5. Organisation zeitabhängiger Daten

Überblick

1. Einführung: Motivation und Problemstellung
2. Indexstrukturen für Transaktionszeit
   - 2.1 Bisherige Methoden
   - 2.2 Datenduplizierende Verfahren
     - Time-Split B-tree
     - Multi-Version B-Tree (MVBT)
   - 2.3 Anfragebearbeitung ohne Duplikate
3. Indexstrukturen für absolute Zeit
   - 3.1 Segmentbaum
   - 3.2 Intervallbaum
   - 3.3 Prioritäts-Suchbaum

5.1. Einführung

Anwendungen im Bereich Banken & Versicherungen

- Kontoführung
  - Anlegen eines neuen Kontos
  - Ändern eines Kontostands **(Erhalten** des alten Wertes)
  - Auflösen eines Kontos
- Anfragen:
  - Kontoauszüge
  - Stand aller Konten am 31.5.95 in der Filiale Leopoldstr.

Multi-Media Anwendungen

- Speicherung zeitabhängiger Daten
  - Video, Audio, …

Pervasive Computing

- Verwaltung mobiler Objekt in Abhängigkeit der Zeitachse

Was kann mit Zeit in Beziehung sein?

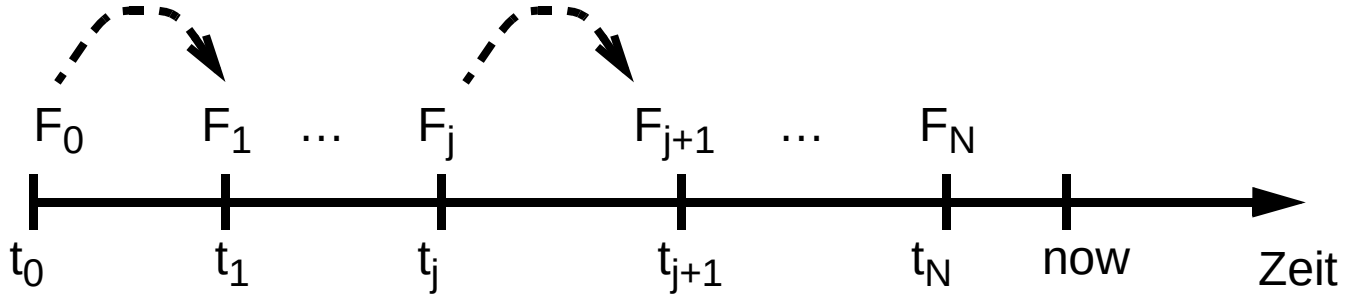
- Attributwerte
- **Datensätze**
- Objekte
- Mengen von Datensätzen
- Datenobjekte des Schemas

Welche Zeitachsen gibt es in Datenbanken?

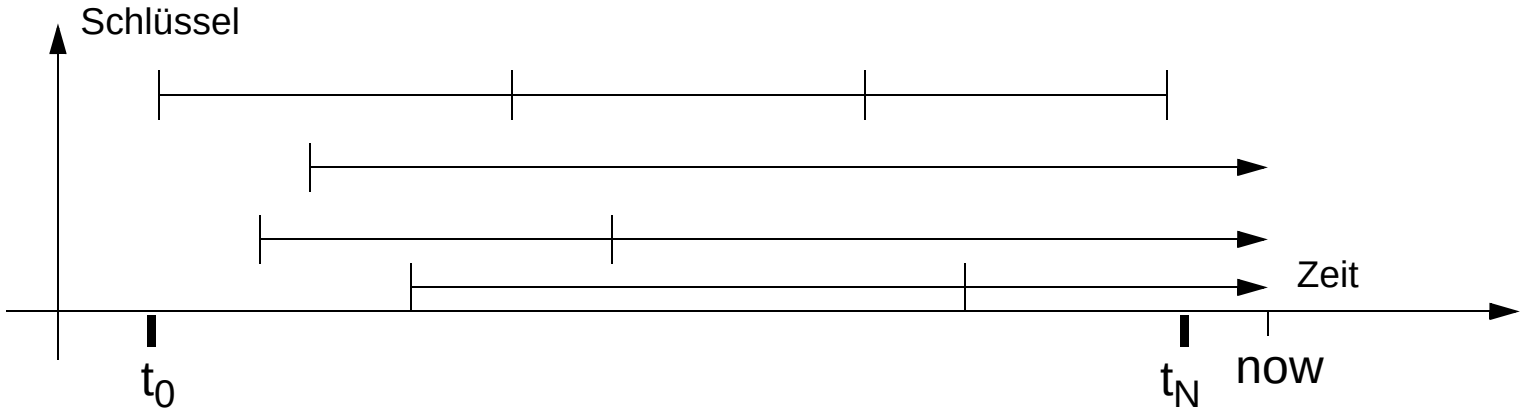
- absolute Zeit
  - Änderungen können sich auf den zukünftigen, gegenwärtigen und historischen Zustand der Datenbank auswirken.
  - hält fest, wann ein Fakt in der realen Welt wirksam wurde.
- Transaktionszeit
  - Änderungen wirken nur auf den gegenwärtigen Zustand der Datenbank ein.
  - hält fest, wann ein Fakt in der Datenbank wirksam wurde.
- Beispiel:
  - Vertrag bei einer Versicherung

5.2 Indexstrukturen für Transaktionszeit

- datei-orientierter Blickwinkel:
  - neue Versionsdatei entsteht durch eine Änderungsoperation



- datensatz-orientierter Blickwinkel:
  - Datensatz gehört zu einer zusammenhängenden Menge von Versionen



Datensatz R besteht aus folgenden Komponenten:

- Schlüssel K
- Informationsteil *info*
- Zeitpunkt t_{ins}, an dem der Datensatz in die Datenbank eingefügt wurde oder an dem der Informationsteil *info* geschrieben wurde.
- Zeitpunkt t_{del}, an dem der Datensatz aus der Datenbank gelöscht wurde oder an dem der Informationsteil *info* überschrieben wurde.

Einfügen eines Datensatzes (K,info) zum Zeitpunkt t

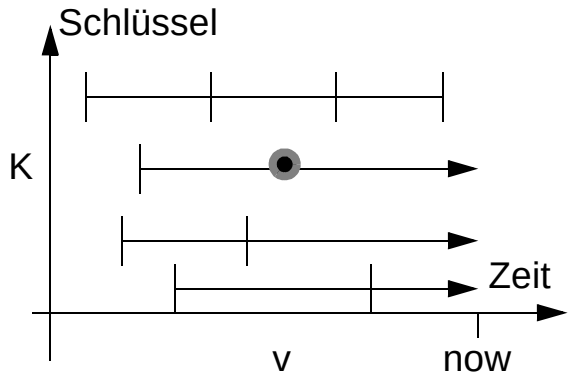
- INC(now); Setze $t_{now} = t$.
- Füge den Datensatz R = (K, info, t_{now}, *) in die Datenbank ein. (F_{now} entsteht).

Löschen eines Datensatzes (K,info) zum Zeitpunkt t

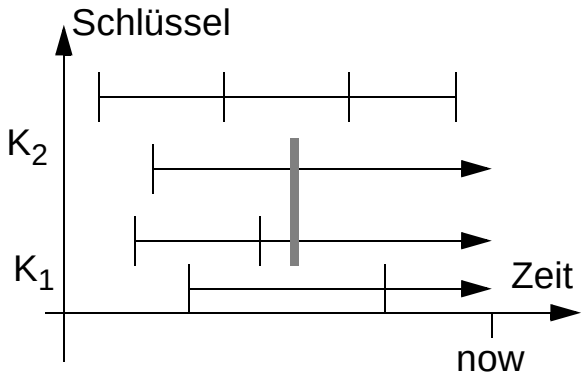
- Falls es ein Datensatz (K, info) im aktuellen Zustand der Datenbank (F_{now}) gibt, d.h. es existiert ein Zeitpunkt t_{ins}, ins $\leq$ now, so daß es R = (K, info, t_{ins}, *) gibt.
  - INC(now); Setze $t_{now} = t$ und R = (K, info, t_{ins}, t_{now}).
  - Zurückschreiben des modifizierten Datensatzes R.

Anforderungen

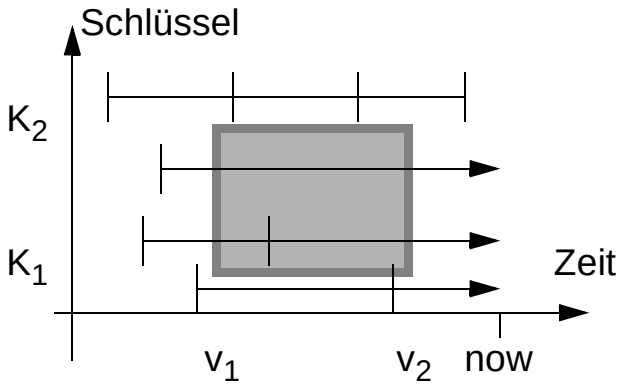
- effiziente Unterstützung von Anfragen



versionsbasierte exakte Anfrage



Schlüssel-Bereichsanfrage



Version-Schlüssel-Bereichsanfrage

- effiziente Unterstützung von Änderungsoperationen
  - Einfügen, Abändern vorhandener Datensätze, Löschen
- niedriger Speicheraufwand für die Versionsdateien
- Annahme: Anfragen oft auf neuen Versionen

Wunscheigenschaften einer effizienten Lösung

- B = Kapazität einer Seite !! **keine** Konstante !!
- N = Anzahl der Versionen

Zugriffszeit

- wie beim B+-Baum (asymptotisch):
  - versionsbasierte exakte Suche (in Version j): $O(\log_B n)$
  - Schlüssel-Bereichssuche in Version j: $O(\log_B n + r)$
  - Änderungsoperationen: $O(\log_B n)$
- n sollte dabei möglichst die Anzahl der Datensätze in Version j bzw. Version *now* sein.
- Version--Schlüssel Bereichsanfragen ?

