Quadtrees

Christian Höner zu Siederdissen

Quadtrees

- Zum Verständnis benötigt...
- Was sind Quadtrees
- Datenstruktur
- Wofür Quadtrees
- Operationen auf dem Baum
- Vor- und Nachteile
- (spezialisierte Formen)

Zum Verständnis benötigt...

- Der Vortrag über Datenstrukturen
- Der Vortrag über Bäume
- Translation, Transformation in der Ebene

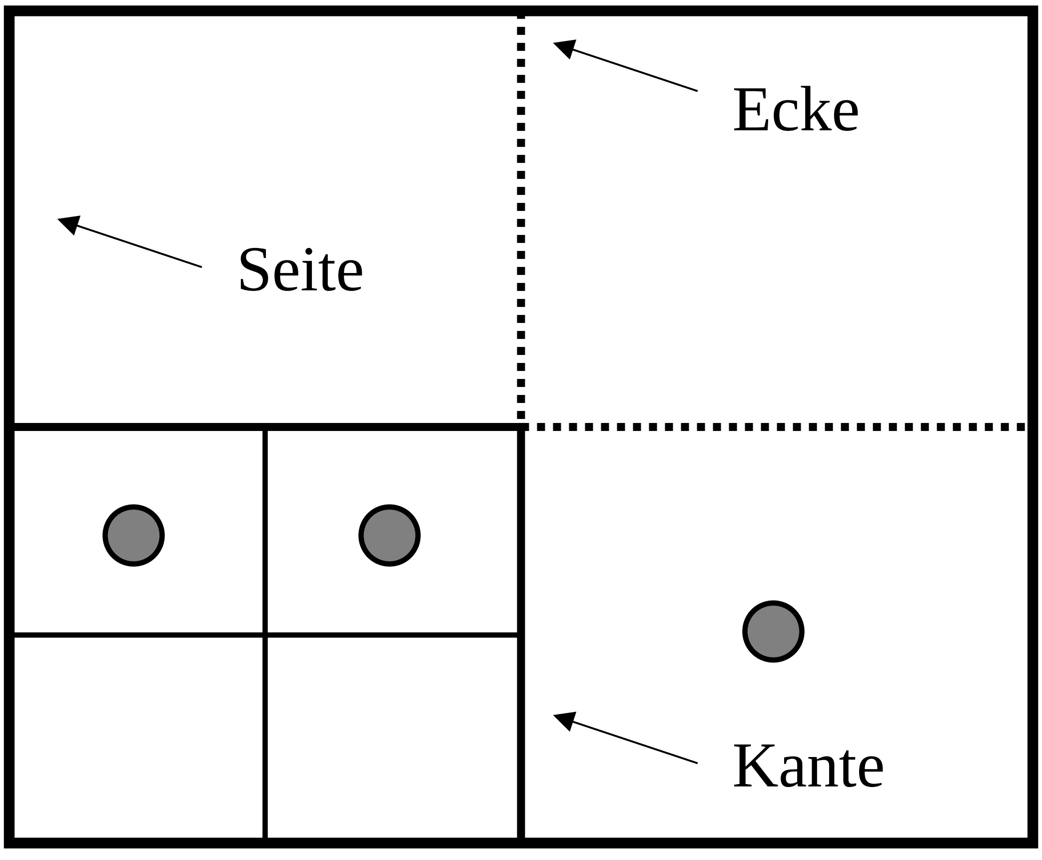
Was sind Quadtrees...

- Dienen der Raumteilung
- Partitionieren Objekte nach ihrer Position in der Ebene
- Zugriff auf Objektgruppen geordnet nach ihrer Position in der Ebene
- Finden von Nachbarobjekten

...Was sind Quadtrees

- Etablieren eine Baumstruktur mit bis zu 4 Kinderknoten (Childnodes)
- Ordnen Kinderknoten nach geometrischer Position
- Knotenebenen werden von „0“ (root) an gezählt

Grafische Darstellung



Quadtree-Struktur

Knoten {

Elementliste	(enthält die Objekte)
Kinderliste	(enthält die max. 4 Kinder)
Vater	(Verweis auf Vater)
BoundingBox	(Position in der Ebene)

}

Wofür Quadtrees

- In der Computergrafik (Landschaften, Spiele, hidden surface removal, ray tracing)
- Bildanalyse
- Finden des nächsten Nachbarn in deutlich kürzerer Zeit als O(n)

Quadtree für eine Punktmenge

- Jeder Knoten enthält maximal 1 Element
- Ansonsten: Unterteilung des Knotens und eventuell auch der Kinderknoten
- Jeder Knoten speichert Informationen über seine Lage (BoundingBox), sowie Kinder und seinen Vaterknoten (bis auf den Wurzelknoten)

Operationen auf Quadtrees

- Einen Quadtree für eine Punktmenge generieren
- Einen Punkt dem Tree hinzufügen
- Einen Punkt aus dem Tree löschen
- Zu einem Punkt einen Nachbarn finden
- Einen Quadtree balancieren

Pseudocode zur Generierung

<u>Teile Knoten (Punktmenge, BoundingBox)</u>

wenn card (Punktmenge) <= 1 „return“

„erstelle Unterknoten“

„verteile Punktmenge nach Unterknoten-Boundingbox“

„für jeden Unterknoten:“ <u>Teile Knoten (...)</u>

Verteilen der Punkte auf Unterknoten

Die vier Unterknoten NE, NW, SW, SE mit den Mittelpunkten:

Xmid := (X+X‘) / 2 Ymid := (Y+Y‘) / 2

Verteilen nach:

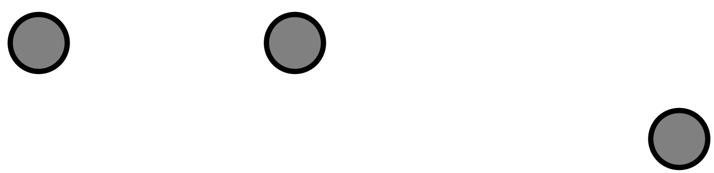
Pne := {p € P : px > Xmid && py > Ymid}

Pnw := {p € P : px <= Xmid && py > Ymid}

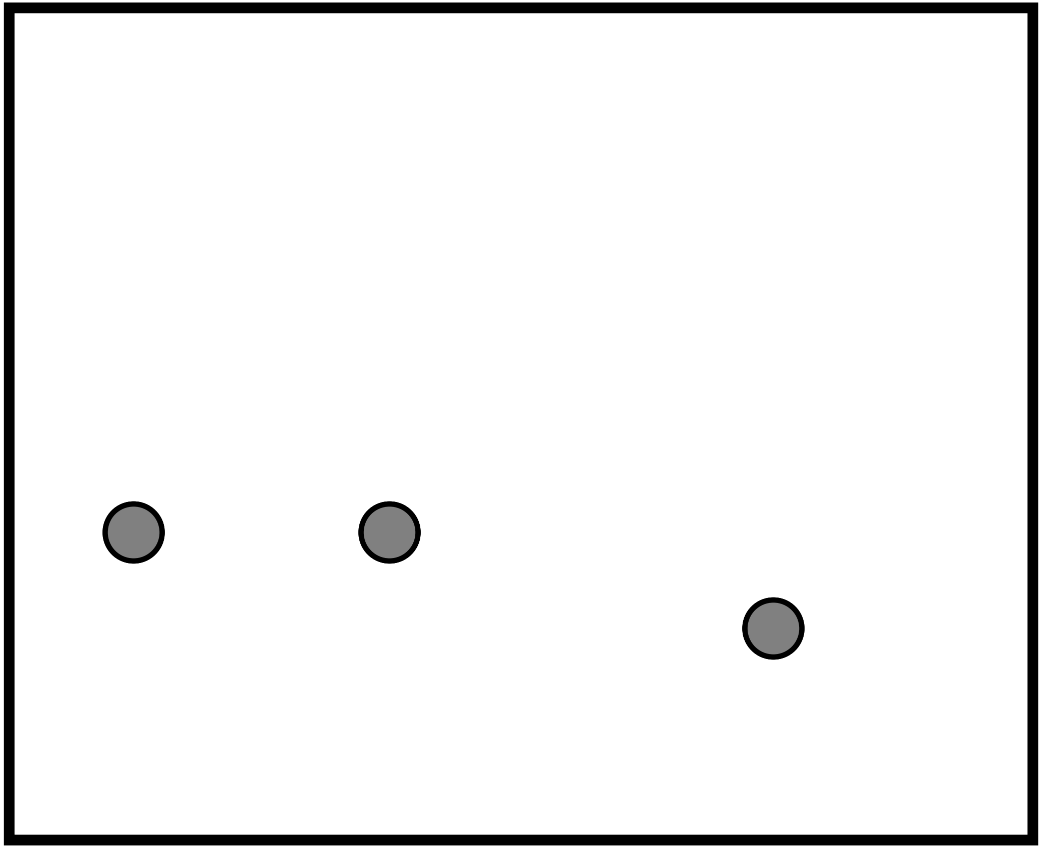
Psw := {p € P : px <= Xmid && py <= Ymid}

