ALGORITHMISCHE ANWENDUNGEN

WINTERSEMESTER 2006/2007

Mehrdimensionale Bäume (*k*-d Trees)

Team: B_blau_Ala0607
Wilhelm Faber 11032935
Ioannis Chouklis 11042438

Inhaltsverzeichnis

1. Der *k*-d Baum

Der k-d-Baum ist ein *k*-dimensionaler Suchbaum. Der *k*-d-Baum kann Punkte in einem Raum beliebiger Dimension speichern, hier wird allerdings nur der Raum R^2 betrachtet. Ein balancierter *k*-d-Baum für n Punkte im R^2 benötigt O(n) viel Speicherplatz. Eine Achsenparallele orthogonale Bereichsanfrage lässt sich in einer Zeit von O(Wurzel(n) + a) beantworten, wobei a die Anzahl der im Anfragerechteck enthaltenen Punkte beschreibt. Es handelt sich um eine statische Struktur; die Operationen Löschen und die Durchführung einer Balancierung sind sehr aufwendig.

1.1. Inhomogene Variante

Die Idee des *k*-d-Baumes ist, die in den inneren Knoten des Binärbaums vorgenommene Teilung der Punktmenge anhand einer Ordnungsrelation durch eine räumliche Teilung zu ersetzen. Abbildung 1.1 veranschaulicht die sukzessive Teilung der Ebene durch Splitgeraden, welche abwechselnd senkrecht und waagrecht erfolgt.

