwei Intervalle } I, J \text{ gilt: } I \;\mathsf{overlaps}\; J \;\vee\; I \;\mathsf{before}\; J \;\vee\; I \;\mathsf{after}\; J$$

Also: $(\mathsf{begin}(I) \leq \mathsf{end}(J) \wedge (\mathsf{begin}(J) \leq \mathsf{end}(I))$
$\vee\; \mathsf{end}(I) < \mathsf{begin}(J) \;\vee\; \mathsf{end}(J) < \mathsf{begin}(I)$.

(re) Alle Intervalle im rechten Teilbaum beginnen aufgrund der Suchbaumanordnung $\geq \mathsf{begin}(J)$.

(li1) Mindestens ein Intervall im linken Teilbaum reicht aufgrund der Suchbaumanordnung von einem Anfang $\leq \mathsf{begin}(J)$ bis $\mathsf{max}(\mathsf{leftchild}(v))$.

(li2) Alle Intervalle im linken Teilbaum enden $\leq \mathsf{max}(\mathsf{leftchild}(v))$.

Falls für J jetzt (**not** $I \,\mathsf{overlaps}\, J$) und (*) $\mathsf{begin}(I) \leq \mathsf{max}(\mathsf{leftchild}(v))$ gilt (**then**-Zweig), folgt:

- Falls I before J, also $\mathsf{end}(I) < \mathsf{begin}(J)$, braucht wegen (re) im rechten Teilbaum nicht gesucht zu werden; nach links ok.

- Falls I after J, also $\mathsf{end}(J) < \mathsf{begin}(I)$, muss es wegen (li1) und (*) links einen Überlappungskandidaten geben; nach links ok.

Falls (*) nicht gilt (**else**-Zweig), also $\mathsf{max}(\mathsf{leftchild}(v)) < \mathsf{begin}(I)$ kann es wegen (li2) nur noch rechts Überlappungskandidaten geben; nach rechts ok.

Erweiterung klassischer Suchbäume (Forts.)

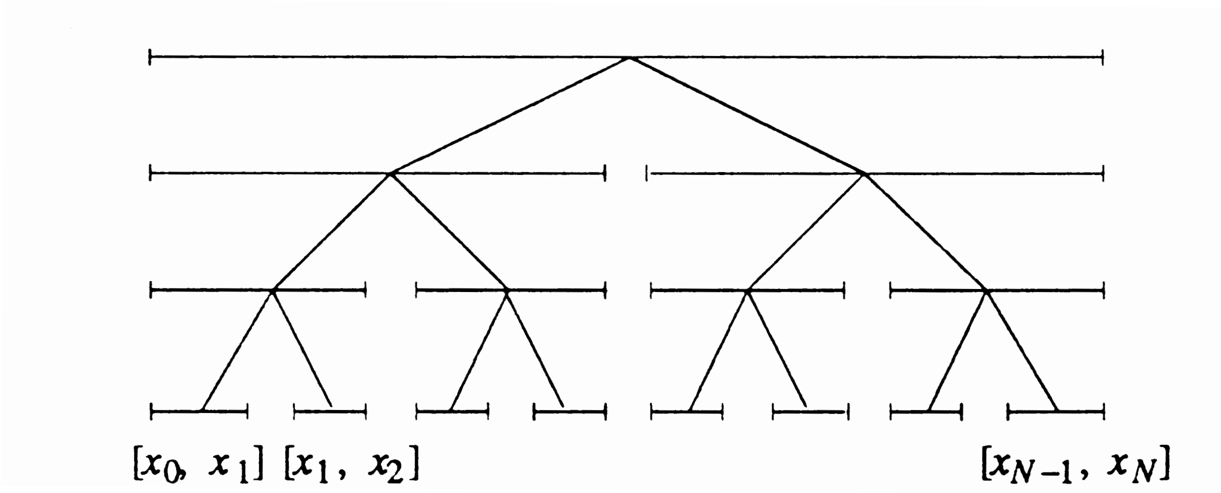
Varianten:

- suche alle I überlappenden Intervalle (z.B. zu $I := [18, 22]$)
  $\rightarrow$ verfolge mehrere Pfade parallel
- merke dazu in jedem Knoten $\mathsf{min..max}$ des zugehörigen Teilbaums, und betrete jeden Teilbaum, dessen $[\mathsf{min}, \mathsf{max}]$ mit I überlappt
- vereinfache: alle Intervalle (plus Nutzdaten) nur in den Blättern; interne Knoten tragen nur $\mathsf{min..max}$-Information
- führe Mehrwegestruktur für externe Speicherung ein
- $\cdots \rightarrow\rightarrow$ **R-Bäume** *(später, für mehrdimensionalen Fall)*

Umbau von Suchbäumen: Segmentbäume

- *Suchoperation:* suche zu "Koordinate" x alle Intervalle J mit $x \in J$
- *semidynamischer Ansatz:* **Segmentbäume** [3]

  Die Endpunkte der zu speichernden Intervalle bilden ein fest vorbereitetes Raster[4] $x_0, \ldots, x_N$. Ein Segmentbaum ist ein binärer Baum, der eine hierarchische Einteilung des Koordinatenbereichs in kleinste Intervalle und daraus paarweise zusammengesetzte Intervalle darstellt[5]; jeder Knoten repräsentiert eines dieser Intervalle.

