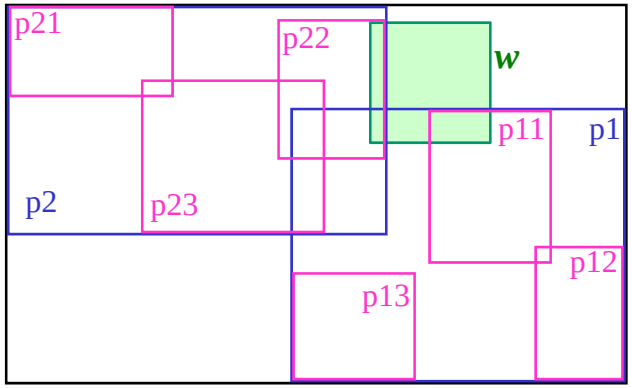
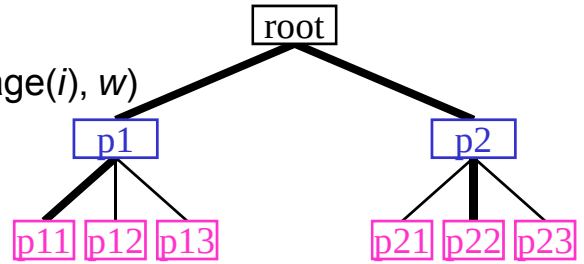
2.2 Indexbasiere Nachbarschaftssuche

2.2.1 Indexbasierte Fensteranfrage

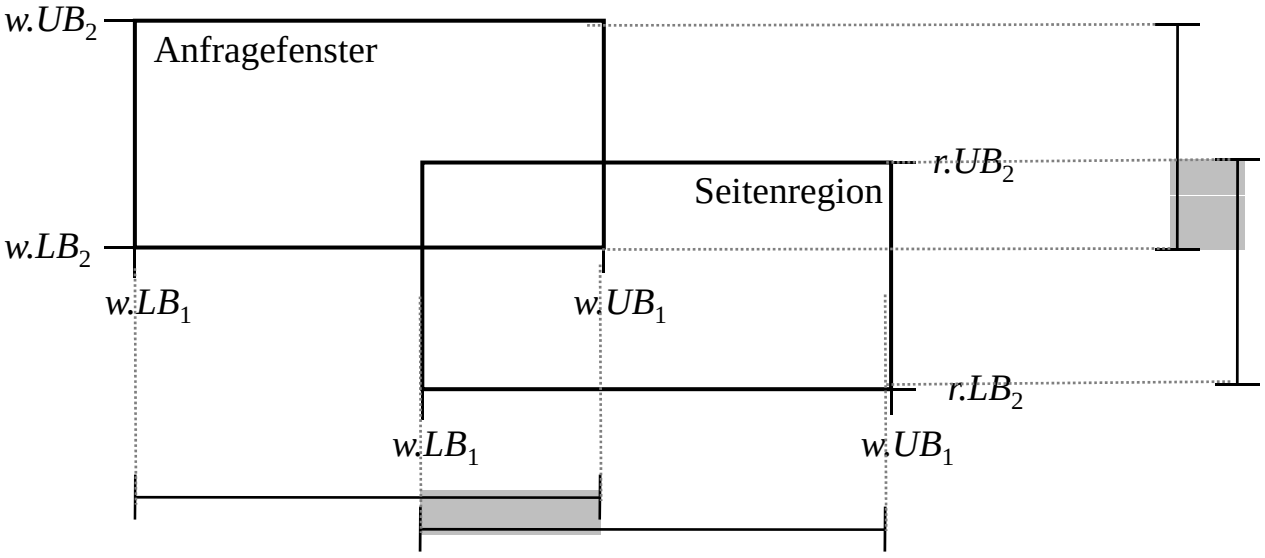
- Algorithmus

```
// w := Anfragefenster
WQ-Index(pa,w)          // pa initialisiert mit Diskadress der Wurzel des R*-Baums
    result = ∅;
    p := pa.loadPage();
    IF p.isDataPage() THEN
        FOR i=0 TO p.size() DO
            o = p.getObject(i);
            IF o ∈ w THEN
                result := result ∪ o;

    ELSE                    // p ist Directoryseite
        FOR i=0 TO p.size() DO
            IF intersect(w, p.getRegion(i)) THEN
                result := result ∪ WQ-Index(p.childPage(i), w)
    RETURN result;
```





