

## 공간데이터 관리 및 응용 – Final 정리

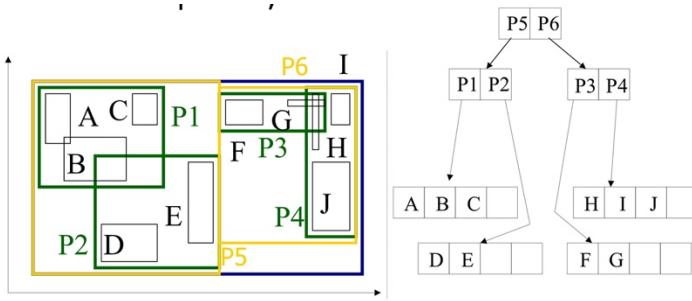
### Ch09. R-Tree

Guttman84에서 처음 소개. 아이디어 : parent 가 overlap 허용.

- Space 활용도 50% 이상 (node의 entry 갯수가 최소 50%) – like B+ tree
- MBR (Minimum Bounding Rectangle) 구성 후 indexing (Easier insertion and split)
- Height balanced (Quadtree 와는 달리 높이가 일정 : all leaf nodes appear on same level)
- Root node는 최소한 두개의 entry
- Multi-way external memory structure.
- indexes MBR
- Dynamic 구조

Fanout : child 최대 갯수

ex) R-tree with fanout 4 : 여러개의 MBR 들이 있을 때 가까이 있는 MBR grouping -> 상위 parent MBR (최대 4 개의 child 가질 수 있음)



- P5, P6 를 갖는 root에는 각각 p5, p6에 해당하는 MBR의 꼭짓점 좌표 저장
- $F = 4 \mid m = 2, M = 4$  (각 노드가 가질 수 있는 entry 갯수는 최소 m 개, 최대 M 개)

R-tree 의 Leaf Node 는  $\{(MBR; obj\_ptr)\}$  의 형태. MBR 정보는 x-low, x-high, y-low, y-high 의 형태로 LowerLeft 좌표와 UpperRight 좌표

R-tree 의 Non-leaf Node 는  $\{(MBR; node\_ptr)\}$  의 형태. leaf node 는 실제 object 를, Non-leaf node 는 child node 를 가리킴

중요한 point:

- 모든 부모 노드가 자식 노드들을 completely cover
- child MBR은 한 개 이상의 부모 노드에게 cover 될 수 있지만 반드시 한개의 부모 아래에만 저장됨 (중복 제거 필요 X)
  - 위의 사진에서 MBR B는 P1, P2에 cover 되지만 완전히 cover 된 P1 아래에 저장됨.
- Point/Window Query 는 multiple path 를 탐색할 수도 있다.

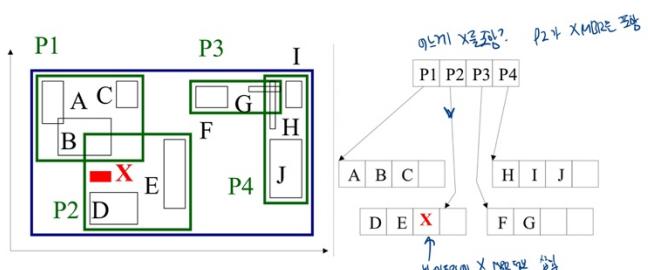
R-tree 의 Search : (Point Query or Window Query)

- Root 부터 Search Window에 겹치는지 꼭짓점 좌표 확인.
- overlap 되는 path 따라 (overlap 안되는 path pruning)
- R-tree 는 경우에 따라 multiple path 를 탐색한다.
- Leaf node 까지 따라가다가 leaf node의 실제 object를 확인해서 결과로

R-tree 의 Insertion : (MBR X 를 추가)