Pse := {p € P : px > Xmid && py <= Ymid}

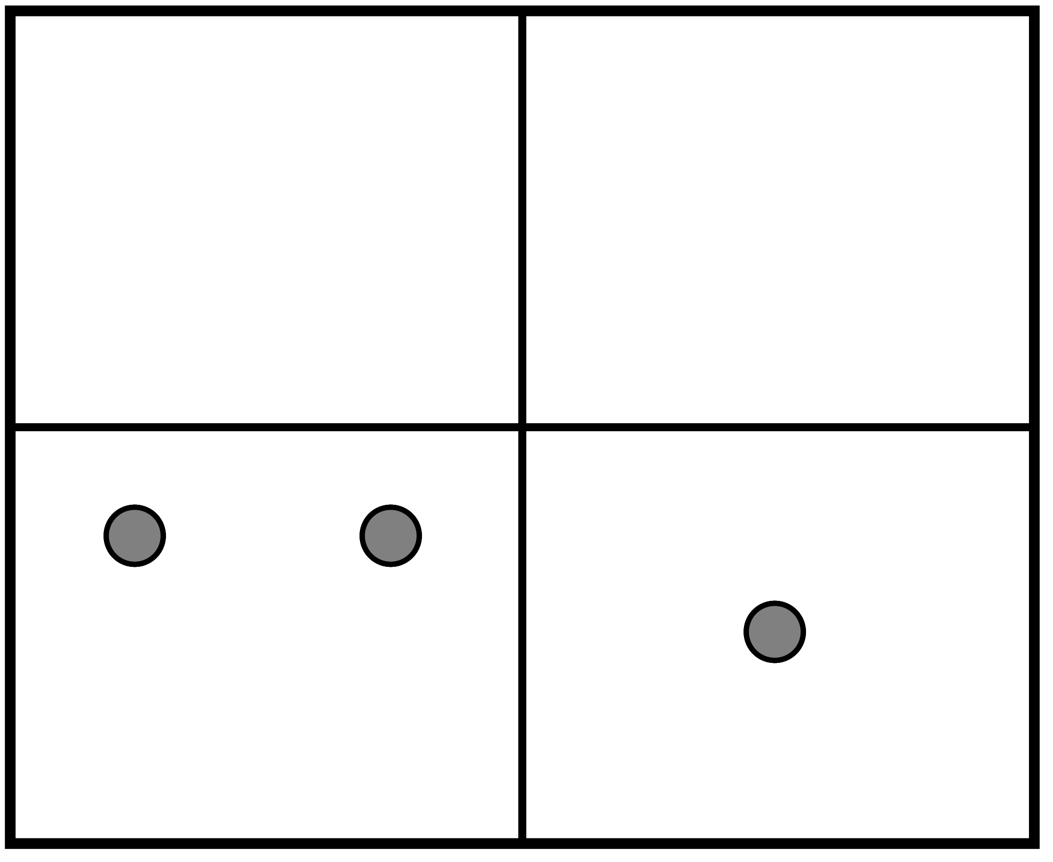
Die Generierung im Bild



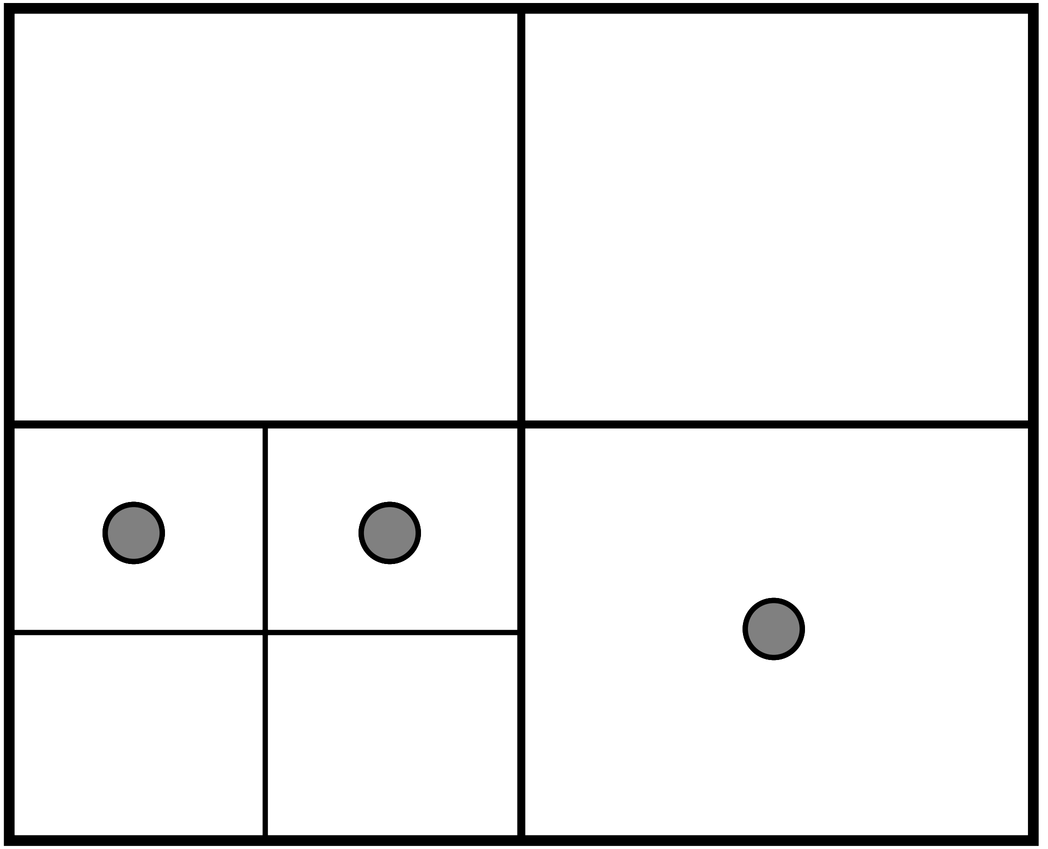
Die Generierung im Bild



Die Generierung im Bild



Die Generierung im Bild



Zeit- und Ressourcenbedarf

- Tiefe: d
- Anzahl der Punkte: n
- Benötigte Knoten: O((d+1)n)
- Benötigte Zeit: O((d+1)n)
- minimale Tiefe: O(log n)
- Maximale Tiefe: O (log (s/c)) mit: c: kleinste Punktdistanz und s: Seitenlänge