Speicheraufwand

- O(N) = linear in der Anzahl der Versionen

5.2.1 Bisherige Ansätze

1. Naiver Ansatz:

- Änderungsoperation ==> Kopieren der kompletten Datei
  - \+ effiziente Abfragebearbeitung
  - \- hohe Speicherkosten

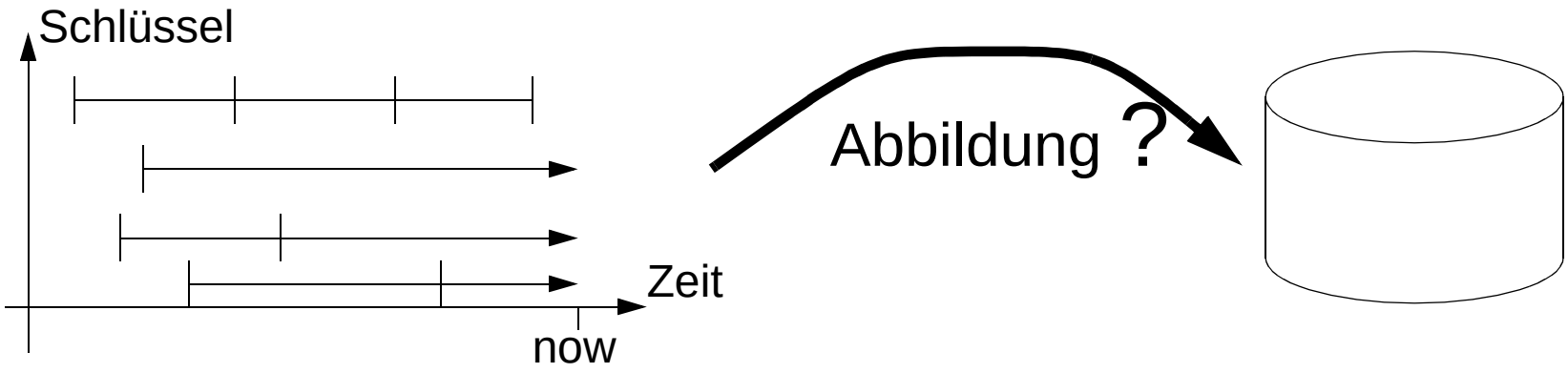
    Für den Fall, dass N Datenobjekte in eine am Anfang leere Datei eingefügt werden, ist der Speicheraufwand $O(N^2)$.

2. Naiver Ansatz:

- Datensatz: (K, in_time, del_time, info)
- verwende zwei-dimensionale Indexstruktur (z. B. R-Baum)
  - \+ geringe Speicherkosten
  - \- hohe Anfragekosten: > O(log N) bei exakter Versionsanfrage
- Regionen der Datenseiten sind i. A. sehr lang!

5.2.2 Datenduplizierende Methoden

- geeignet für Klasse von Zugriffsstrukturen (inkl. B^+-Bäume)



- zweidimensionaler Datensatz mit Ausdehnung: <K, in_time, del_time, info>

Idee:

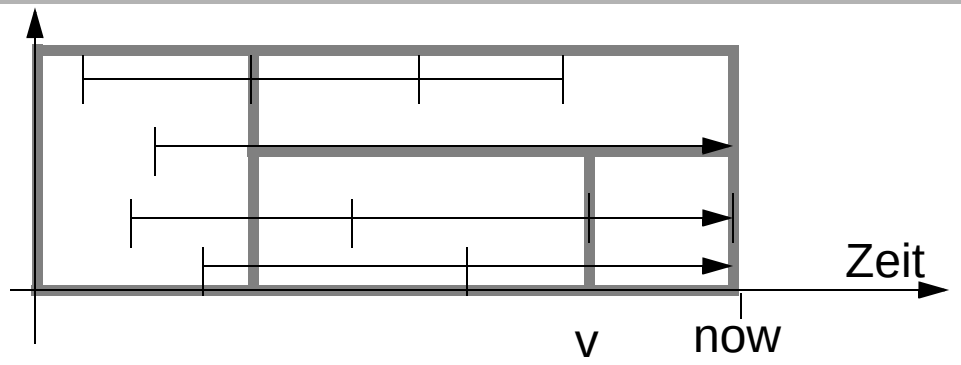
- unterteile den Datenraum vollständig in nicht-überlappende Regionen
- ordne jeder Region eine Seite zu und speichere die zweidimensionalen Datensätze in den Seiten, so daß

  Satz schneidet Seitenregion ==> abspeichern in der entsprechenden Seite

Was passiert beim Überlauf einer Seite?

Zeit-Spalten

Zeit-Spalten



Wähle einen Zeitpunkt t_{split} und teile die Datensätze aus der Seite L auf die Seiten L und R folgendermaßen auf:

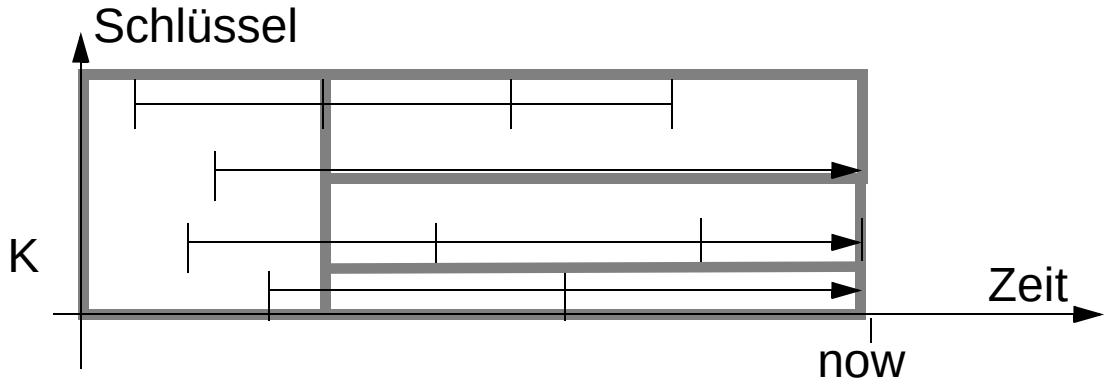
- alle Datensätze R mit $R.t_{del} < t_{split}$ verbleiben in der Seite L.
- alle Datensätze R mit $R.t_{ins} \geq t_{split}$ kommen in die neue Seite R.
- alle anderen Datensätze verbleiben in L und kommen zusätzlich nach R.

Eigenschaften:

- \- hohe Speicherkosten durch Erzeugung der Duplikate
- \+ niedrige Anfragezeiten (durch Clusterung)

Schlüssel-Spalten

Schlüssel-Spalten



Wähle einen Schlüssel K_{split} und teile die Datensätze aus der Seite L auf die Seiten L und R (analog zum B+-Baum):

– alle Datensätze R mit $R.K < K_{split}$ verbleiben in der Seite L.

– alle Datensätze R mit $R.K \geq K_{split}$ kommen in die neue Seite R.

Eigenschaften:

\+ niedrige Speicherkosten

\- hohe Anfragekosten

5.2.2.1 Time-Split-B-Tree (TSBT)

- Lomet & Salzberg (1989 / 1990 / 1993) haben basierend auf von Eastman's Write-Once B-tree (1986) folgenden Ansatz vorgestellt.

- Regeln für das Aufspalten beim TSBT:

  Falls historische Datensätze existieren:

  1. Zeit-Spalten: Zeitpunkt der letzten Änderungsoperation in der Seite L.
  2. #Datensätze > 2/3*B ==> Schlüssel-Spalten

- Nachteile des Verfahrens
  - logisches Löschen durch leere Stellvertreter-Datensätze
  - keine garantierte Clusterung alter Versionen
  - gesonderte Behandlung von Index- und Datenseiten
    - komplizierte Algorithmen
    - worst-case Verhalten ?

5.2.2.2 Multiversion B-Tree (MVBT)

- Becker, Geschwind, Ohler, Seeger, Widmayer (1993, 1996)

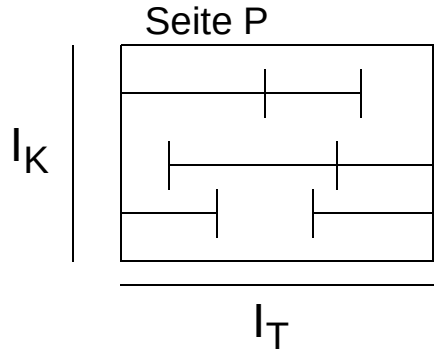
Entwurfsziele:

- Anfrageleistung: (asymptotisch) wie beim B+-Baum (in jeder Version)
- Speicherkosten: O(N) = linear in der Anzahl der Versionen

notwendige Bedingungen:

- eine Seite P mit Region I_T x I_K

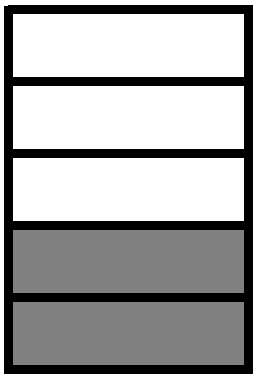
  für alle v $\in$ I_T ==>
  es gibt $d = \Theta(B)$ Datensätze in P,
  die zur Version v gehören

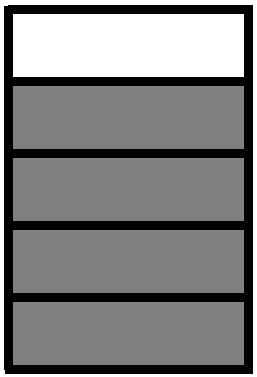


- kein Aufspalten von ***frisch*** aufgespalteten Seiten (**starke Versionsbedingung**)

Behandlung eines Überlaufs

- d = B/5 (B = Kapazität der Seiten)
- **Starke Versionsbedingung** (genau):
  Eine frisch aufgespaltete Seite besitzt zwischen 2d und 4d Datensätzen.

Mindestfüllgrad 

Höchstfüllgrad 

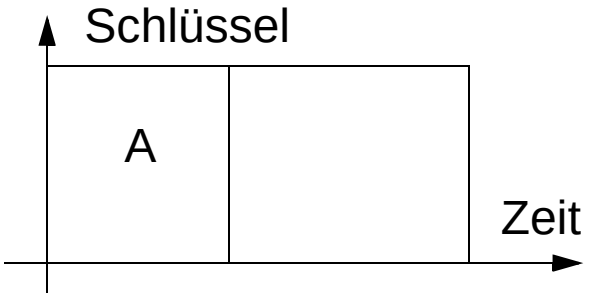
- Zeit-Spalten einer Seite A: verwende aktuellen Zeitpunkt (*now*)
  - #Datensätze > 4d ==> Schlüssel-Spalten
    oder
  - #Datensätze < 2d ==> beseitige diese “unterfüllte” Seite.

- “unterfüllte” Seite: Verschmelzen mit einer Partnerseite
  - #Datensätze > 4d ==> Schlüssel-Spalten.