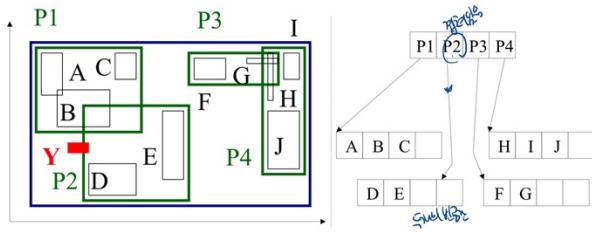
- Root 부터 어느 MBR에 추가하려는 x가 포함되는지를 확인
- 빈 엔트리에 x MBR 정보 삽입

ex1) X 삽입



Root에서 확인 : P2 MBR에 완전히 Cover -> 빈 엔트리에 X MBR 정보 삽입

ex2) Y 삽입

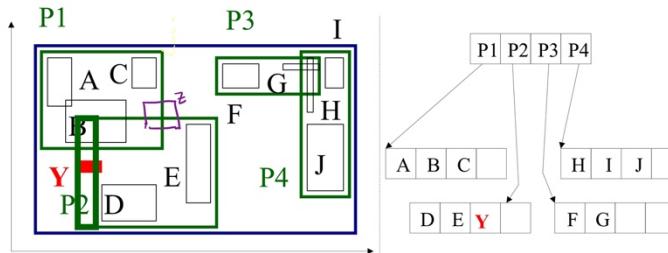


이전 예시의 X 와는 달리 Y 의 경우 P1, P2, P3, P4 MBR 에 완전히 cover 하는게 없다..

→ P2 에 걸쳐져만 있음

1) 같은 방식으로 Y 를 일단 걸치는 P2 MBR 의 빈 엔트리에 삽입한다.

2) Y 삽입 후, parent MBR 인 P2 가 Y 를 완전히 cover 할 수 있도록 parent MBR 를 확장한다.



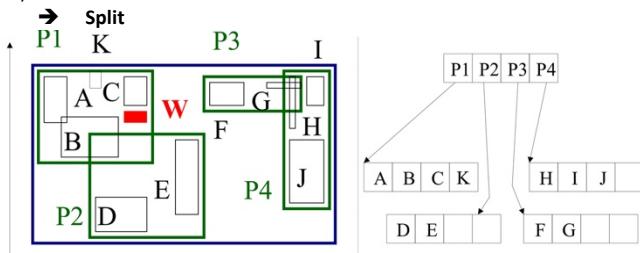
만약 P2 위에 parent 가 더 있다면, 모든 parent 들에 대해 확장을 해야한다.

ex3) 만약 위의 사진에서 Z (보라색 MBR) 같은 MBR 를 삽입해야한다면?

- z 는 P1, P2 모두 에 걸쳐있다.. => 어떻게 결정?

parent 의 확장영역이 작은 entry 로 선택. (P1, P2 를 각각 확장했을 때 더 조금 확장되는 쪽으로 선택!) -> 만약 같다면, random

ex4) Insertion when Node is FULL



두 가지로 split 가능 -> (AB)(CKW) or (ABK)(CW)

- (A,B) (C,K,W) 로 split 된다면, multiple path 가 생긴다. (overlap 구역 생김) -> no (bad split)
- (A,B,K) (C,W)로 split 하는 것이 더 좋다.

Node P1 에 A, B, C, K, W MBR 들이 존재할 때, 이 5 개의 MBR 을 어떤식으로 split 할 것인가? Split 기법 네 가지:

- Plane sweep : 50%의 직사각형 만큼 plane sweep 하면서 check 하여 split
- Linear Split : O(m)
- Quadratic Split : O(m<sup>2</sup>)
- Exponential Split : 만들 수 있는 모든 grouping 을 다 확인. 가장 optimal 한 최적의 split 하지만 시간이 오래 걸림.
  - o  $2^{M-1}$  개의 경우의 수

일반적으로 Linear Split, Quadratic Split 을 많이 사용하는데, 이것은 Seed 를 선택하는데 차이가 있다.