Einen Punkt hinzufügen

Knoten <- root

<u>Punkt hinzufügen (Knoten)</u>

Wenn „Knoten leer“ oder „nicht vorhanden“

„füge Punkt in Knoten ein“

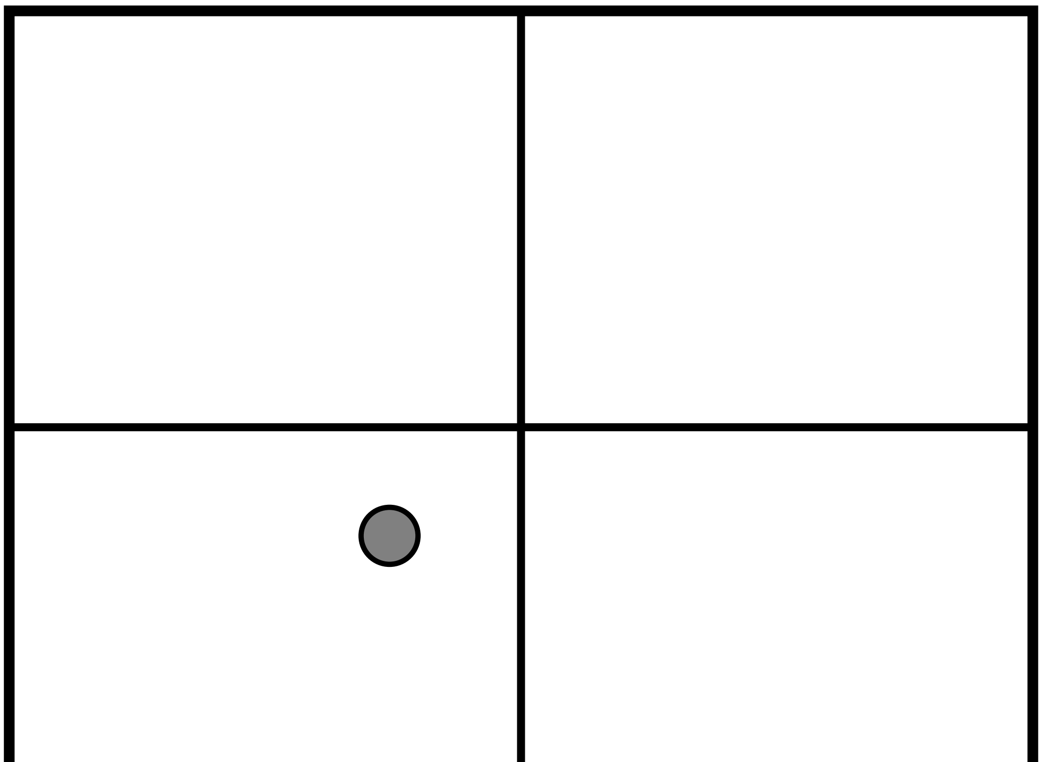
Sonst „finde den Punkt einschließenden Kindsknoten n“

„erstelle 4 Kinderknoten“

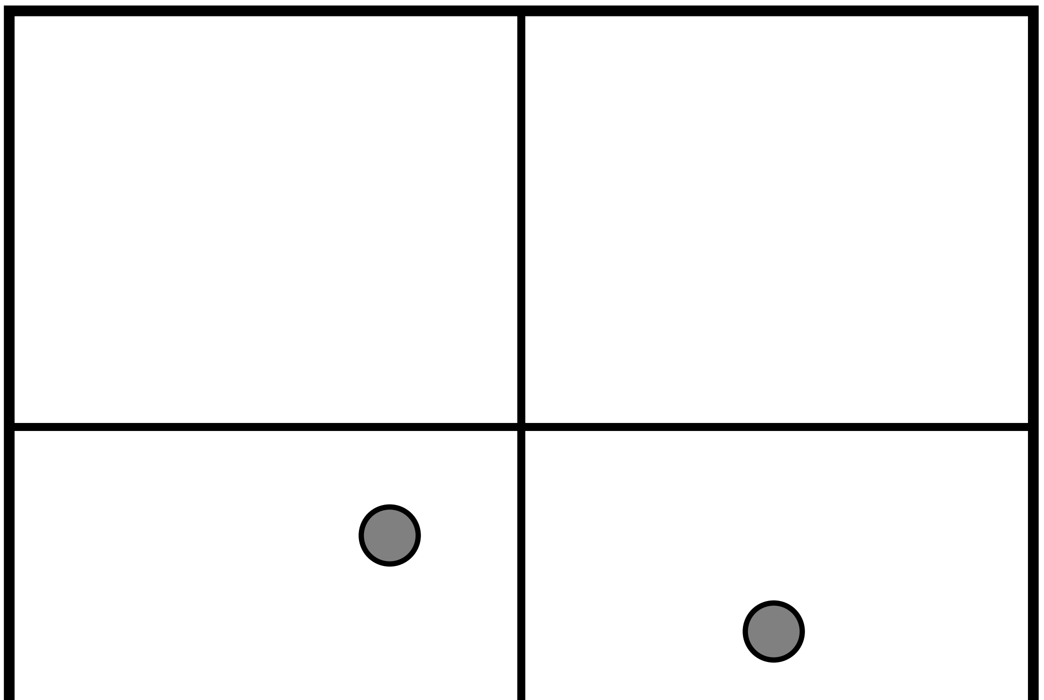
„füge vorhandenen Punkt in passenden Knoten ein“