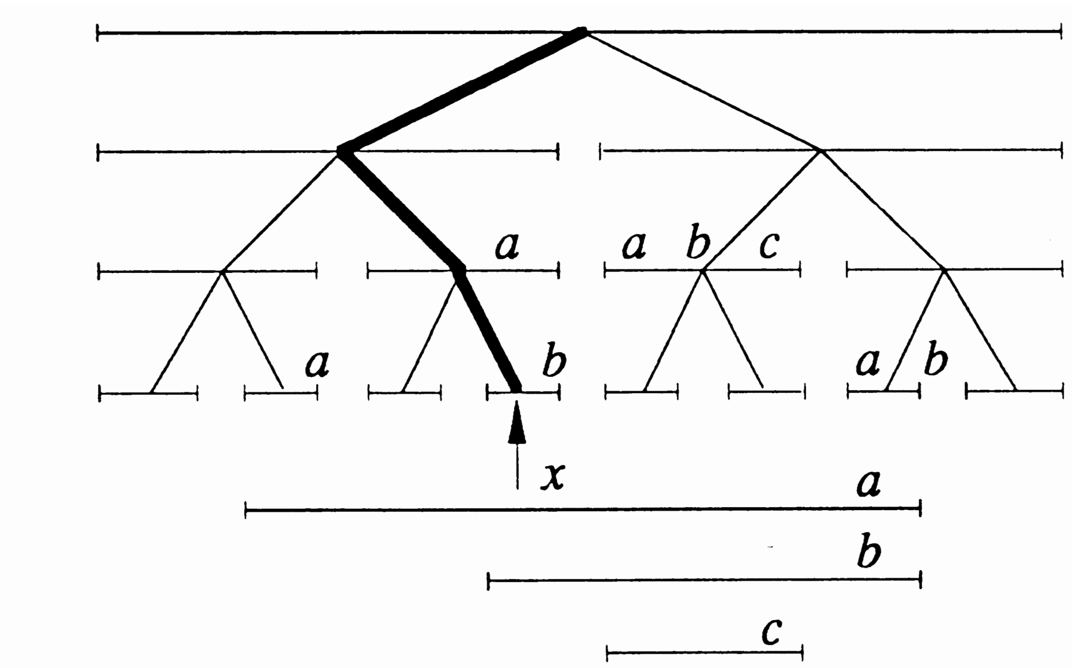


[3] nach Bentley; vgl. Datenstrukturen-Bücher von Güting und von Ottmann
[4] ein im Gegensatz zur Abbildung nicht unbedingt äquidistantes Raster
[5] falls die Anzahl keine Zweierpotenz ist, ist der Baum auf der rechten Seiten mit trivialen Intervallen aufzufüllen

Segmentbäume

Ein Intervall a wird im Segmentbaum gespeichert, indem ein Verweis darauf in jedem höchsten von a “bedeckten Knoten” vermerkt wird.

Zur Suche, welche aktuellen Intervalle eine Koordinate x enthalten, wird der Pfad der Knoten verfolgt, deren zugehörige Intervalle x enthalten, von der Wurzel bis zur Blattebene; alle in diesen Knoten vermerkten Intervalle werden gemeldet.



Zeitaufwand: $O(\log N + k)$, wobei $k =$ Anzahl der gefundenen Intervalle

Segmentbäume (Forts.)

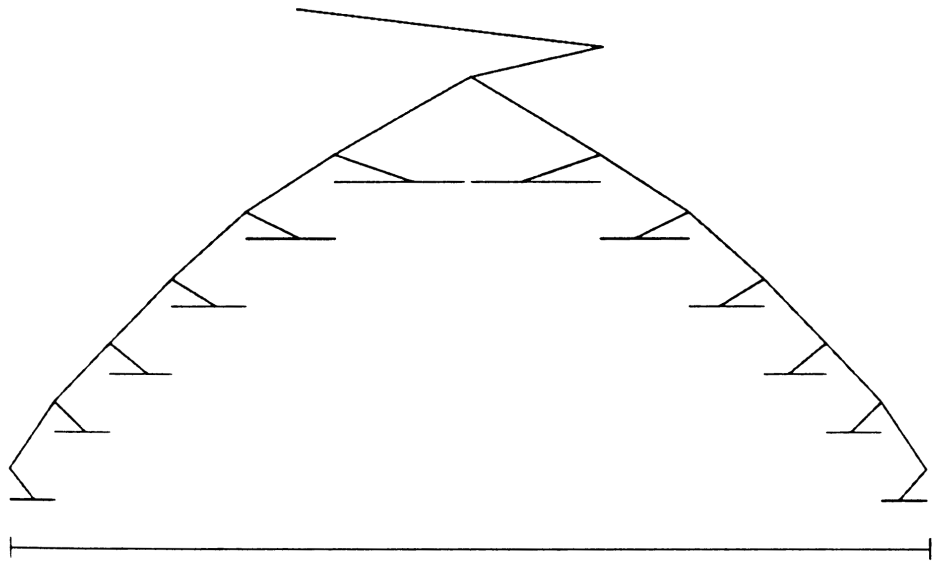
Obige Suche:[6]