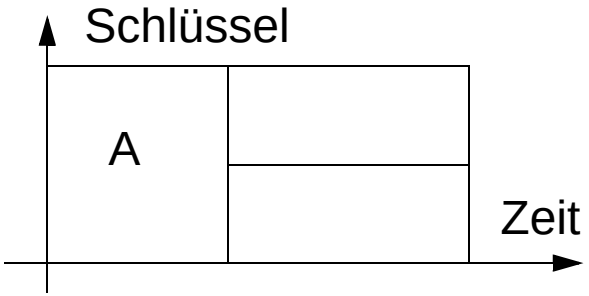
Möglichkeiten der Reorganisation

- Überlauf der Seite A mit L_A ***lebenden*** Einträgen

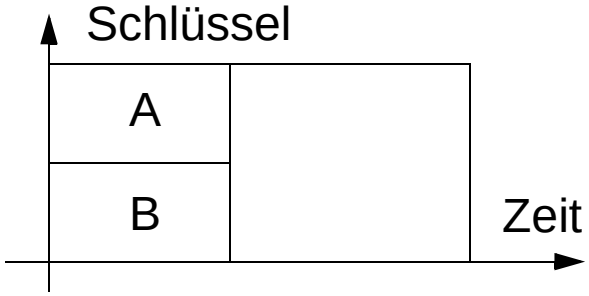
reines Zeit-Spalten: $2d < L_A < 4d$



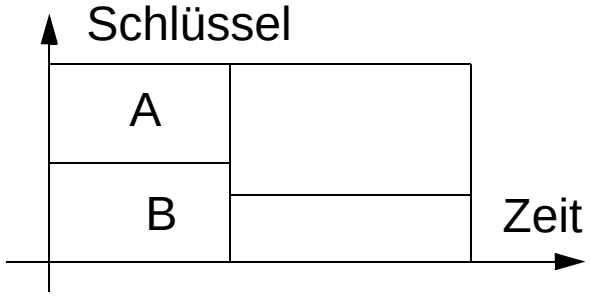
Zeit- & Schlüssel-Spalten: $L_A > 4d$



reines Verschmelzen mit B
$L_A < 2d$, $L_A + L_B < 4d$



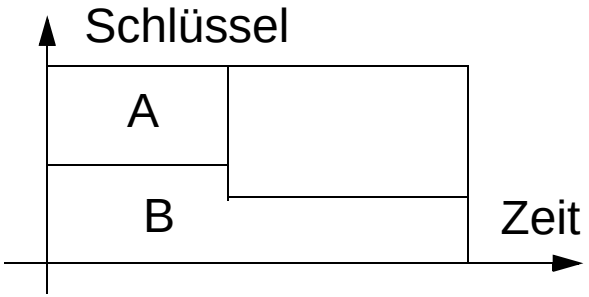
Verschmelzen & Schlüssel-Spalten
$L_A < 2d$, $L_A + L_B \geq 4d$



Vermeiden von Kopieroperationen

- Varman & Verma haben 1996 folgende Verbesserung beim Verschmelzen (3./4. Fall auf der letzten Folie) vorgeschlagen:
  - Falls $d \leq L_A < 2d$ und $L_B \geq 3d$
    Kopiere L_A Einträge aus A und **nur** $2d - L_A$ Einträge aus B in einen neuen Block.

<u>Verschmelzen & modifiziertes Schlüssel-Spalten</u>
$L_A < 2d$, $L_B \geq 3d$



- Ersparnis
  - $L_B + L_A - 2d$ ($\geq 2d$) Einträge aus der Seite B müssen nicht kopiert werden

Erster Algorithmus für Anfragen

- Um eine versionsbasierte exakte Anfrage und eine Schlüssel-Bereichsanfrage zu beantworten wird in zwei Schritten verfahren
  - Suche nach der ***Startwurzel***, die zur gewünschten Version gehört.
  - Tiefendurchlauf des Teilbaums.
- Algorithmus DF(Entry f; key_range R; time_period P);

```
N = GetPage(f);
IF (N is a data page) THEN
    …
ELSE
    FOR EACH entry g in N DO
        compute the key range [low, up) of entry g;
        compute the life-span [t_ins,t_del) of g;
        IF ([low, up) ∩ R ≠ ∅) AND ([t_ins, t_del) ∩ P ≠ ∅)) THEN
            DF(g, R, P);
    END;
END;
```

Worst-Case Leistung des MVBT

- m_i = #Datensätze in der i-ten Version
- #Diskzugriffe für eine exakte Versionsanfrage in Version i (unter der Annahme, dass die ***Startwurzel*** bereits gefunden wurde)

$$2 + \lceil \log_d m_i \rceil$$

- #Diskzugriffe für die i-te Änderungsoperation:

$$2 + 5\lceil \log_d m_i \rceil$$

- Speicherplatzkosten: O(N)

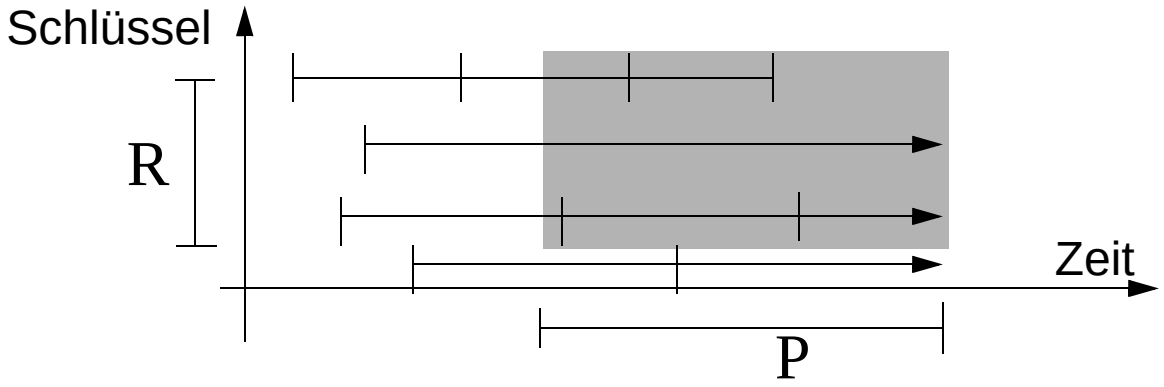
Für B-Bäume ist dieser Ansatz asymptotisch optimal:

- Ein B-Baum zur Verwaltung lediglich einer Version zeigt kein besseres Leistungsverhalten!

5.2.3 Anfragebearbeitung ohne Duplikate

Version-Schlüssel-Bereichsanfragen:

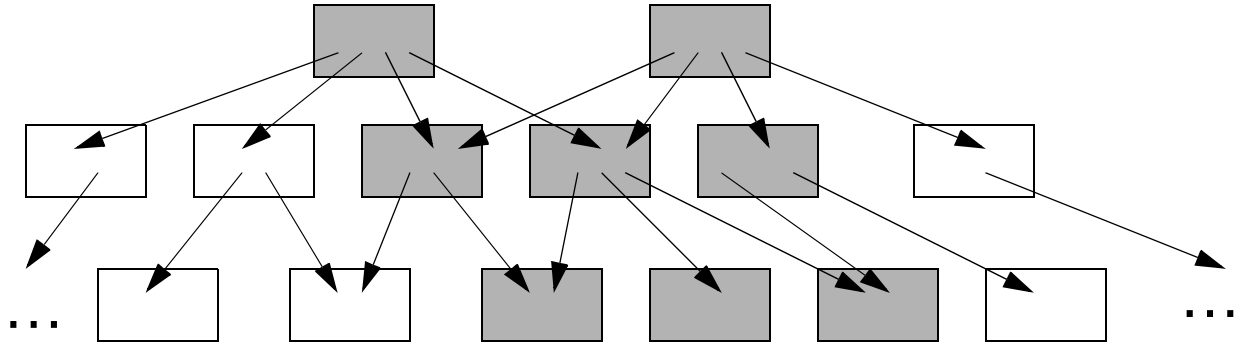
- Finde alle Datensätze mit Schlüssel K im Bereich R zur Zeitperiode P



erfordert Zugriff auf eine

- Menge von Datensätzen
- Menge von Datenseiten

Probleme beim Tiefendurchlauf



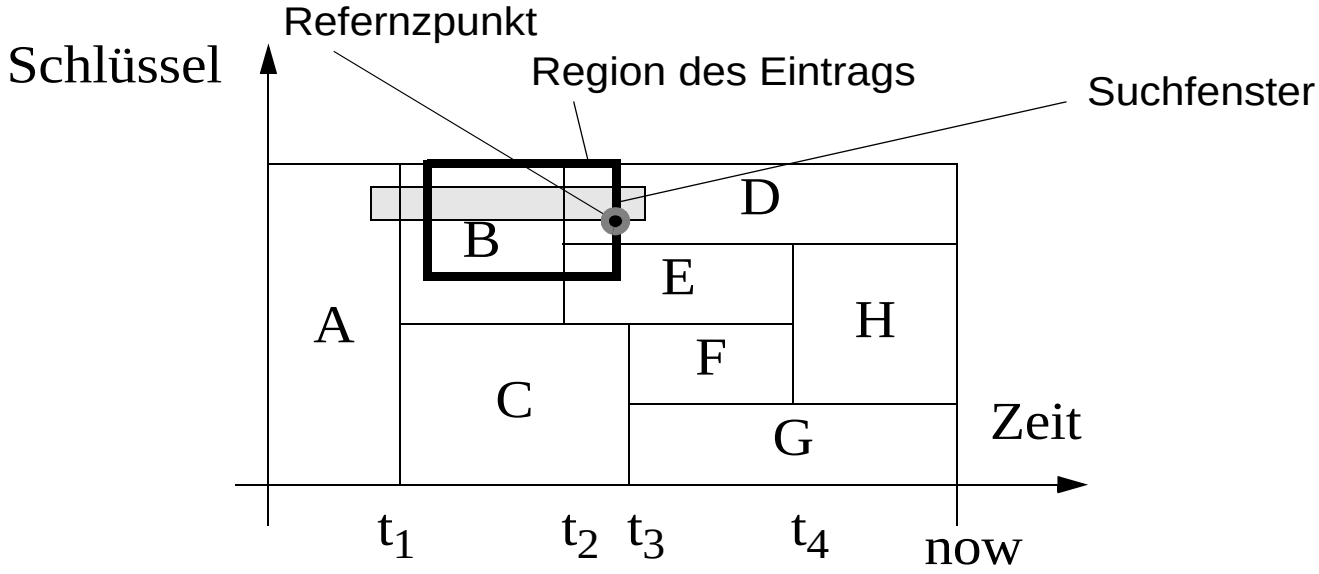
- Duplikate in der Antwortmenge
  - sind oft unerwünscht
- Duplikate im Index
  - führen zu erheblich höheren E/A-Kosten, da Knoten mehrmals besucht werden

Wie kann der Zugriff auf Duplikate vermieden werden?

- Sortieren
- Hashtabelle für Antworten und qualifizierende Indexeinträge

Referenzmethode beim Tiefendurchlauf

- Referenzpunkt eines Eintrags aus der Seite A ist der linke untere Eckpunkt des Schnittrecktecks der **Region des Eintrags** und des Suchfensters
  - Falls der Referenzpunkt in der **Seitenregion von A** liegt, wird der Eintrag akzeptiert und weiterverfolgt in der Tiefensuche.