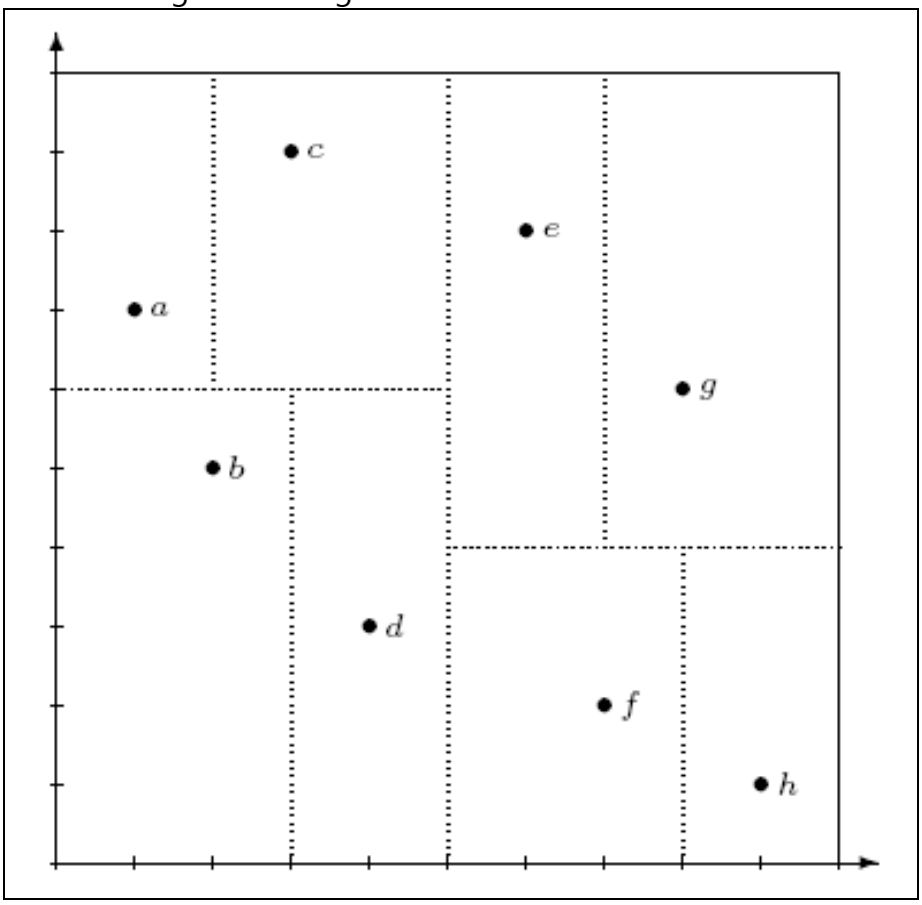


Abbildung 1.1: *k*-d-Baum - Räumliche Unterteilung durch Splitgeraden

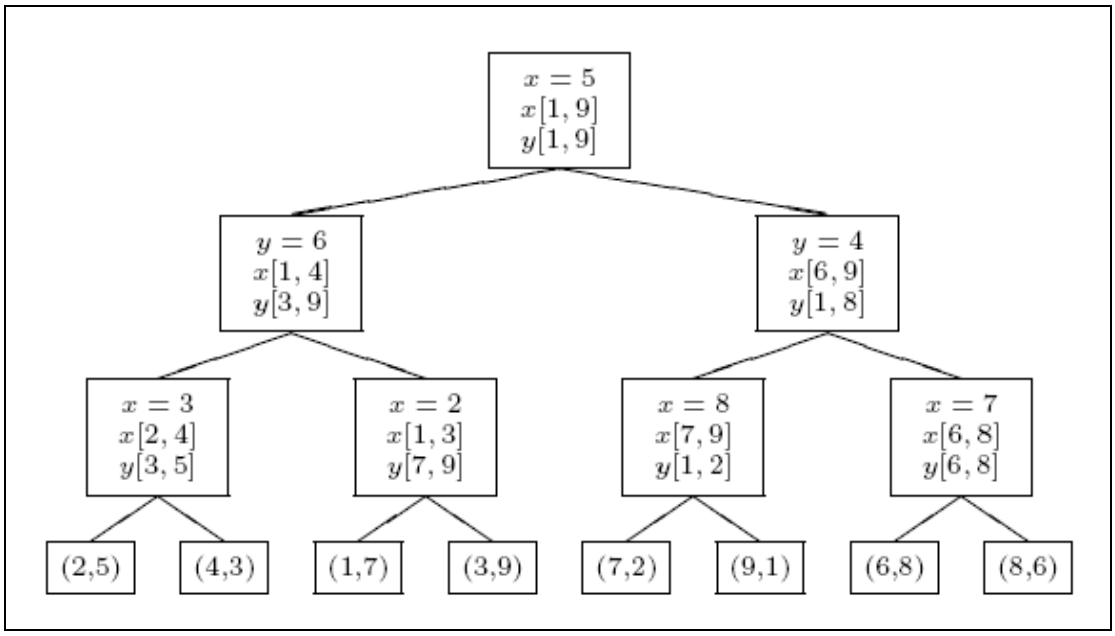


Abbildung 1.2: Der aus Abbildung 1.1 resultierende *k*-d-Baum

Es ist darauf zu achten, dass der *k*-d-Baum stets balanciert aufgebaut wird, d.h. dass sich in jedem inneren Knoten die Anzahl der Blätter seiner Teilbäume um maximal eins unterscheidet. Die inneren Knoten des *k*-d-Baumes enthalten die Splitkoordinate (x oder y) und den Splitwert als Wegweiser für die Suche eines einzelnen Punktes. Zusätzlich wird in den inneren Knoten der von den im Teilbaum enthaltenen Punkten eingenommene Bereich gespeichert, welcher als Wegweiser für Bereichsanfragen dient.

1.2 Homogene Variante

Probleme gibt es jedoch, wenn Punkte identische Koordinaten aufweisen, denn der Wert einer Splitgerade darf nicht als Koordinate eines Punktes vorkommen. Um in dieser Situation dennoch einen balancierten Baum aufbauen zu können, muss der *k*-d-Baum erweitert werden. Sollten Punkte auf einer Splitgerade zu liegen kommen, so werden diese im zugehörigen inneren Knoten gespeichert. Der zuvor binäre Baum wird dadurch ternär. Ein erweiterter *k*-d-Baum wird als balanciert bezeichnet, wenn für jeden inneren Knoten die Teilbäume seiner äußeren Kindknoten jeweils höchstens halb so viele Punkte enthalten wie der gesamte Teilbaum dieses inneren Knotens.