procedure *report* $(p : Knoten; x : Punkt)$;
$\{$*anfangs ist p die Wurzel des Segment-Baumes*$\}$
gebe alle Intervalle der Liste von p aus;
if *p ist Blatt*
  then *fertig*
  else
    begin
      if (p *hat einen linken Sohn* p_λ) **and** $(x \in I(p_\lambda))$
        then $report(p_\lambda, x)$;
      if (p *hat einen rechten Sohn* p_ρ) **and** $(x \in I(p_\rho))$
        then $report(p_\rho, x)$
    end

[6] $I(p)$ bezeichne das durch den Knoten p repräsentierte Intervall über dem gegebenem Raster.

Segmentbäume (Forts.)

Auch das Einfügen und das Löschen eines Intervalls sind in $O(\log N)$ möglich, denn es müssen nur höchste bedeckte Knoten aufgesucht werden, und diese liegen an einem “gegabelten Pfad” der folgenden Form:



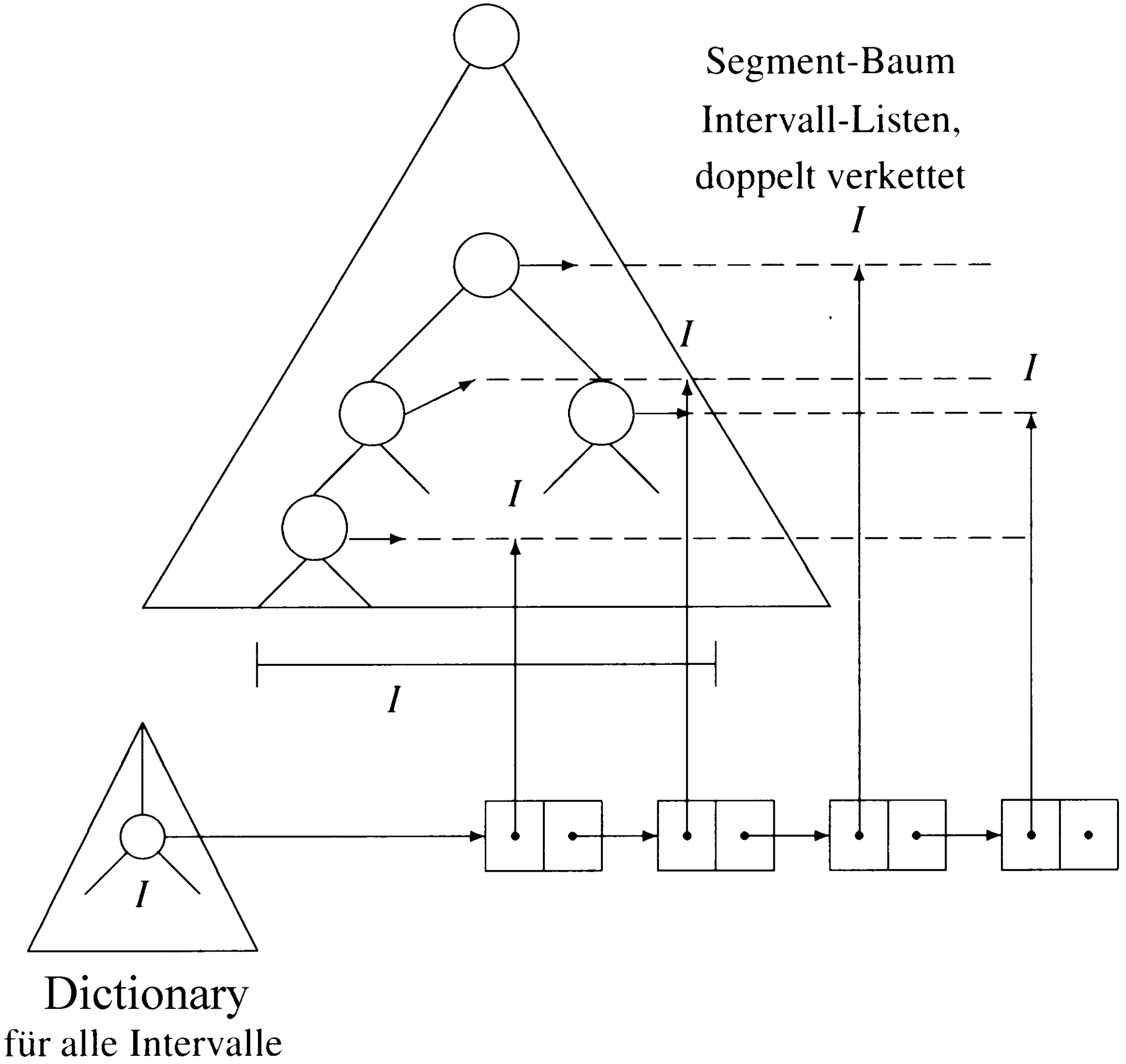
Jedes zu speichernde Intervall ist somit an höchstens $O(\log N)$ Knoten (in geeigneten Listen) vermerkt.