<u>Punkt hinzufügen (n)</u>

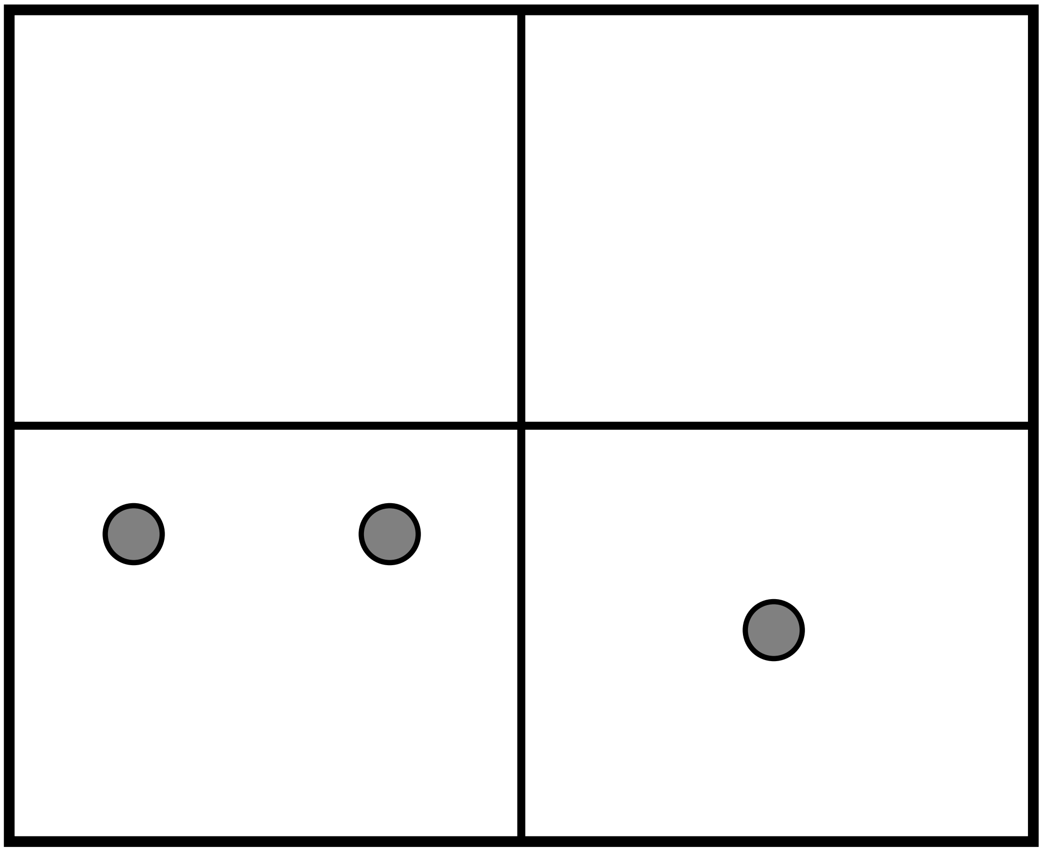
Hinzufügen im Bild



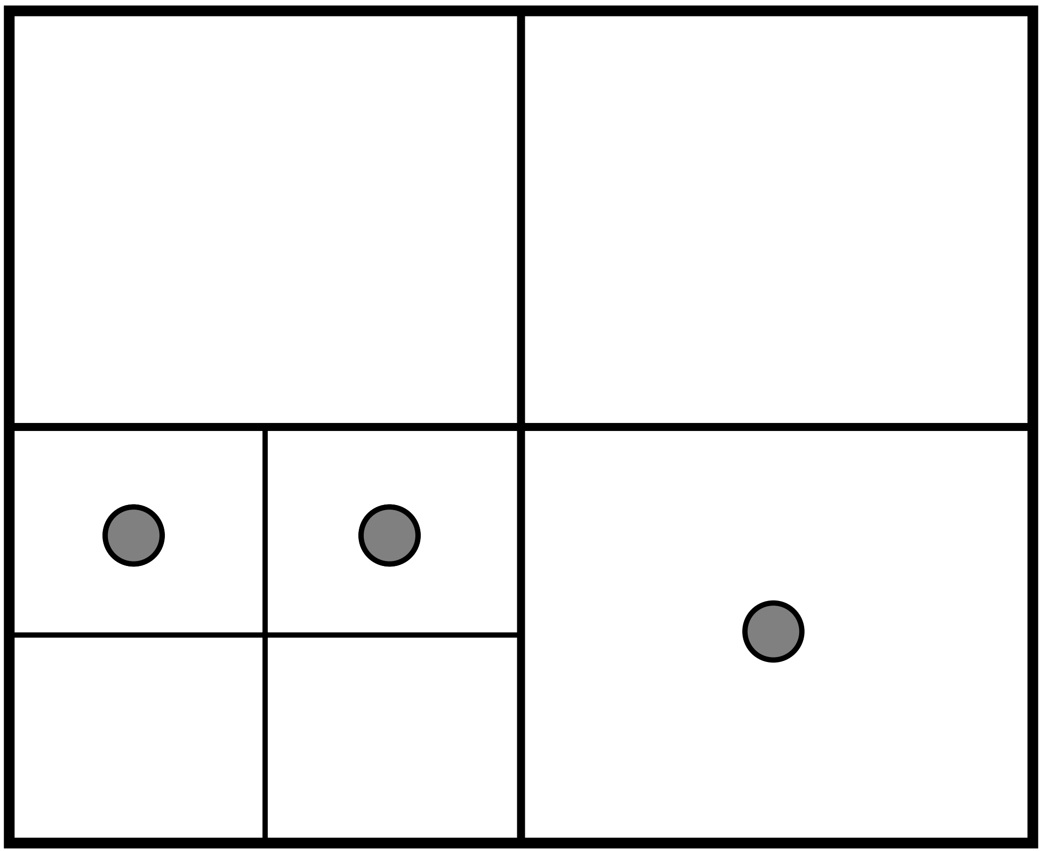
Hinzufügen im Bild



Hinzufügen im Bild



Hinzufügen im Bild



Zeit- und Ressourcenbedarf

- Knotentraversierung: O (log n)
- Im worst case O(n)
- Einfügeoperation ist konstant
- Vielleicht Neubalancierung des Baumes nötig
- Gesamt: Zeitbedarf O (log n) + O ((d+1)m) und Speicherbedarf: O(1)

Einen Punkt löschen

Knoten <- root

Punkt löschen (Knoten)

Wenn „nicht vorhanden“ „return“

Wenn „Punkt in Knoten“ „lösche Punkt“

Sonst „finde den Punkt einschließenden Kindsknoten n

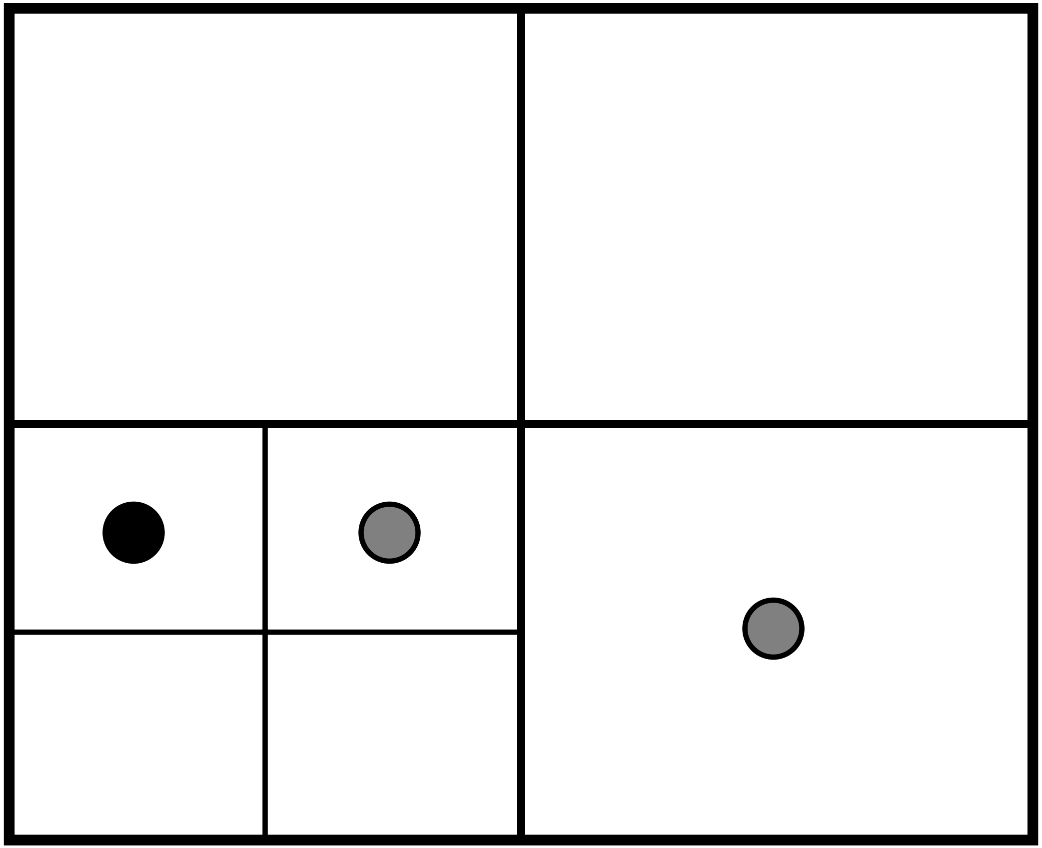
    Punkt löschen (n)