- Split 한 두 개의 노드의 total area 를 최소화
  - Split 한 두 개의 노드의 overlapping 되는 영역을 최소화
- 이 두 가지 요구사항들이 항상 compatible 하지는 않다... 일반적으로 R-tree splitting 은 total area 최소화를 우선시.

Split 에서는 두 개의 직사각형 (대표 MBR)들을 seed 로 선택해서 각 seed 와 closest MBR 를 group 에 추가 한다. 이 때, 포함시킴으로서 증가하는 area 를 최소화하는 것이 “closest”

Seed 를 고르는 방법? (여기서 Linear vs Quadratic 으로 기법이 나뉨) – 2dimension 가정

- **Linear** : 쪽 훑으면서 x 축 입장에서 가장 작은 MBR 과 가장 큰 MBR, y 축 입장에서 가장 작은 MBR 과 가장 큰 MBR. 각 side 의 lowest 와 highest 의 separation 을 normalize 하여 비교. -> 더 큰 쪽으로
  - o sorting 필요 없이 n 개에 대해 한 번 확인 ->  $O(n)$
- **Quadratic** : 모든 E1, E2 pair 들에 대해 직사각형 MBR(E1, E2) 와  $d = J-E1-E2$  를 계산하여 가장 큰 d 값을 가지는 E1, E2 선택
  - o n 개 중에 2 개씩 모든 조합 확인 ->  $nC_2 = O(n^2)$

이러한 방식에 따라, R-tree 의 **Insertion** :

- **ChooseLeaf** 를 통해 entry E 에 삽입할 leaf node 를 찾는다.
- leaf node 가 꽉찼다면, **Split**. 빈자리가 있다면 삽입
- parent node 들에 대해서도 조정

**R-tree Deletion:**

- entry E 를 포함하는 leaf node 를 찾아서 노드에서 E 를 제거.
- 만약 **underflow?** (노드가 가질 수 있는 entry 최소 범위가 존재한다.)
  - o R-tree 에서는 각 노드가 가질 수 있는 entry 갯수 m 이상, M 이하.
  - o 만약 하나를 삭제해서 m-1 이 되는 경우 : **underflow!!**
  - o 이 경우 해당 노드 entry 와 parent entry 를 삭제하고, 나머지 entry 들을 다시 R-tree 에 다시 삽입.

**R-tree 의 여러 종류**

- **R+ 트리** : overlapping 허용 X (internal node 겹치는 것 없음). 반드시 split object
- **R\* 트리** : R-tree 구조에서 삽입/삭제를 최적화. 실생활에 많이 사용됨. (area 뿐만 아니라 perimeter 도 최소화)
- **Hilbert R 트리** : R-tree 의 삽입 순서를 data 의 **Hilbert Value** 순서로 삽입.

**R\*-Tree** : original R-tree 는 enclosing 직사각형의 area 를 최소화하려고 한다. 여기에 추가로 **insertion** 에 있어서 **split** 을 최적화 최적화 영역: (이 모든 것을 동시에 만족하기는 쉽지 않음.)

- 인덱스 MBR 0이 cover 하는 영역
- 디렉토리 MBR 간의 겹치는 영역
- 노드의 directory rectangle 의 둘레 (perimeter) 최소화 : 직사각형의 둘레합 최소화-> 정사각형
- Storage Utilization (노드 entry 최대로 활용)

**R\*-Tree** 에서 노드 삽입을 위한 **ChooseSubtree**: (어떤 branch 로?)

- if next node : leaf node, 고려해야 할 것!
  - o overlap enlargement 최소
  - o area enlargement 최소 (reduced dead space)
  - o smaller area
- if next node is not leaf node (non-leaf node)
  - o area enlargement 최소
  - o smaller area