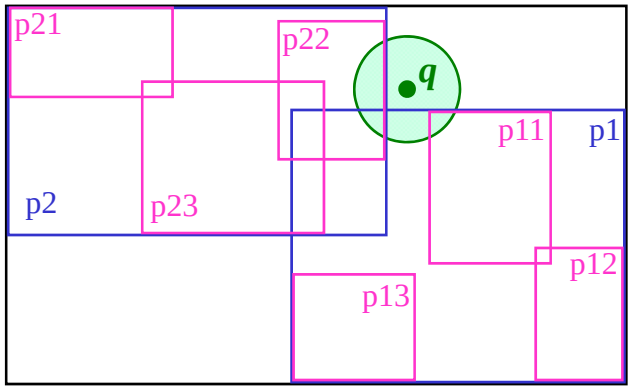
- Auswertung des Intersect-Prädikates *intersect*():
  - Test ob Anfragefenster *w* ein anderes achsenparalleles Rechteck *r* (z.B. R*-Baum-Seitenregion) schneidet
  - Gegeben:
    - Anfragefenster $w = [(w.LB_1, w.UB_1), \ldots, (w.LB_d, w.UB_d)]$
    - Seitenregion $r = [(r.LB_1, r.UB_1), \ldots, (r.LB_d, r.UB_d)]$
  - Anfragefenster w scheidet Seitenregion r genau dann, wenn sich alle entsprechenden Projektionen auf die d eindimensionalen Räume schneiden.

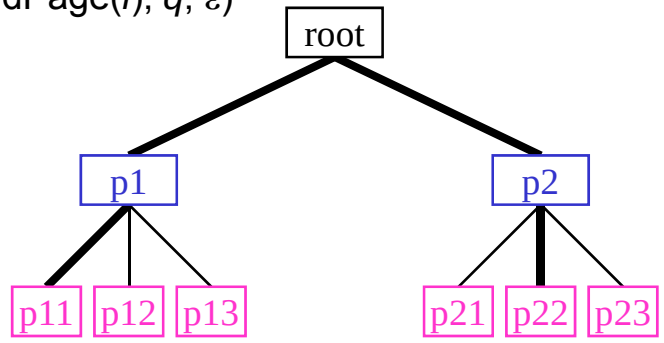


2.2.2 Indexbasierte Bereichsanfrage

- Algorithmus:

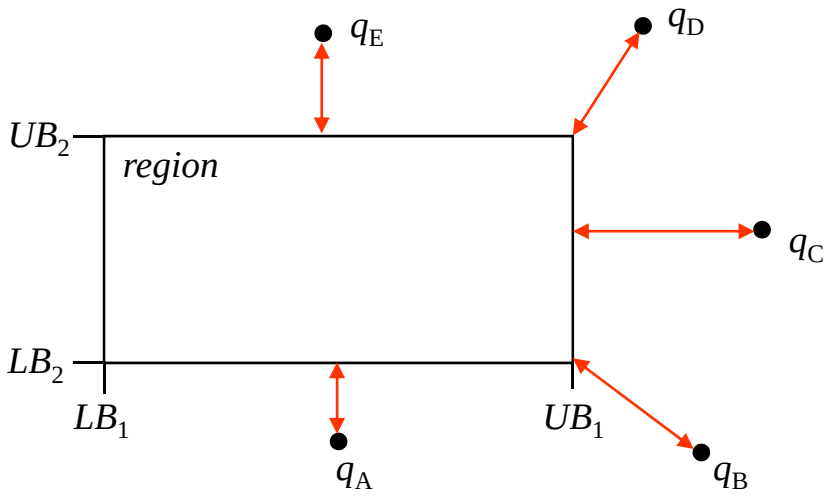
```
RQ-Index(pa,q)          // pa initialisiert mit Diskadress der Wurzel des R*-Baums
    result = ∅;
    p := pa.loadPage();
    IF p.isDataPage() THEN
        FOR i=0 TO p.size() DO
            o = p.getObject(i);
            IF dist(q,o) ≤ ε THEN
                result := result ∪ o;

    ELSE                    // p ist Directoryseite
        FOR i=0 TO p.size() DO
            IF MINDIST(q, p.getRegion(i)) ≤ ε THEN
                result := result ∪ RQ-Index(p.childPage(i), q, ε)
    RETURN result;
```