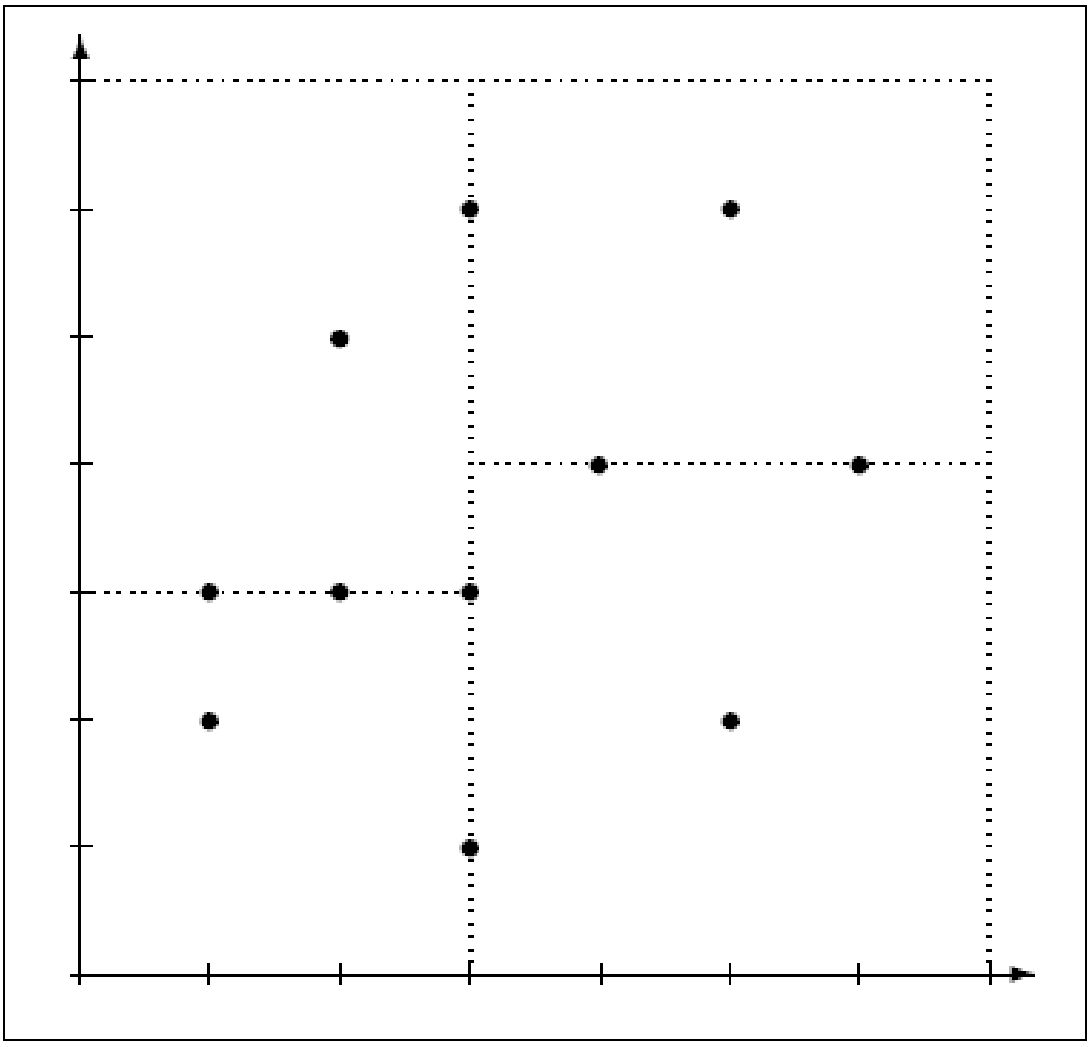


Abbildung 1.3: *k*-d-Baum - Punktmenge mit identischen Koordinatenwerten

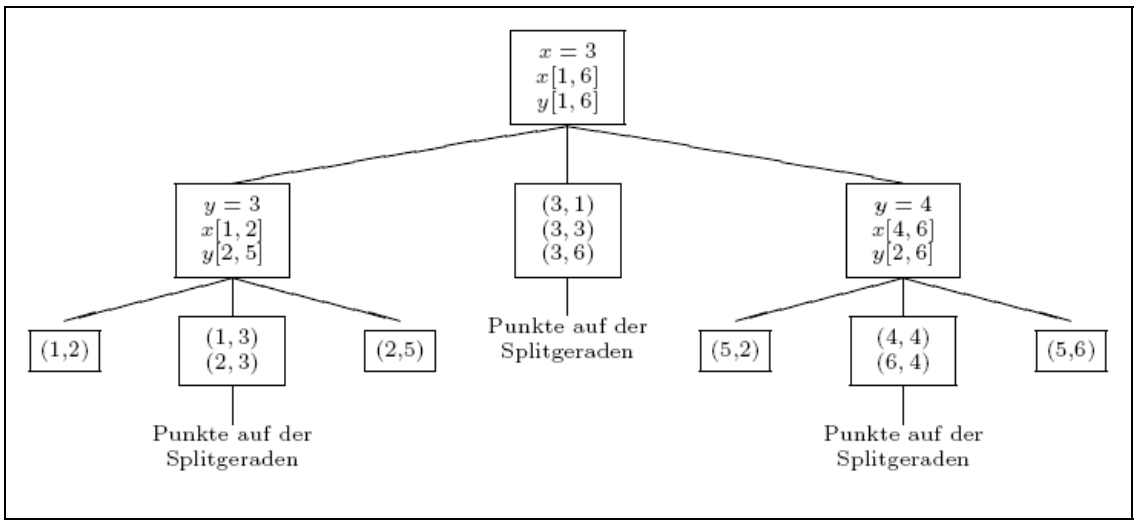


Abbildung 1.4: Der aus Abbildung 1.3 resultierende erweiterte *k*-d-Baum

2. Operationen

Die Operationen auf einem 2-d Baum laufen analog zum binärem Baum ab:

- *Insert:*
  Suche mit Schlüssel [x;y] unter Abwechslung der Dimension die Stelle, wo der [x=y]-Knoten sein müsste und hänge ihn dort ein.

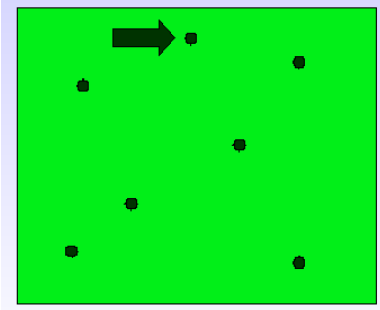
- *Exakt Match* (z.B. finde Record [15;5]):
  Suche mit Schlüssel [x;y] unter Abwechslung der Dimension bis zu der Stelle, wo der [x;y]-Knoten sein müsste.