**SplitNode** : split 할 축을 먼저 선택해서, 선택한 axis 기준으로 두개의 group 으로 split

**ChooseSplitAxis** : 축 선택하는 방법?

- 축별로 직사각형 (MBR)들을 정렬하고,
- 두 개의 group (각 group 이 최소한 m 개의 직사각형을 갖도록) 나눈다.
- 이때, **group 을 분리하는 방법은 최대  $M-2m+2$  개**
  - o M+1 개를 두 개의 group 으로 나누면,
    - (m, M+1-m), (m+1, M+1-(m+1)), ... (m+k, M+1-(m+k))
    - k = M-2m+1, 총 : **M-2m+1 +1**
- 이렇게 나눈 그룹들의 margin-value (둘레) pair 들의 총합 sum 을 계산하여 최소화하는 group 과 축 선택.

**ChooseSplitIndex** : 선택한 축들 중에서 **overlap-value** 를 최소화 하는 grouping 선택

ex)

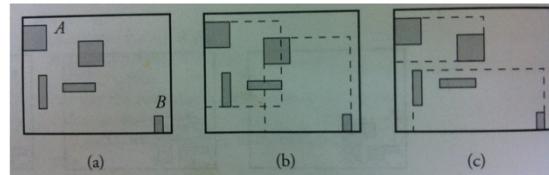


Figure 6.29 Splitting strategies: overflowing node (a), a split of the R-tree (b), and a split of the R\*-tree (c).

- (a)가 overflow 하여 split 을 하기 위해 축 선택을 할때,  
(b)는 R-tree 가 split 하는 방법이다. (x 축을 기준으로) -> overlapping 영역 존재  
(c)는 R\*-tree 가 하는 방식 (y 축 기준으로) -> overlapping 영역 없음

#### Forced Reinsert :

R\*-Tree 에서 insert 때 split 가 생기는 경우, 잠깐 insert 를 멈추고 entry 를 몇 개 삭제하고 MBR overflow 를 줄인 후 재삽입하는 방법

ex)

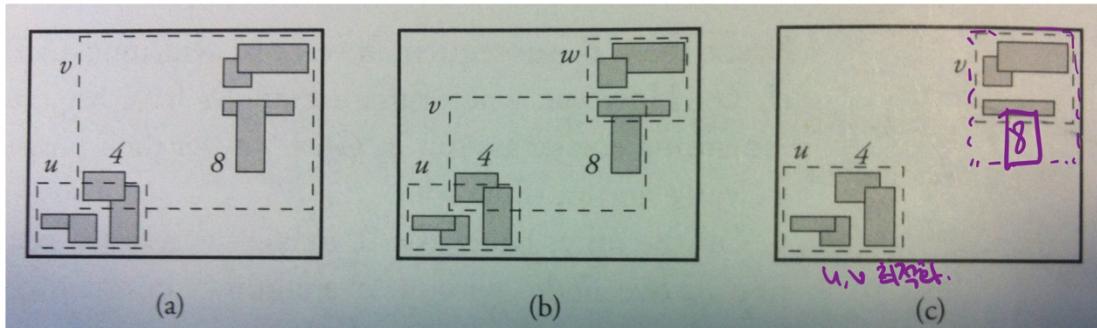


Figure 6.30 The R\*-tree reinsertion strategy: insertion of 8 (v overflows) (a), a split of the R-tree (b), and R\*-tree forced reinsertion of 4 (c).

(a)에서 8 을 삽입함으로서 v node 가 overflow 가 되었을때, R-tree 에서는 (b)그림과 같이 v MBR 를 v 와 w 로 split 을 한다.  
하지만 R\*-tree 에서는 **Forced Reinsert** 를 수행.

- 즉, v 에 8 을 삽입하기 전에 v 에서 임시적으로 4 를 삭제한다. (dead space 를 줄일 수 있음).
- 그러면 (c)에서 보라색 영역과 같이 v MBR 의 크기가 굉장히 줄어든다.
- 이때 4 를 재 삽입한다면, v 가 아닌 u 에 포함되고, 8 을 삽입하면, u,v 가 모두 최적화 된 형태를 보임
- 임시적으로 entry 를 삭제하고 재삽입함으로서 u 와 v 의 MBR 크기 최적화하면서 split 되는...