Segmentbäume (Forts.)

procedure *Einfügen* $(I : \mathit{Intervall}; p : \mathit{Knoten})$;
{*anfangs ist p die Wurzel des Segment-Baumes*}
  if $I(p) \subseteq I$ [7]
    then
      füge I in die Intervall-Liste von p ein und fertig
    else
      begin
        if (p *hat linken Sohn* p_λ) **and** $(I(p_\lambda) \cap I \neq \emptyset)$
          then $\mathit{Einfügen}(I, p_\lambda)$;
        if (p *hat rechten Sohn* p_ρ) **and** $(I(p_\rho) \cap I \neq \emptyset)$
          then $\mathit{Einfügen}(I, p_\rho)$
      end

[7] $I_1 \subseteq I_2$: I_1 ist vollständig in I_2 enthalten; $I_1 \cap I_2 \neq \emptyset$: die Intervalle I_1, I_2 überlappen sich.

Segmentbäume (Forts.)



(Hilfsstruktur zur Unterstützung des Löschens)

Umbau von Suchbäumen: Intervallbäume v2

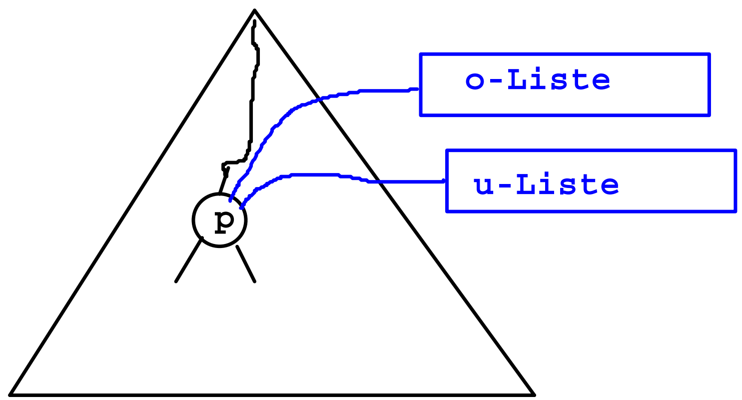
- *Suchoperation:* suche zu “Koordinate” x alle Intervalle J mit $x \in J$
- *dynamischer Ansatz:* **Intervallbäume v2** [8]

  Verwendet wird ein dynamischer, ausgeglichener Suchbaum *aller* Intervallgrenzen als Schlüssel. An jedem Knoten p dieses “Skeletts” wird eine Liste von Intervallen zweimal gespeichert:

  – o-Liste: nach absteigenden oberen Endpunkten sortierte Liste

  – u-Liste: nach aufsteigenden unteren Endpunkten sortierte Liste

  Ein Intervall $[l, r]$ kommt in die u/o-Liste des höchsten Knotens p (d.h. des Knotens minimaler Tiefe), für den $p.key \in [l, r]$ gilt.