    wenn „mindestens drei Kinderknoten leer“

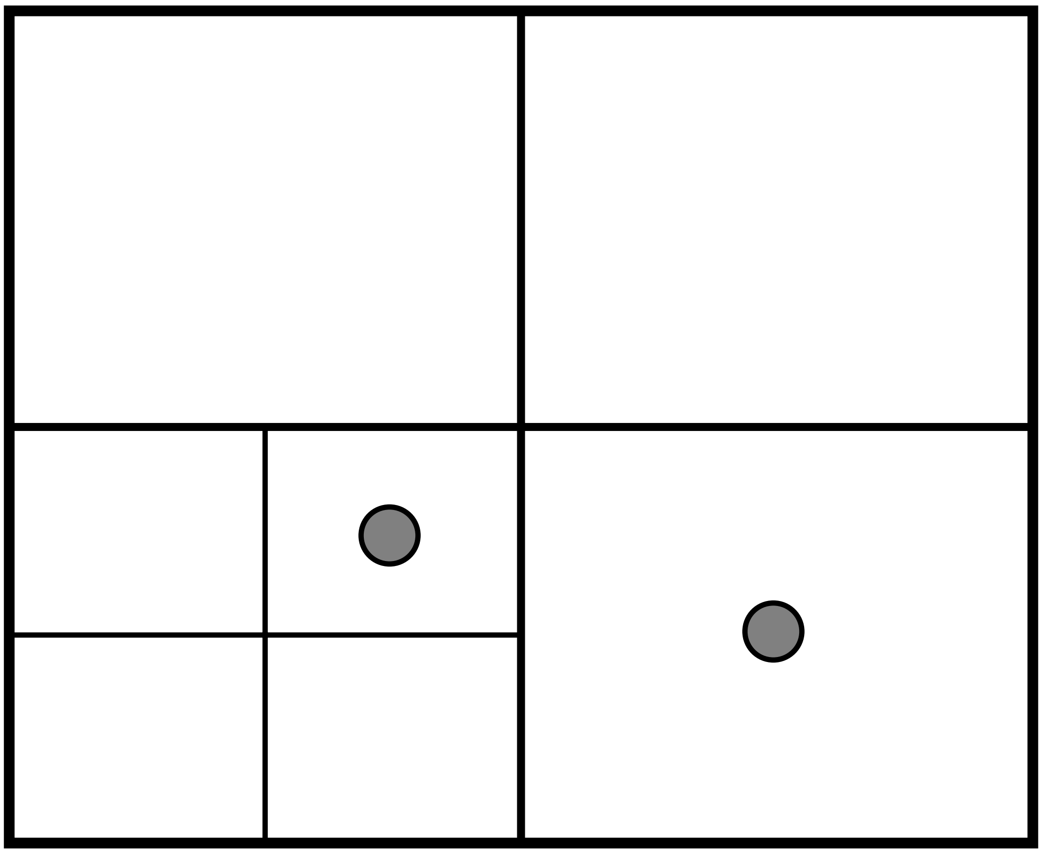
        „kopiere den vorhandenen Punkt ‚hoch‘“

        „lösche Kinderknoten und werde Blatt“

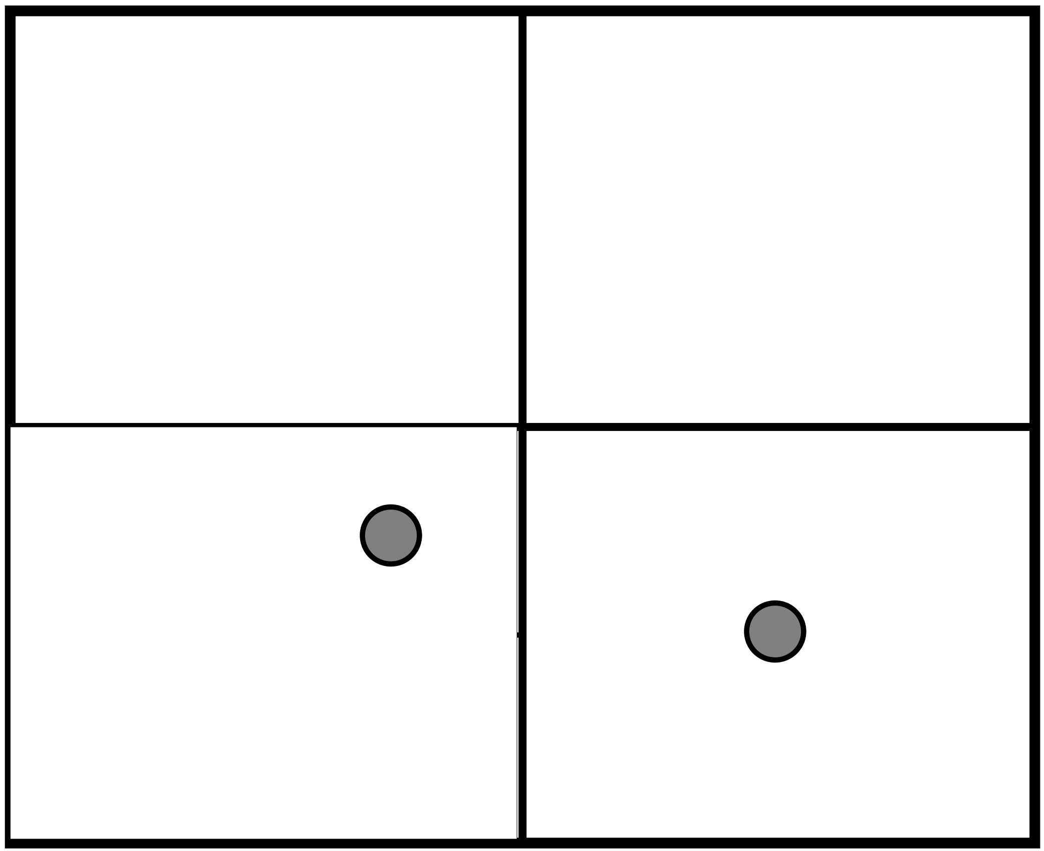
Löschen im Bild



Löschen im Bild



Löschen im Bild



Zeit- und Ressourcenbedarf

- Knotentraversierung: O (log n)
- Im „worst case“ O (n)
- Löschoperation ist konstant
- Vielleicht Neubalancierung des Baumes nötig
- Gesamt: Zeitbedarf O (log n) + O ((d+1)m) und Speicherbedarf: O(1)

Den nächsten Nachbarn finden

Nördlicher Nachbar (Knoten)

Wenn Knoten = Wurzel „return null“

Wenn Knoten = SW „return NW“

Wenn Knoten = SE „return NE“

Temp ← Nördlicher Nachbar (Vater(Knoten))

Wenn Temp = null *oder* Temp = „Blatt“

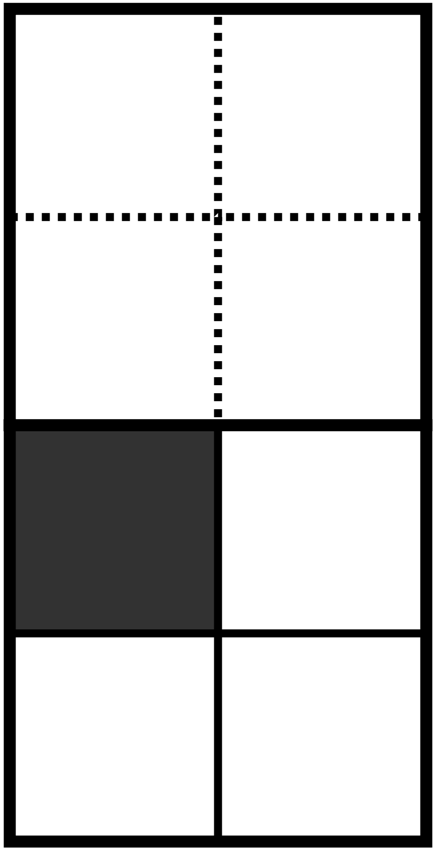
  „return Temp“

Sonst

  wenn Knoten = NW „return SW von Temp“

  sonst „return SE von Temp“

Den nächsten Nachbarn finden



Zeit- und Ressourcenbedarf

- Tiefe: d
- Knoten: v

- Benötigte Zeit: O(d+1) für jeden v

Einen Quadtree balancieren

- Jedes Blatt soll als Nachbar einen haben, der eine Seitenlänge von maximal einer gewissen Differenz aufweist
- Oft ist diese Differenz „2“
- Ob balanciert wird, hängt von der Verwendung des Trees ab und somit der Nutzung der gespeicherten Daten