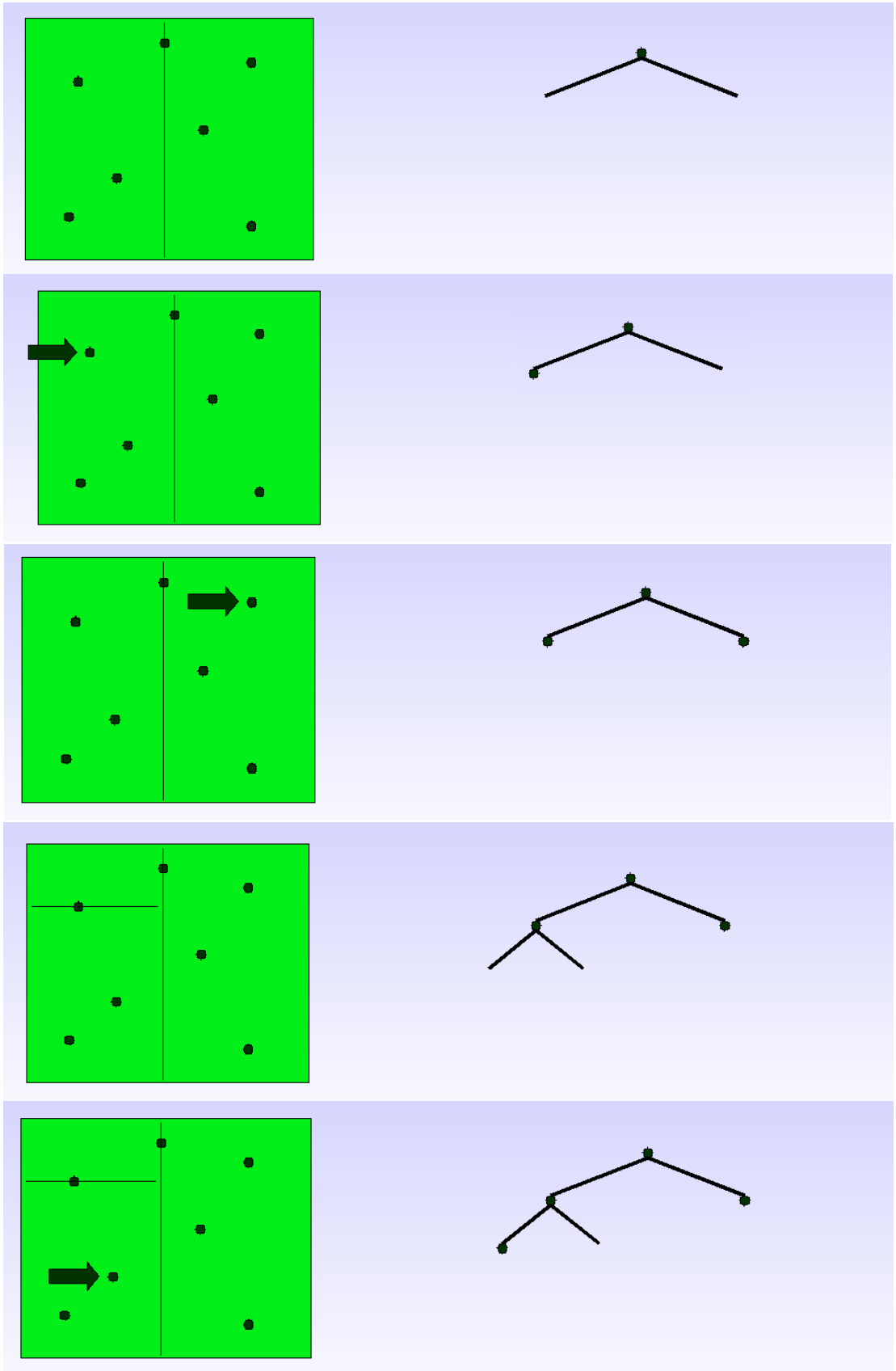
- *Partial Match* (z.B. finde alle Records mit x = 7):
  An den Knoten, an denen nicht bzgl. x diskriminiert wird, steige in beide Söhne ab; an den Knoten, an denen bzgl. x diskriminiert wird, steige in den zutreffenden Teilbaum ab.

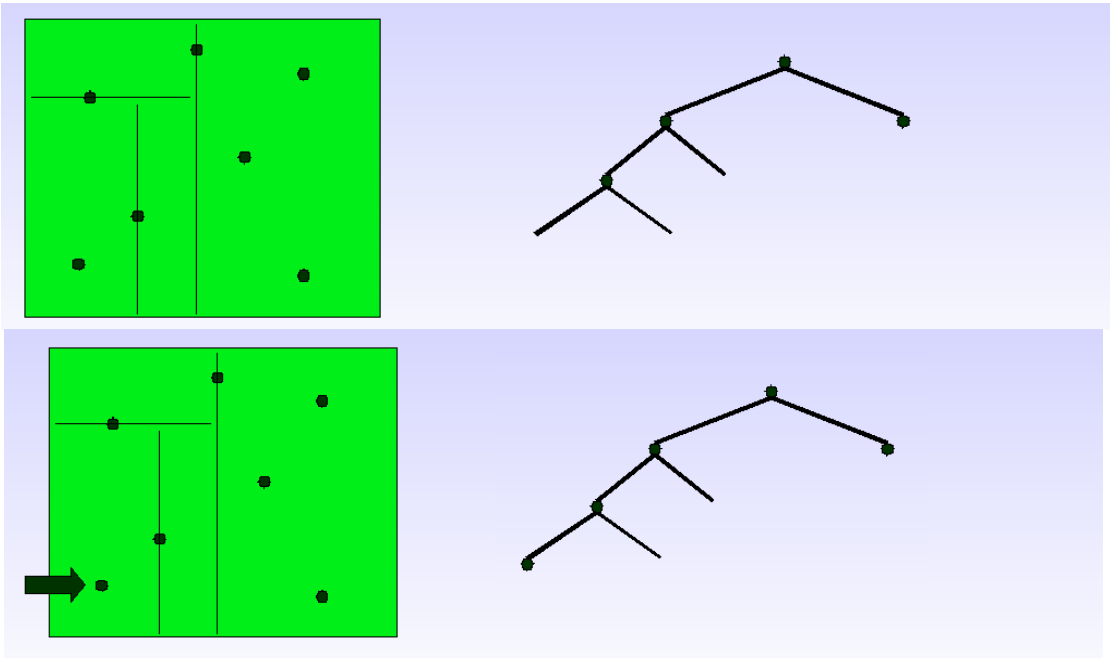
- *Range-Query* (z.B. finde alle Records [x; y] mit 7 x 13; 5 y 8):
  An den Knoten, an denen die Diskriminatorlinie das Suchrechteck schneidet, steige in beide Söhne ab, sonst steige in den zutreffenden Sohn ab. Beobachtung: Laufzeit k+log n Schritte bei k Treffern!

- *Best-Match* (z.B. finde nächstgelegenes Record zu x = 7; y = 3):
  Dies entspricht einer Range-Query, wobei statt eines Suchrechtecks jetzt ein Suchkreis mit Radius gemäß Distanzfunktion vorliegt. Während der Baumtraversierung schrumpft der Suchradius. Diese Strategie ist erweiterbar auf k-best-Matches.

3. Knoten einfügen – Beispiel







4. Bereichsabfragen

Im Gegensatz zum Binärbaum kann aus dem k-d-Baum effizient eine Punktmenge innerhalb eines gewünschten orthogonalen achsenparallelen Bereiches extrahiert werden. Abbildung 4.1 veranschaulicht die Vorgehensweise am Beispiel des Anfragebereiches mit den Grenzen x [0.5, 7.5], y [2.5, 9.5]. Der zur Punktmenge gehörende k-d-Baum ist auf Seite 1 abgebildet.