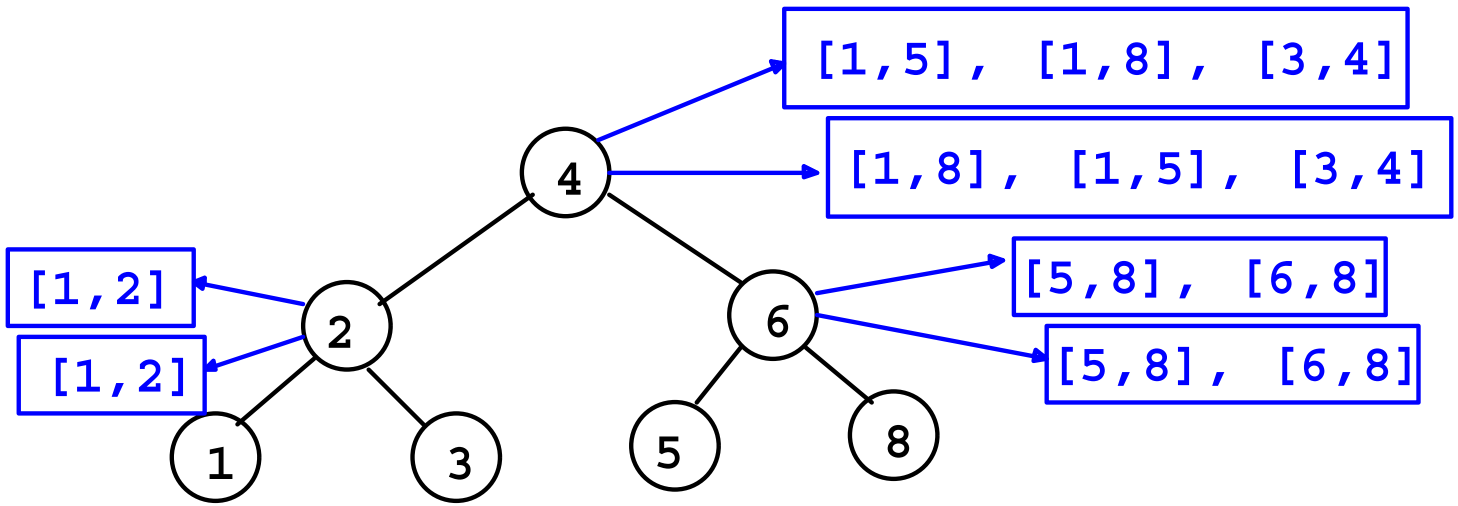


[8]Version nach Edelsbrunner; vgl. Datenstrukturen-Buch von Ottmann

Intervallbäume v2 (Forts.)

Beispiel:

{[1,2], [1,5], [3,4], [5,8], [6,8], [1,8]}

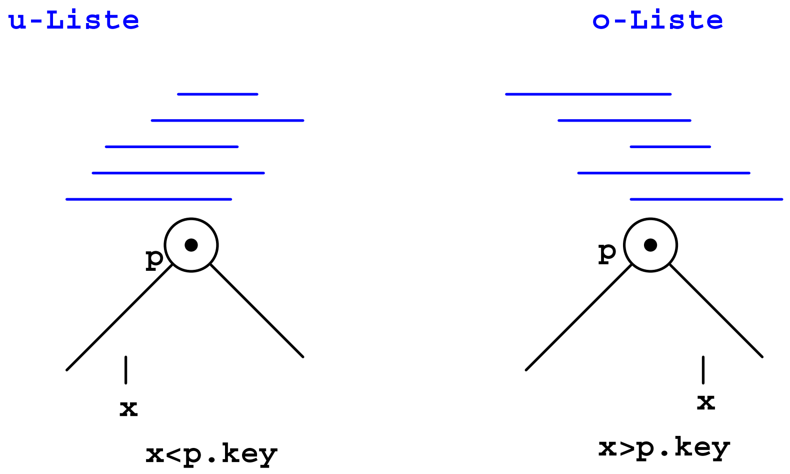


Intervallbäume v2 (Forts.)

procedure *Einfügen* (I : *Intervall*; p : *Knoten*);
{*anfangs ist p die Wurzel des Intervall-Baumes; I ist ein Intervall mit linkem Endpunkt .I und rechtem Endpunkt I.*}
if $p.key \in I$
  then
    füge I entsprechend seinem unteren Endpunkt in die u-Liste von p und entsprechend seinem oberen Endpunkt in die o-Liste von p ein und fertig!
  else
    if $p.key < .I$
      then $Einfügen(I, p_\rho)$
        else $\{p.key > I.\}$
          $Einfügen(I, p_\lambda)$

zusätzlich/schwieriger: Vorab-Einfügen neuer Intervallgrenzen
→ erfordert ggf. Restrukturierungen wie in ausgeglichenen Suchbäumen, einschl. teilweises Neu-Einfügen von Intervallen, was (ebenso wie das Löschen) wiederum ein Dictionary aller Intervalleinträge benötigt

Intervallbäume v2 (Forts.)



procedure *report*$(p : Knoten; x : Punkt)$;
if $x = p.key$
  then
    gebe alle Intervalle der u-Liste (oder alle Intervalle der o-Liste)
    von p aus und fertig!
  else
    if $x < p.key$
      then
        gebe alle Intervalle I der u-Liste von p mit
          $.I \leq x$ *aus* $\{$*das ist ein Anfangsstück dieser Liste!*$\}$
          report (p_λ, x)
      else $\{x > p.key\}$
        gebe alle Intervalle I der o-Liste von p mit
          $I. \geq x$ *aus* $\{$*das ist ein Anfangsstück dieser Liste!*$\}$
          report (p_ρ, x)

Intervallbäume v2 (Forts.)