- MINDIST
  - Test ob Anfragebereich (q,ε) sich mit Seitenregion schneidet
  - Minimale Distanz zwischen Anfragepunkt und allen Punkten der Seitenregion (=> Lower Bound!!!)
  - Beispiel: Berechnung der MINDIST für L_2-Norm

$$\text{MINDIST}(region, q) = \sqrt{\sum_{0<i\le d} \begin{cases} (region.LB_i - q_i)^2 & \text{if} \quad q_i \le region.LB_i \\ 0 & \text{if} \quad region.LB_i \le q_i \le region.UB_i \\ (q_i - region.UB_i)^2 & \text{if} \quad region.UB_i \le q_i \end{cases}}$$



2.2.3 Indexbasierte *k*-Nächste-Nachbarn Anfrage

– Einfache Tiefensuche

- Unterschied zur Range-Query
  - Nächste Nachbar kann beliebig weit vom Anfragepunkt weg liegen
  - Gestalt der Query zunächst unbekannt
  - Es kann zunächst nicht anhand der Seitenregion entschieden werden, ob eine Seite gebraucht wird
  - Ob eine Seite gebraucht wird, hängt auch von dem Inhalt der anderen Seiten ab
  - Kennt man NN-Distanz, würde Range Query ausreichen
  - Kennt man ein beliebiges Objekt, kann man dessen Abstand als obere Schranke für die NN-Distanz nutzen
  - Kennt man mehrere Objekte, kann man den geringsten Abstand als obere Schranke für die NN-Distanz nutzen

- Umformulierung des RQ-Algorithmus:
  - Verwende als ε die kleinste Distanz zu den bisher gefunden Nachbarn

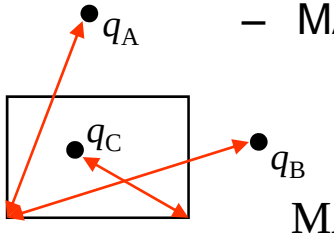
```
Globale Variable: stopdist = +∞;

NN-Index-Simple-TS(pa, q)          // pa = Diskadress z.B. der Wurzel des Indexes
    result = ∅;
    p := pa.loadPage();
    IF p.isDataPage() THEN
        FOR i=0 TO p.size() DO
            IF dist(q, p.getObject(i)) ≤ stopdist THEN
                result := getObject(i);
                stopdist = dist(q, p.getObject(i));
    ELSE                           // p ist Directoryseite
        FOR i=0 TO p.size() DO
            IF MINDIST(q, p.getRegion(i)) ≤ stopdist THEN
                result := NN-Index-Simple-TS(p.childPage(i), q)
    RETURN result;
```

- Nachteil des einfachen Tiefensuch-Algorithmus
  - Initialisierung: stopdist = $+\infty$
  - Dadurch: Start mit beliebigem Pfad
  - Folge: die ersten gefundenen Objekte sind meist sehr weit vom Anfrageobjekt entfernt => stopdist ist wenig selektiv
  - Verbesserung: beginne Pfad, der möglichst nah zum Anfrageobjekt liegt

– Tiefensuche (Depth-first search nach [RKV 95])

[Roussopoulos, Kelley, Vincent. Proc. ACM Int. Conf. Management of Data (SIGMOD), 1995]