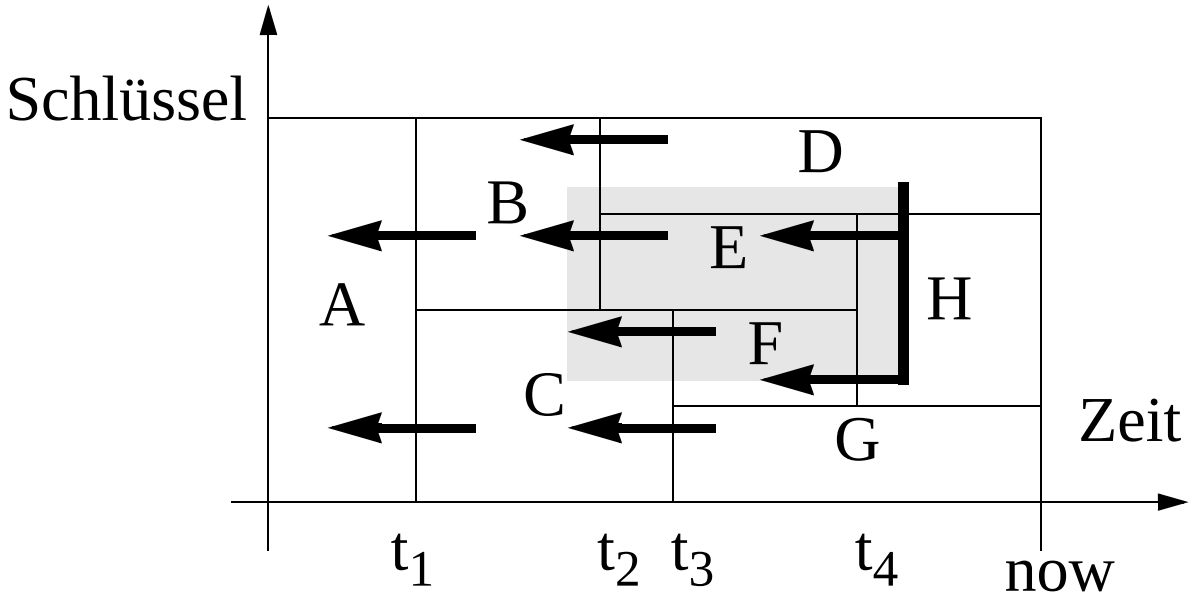
- Vorteil: anwendbar für viele Zugriffsstrukturen
- Nachteil: hoher Aufwand für das Durchsuchen des Index

Link-Methode

Idee:

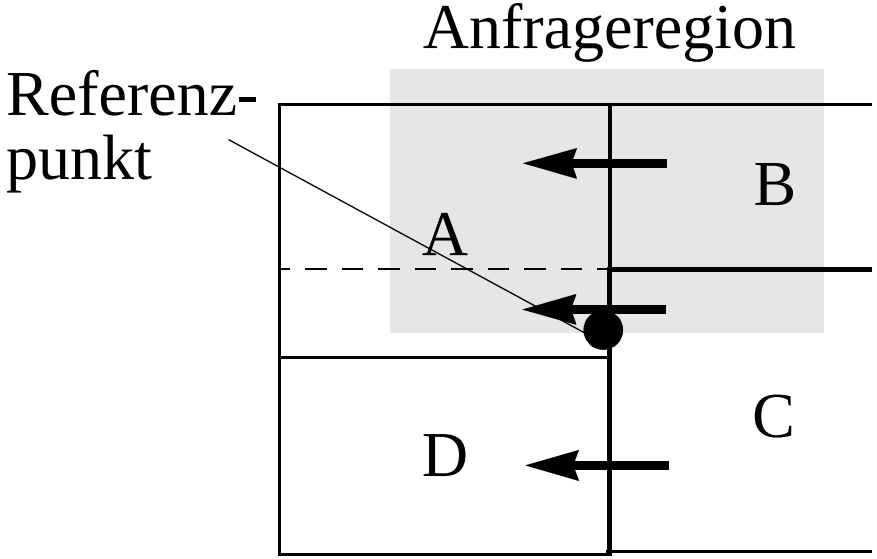
zuerst: Schlüssel-Bereichsanfrage mit dem rechten Rand des Suchbereichs

danach: verfolge Zeiger zu historischen Vorgängerseiten



- wichtige Eigenschaft für B-Baum basierte Verfahren:
  - es gibt höchstens 2 historische Vorgängerseiten

- Zugriff auf Duplikate können vermieden werden



- Aufwand für eine Zeit-Schlüssel-Bereichsanfrage $[r_{low}, r_{up}] \times [v_1, v_2]$
  1. Aufwand für die Schlüssel-Bereichsanfrage:
     $O(log_B\, m_{v1} + a_1 / B)$
  2. Aufwand für das Verfolgen der Links
     $O((v_2 - v_1) / B)$
- Link-Methode nicht für den R-Baum geeignet
  - Überlappung der Seitenregionen
  - keine vollständige Partitionierung

5.3 Indexstrukturen für absolute Zeit

Zielvorgabe

- Entwurf einer Datenstruktur zur Verwaltung von Zeitintervallen im Hauptspeicher mit folgenden Eigenschaften:
  - Speicherplatzbedarf O(n)
  - **Einfügen** und **Löschen** in O(log n)
  - Unterstützung von **Punktsuche** PSuche(*t*)
    - Gegeben einen Zeitpunkt *t*. Finde alle Intervalle *I* mit $t \in I$.
    - Aufwand: O(log n + r), wobei r die Anzahl der Antworten ist.
  - Unterstützung von **Intervallsuche** ISuche(*P*)
    - Gegeben eine Zeitintervall *P*. Finde alle Intervall *I* mit $P \cap I \neq \emptyset$.
    - Aufwand: O(log n + r), wobei r die Anzahl der Antworten ist.
- Entwurf einer **externen** Datenstruktur mit den obengenannten Eigenschaften, wobei
  - log n durch logB n
  - r durch r/B

  zu ersetzen ist.

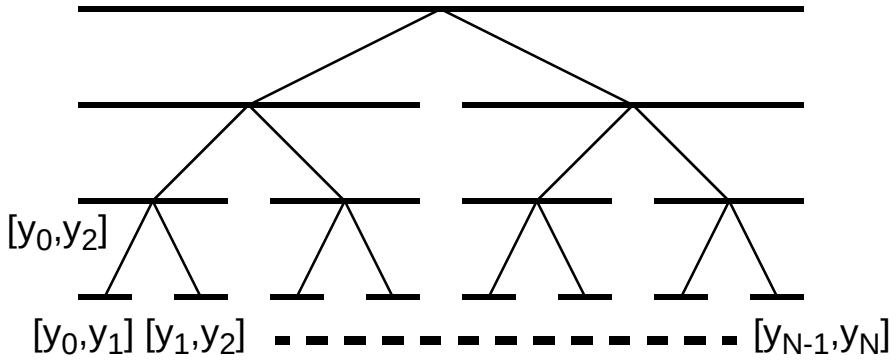
Querbezüge in der Informatik

- Das Problem tritt insbesondere beim Lösungsparadigma “Plane Sweep”im Bereich der Computational Geometry auf.
  - Gegeben eine Menge von Rechtecken im 2-dimensionalen Datenraum. Berechne alle Paare sich schneidender Rechtecke.
  - Sortieren der Daten in x-Richtung und Verwaltung der durch Projektion der Rechtecke auf die y-Achse entshenden Intervalle in einer sogenannten Sweepline-Statusstruktur
  - Sweepline-Statusstruktur ist im wesentlichen eine Datenstruktur zur Verwaltung von Intervallen.

  Durch Plane-Sweep könnnen solche zweidimensionale Problemstelungen auf eindimensionale Mengenprobleme zurückgeführt werden.

5.3.1 Segmentbaum (I)

- Beim Initialisieren der Struktur wird ein vorgefertigtes Skelett eines Baums zur Verfügung gestellt.
  - Man spricht dann auch von einer halbdynamischen Datenstruktur
- Sei $y_0,\ldots y_N$ eine Menge von Punkten im 1-dimensionalen Datenraum mit $y_{i-1} < y_i$ für $1 \leq i \leq N$. Dann kann über diese Menge ein binärer Baum mit minimaler Höhe erzeugt werden, so daß ein Intervall $[y_{i-1},y_i]$ einem Blatt zugeordnet ist.

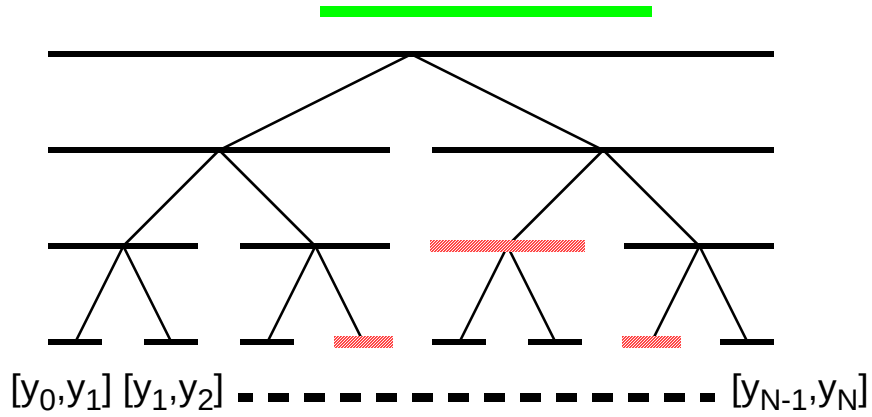


- Einem inneren Knoten des Baums ist jenes Intervall (des Datenraums) zugeordnet, welches sich aus der Vereinigung der Intervalle seiner Söhne ergibt
- Jedem Knoten wird eine Liste von Intervallen (Datenintervallen) zugeordnet.

Segmentbaum (II)

Zuordnung der Datenintervalle

- Ein Datenintervall I = [l,u] wird in der Liste eines Knotens abgelegt, falls I das zugehörige Intervall des Knotens überdeckt, jedoch das Intervall des Vaterknotens **nicht** überdeckt.



PSuche(*t*) im Baum

- Bestimme den Pfad von der Wurzel zu dem Blatt, so daß das zugeordnete Intervall des Knotens stets *t* enthält. Die zugehörigen Knotenlisten enthalten genau die Antwortmenge der Anfrage.
- Aufwand: O(log N + r), wenn r die Größe der Antwortmenge ist

Segmentbaum (III)

Aufwandsanalyse

- Aufbau des leeren Baums:
  - O(N) Zeit und O(N) Speicherplatz
- Einfügen in den Baum:
  - Maximal zwei Knoten auf jeder Stufe des Baums sind durch das Einfügen betroffen
  - Einfügen in die Liste benötigt O(1) Zeit
    - $\Rightarrow$ O(log N) Zeit
    - $\Rightarrow$ Gesamtspeicherplatzbedarf: O(N + N log N)
- Löschen aus dem Baum:
  - analog zu Einfügen
    - $\Rightarrow$ O(log N) Zeit
- Punktanfrage:
    - O(log N + r) Zeit

5.3.2 Intervallbaum

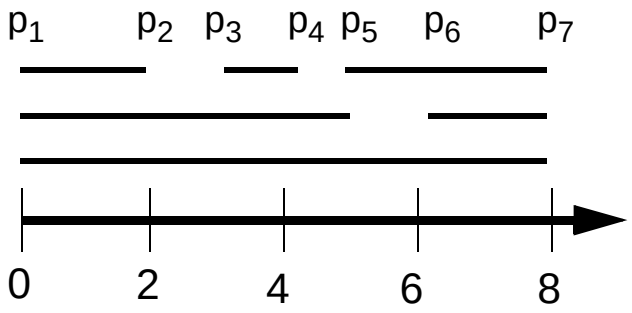
Motivation

- Gibt es eine Datenstruktur mit linearem Speicheraufwand, die anstelle des Segmentbaums beim **Punkteinschlußproblem** verwendet werden kann?