- *Alternative*[9]: rechne mit allen möglichen x-Koordinaten, abgebildet auf einen Bereich $[1, 2^h - 1]$, nutze vollständigen Suchbaum darüber (wie im Beispiel) — jedoch nur virtuell !
- $\Rightarrow$ navigiere arithmetisch, um "fork node" (Speicherknoten) zu einem Intervall $[\text{lower}, \text{upper}]$ mit $\text{lower} \leq \text{node} \leq \text{upper}$ zu ermitteln :

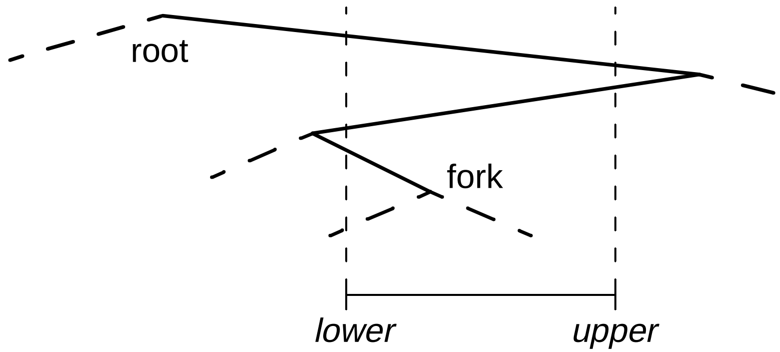


Figure 3: Fork node of an interval in the tree.

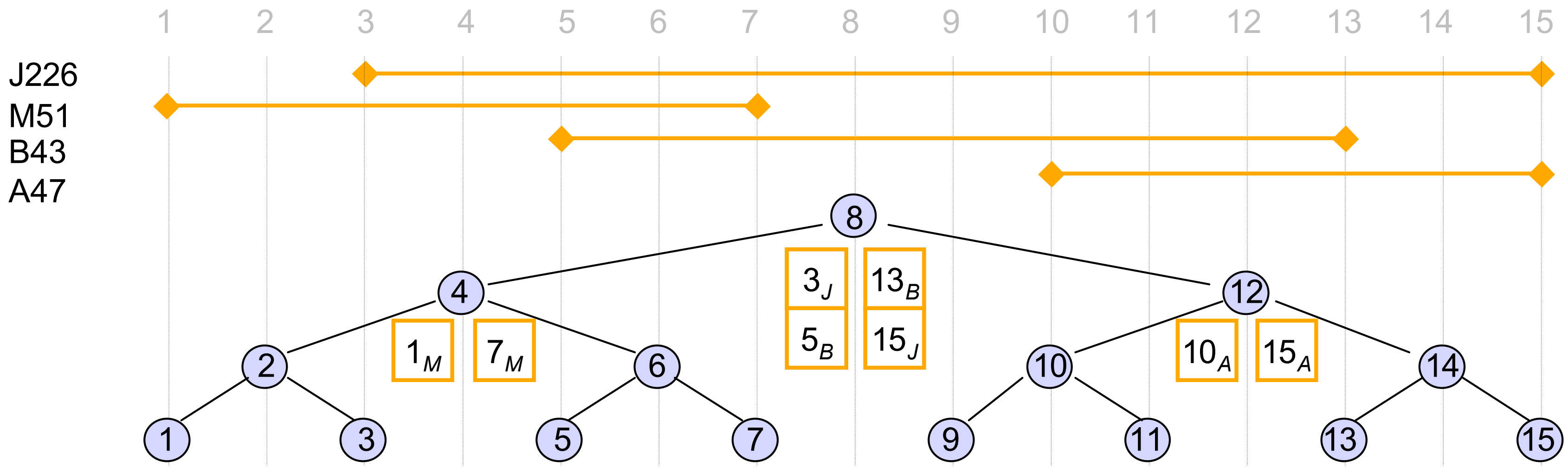
```
FUNCTION int forkNode (int lower, int upper) {
  int node = root;
  for (int step = node/2; step >= 1; step /= 2)
    if (upper < node) node –= step;
    elsif (node < lower) node += step;
    else break;
  return node;
}
```

Figure 4: Computing the fork node of an interval.

- speichere i.w. nur die u/o-Listen — was in einer relationalen, mit klassischen Indexen versehenen Datenbank möglich ist !

[9] nach Kriegel et al in Proc. VLDB 2000

Intervallbäume v2 (Forts.)



- *Primärstruktur:*
  Binärbaum mit Wurzel 2^{h-1} über dem Bereich $[1..2^{h-1}]$
- *Intervalle:* Jedes Intervall ist genau einem Knoten zugeordnet.
- *Sekundärstruktur:*
  Zwei sortierte Listen von Intervallgrenzen an den Knoten
- *Relationale Speicherung:* Eine Intervalle-Tabelle mit zwei Indexen

Intervallbäume v2: Relationale Speicherung

- *DB-Schema:*

```
CREATE TABLE Intervals (node int,
         lower int, upper int, id int);
CREATE INDEX lowerIndex ON Intervals (node,lower,id);
CREATE INDEX upperIndex ON Intervals (node,upper,id);
```