R 트리와 R\* 트리는 모두 incremental dynamic update 를 지원함. (구조의 발전이 오히려 삽입/삭제가 증가할 수록 성능 저하..)  
만약, MBR 들이 stable with time 인 경우, R-tree 생성전에 data 전처리를 하여 조금 더 optimized R-tree 구조를 만드는 것 가능

#### → R tree packing : STR (Sort-Tree-Recursive) 알고리즘

N : dataset 크기, M : node capacity 일때, P = ceiling of N/M : leaf MBR entry 의 최소 갯수

vertical slice 와 horizontal slice 의 갯수가 둘다 ceiling of Root P 가 되도록.

Packing 방법 : 1) MBR 의 중앙점 좌표를 기준으로 정렬 후, 2) 정렬된 리스트를 ceiling of Root P 개의 group 으로 partition.

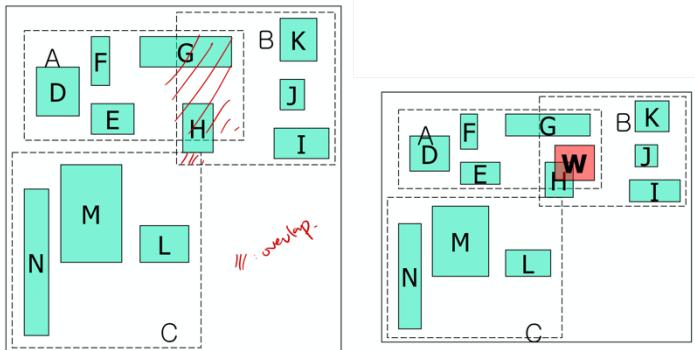
#### → 이런식으로 R-tree 의 upper level 도 재귀적으로 구성

이런 packing 을 통해 모든 leaf 들이 전부 채워지고, space utilization 을 최대로 할 수 있다.

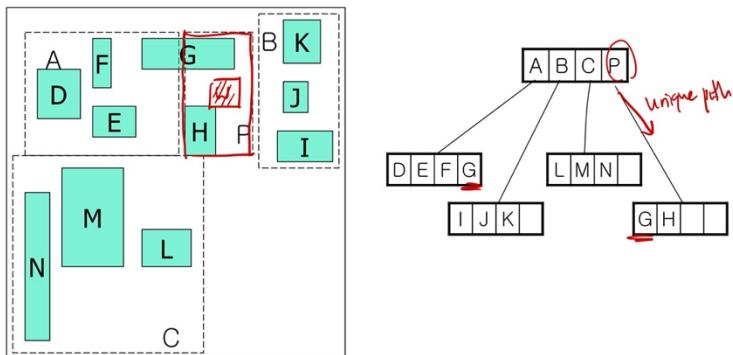
#### R+ Tree :

- Search 에서 multiple path 를 하지 않는다.
- 대신, Object 들이 여러개의 node 에 저장될 수 있음.
- 같은 tree level 에 있는 MBR node 들은 overlap 하지 않는다.
- Insertion, deletion cost/횟수가 많아서 실제로는 R\*tree 에 비해 잘 쓰이진 않고, 구조 유지도 복잡하다는 문제점이 있다.

ex) R-tree 와 R+Tree 를 비교해보면,



이와 같이 R-tree 의 경우 overlap 하는 공간이 존재하고, Window Query W 에 대해, overlapping 하는 부분, A 와 B 를 둘 다 확인해야했다. (Multiple Path 탐색)