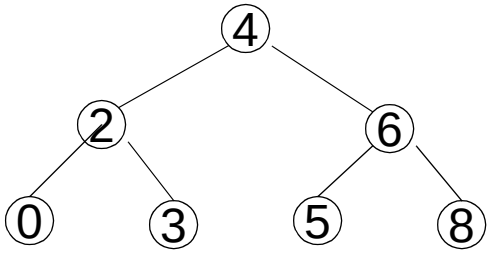
Aufbau des Skeletts eines Baums

- Sei $K_1, K_2, \ldots, K_n$ eine Menge von Kanten mit $K_i = [\mathbf{u_i,o_i}]$, $1 \leq i \leq n$.
- Sei $P = \{ p_1, \ldots, p_s \}$ die Menge der Punkte, die Eckpunkt von zumindest einer Kante sind, $s \leq 2n$. Sortiere P, so daß $p_i < p_{i+1}$ gilt.
- Erzeuge einen binären Suchbaum mit minimaler Höhe über die Punkte $p_1,\ldots,p_s$

Menge der Intervalle



Skelett eines binären Baums



Einfügen

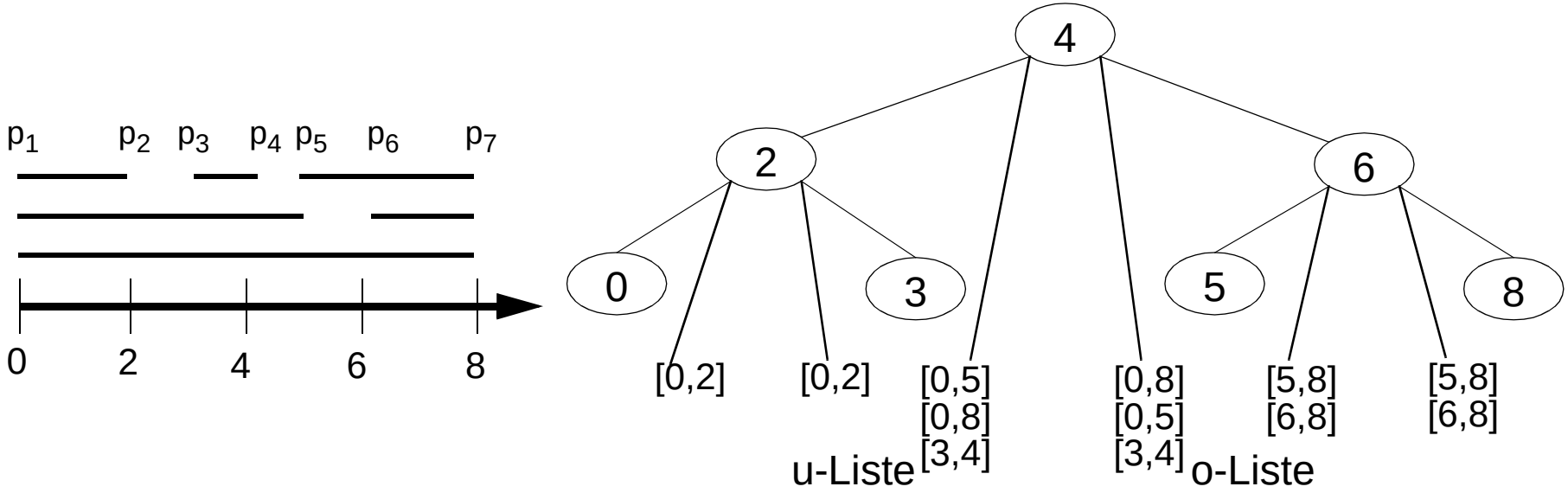
Einfügen

- Jeder Knoten Node besitzt einen Punkt (**Node.p**) und eine Menge von Kanten $\{K_1,\ldots,K_m\}$, wobei jede Kante den Punkt Node.p enthalten.
- Die Menge $\{K_1,\ldots,K_m\}$ wird durch zwei sortierte Listen implementiert:
  - **u-Liste**: alle Kanten sind nach den u-Werten aufsteigend sortiert
  - **o-Liste**: alle Kanten sind nach den o-Werten absteigend sortiert
- Eine Kante K wird in die beiden Listen des Knotens eingetragen, dessen Wert beim Durchlauf von der Wurzel beginnend als erster in der Kante enthalten ist:
- *Algorithmus Insert (K, Node) //* erster Aufruf erfolgt mit Node = Wurzel)

```
IF K.u ≤ Node.p ≤ K.o THEN
    InsertIntoList (u_list, K.u);
    InsertIntoList (o_list, K.o);
ELSIF K.o < Node.p THEN
    Insert (K, Node.left)
ELSE
    Insert (K, Node.right)
```

Aufwand für das Einfügen

Beispiel für das Einfügen



Aufwand für das Einfügen

- Verwende zur Organisation der Listen balancierte binäre Bäume (AVL-Baum)
  - $\Rightarrow$ Einfügen in die o- und u-Liste benötigt O(log n) Zeit
- Erreichen des Knotens im Intervallbaum: O(log n) Zeit
- Gesamtbedarf:O(log n) Zeit und O(n) Speicherplatzbedarf
- Löschen wird analog zum Einfügen vorgenommen

Punktsuche im Intervallbaum (IV)

Punktsuche PSuche(*t*)

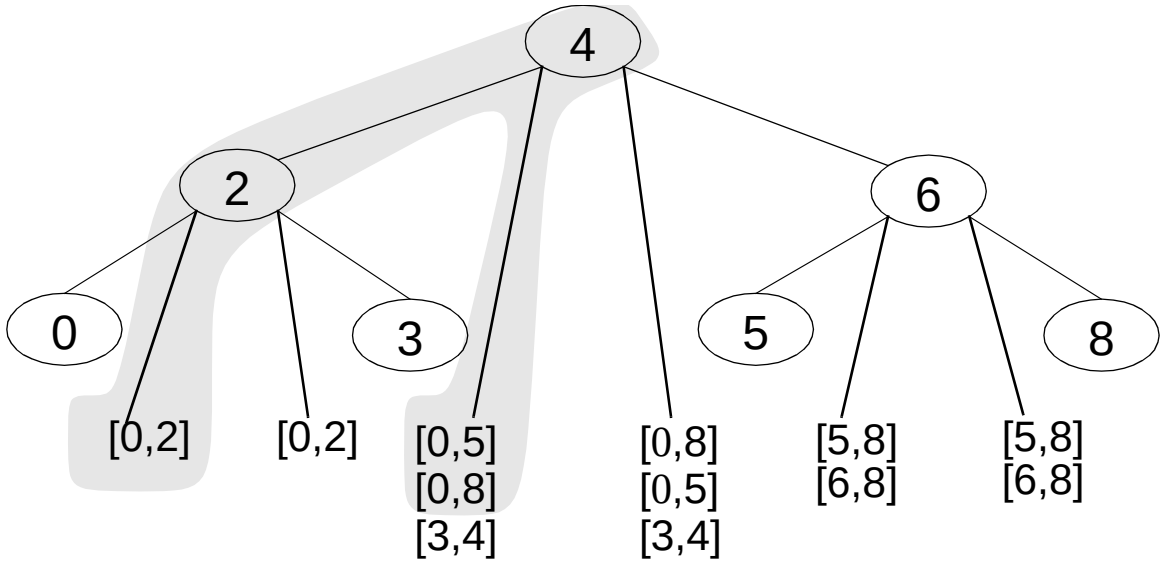
Algorithmus PointQuery (Node, t)

IF *t* = Node.p THEN
    Gib alle Intervalle der u-Liste aus;
ELSIF *t* < Node.p THEN
    Gib sequentiell den Anfang der u-Liste aus, bis ein Intervall rechts von *t* liegt;
    PointQuery (Node.left, *t*);
ELSE
    Gib sequentiell den Anfang der o-Liste aus, bis ein Intervall links von *t* liegt;
    PointQuery (Node.right, *t*);

Aufwand für Punktsuche

Beispiel für Punktsuche mit $t = 2$

Suche alle Intervalle, die Punkt 2 enthalten.



Aufwand für das Suchen

- Jede Antwort wird nur einmal gelesen.
- Für jeden Knoten des Intervallbaums wird maximal auf ein Intervall zugegriffen, daß nicht Antwort ist
- Die Anfrage ist auf einen Pfad des Baums beschränkt

$\Rightarrow$ Gesamtaufwand: O(log n + r) (r = Größe der Antwortmenge)