Einen Quadtree balancieren

<u>Quadtree Balancieren</u>

„alle Blätter in eine Liste geben"

Während „Liste nicht leer"

  „nehme ein Blatt aus der Liste"

  wenn „Blatt muss gesplittet werden"

    „erstelle internen Knoten aus Blatt"

    wenn „Knoten einen Punkt enthält"

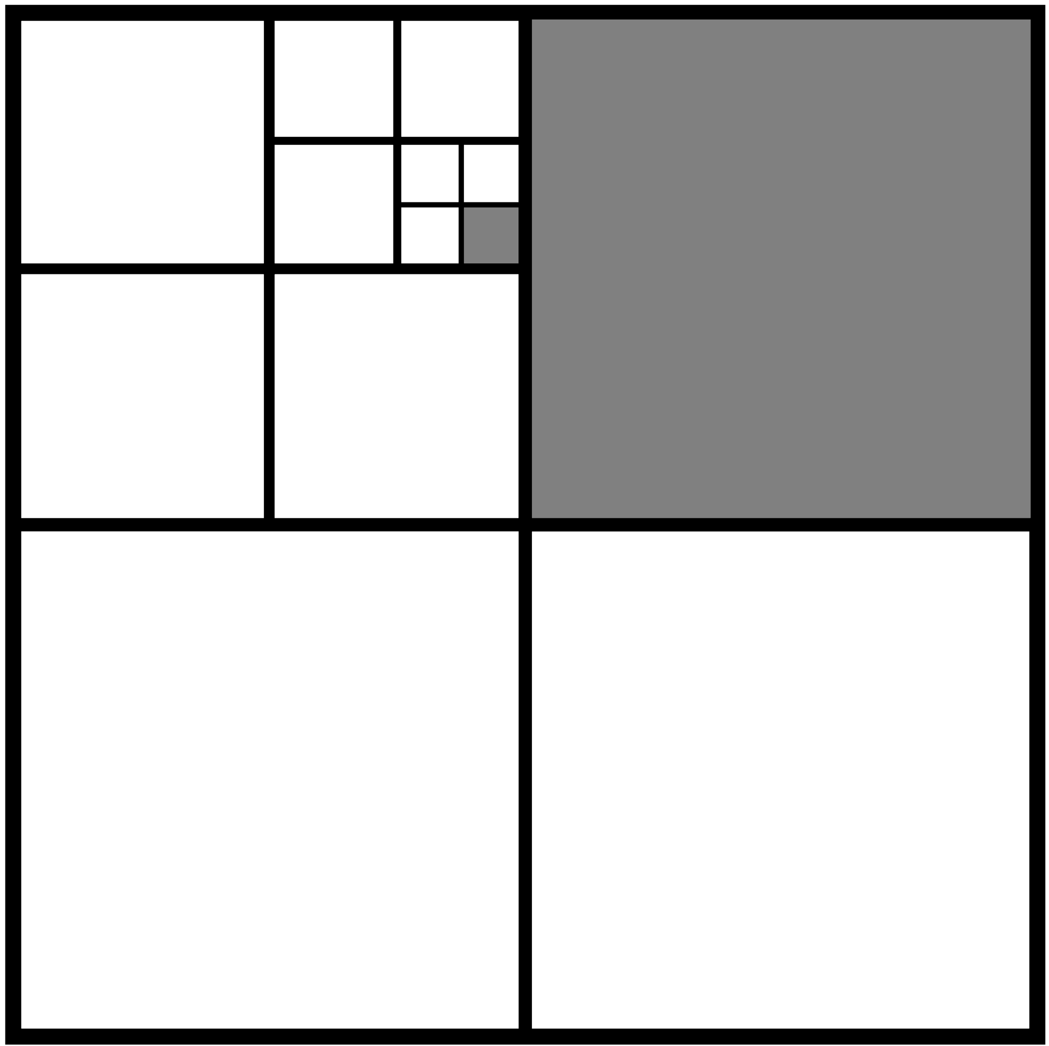
      „Punkt in passendes Blatt einfügen"

    „füge die neuen Blätter in die Liste ein"

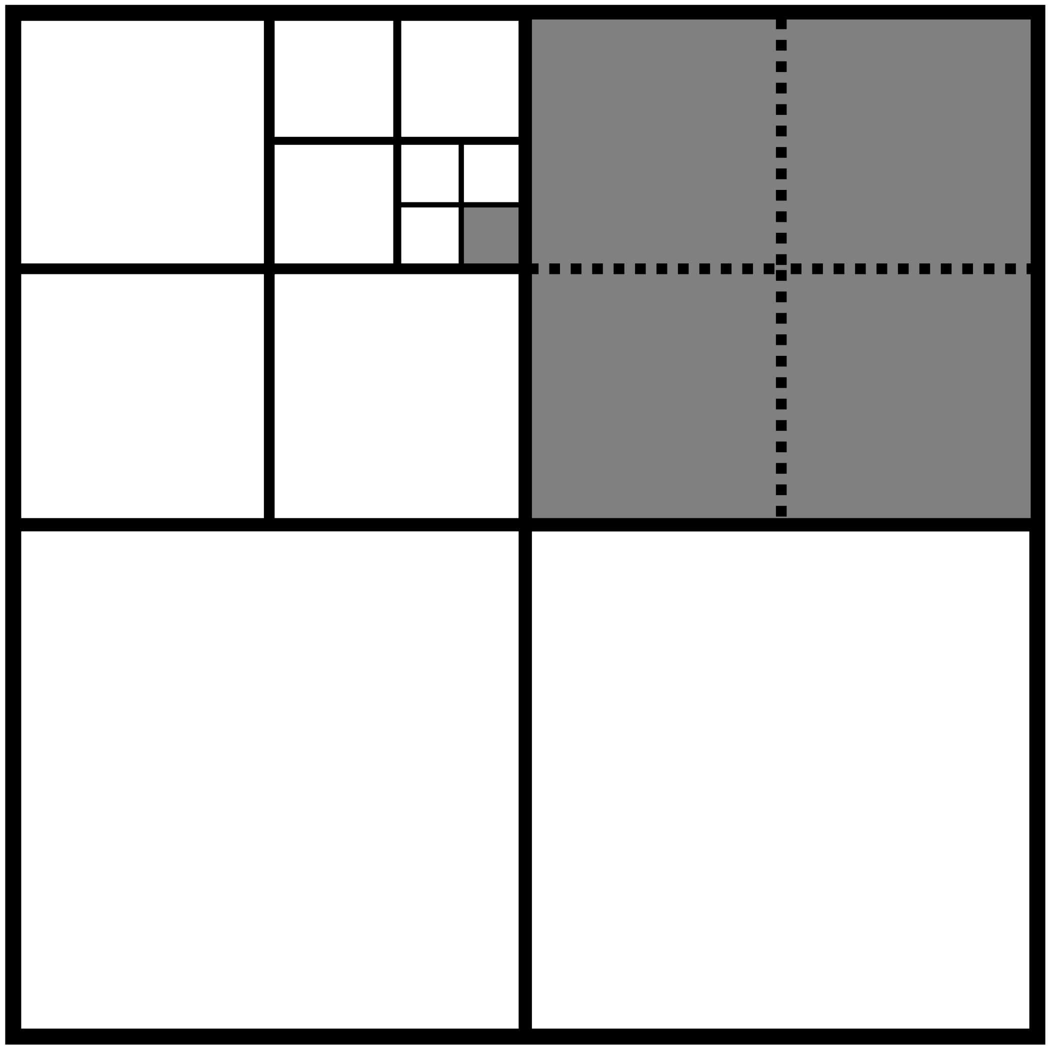
  wenn „Knoten" hat nun Nachbarn, die gesplittet werden müssen"