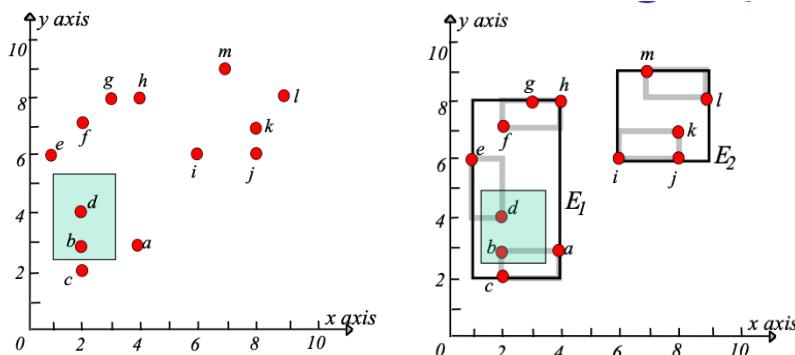
R+Tree 의 경우, 겹치는 부분을 따로 MBR 에 구성한다. (위 사진에서 MBR P 를 새롭게 만듬)

Window Query W 에 대해 unique path 만 탐색해도 된다. 하지만 중복 저장을 한다는 단점 + 실제 성능이 그렇게 좋지는 않다.

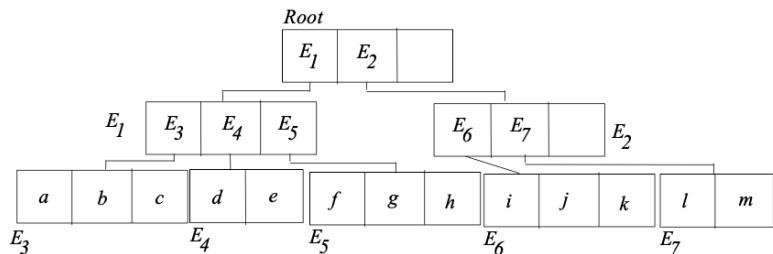
## Ch10. R-Tree Spatial Query Processing

Range Query 를 진행할때, no index 구조에서는 모든 object scan 필요 -> 비효율적

**1. R-tree Range Query :** 다음과 같이 Range Query 가 주어질때, R-tree 구조로 먼저 object 들을 인덱싱 해놓은다면 더 효율적으로 처리



위에서 사용된 clustering 방법은 proximity 이다. 위처럼 MBR 를 구성하였다면, 다음과 같이 R-tree 를 구성할 수 있다.



이 R-tree 에서 Range Query 하는 방법? Root 부터 노드를 확인해서 MBR 좌표 범위 안에 들어가는지 확인

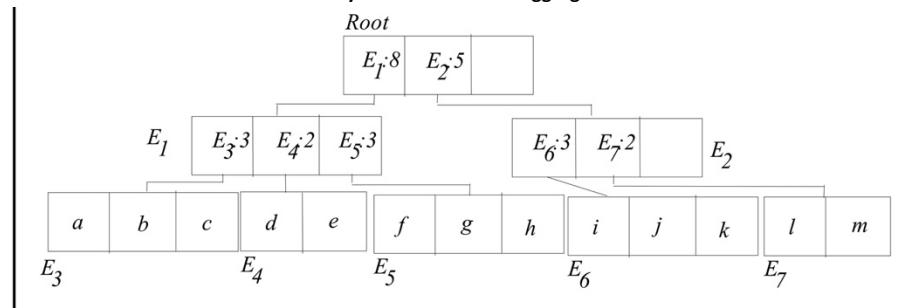
→ 범위 밖인 것 pruning 가능 -> 다음 path 로 이동. (leaf node 까지) -> object 확인

## 2. Aggregation Query? 질의 처리 유형 중 집계 유형 (COUNT, SUM, AVG, MIN, MAX)

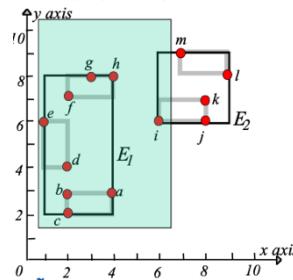
ex) Massachusetts 안에 있는 호텔의 총 갯수

가장 쉬운 방법은 range query 로 처리해서 갯수를 세는 것이겠지만,

애초에 R-tree 를 구성할 때 index entry 에 그 sub-tree 의 aggregate 를 저장해 놓으면 모든 leaf 확인 필요 없다.