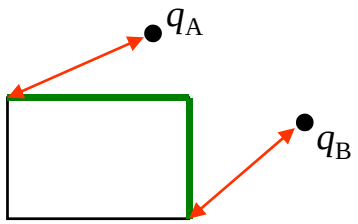
- Vermeidet langsame Einschränkung des Suchraums durch
  - Verwendung der Seitenregionen zur Abschätzung der k-NN-Distanz
  - Priorisierung der Tiefensuche nach Distanz der Seitenregion zur Query
- Neben MINDIST weitere Abschätzungen der NN-Distanz durch:
  - MAXDIST
    - » Maximale Distanz zwischen Query und allen Punkten der Seitenregion
    - » NN-Distanz kann nicht schlechter als MAXDIST werden



$$\mathrm{MAXDIST}(region, q) = \sqrt{\sum_{0<i\leq d} \max\{(q_i - region.UB_i)^2, (q_i - region.LB_i)^2\}}$$

  - MAXDIST aller bisher bekannten Seitenregionen wird zum frühzeitigen Abschneiden von Teilbäumen (Verbesserung der Pruning-Distanz) verwendet.

– MINMAXDIST
  » MBRs als Seitenregionen: maximale NN-Distanz noch besser abzuschätzen
  » Auf jeder Kante des MBR muss ein Punkt liegen (sonst ist MBR nicht minimal)
  » Intuition: „nächstliegende Kante, weitester Punkt“



$$MINMAXDIST(region, q) = \sqrt{\min_{0<k\le d}(|q_i - rm_i|^2 + \sum_{\substack{i\neq k \\ 0<i\le d}} |q_i - rM_i|^2)}$$

wobei $$rm_i = \begin{cases} region.LB_i & \text{if } q_i \le \frac{region.LB_i + region.UB_i}{2} \\ regionUB_i & \text{else} \end{cases}$$

$$rM_i = \begin{cases} region.LB_i & \text{if } q_i \ge \frac{region.LB_i + region.UB_i}{2} \\ regionUB_i & \text{else} \end{cases}$$

– Für andere Geometrien (nicht MBRs) sind MINDIST und MAXDIST analog definierbar; MINMAXDIST allerdings nicht
– Abschätzung von stopdist durch Minimum aus stopdist und MINMAXDIST (bzw. MAXDIST) aller bisher bekannten Seitenregionen (pruningdist)
– Vor dem rekursiven Abstieg: sortieren der Kindseiten nach MINDIST (experimentell als bestes Prioritätsmaß ermittelt)

- Algorithmus (NN-Variante, d.h. für k = 1):

```
Globale Variablen: pruningdist = +∞;
result = ∅; //

NN-Index-RKV (pa, q, k)            // pa = Diskadress z.B. der Wurzel des Indexes
   p := pa.loadPage();
   IF p.isDataPage() THEN
      FOR i=0 TO p.size() DO
         IF dist(q, p.getObject(i)) ≤ pruningdist THEN
            result := p.getObject(i);
             pruningdist = dist(q, result);
   ELSE                            // p ist Directoryseite
      FOR i=0 TO p.size() DO
            IF MINMAXDIST(q, p.getRegion(i)) < pruningdist THEN
            pruningdist = MINMAXDIST(q, p.getRegion(i));
      quicksort(p.getObjectArray(), MINDIST);
      FOR i=0 TO p.size() DO
         IF MINDIST(q, p.getRegion(i)) ≤ pruningdist THEN
            NN-Index-RKV-TS(p.childPage(i), q);
   END IF;
```

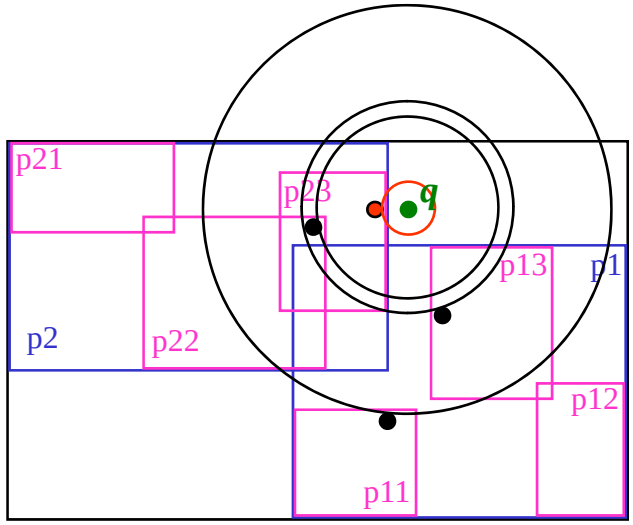
- Algorithmus (kNN-Variante, d.h. für k ≥ 1):