    „füge Nachbarn in Liste ein"

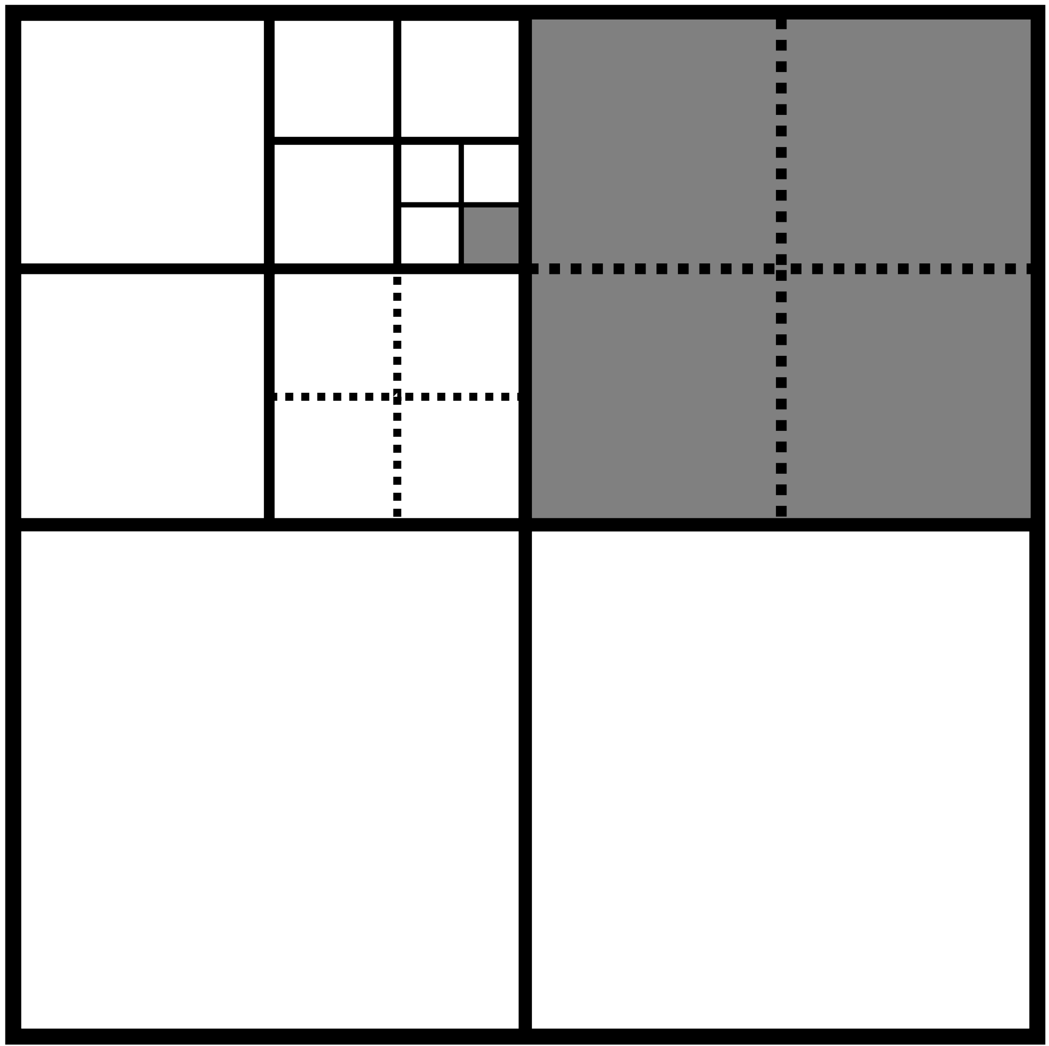
Balancieren im Bild



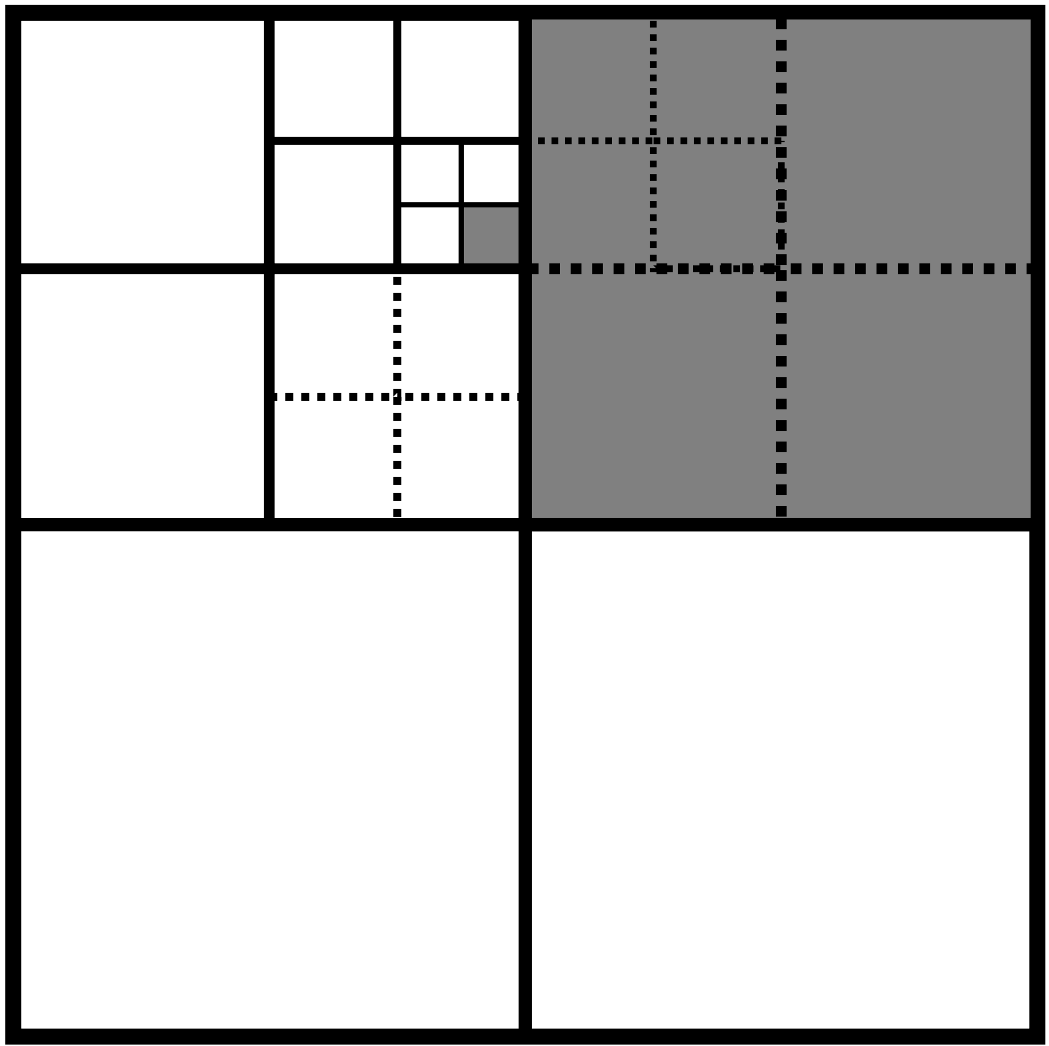
Balancieren im Bild



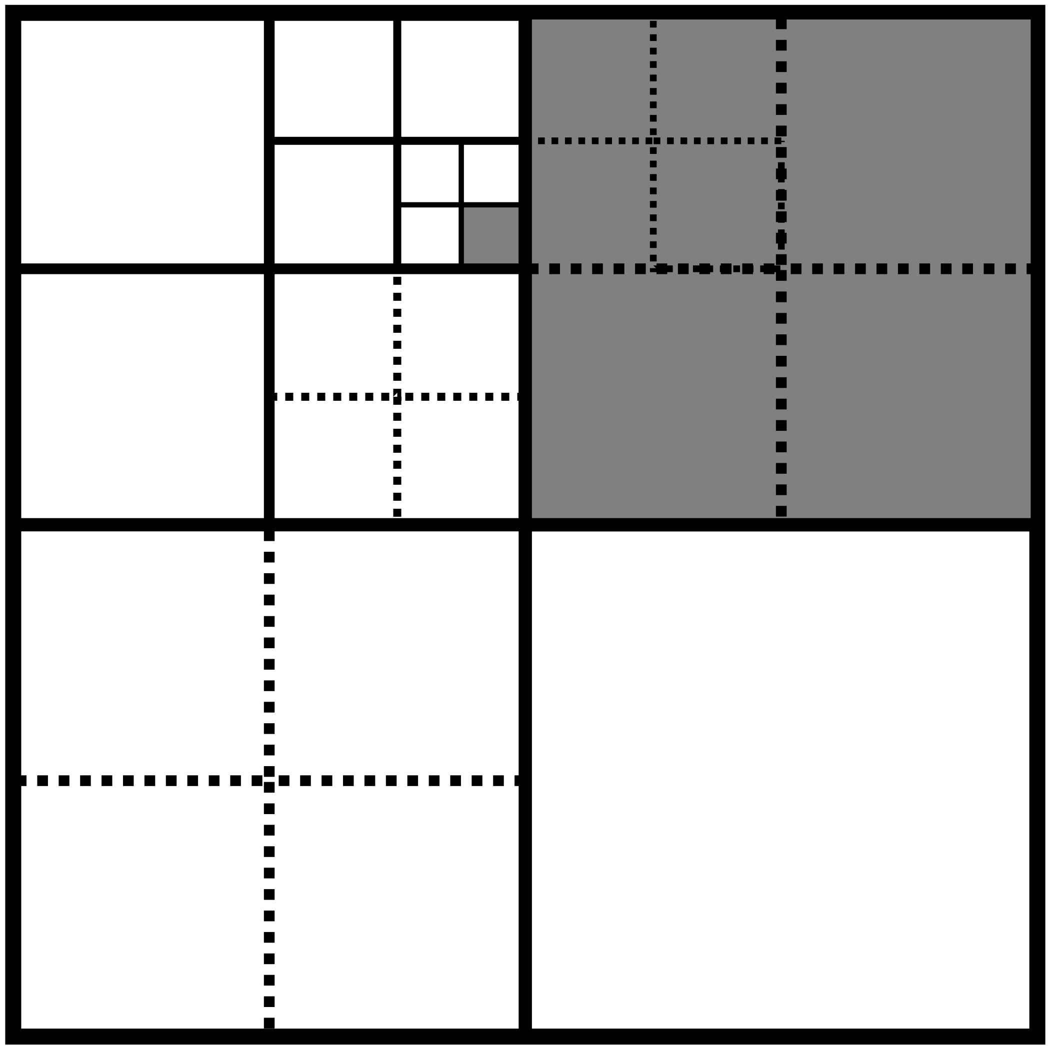
Balancieren im Bild



Balancieren im Bild



Balancieren im Bild



Zeit- und Ressourcenbedarf

- Anzahl der Knoten im Baum: m

- Anzahl der Knoten im balancierten Baum: O(m)
- Benötigte Zeit: O((d+1)m)
- Maximale Zeit: Baum der Tiefe d mit allen Blättern in der maximalen Tiefe

Zusammenfassung...

- Teilt den Raum, nicht die Daten
- Für spezielle Anwendungen gut geeignet
- Bei diesen teilweise sehr schnell (O(1))
- Für Intersektionstests geeignet (ray tracing)
- Zugriffsoperationen in der Klasse O (log n)
- Einfach zu implementieren (eine der ersten solcher Strukturen)

...Zusammenfassung

- Die Raumabhängigkeit kann leicht zu degenerierten Bäumen führen
- Nutzen deshalb sehr abhängig von den Datensätzen die gespeichert werden
- Bessere Strukturen teilen oftmals die Daten (siehe folgenden Vortrag)

Spezialisierte Quadtrees

- Reguläre Quadtrees
- Objekte statt Punkte im Baum
- „lose“ Quadtrees
- Quadtrees mit direktem Zugriff
- Octrees

Reguläre Quadtrees

- Es sind immer alle Blätter in der maximalen Tiefe ausgebildet (sehr balancierter Tree)
- In den Blättern werden Referenzen auf die Nachbarn gespeichert
- Sehr schnelle Intersektionstests möglich
- Viele leere Blätter oder
- In einem Blatt mehrere Datenelemente

Effizienz regulärer Quadtrees

- Knotenzugriff in $O(\log n)$
- Intersektionstests zu Nachbarknoten: $O(1)$
- Speicherbedarf von $2n$ gegenüber $n \log n$

Objekte statt Punkte