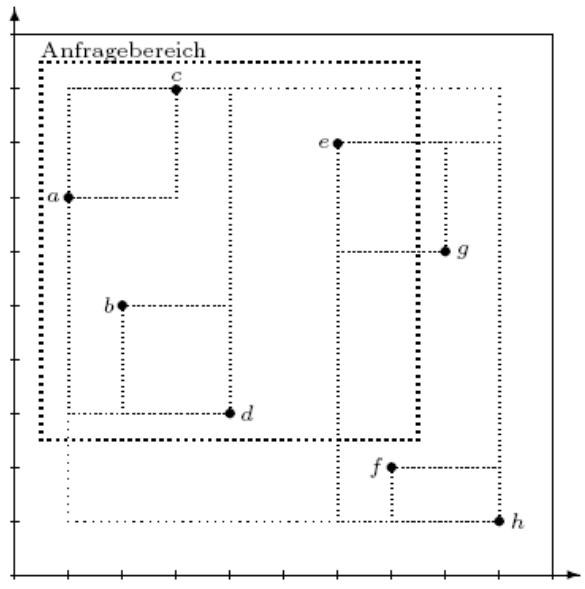


Abbildung 4.1: Anfragebereich und Bereiche der inneren Knoten

Die Bereichsanfrage wird rekursiv ausgeführt und startet im Wurzelknoten. Dieser enthält wie jeder andere innere Knoten des Baumes den Bereich, in dem sich die Punkte des Teilbaumes befinden, als Wegweiser für die Suche. Dieser Bereich kann beim Vergleich mit dem Anfragebereich immer einer der drei folgenden Kategorien zugeteilt werden:

1. Der Bereich des Knotens liegt komplett außerhalb des Anfragebereiches: In diesem Fall kann es im Teilbaum dieses Knotens keine Punkte geben, die innerhalb des Anfragebereiches liegen. Die Bereichsanfrage bricht an dieser Stelle ab.

2. Der Bereich des Knotens liegt komplett innerhalb des Anfragebereiches: In diesem Fall liegen alle Punkte des Teilbaumes des Knotens innerhalb des Anfragebereiches und können zur Ergebnismenge hinzugefügt werden. Die Bereichsanfrage bricht an dieser Stelle ab.

3. Der Bereich des Knotens und der Anfragebereich überschneiden sich: In diesem Fall muss die Bereichsanfrage rekursiv an die Kindknoten weitergeleitet werden. Dies geschieht so lange, bis entweder Fall 1 oder Fall 2 für die betrachteten Knoten zutrifft. Sollte die Anfrage bis in die Blätter des Baumes weitergeleitet werden, so wird für den jeweiligen Punkt geprüft, ob er im Anfragebereich liegt. Wenn ja, wird er zur Ergebnismenge hinzugefügt.

Für den oben dargestellten Anfragebereich würde die Bereichsanfrage so verlaufen: Der Bereich des Wurzelknotens überschneidet sich mit dem Anfragebereich, daher wird die Anfrage rekursiv an die Kindknoten weitergeleitet. Der linke Kindknoten befindet sich komplett innerhalb des Anfragebereiches, daher werden die Punkte a, b, c und d zur Ergebnismenge hinzugefügt. Der rechte Kindknoten überschneidet sich wiederum mit dem Anfragebereich, daher werden rekursiv die Kindknoten in Betracht gezogen. Im linken Kindknoten geht die Rekursion weiter in die Tiefe bis auf die Ebene der Blätter, der Punkt e liegt innerhalb und wird zur Ergebnismenge hinzugefügt, der Punkt g liegt außerhalb und bleibt daher unberücksichtigt. Der Bereich jenes Teilbaumes, der die Punkte f und h enthält, liegt komplett außerhalb des Anfragebereiches und somit terminiert die Bereichsanfrage an dieser Stelle. Als Ergebnis der Anfrage liegt die Menge {a, b, c, d, e} vor.

5. Insert - Methode

Beim Einfügen eines Knotens fängt man mit Root-Knoten an. Sollte der Root-Knoten leer sein, also leeres Baum. So wird der einzufügende Knoten zu Root. Seine Kinder werden auf Null gesetzt. Die Aufteilungsebene wird auf Null gesetzt und der Bereichsarray für die Unterbäume angepasst. Als Rückwert der Funktion wird der Root-Knoten geliefert.