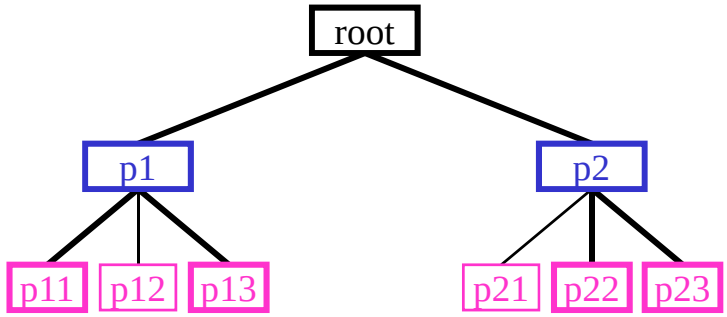
Globale Variablen: pruningdist = +∞;

result = ∅; // Heap mit (oid,dist()) tupel, erster Eintrag hat höchsten dist()-wert, maximale Heapgröße = k (Bei Überlauf wird letztes Element gelöscht)

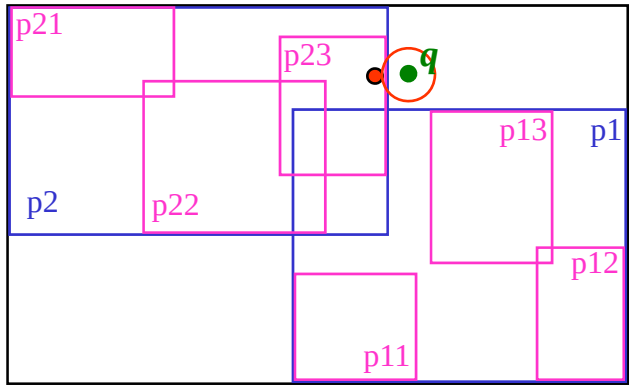
```
k-NN-Index-RKV (pa, q, k)          // pa = Diskadress z.B. der Wurzel des Indexes
  p := pa.loadPage();
  IF p.isDataPage() THEN
    FOR i=0 TO p.size() DO
      IF dist(q, p.getObject(i)) ≤ pruningdist THEN
        result .insert((getObject(i));
        pruningdist = dist(q, result.first);
  ELSE                              // p ist Directoryseite
    FOR i=0 TO p.size() DO    // Vorausgesetzt: k << # Objekte in
                                      // den Teilbäumen unter p
      IF MAXDIST(q, p.getRegion(i)) < pruningdist THEN
        pruningdist = MAXDIST(q, p.getRegion(i));
    quicksort(p.getObjectArray(), MINDIST);
    FOR i=0 TO p.size() DO
      IF MINDIST(q, p.getRegion(i)) ≤ pruningdist THEN
        NN-Index-RKV-TS(p.childPage(i), q);
  END IF;
```

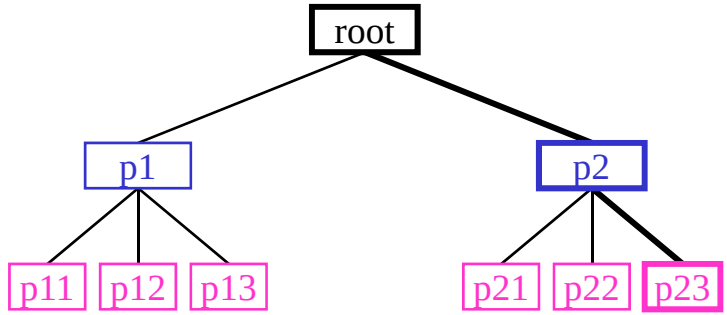
- Ablaufbeispiel
  - Einfache Tiefensuche