- Zweidimensionale Objekte statt Punkte werden gespeichert
- Ausdehnung und Mittelpunkt des Objektes zur Berechnung
- Speicherung nicht nur in den Blättern, sondern auch in zwischen gelagerten Knoten (auch im „root“)
- Mehr als 1 Objekt je Knoten möglich

Effizienz bei Objekten

- Ein Faktor k: die Anzahl der Objekte im jeweiligen Knoten
- Alle Operationen verlängern sich um den Faktor k (Löschen in k log n statt n)
- k liegt im „worst case“ in O(n)

„lose“ Quadtrees

- Vereinfachte Speicherung von Objekten
- Die umschließenden Boxen sind variabel
- Objekte „sinken“ tiefer herab, die sonst aufgrund ihrer Position „höher“ im Baum gespeichert würden

Effizienz bei losen Quadtrees

- Faktor k: wieder die Anzahl der Objekte im jeweiligen Knoten
- Wie Quadtrees bei Objekten (k log n statt log n)
- Faktor k deutlich kleiner, oft 1

Quadtrees mit direktem Zugriff

- Unterart der regulären Quadtrees
- Ermöglichen Zugriff auf einen Knoten in O(1)
- Besonders mächtig, wenn dynamisch Objekte im Baum geändert werden (hinzufügen, löschen)

Effizienz von DA-Quadtrees

- Faktor k: Anzahl der Objekte im jeweiligen Knoten
- Knotenzugriff in O(1)
- Sonst wie reguläre Quadtrees
- In jedem Knoten mehrere Objekte möglich: Objektzugriff verlängert um Faktor k mit „worst case“ O(n)

Octrees

- Erweitern den Quadtree ins Dreidimensionale
- Alle Operationen und Unterarten auch für den Octree möglich
- Effizienzverhalten entspricht dem jeweiligen Quadtree

Weitere Informationen

- Game Programming Gems, Vol 1
- Game Programming Gems, Vol 2
- www.gamedev.net/reference
- De Berg, p.291 – 306 (Semesterapparat)