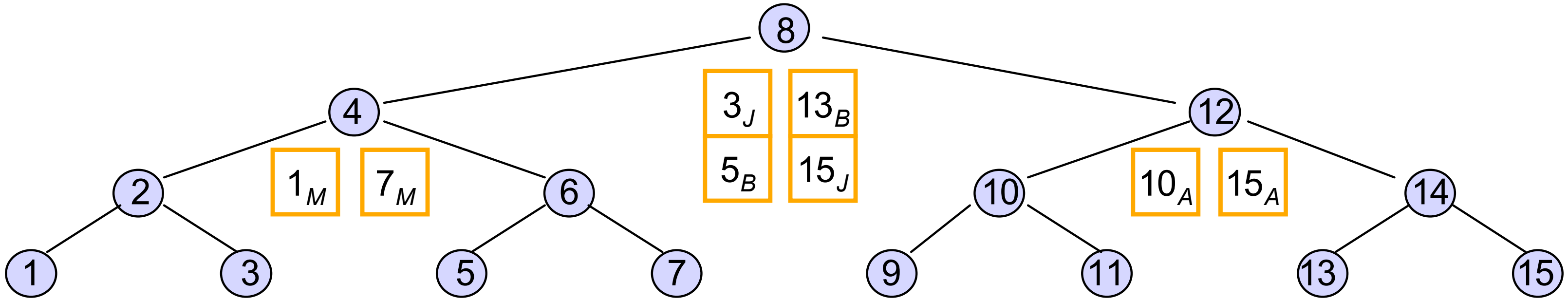
Zwei relationale Indexe speichern die Intervallgrenzen.

- *Einfügen:*

```
INSERT INTO Intervals
  VALUES (forkNode(:lower,:upper), :lower,:upper,:id);
```

- *zu implementierende Suchoperation:*
suche zu Intervall $I = [:\mathsf{lower}, :\mathsf{upper}]$ alle Intervalle J mit I overlaps J

Intervallbäume v2: Relationale Speicherung (Forts.)



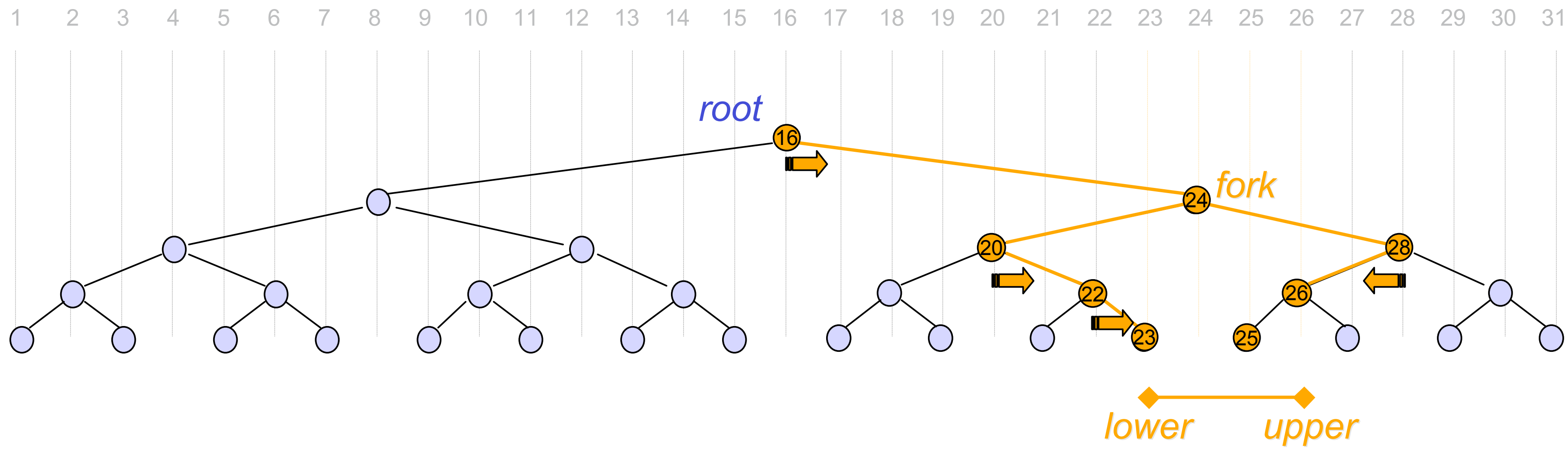
Intervals	node	lower	upper	id
	12	10	15	A47
	8	5	13	B43
	8	3	15	J226
	4	1	7	M51

lowerIndex	node	lower	id
	4	1	M51
	8	3	J226
	8	5	B43
	12	10	A47

upperIndex	node	upper	id
	4	7	M51
	8	13	B43
	8	15	J226
	12	15	A47

(Indexe sind eigentlich keine Tabellen, sondern typischerweise B*-Bäume.)

Intervallbäume v2: Anfragebearbeitung



Vorbereitung:

- Steige arithmetisch von root bis lower und bis upper ab und sammle die Knoten links/rechts in temporären internen Tabellen left/rightNodes (mit Attribut node, Größe $O(h)$).