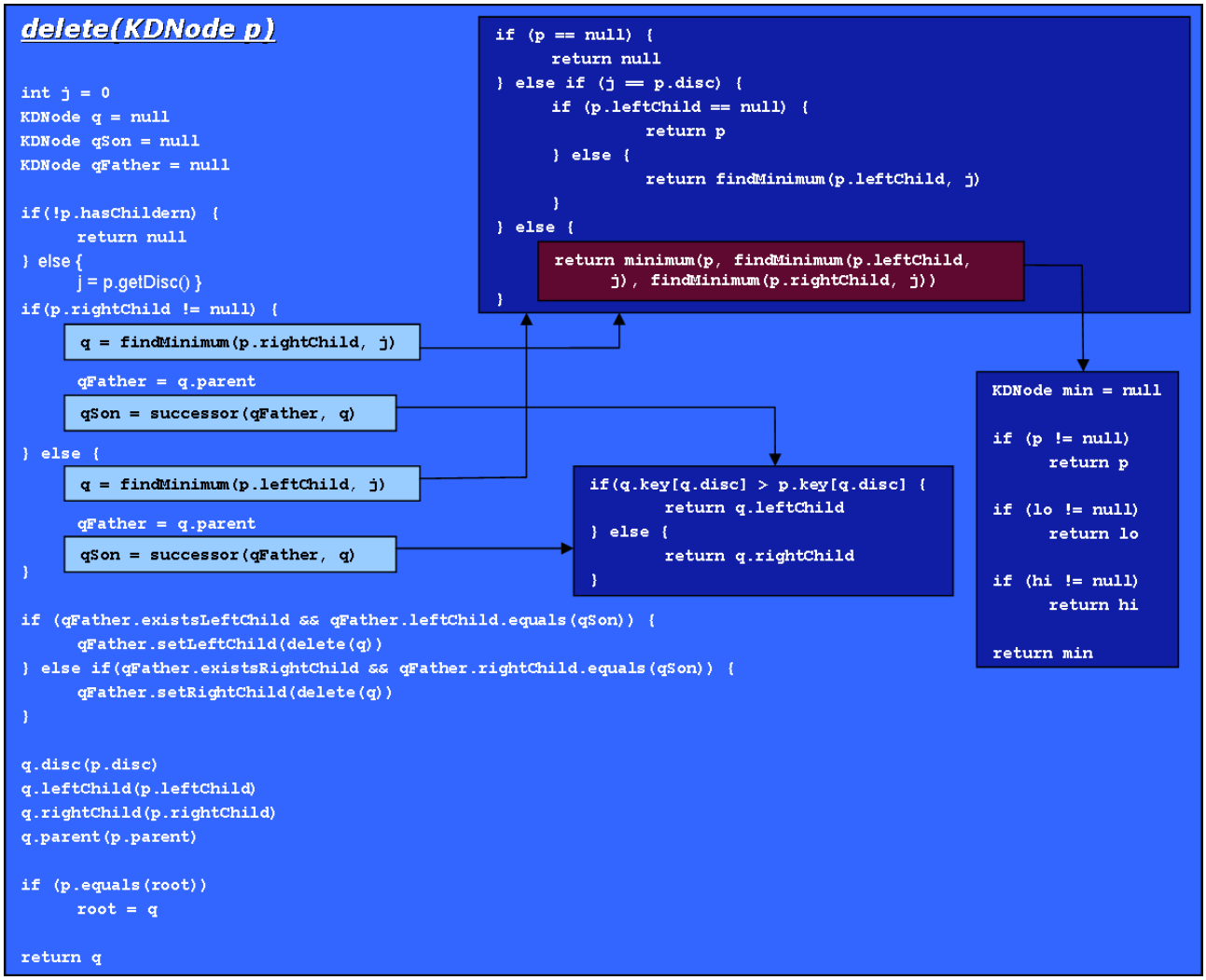
```
insert(KDNode p)

KDNode q
KDNode son

if(root == null) {
        root = p
        p.leftChild = null
        p.rightChild = null
        p.disc = 0
        p.createBounds()
        return root
} else {
        q = root
}

while(q != null) {
        if(p.equals(q)) {
                return q
        } else {
        son = successor(q, p)
        }

        if(son == null) {
                p.disc = nextDisc()
                if(q.key[q.disc] > p.key[q.disc]) {
                                q.leftChild = p
                } else {
                                q.rightChild = p
                }
                p.parent = q
                p.createBounds()
                son = p
                nodes++
                return son
        } else {
                q = son
        }
}
```

Falls ein Root-Knoten bereits exestiret, so wird Q auf Root-Knoten gesetzt. Danach folgt eine While-schleife in der man Ebene für Ebene den Baum nach unten durchläuft bis die passende Stelle für das Einfügen der neuen Knoten gefunden ist.

Bei einem zweidimensionalem Baum mit den Koordinaten x und y, werden in der ersten Ebene die x-Koordinaten verglichen und je nach dem, ob der einzufügende Knoten größer oder kleiner ist, so wird der Knoten entweder rechts oder links als Kind eingefügt. In der zweiten Ebene werden die y-Koordinaten verglichen und nach dem gleichen Prinzip entschieden, wo der Knoten einzufügen ist. Dies geschieht mit der Hilfe der Methode „SUCCESSOR".