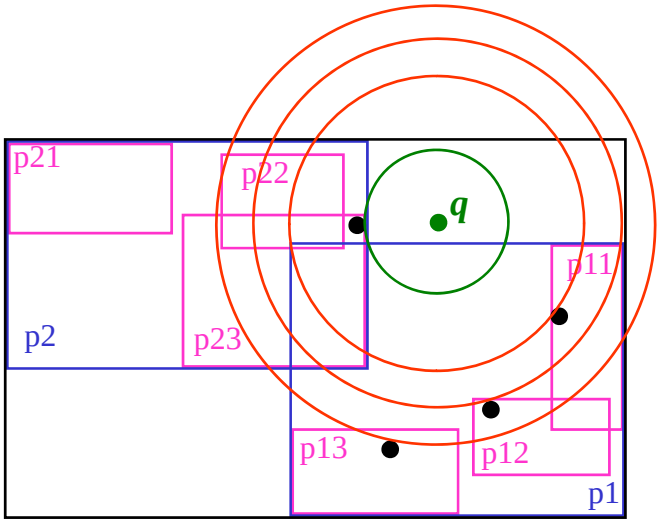


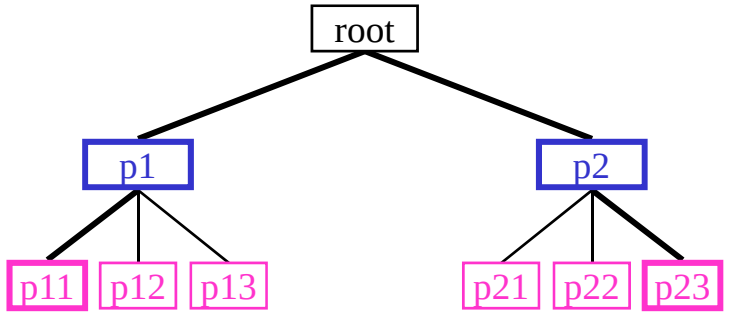
  - Tiefensuche nach RKV





- Fazit:
  - Priorisierung mit MINDIST bewirkt Reduktion der Seitenzugriffe von 7 auf 3
  - MAXDIST (bzw. MINMAXDIST) kann grundsätzlich die Pruning-Distanz verbessern, verhindert hier aber keine Seitenzugriffe
  - Trotz Priorisierung: Tiefendurchlauf kann prinzipiell stark fehlgeleitet werden wenn z.B. eine Seite auf dem ersten Level sehr nah am Queryobjekt liegt, ihre Kindseiten aber relativ weit weg





  - Bei Start mit p_2 hätte keine der Kindseiten von p_1 geladen werden müssen

– Priorität-basierte Suche (Best-first search nach [HS 95]

[Hjaltason, Samet. Proc. Int. Symp. on Large Spatial Databases (SSD), 1995]

- Statt rekursivem Durchlauf: Liste der aktiven Seiten (active page list APL)
  - Seite p ist aktiv genau dann wenn folgende Bedingungen erfüllt sind:
    - » p wurde noch nicht geladen
    - » Elternseite von p wurde bereits geladen
    - » MINDIST(q, p.getRegion()) ≤ pruningdist
  - APL wird mit Wurzel des Indexes initialisiert
  - Seiten in APL nach MINDIST zum Anfrageobjekt aufsteigend sortiert
  - Algorithmus entnimmt immer die erste Seite aus APL (mit kleinster MINDIST)
  - Entnommene Seite wird geladen und verarbeitet: („verfeinert“)
    - » Datenseiten werden wie bisher verarbeitet
    - » Directoryseiten: Kindseiten mit MINDIST ≤ pruningdist in APL einfügen
  - Ändert sich pruningdist werden Seiten mit MINDIST > pruningdist alternativ:
    - » aus APL entfernt
    - » als gelöscht markiert
    - » ohne explizite Markierung später ignoriert