5.3.3 Prioritäts-Suchbaum

Entwurf einer effizienten (halbdynamischen) Datenstruktur

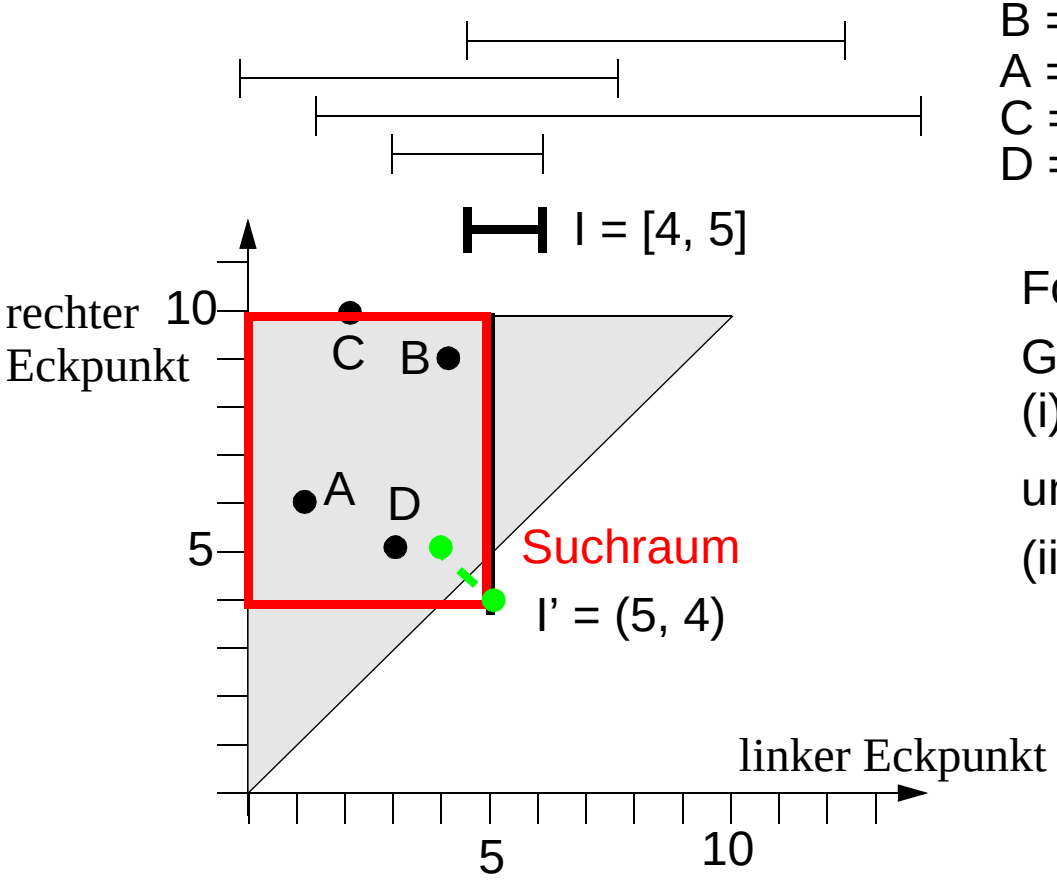
- Verwaltung einer Menge M von Intervallen (maximale Anzahl: n)
  - Speicherplatz: O(n)
- Einfügen und Löschen von Intervallen
  - Zeitaufwand: O(log n)
- Effiziente Unterstützung von Intervallsuchen.
  - Zeitaufwand: O(log n + r), wobei r die Anzahl der Antworten ist.

Diese Anforderungen werden durch den von McCreight 1985 vorgestellten Prioritäts-Suchbaum (Priority Search Tree) erfüllt. Wesentliche Ideen dabei sind:

- Darstellung eines Intervalls I = [x, y] als Punkt (x,y) in einem zweidimensionalen Datenraum
- Kombination von Suchbaum (bzgl. y-Achse) und Heap (bzgl. der x-Achse) in einer Datenstruktur.

Transformation der Intervalle

Intervall I = [x,y] als zweidimensionaler Punkt P = (x, y) dargestellt.



B = [4, 9]
A = [1, 6]
C = [2, 10]
D = [3, 5]

Formulierung einer Anfrage:

Gesucht sind alle Punkte P mit

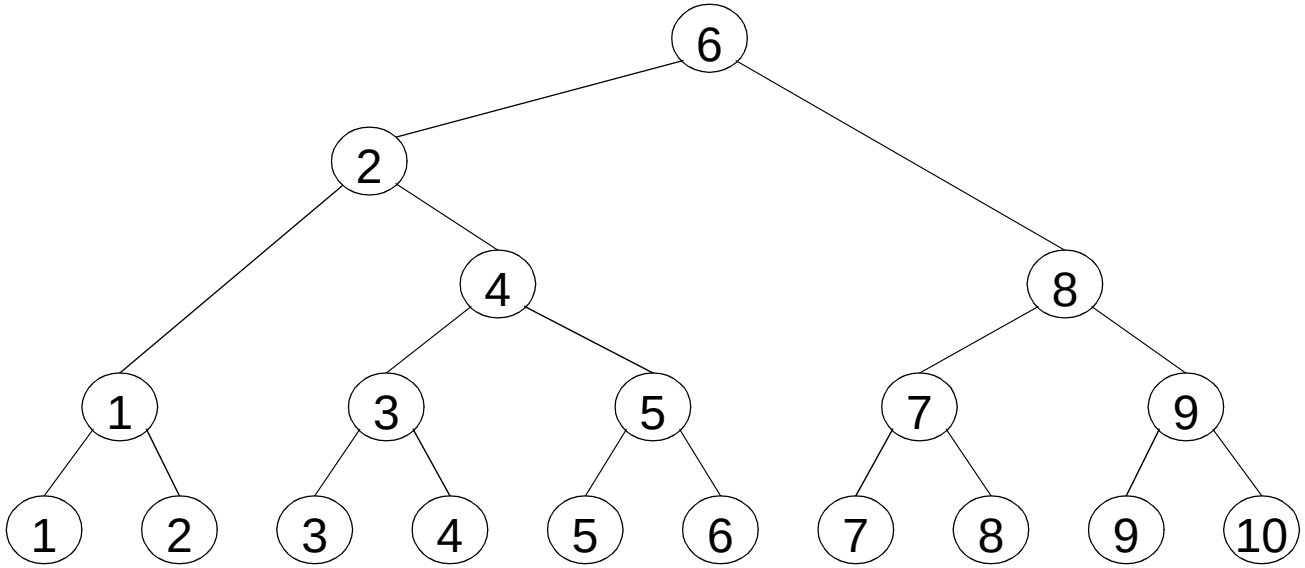
(i) $y_{low} \leq P.y \leq y_{high}$

und

(ii) $P.x \leq x_{low}$

Das Skelett

- Ein Prioritäts-Suchbaum ist ein **Blattsuchbaum** bzgl. der y-Achse und ein **Min-Heap** bzgl. der x-Achse.
- Entsprechend zu dem Intervallbaum wird ein Skelett über der Menge aller von den Intervallen angenommenen y-Werte aufgebaut.
  - jeder Wert wird einem Blatt zugeordnet
  - der Wert eines inneren Knotens (**Splitwert**) ergibt sich aus dem Maximum des linken Teilbaums



Einfügen

Vereinfachende Annahme (o.B.d.A.):

- Jeder y-Wert des Datenraums wird höchstens einmal angenommen.
  - Nutze lexikographische Ordnung aus, wobei die x-Koordinate niedrigere Priorität als die y-Koordinate besitzt.

Eigenschaft der Datenstruktur:

- Jedem Knoten des Prioritäts-Suchbaum wird neben dem Splitwert höchstens ein zwei-dimensionaler Punkt zugeordnet.
- Nach dem Aufbau des Skeletts ist noch keinem Knoten ein Punkt zugeordnet.
- Danach werden die Punkte in die Struktur eingefügt, wobei
  - Der erste Punkt stets in der Wurzel abgelegt wird.
  - Beim Einfügen weiterer Punkte ist darauf zu achten, daß der Prioritäts-Suchbaum seine Heap-Eigenschaft bewahrt .

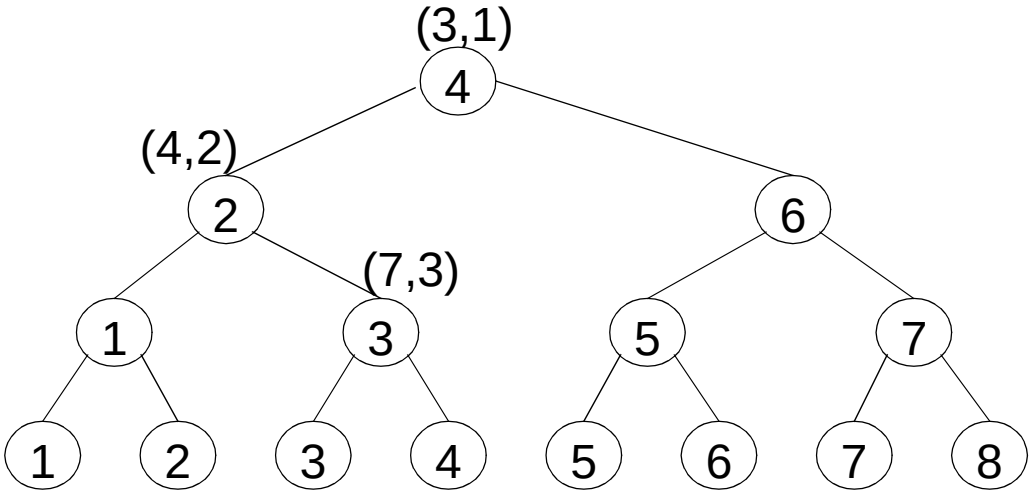
Algorithmus

Algorithmus InsertPST (Node, point)

```
// point besteht aus zwei Komponenten x und y
// Node besteht aus vier Komponenten: Punkt p, Splitwert split
// dem linken und rechten Teilbaum (left, right)
IF (Node.p is not defined) THEN
    Node.p := point
ELSE
    IF Node.x <= point.x THEN
        // Point läuft in einen der Teilbäume hinein
        IF (Node.split >= point.y) // Suchbaumkriterium
            InsertPST(Node.left, point)
        ELSE
            InsertPST(Node.right, point)
    ELSE
        tmp := Node.p;
        Node.p := point;
        InsertPST(Node, tmp)
```

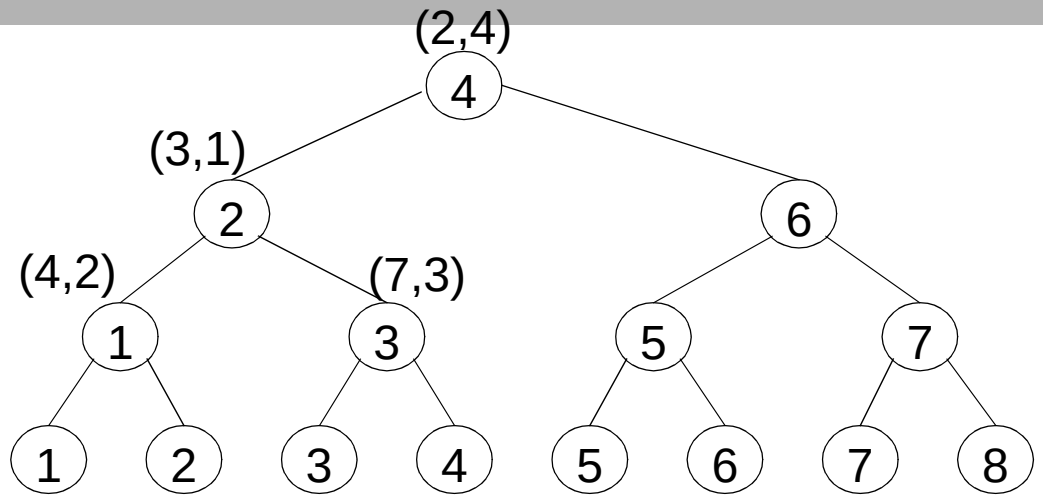
Beispiel (I)

- Punktmenge: {(3,1), (4,2), (7,3), (2,4), (1,5), (6,6), (5,7), (4,8)}
  - bezieht sich jetzt **nicht** auf Punkte, die aus der Transformation von Intervallen gewonnen wurden.
- Datenraum der y-Werte: {1,2,…,8}
- nach dem Einfügen der ersten drei Punkte:

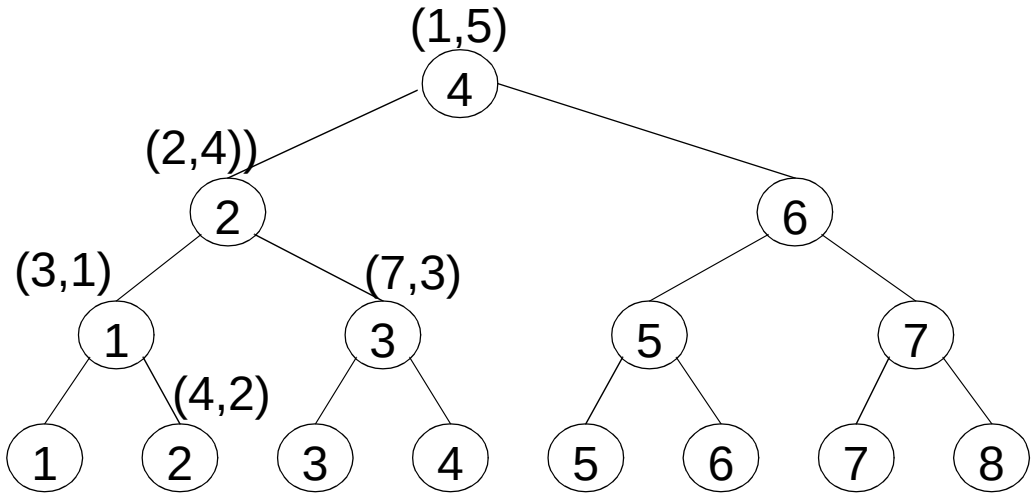


- Beim Einfügen von **(2,4)** ist die Prioritätseigenschaft bereits an der Wurzel verletzt.
  - (2,4) wird an der Wurzel abgespeichert
  - Einfügen wird mit Punkt **(3,1)** fortgesetzt

Beispiel (II)

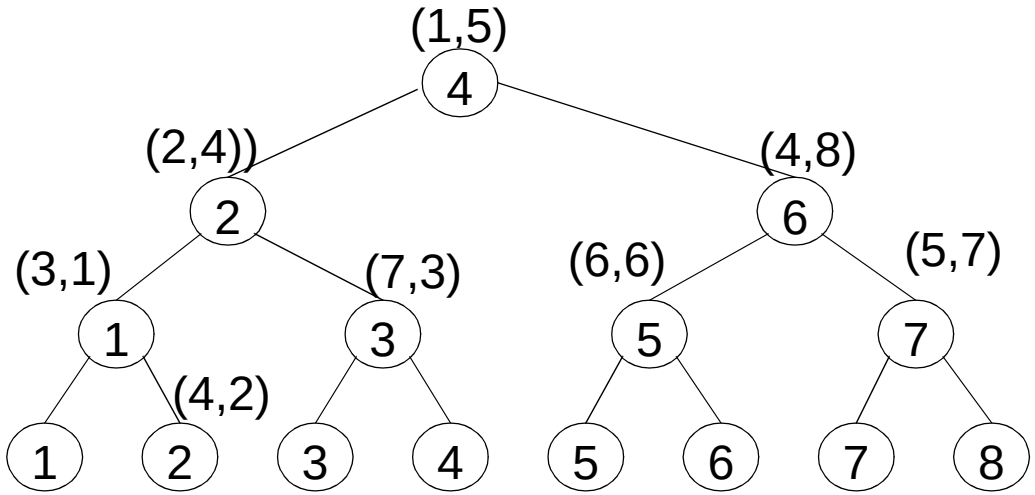


- nach dem Einfügen des Punkts (1,5)



Beispiel (III)

Nach dem Einfügen aller Punkte ergibt sich folgender Prioritäts-Suchbaum



Löschen

Zunächst wird beim Löschen stets eine Suche durchgeführt, wobei hierzu nur der y-Wert (und die Splitwerte der internen Knoten) benutzt wird.

Sei N der Knoten mit dem zu löschenden Punkt. Es werden folgende Fälle betrachtet:

- Falls nur N.left oder N.right einen Punkt besitzt, wird der entsprechende Punkt nach N "hochgezogen".
- Falls beide Kinder Punkte besitzen, wird derjenige Punkt mit dem kleineren x-Wert hochgezogen.

Das Löschen setzt sich dann in dem entsprechenden Zweig fort.

Beispiel:

Betrachten wir das Löschen des Punkts **(1,5)** in unserem Beispiel.

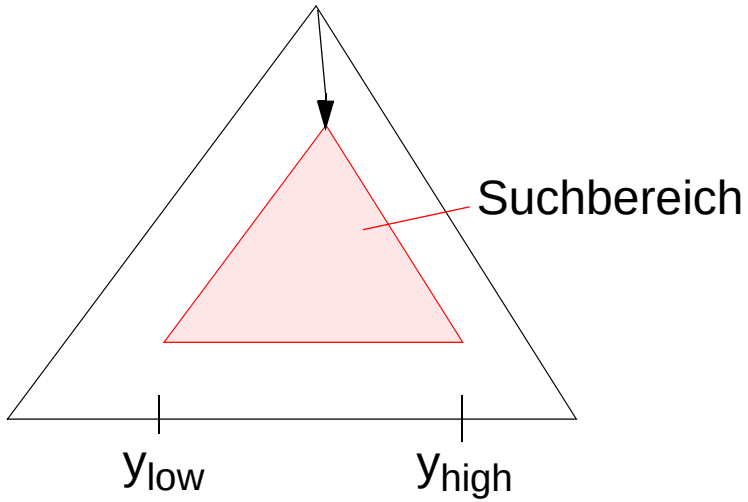
Aufwand beim Einfügen und Löschen

- O(log n), da diese Operationen auf einen Pfad beschränkt sind.

Suchen

Gesucht sind alle Punkte mit y-Komponente in einem vorgegebenen Bereich $[y_{low}, y_{high}]$ und mit Priorität $\leq x_{low}$.

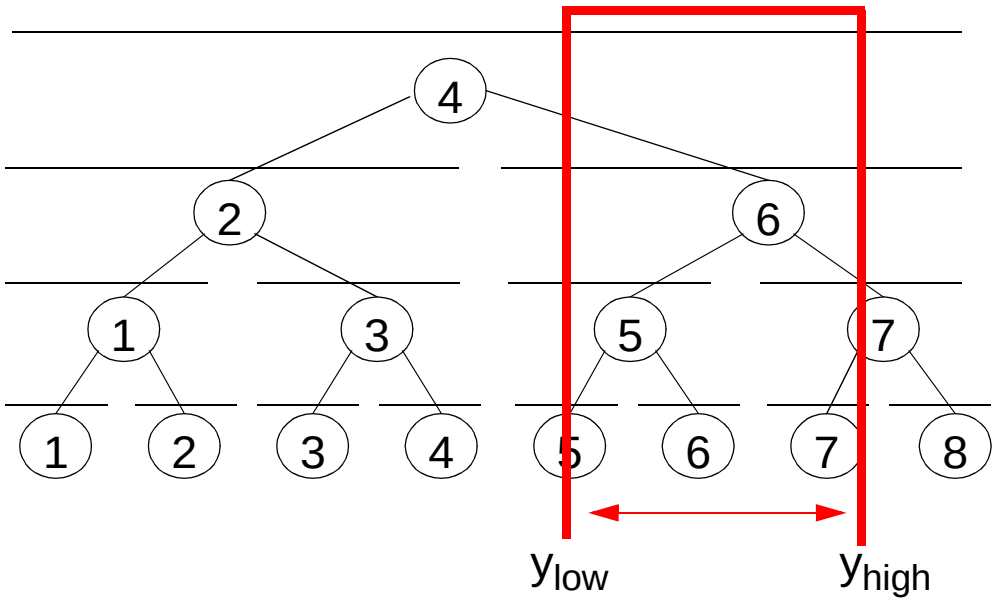
- Suche mit dem Bereich $[y_{low}, y_{high}]$ zunächst die Wurzel des (kleinsten) Teilbaums, in dem alle Antworten liegen. Es wird dabei nur die Suchbaumeigenschaft des Prioritäts-Suchbaums genutzt.
- Dieser Teilbaum wird mittels eines Tiefendurchlaufs rekursiv durchlaufen,wobei die Rekursion bei einem Knoten mit Priorität > x_{low} **nicht** fortgesetzt wird..



Aufwand beim Suchen

Wichtige Beobachtungen

- Jedem Knoten eines Prioritäts-Suchbaums kann ein Intervall (Potential) möglicher y-Werte zugeordnet werden.
- Bei einer Suchanfrage mit dem Bereich $[y_{low}, y_{high}]$ sind Antworten nur in den Knoten, deren Potential sich mit dem Suchbereich schneidet.
  - Relevante Knoten sind rechts vomm Suchpfad von y_{low} und links vom Suchpfad von y_{high}.



Aufwand beim Suchen (II)

- Ein Knoten mit einem zu hoch priorisierten Punkt wird nicht mehr betrachtet.
  - Offensichtlich brauchen dann auch keine Kinder dieses Knoten untersucht werden.
  - Zu jedem Knoten, dessen Punkt sich als Antwort qualifiziert, müssen höchstens zwei nicht qualifizierende Knoten betrachtet werden.
  - Besuchte Knoten, die keine Antwort liefern, liegen am Rand des von der Anfrage zu durchlaufenen Teilbaums oder am Ende eines Pfads des Teilbaums.

Es folgt damit folgendes Resultat

- Eine Suchanfrage in einem Prioritäts-Suchbaum wird mit O(log n + k) Aufwand beantwortet.

Volldynamische Prioritätssuchbäume

Idee:

- Beim Einfügen eines Punktes (x,y) wird im ersten Schritt die Struktur des Suchbaums um das Blatt mit dem Wert y and der folgenden Stelle erweitert.

```
Node = root;
WHILE (Node is not a leaf) DO
    IF (Node.split >= point.y) // Suchbaumkriterium
        Node = Node.lef
    ELSE
        Node = Node.right
```

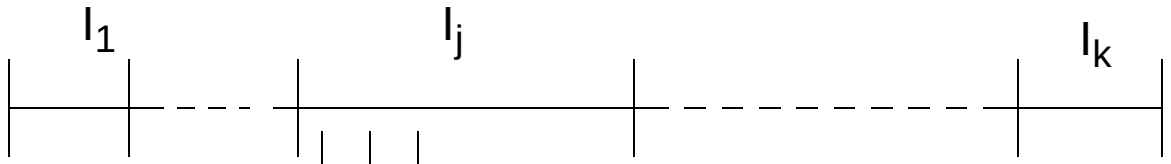
- Danach wird der Punkt analog zum halbdynamischen Verfahren in die Struktur eingefügt.
- Beispiel:
  - (4,6), (3,7), (2,2), (6,4), (5,1), (9,3), (1,5)
- Im Mittel ist die Höhe des Baumes logarithmisch in der Anzahl der Datensätze.