위의 R-tree 처럼 sub-tree 의 index 총 갯수를 저장해놓은다면, 탐색 횟수가 줄어든다!!



위의 Aggregation Query 에 대해 원래는 E1 -> E3, E4, E5 와, E2 도 확인해서 E6, E7 도 확인해야했다.

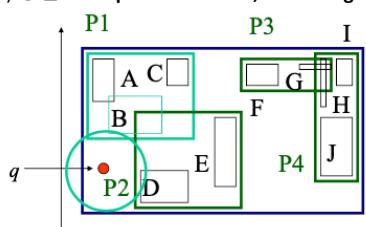
하지만 위처럼 aggregate 값을 저장해놓으면 leaf node 확인 없이 E1 MBR 자체가 모두 cover 되기 때문에 확인 없이 노드 갯수 : 8 과 E2 -> E6, E7 만 확인해 E6 값도 바로 찾을 수 있다. (Subtree pruning)

이처럼 Aggregation 정보 (count 뿐만 아니라, sum, avg 같은..) 들을 추가로 저장해놓으면 탐색 횟수를 줄일 수 있다!!

## 3. NN- Query

NN Query : query location 에 대해 가장 가까운 object 들을 구하는 것 (일반적으로 거리는 euclidean distance)

### 1) 방법 1 : Depth First Search, then Range Query



R 트리를 DFS 를 통해 탐색, q 기준 원을 그려서 Range Query. 원 안에 들어있는 MBR (B,D)를 확인한다.

→ KNN 질의처리를 이렇게 한다고 할 때도, DFS 로 내려가서 k 개의 point 를 확보한 후, 해당 k 개의 point 들을 포함하는 원을 그린다. 이때 반드시 원 이내에 있는 MBR 에 KNN 질의처리 결과가 포함된다.. (당연히 full scan 보다는 빠르다)

### 2) 방법 2: Branch and Bound

- MBR Face Property : “어느 MBR 이던 그 MBR 의 변에 최소 한 개 이상의 실제 spatial object 가 있어야한다. (MBR 특성)”

Branch and Bound 기법은 MBR Face Property 아이디어를 바탕으로 검색 improve...

- MBR 은 필요한 경우에만 방문
- Pruning 할 때는? MINDIST 와 MINMAXDIST 를 활용

**MINDIST (P, R):** point P 와 rectangle R 사이의 최소 distance

- if P is inside R : MINDIST = 0
- if P is outside of R : MINDIST(P, R) = dist(R 의 둘레 중 가장 가까운 점, P 점)

$$\forall o \in R, MINDIST(P, R) \leq \|(P, o)\|$$

→ R 에 있는 가장 closest point 는 최소 MINDIST 이상 떨어져있다...

**MINMAXDIST (P, R)** : point P 와 MBR R 이 있을 때, 직사각형 R 까지의 가까운 거리들 중 가장 먼 거리

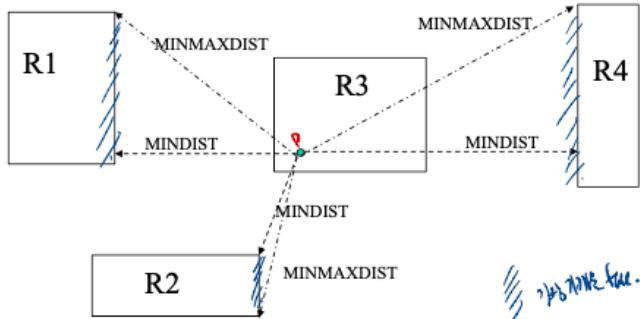
→ 점 P 를 기준으로 R 의 각 dimension 들 중 가장 가까운 변을 찾고, 이 변에서 점까지 max distance (가장 멀리 있는 점까지)

$$\exists o \in R, \| (P, o) \| \leq \text{MINMAXDIST}(P, R)$$

→ smallest possible upper bound of  $\text{dist}(P, R)$

object 와 P 까지의 거리가 MINMAXDIST 이하인 R 내부의 최소한 한 개 이상의 object 가 보장된다. ("R 속에 반드시 object 하나 있다")