- Algorithmus:

Globale Variablen: pruningdist = $+\infty$;

result = $\varnothing$; // Heap mit (o,dist(q,o)) tupel, erster Eintrag hat höchsten dist()-wert, maximale Heapgröße = k (Bei Überlauf wird letztes Element gelöscht).

k-NN-Index-HS(pa, *q*) // pa = Diskadress z.B. der Wurzel des Indexes

apl = **LIST OF** (dist:Real, da:DiskAdress) **ORDERED BY** dist **ASCENDING**

apl = [(0.0, pa)]

WHILE NOT apl.isEmpty() **AND** apl.first().dist ≤ pruningdist **DO**

p := apl.pop_first_element;

IF p.isDataPage() **THEN …**

* siehe k-NN-Index-RKV *

ELSE // p ist Directoryseite

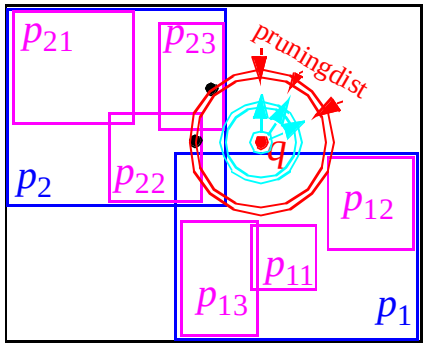
FOR *i*=0 **TO** p.size() **DO**

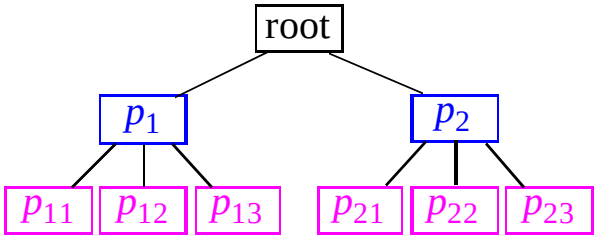
IF MINDIST(*q*, p.getRegion(*i*)) ≤ pruningdist **THEN**

apl.insert(MINDIST(*q*, p.getRegion(*i*)), p.childPage(*i*));

END IF;

- Beispiel





APL:

1. root
2. p_1 p_2
3. p_2 p_{12} p_{13} p_{11}
4. p_{23} p_{22} p_{12} p_{13} p_{11} p_{21}
5. p_{22} p_{12} p_{13}

- Eigenschaften
  - Allgemein
    - » Seiten werden nach aufsteigendem Abstand geordnet zugegriffen (blaue Kreise)
    - » pruningdist wird kleiner, sobald nähergelegenes Objekt gefunden (rote Kreise)
    - » Anfragebearbeitung stoppt, wenn beide Kreise sich treffen. Optimalität!!!
  - Speicherbedarf
    - » Wie bei Breitensuche kann gesamter unterste Directorylevel in APL stehen
    - » Dieser Fall is allerdings unwahrscheinlicher als bei Breitensuche
    - » Speicherkomplexität O(*n*) (Tiefensuche O(log *n*)

- Optimalität des Verfahrens

  [Berchtold, Böhm, Keim, Kriegel. ACM Smp. Principles of database Systems (PODS), 1997]

  – Prioritätssuche nach [HS 95] ist optimal bzgl. der Anzahl der Seitenzugriffe

  – Beweis (Überblick):

    » Lemma 1: jeder korrekte Algorithmus muss mind. die Seiten laden, die von der NN-Kugel um q berührt werden

    » Lemma 2: das Verfahren greift auf Seiten in aufsteigendem Abstand von q zu

    » Lemma 3: keine Seite s wird zugegriffen, mit MINDIST(q, s) > NN-Distanz(q)

  – **Lemma 1:** Ein korrekter NN-Algorithmus muss mind. die Seiten s laden, die MINDIST(q, s) $\leq$ NN-Distanz(q) erfüllen.