5.4 Worst-case optimale Verfahren

- Arge & Vitter haben 1996 eine worst-case optimales externes Verfahren für die Verwaltung von Intervallen vorgestellt.
  - Unterstützung von Punktanfragen in $O(\log_B n + r/B)$
  - Einfügen und Löschen in $O(\log_B n)$
  - linearer Speicherplatz
- Verfahren ist mehr von theoretischen Interesse und zumindest in seiner ursprünglichen Form nicht sehr praktikabel
  - basiert auf dem Konzept gewicht-balancierter B-Bäume:
    Anzahl der Daten in benachbarten Teilbäumen ist asymtotisch gleich (unabhängig von der Höhe)
  - Verwendung des $\sqrt{B}$-Tricks: Fan-out von $\sqrt{B}$ verschlechtert nicht die asymptotische Laufzeit
  - Speziallösung für die Verwaltung von k Intervallen mit $k \leq B^2$.

Verzweigungsgrad

- Entsprechend zum Intervallbaum kan man sich die wesentliche am besten an der halbdynamischen Variante illustrieren.
- Jedem Knoten werden $\sqrt{B}$ Punkte zugeordnet.
  - Ein Intervall wird in dem höchsten Knoten abgespeichert, der das Intervall mit seinen Punkten aufspiest.
  - Jedes Intervall wird 3x repräsentiert.



Liste der Intervalle, die in Ij enden.

Liste der Intervalle, die in Ij anfangen.

(k-j) Listen L_1, …, L_{k-j} von Intervallen, wobei Liste L_m die Intervalle enthält, die in I_j anfangen und in I_{j+m} enden.

5.5 Anwendung

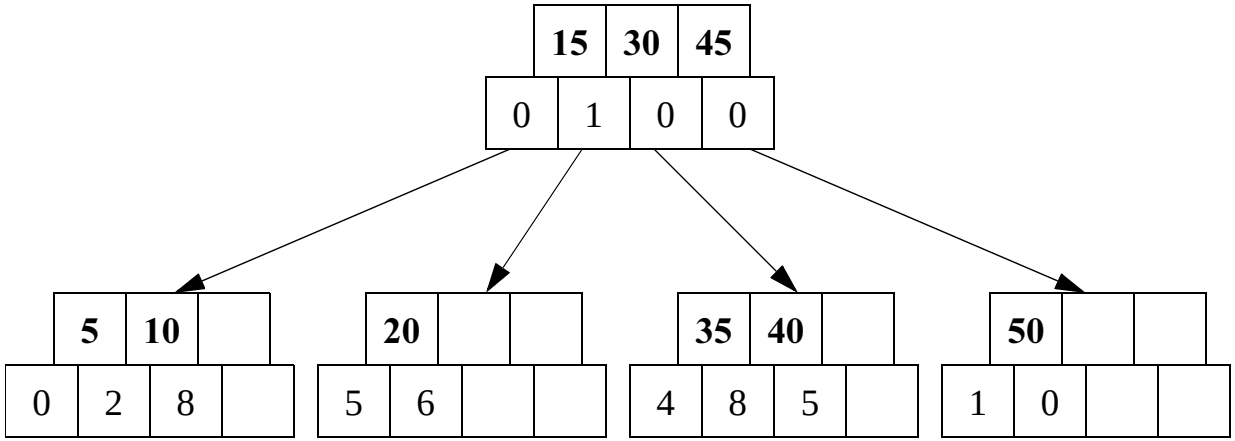
- SB-Baum (Yang & Widom, 2001) ist eine Mischform eines Segmentbaums und eines B+-Baums zur Verwaltung von zeitabhängigen Aggregaten
- Problemstellung
  Patienten wird pro Tag über einen Zeitraums eine gewisse Dosis von Medikamenten verabreichtet.
  - Berechne die Gesamtdosis von Libobay am 1.1.1999.
  - Berechne die Dosis für Libobay für jeden Monat des Jahres 1999.
- SB-Baum
  - Einfügen und Löschen von Intervallen, die mit einem Aggregat assoziiert sind, mit Aufwand $O(\log_B n)$.
  - Effiziente Unterstützung von Punktanfrage: Gib mir den Wert des Aggregats zum Zeitpunkt t
- Im Folgenden betrachten wir als Aggregat die Summe
  - Aggregate wie Durschnittswert und Varianz lassen sich mit konstanten Mehraufwand (Speicher) entsprechend verwalten.
  - Bei Min/Max können nicht efektiv Löschoperationen unterstütz werden.

SB-Baum

- Ein interner Knoten N besitzt mindestens B/2 (außer der Wurzel) und bis zu B zusammenhängende Intervalle. Jedes Intervall ist assoziiert mit einem Aggregat und einem Zeiger auf einen Teilbaum.
  - Die Zeitintervalle in einem Teilbaum müssen alle echt kleiner sein als ´die Intervallgrenze des entsprechenden Eintrags im Elternknoten.
- Ein Blatt entspricht einem internen Knoten, wobei die Einträge keinen Verweis auf einen Teilbaum besitzen.

Beipiel

Name	Dosierung	Periode
Alfred	2	[10,40)
Biance	3	[10,30)
Carnen	1	[20,40)
Daniel	2	[5,15)
Edgar	4	[35,45)
Florian	1	[10,50)



Punktsuche, Einfügen und Löschen

- Berechnen der Summe zum Zeitpunkt t
- Algorithmus Summe(Node N, Time t)

  Sei I_j das Intervall aus N, das t beinhaltet und vj das assoziierte Aggregat.

  Falls N ein Blatt,

  return v_j

  Ansonsten (c_j ist der assozierte Verweis),

  return v_j + Summe(N.c_j, t)

Einfügen

- Beim Einfügen eines Intervalls müssen die Aggregate angepasst werden. Es muß nicht notwendigerweise die Baumstruktur verändert werden.
- Algorithmus Insert(Node N, TimePeriod p, Value v)

  $I_j \subseteq p$: vj += v

  $I_j \cap p \neq \emptyset$ und N ist kein Blatt: Insert(N.c_j, p, v)

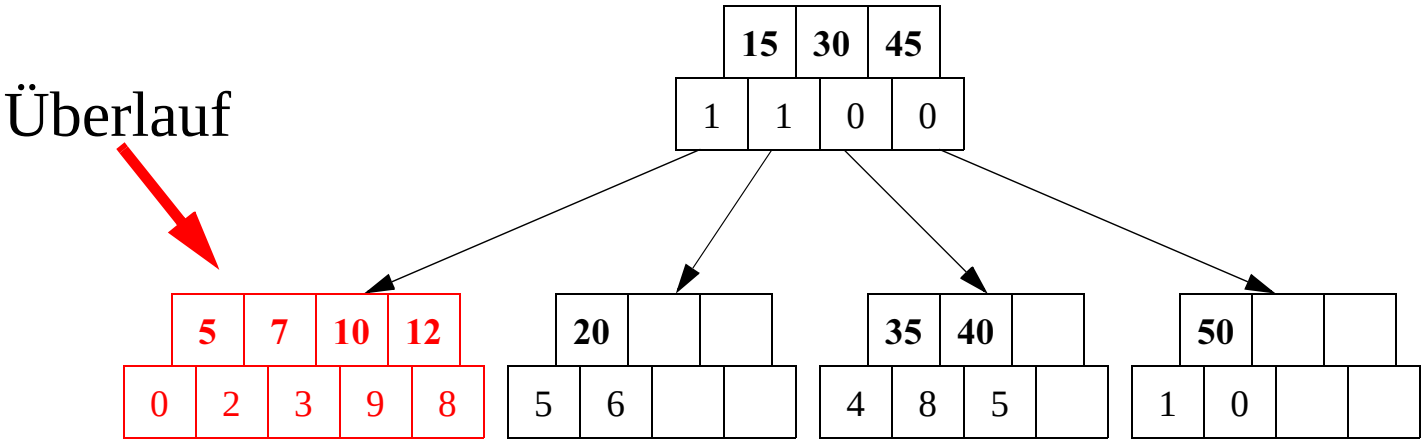
  $I_j \cap p \neq \emptyset$ und N ist ein Blatt: Passe die Struktur des B+-Baums an.

Löschen

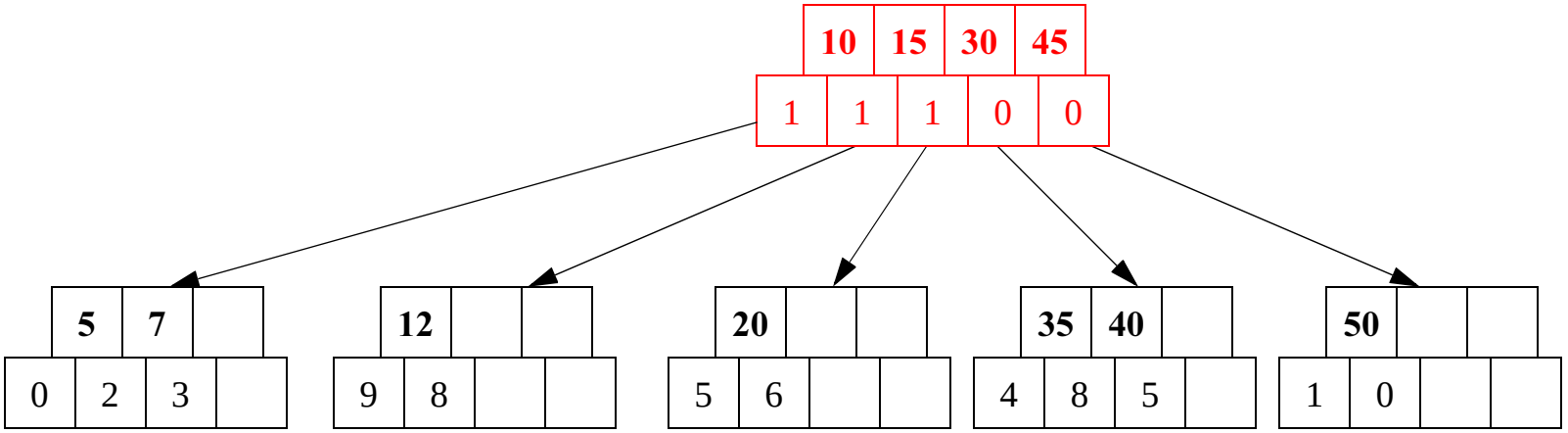
- Löschen wird durch ein Insert implementiert, wobei der Wert in negierter Form verwendet wird.

Strukturelle Änderungen im Baum

- Aufspalten eines Knotens: ähnlich wie in einem B+-Baum, aber Aggregate müssen noch zusätzlich angepasst werden.
- Beispiel:
  - Insert ([5,15), 1) und dann Insert([7,12), 1)



– Beseitigung des Überlaufs



Verschmelzen

- Blätter
  - Verschmelzen von zwei benachbarten Intervalle mit gleichem Aggregat
- Interne Knoten
  - Anpassung der Replikate