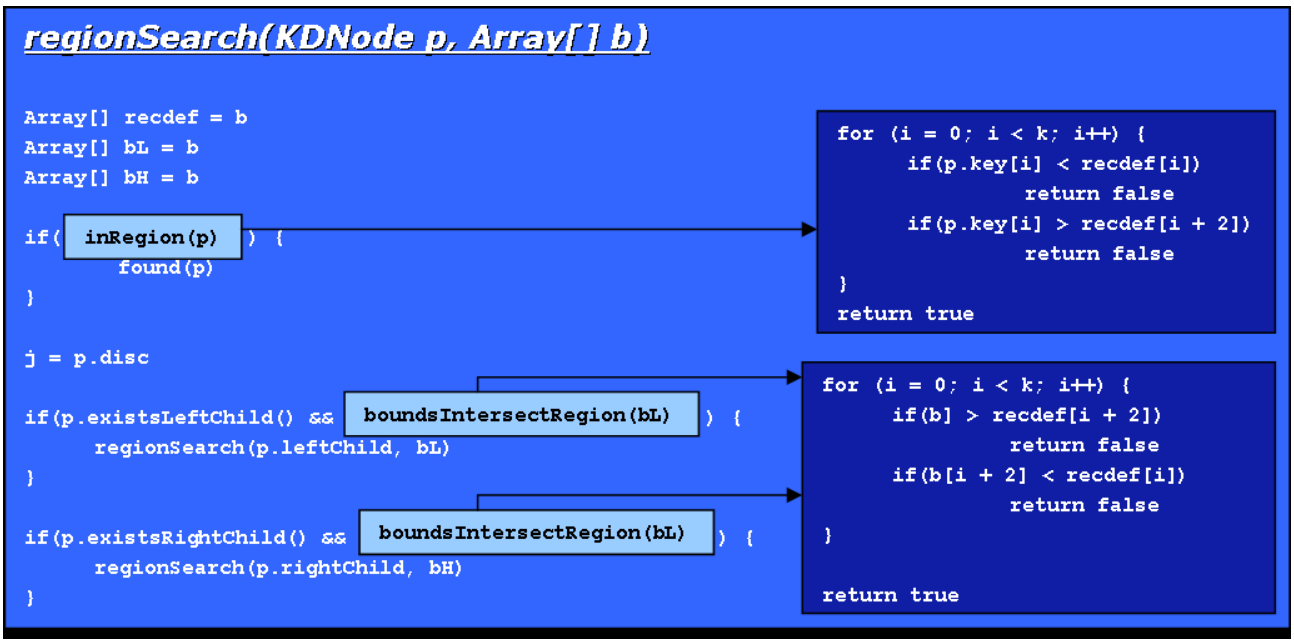
Nach der kurzen Analyse fählt es sofort auf, dass While-Schleife eine Rekursion simuliert, in dem es Knoten für Knoten entweder in den linken oder rechten Teilbaum herunter steigt. Es ist allgemein bekannt das bei Rekursionen die Laufzeit ungefähr der Laufzeit O(log n) entspricht. Deswegen schätzen wir die Funktion mit der Laufzeit O(log n) ein.

6. Delete – Methode



Die Entfernungsmethode ist sehr kompliziert, besonders beim entfernen der Wurzel muss man einen sehr großen Aufwand betreiben um den Ersatzknoten für die Wurzel zu finden. Man steigt von dem zu entfernendem Knoten den Baum herunter und sucht in dem linken Teilbaum nach dem kleinsten Element oder in dem rechten Teilbaum nach dem größten Element der selben Ebene wie die Ebene von dem Knoten, der entfernt werden soll.

7. RegionSearch



An die Methode wird ein aktueller Knoten übergeben und ein Anfragebereich in dem gesucht werden soll. Es wird geprüft ob der aktuelle Knoten sich in dem Anfragebereich befindet. In der If-Abfragen wird geprüft ob bei dem aktuellen Knoten Kinder vorhanden sind und ob der Anfregbereich mit dem Bereich des Knotens zumindest teilweise übereinstimmt. Falls dies der Fall ist, dann wird die Funktion rekursiv mit der Übergabe von Kindknoten aufgerufen. Nach der asymptotische Analyse beträgt die Laufzeit ohne die Betrachtung der Aufrufe der Funktionen inRegion() und boundsIntersectRegion(), O(log n). In der For-Schleifen der Hilfsfunktionen ist k, die Anzahl der Ebenen. Rechnet man diese Operationen dazu so erhält man die Laufzeit O(3k log n).

9. Vergleiche

9.1 Bereichsanfragen

Abbildung 9.1 stellt die Ausführungsdauer der Bereichsanfragen von Vektor, *k*-d Baum und Bereichsbaum gegenüber. Die per Zufallsgenerator erzeugte Punktmenge befindet sich im Bereich x[0, 1], y[0, 1] und ist annähernd gleich verteilt. Das Zentrum der Bereichsanfragen befindet sich stets im Punkt (0.5, 0.5), die Seitenlänge des Anfragebereiches variiert von 5% bis 100% der Seitenlänge der von der Punktmenge eingenommenen Fläche. Die vierte Linie im Diagramm stellt die Anzahl der Punkte innerhalb des Anfragebereiches dar, welche von links nach rechts sukzessive zunimmt.

Betrachtet man die Abfrage kleiner Bereiche, so tritt das erwartete Verhalten ein: Die langsamste Variante sind die Bereichsanfragen an Vektoren, schneller sind die Anfragen an den *k*-d Baum, und die besten Resultate werden vom Bereichsbaum erzielt.

Bei der Abfrage großer Bereiche verlieren jedoch die Bäume an Effizienz. Dies liegt daran, dass die rekursive Weiterleitung der Anfrage innerhalb des Baumes nur noch selten abgebrochen werden kann und oft bis auf die Ebene der Blätter fortgesetzt werden muss. Der Vektor hingegen kann den benötigten Bereich durchlaufen und effizient eine große Menge von Punkten ausgeben. Selbst wenn der ganze Bereich des Baumes innerhalb des Anfragebereiches liegt, dauert es länger, die Punkte aus den Tiefen des Baumes hervorzuholen, als alle Punkte eines Vektors auszulesen und ihre zweite, nicht der Sortierrichtung des Vektors entsprechende Koordinate dahingehend zu prüfen, ob sie innerhalb des Anfragebereiches liegt.