    Beweis: Angenommen eine Seite s mit MINDIST(q, s) $\leq$ NN-Distanz(q) wird nicht geladen. Dann kann diese Seite Punkte enthalten (als Datenseite; Directoryseiten können im entspr. Teilbaum Punkte speichern), die näher am Anfragepunkt liegen als der nächste Nachbar. Der nächste Nachbar ist also nicht als solcher validiert, da über Punkte in einem Teilbaum keine Infos bekannt sind, außer dass sie in der entsprechenden Region liegen.

– **Lemma 2:** Das Verfahren greift auf die Seiten des Index aufsteigend sortiert nach MINDIST zu.

  Beweis: Die Seiten werden in aufsteigender Reihenfolge aus der APL entnommen. Es muss also nur sichergestellt werden, dass nach Entnahme von Seite *s* keine Seiten *s'* mehr in APL eingefügt werden, mit MINDIST(*q*,*s'*) < *r* := MINDIST(*q*,*s*). Alle Seiten, die nach Entnahme von s in APL eingefügt werden, sind entweder Kindseiten von *s* oder Kindseiten von Seiten *s''* mit MINDIST(*q*,*s''*) ≥ *r*. Da die Region einer Kindseite in der Region der Elternseite vollständig eingeschlossen ist, ist die MINDIST einer Kindseite nie kleiner als die der Elternseite. Daher haben alle später eingefügten Seiten eine MINDIST ≥ *r*.

– **Lemma 3:** Das Verfahren greift auf keine Seite *s* zu, mit MINDIST(*q*,*s*) > NN-Distanz(*q*).

  Beweis: Nach Lemma 2 können nach Zugriff auf Seite *s* nur Punkte *p* gefunden werden, mit dist(*q*, *p)* > MINDIST(*q*, *s*). Wäre vor Zugriff auf *s* ein Punkt *p* mit dist(*q*, *p)* < MINDIST(*q*, *s*) gefunden worden, dann wäre s aus der APL gelöscht worden bzw. der Algorithmus hätte vor der Bearbeitung von *p* angehalten.

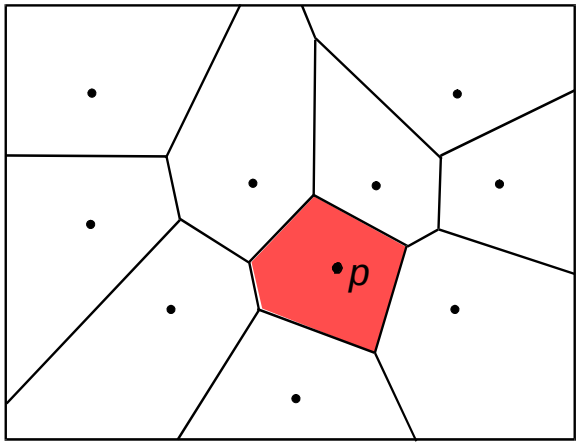
  □

– Aus Lemma 1-3 ergibt sich, dass der Algorithmus nach [HS 95] optimal bzgl. der Anzahl der Seitenzugriffe ist.

– Hybrider Algorithmus: Voronoi-Diagramme

[Berchtold, Ertl, Keim, Kriegel, Seidl. Proc. Int. Conf. Data Engineeering (ICDE), 1998]

- Nur für Vektordaten!!!
- Idee:
  - Berechne für jeden Punkt *p* den Teil des Datenraumes in dem *p* der nächste Nachbar ist (Voronoi-Zellen)
  - Speichere Voronoi-Zellen in DB
  - NN-Anfrage entspricht Punktanfrage mit *q* auf den Voronoi-Zellen
    => Punkt *p*, in dessen Voronoi-Zelle *q* liegt, ist der NN von *q*



- Problem:
  - Voronoi-Zellen sind konvexe Polygone
  - Ab $d > 2$ sehr komplex (große Anzahl Eckpunkte)
- Lösung: Approximation der Voronoi-Zellen (z.B. mit MBR)
  - Nur Filterschritt, da MBRs sich überlappen können, und q in mehrere dieser MBRs liegen kann
    => Verfeinerung der Kandidaten mit exakten Punktdistanzen