Eigentliche Anfrage:

- Für Intervalle i in Knoten links von lower: Teste i.upper $\geq$ lower
- Für Intervalle i in Knoten rechts von upper: Teste i.lower $\leq$ upper
- Für Knoten von lower bis upper: Ausgabe aller Intervalle

Intervallbäume v2: Anfragebearbeitung (Forts.)

Implementierung als indexunterstützte Union/Join-Anfrage:

```
  SELECT id FROM leftNodes left, Intervals i,
  WHERE i.node=left.node AND i.upper>=:lower
UNION
  SELECT id FROM rightNodes right, Intervals i
  WHERE i.node=right.node AND i.lower<=:upper
UNION
  SELECT id FROM Intervals i
  WHERE i.node BETWEEN :lower AND :upper
```

unterstrichene Attributzugriffe mit Indexunterstützung:

$$\text{leftNodes} \bowtie_{\ldots}^{\text{REL,IND(upperIndex)}} \text{Intervals}$$

$$\cup\ \text{rightNodes} \bowtie_{\ldots}^{\text{REL,IND(lowerIndex)}} \text{Intervals}$$

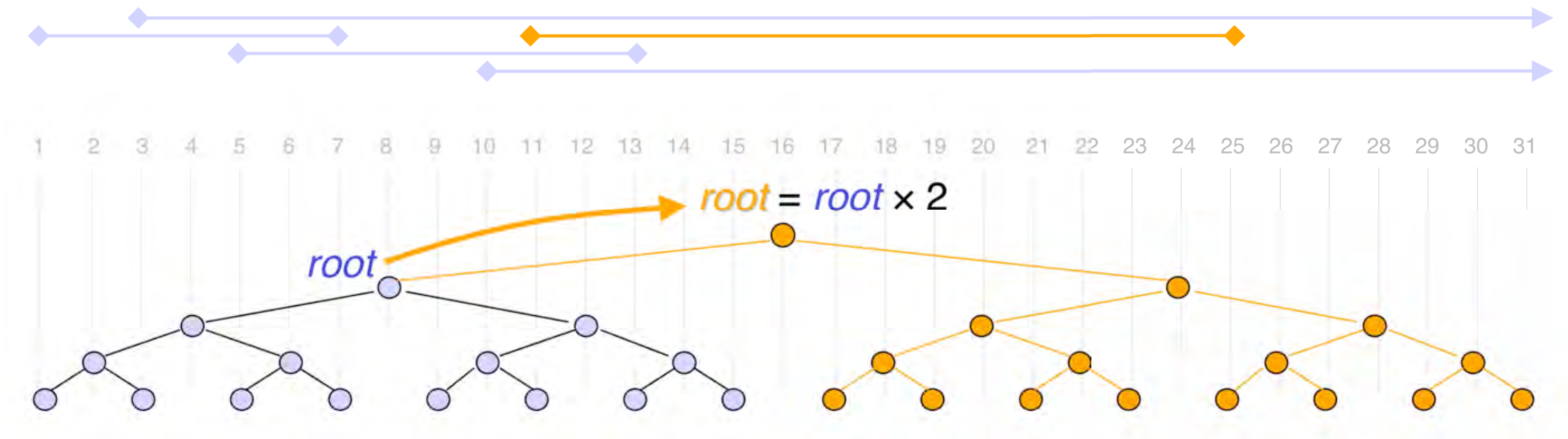
$$\cup\ \sigma_{\ldots}^{\text{IND(lowerIndex)}} (\text{Intervals})$$

Intervallbäume v2: Aufwand

Wie verhält sich das System bei großen Datenmengen?

- Gegeben sei eine Platte mit Blockgröße b, ein relational gespeicherter Intervallbaum der Höhe h und eine Menge von n Intervallen.
- Speicherplatzbedarf: $O(n/b)$ Plattenblöcke.
  - Platzbedarf ist linear abhängig von der Datenbankgröße.
- Einfügen oder Entfernen eines Intervalls: $O(\log_b n)$ Plattenzugriffe.
  - beachte Wartung der Indexe
  - Einfügezeit ist logarithmisch abhängig von der Datenbankgröße.
- Schnittanfrage mit r Ergebnissen: $O(h \cdot \log_b n + r/b)$ Plattenzugriffe.
  - vgl. obigen Ausführungsplan + Ergebnisausgabe
  - Suchzeit wird dominiert von der Größe r der Ergebnismenge.

Intervallbäume v2: Erweiterung des Datenraums



- Verdoppelung wirkt sich wegen virtueller Behandlung nur auf die Baumberechnungen aus (Höhe nur +1)
- keine Indexreorganisation
- Erweiterung nach oben zugeschnitten auf temporale Datenbanken
- Intervalle bis FOREVER hängen an einem Spezialknoten (wie maxint, aber nicht im Baum berücksichtigt!), der den rightNodes extra hinzugefügt wird