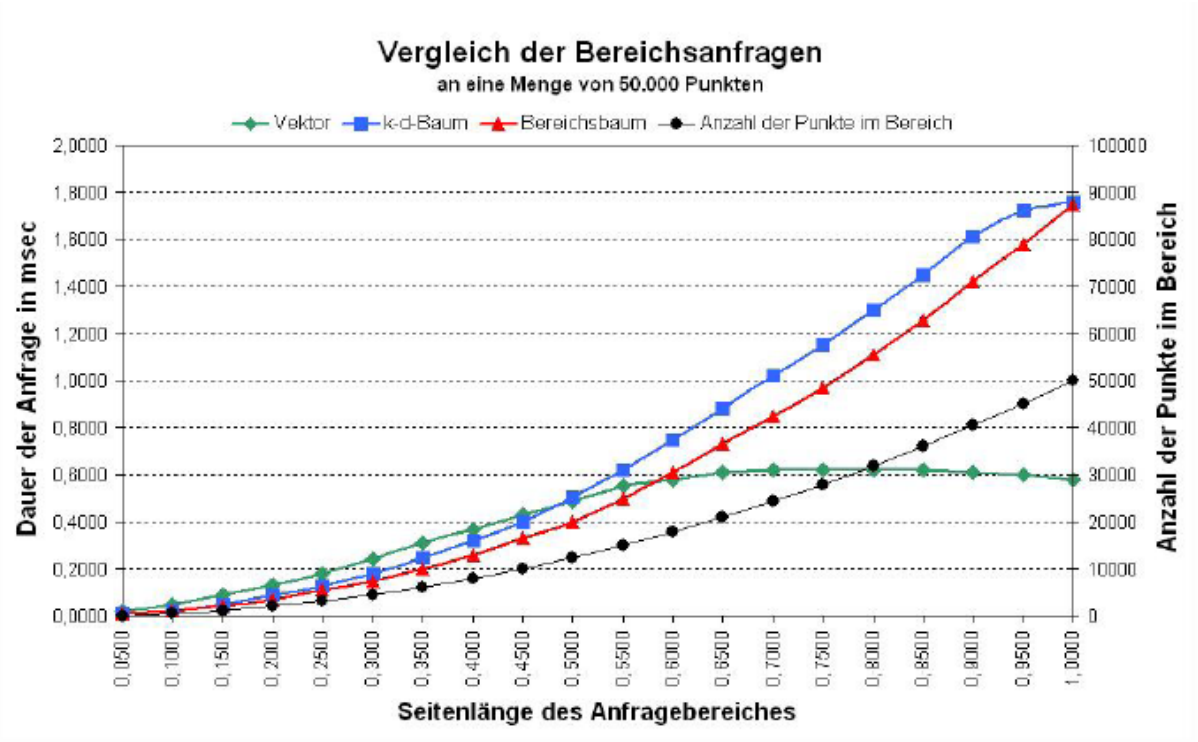


Abbildung 9.1: Datenstrukturen - Effizienz von Bereichsanfragen

Kleine Bereichsanfragen liefern das erwartete Verhalten. Bei größeren Anfragebereichen hingegen verlieren die Suchbäume an Effizienz, da das Extrahieren der Punkte aus den Blattknoten zu viel Zeit benötigt. Interessant ist, dass die Dauer der Anfragen an Vektoren bei steigender Seitenlänge

abnimmt. Für dieses Verhalten konnte jedoch keine Erklärung gefunden werden.

9.2 Speicherbedarf

Abbildung 9.2 stellt den Speicherbedarf von Vektor, *k*-d Baum und Bereichsbaum gegenüber. Während Vektor und *k*-d Baum nur relativ wenig Speicherplatz benötigen und somit mehrere Millionen Punkte speichern können, so werden beim Bereichsbaum relativ schnell die Grenzen des vorhandenen Arbeitsspeichers gesprengt (für eine Menge von 1,3 Millionen Punkte wird bereits mehr als ein Gigabyte Speicher benötigt).

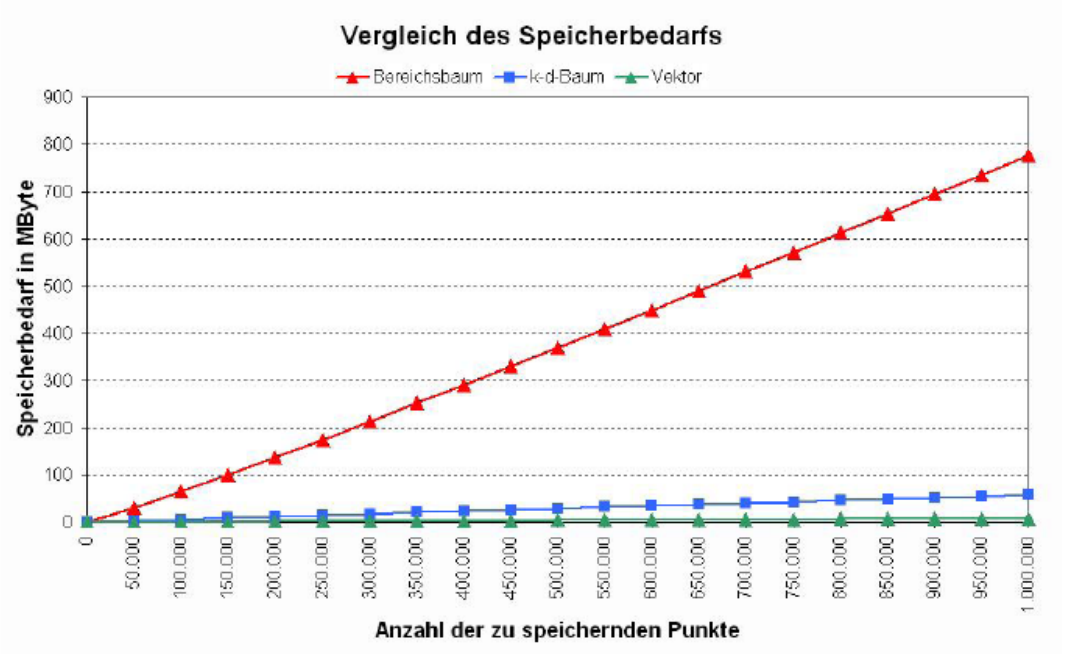


Abbildung 9.2: Vergleich des Speicherbedarfs