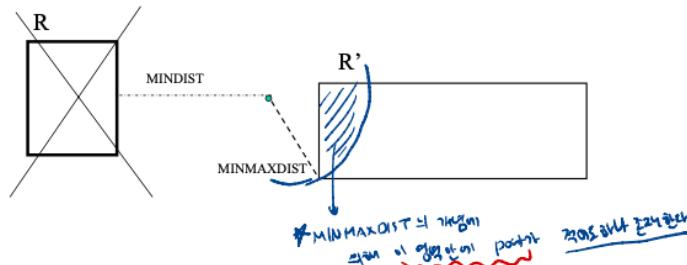
\*  $\text{MINDIST}(P, R) \leq \text{NN}(P) \leq \text{MINMAXDIST}(P, R)$ .



→ NN(P)의 lower bound 와 upperbound 가 생긴다

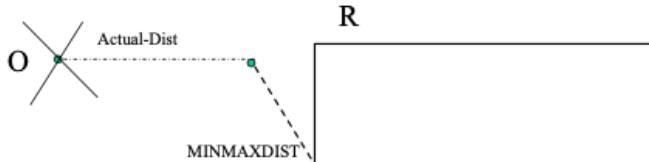
**Pruning in NN Search : Branch and Bound** 세 가지 pruning

1) Downward pruning : An MBR R is discarded if there exists another R' s.t.  $\text{MINDIST}(P, R) > \text{MINMAXDIST}(P, R')$



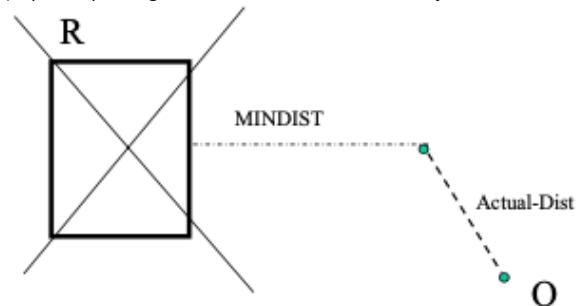
⇒ R 의 어느 점을 가져오더라도 색칠된 영역 안의 point 보다 거리가 멀다. (R pruning 가능)

2) Downward pruning : An object O is discarded if there exists an R s.t. the  $\text{Actual-Dist}(P, O) > \text{MINMAXDIST}(P, R)$



Pruning 1 과 같은 이유로 object 와의 실제 distance 가 MINMAXDIST(P, R)보다 작은 R 이 있다면 object pruning 가능.

3) Upward pruning : An MBR R is discarded if an object O is found s.t. the  $\text{MINDIST}(P, R) > \text{Actual-Dist}(P, O)$



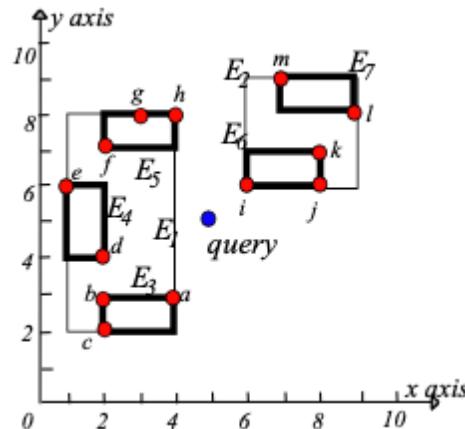
Pruning 2 와 비슷하지만, 실제로 어느 object 까지의 거리가 MINDIST(P, R)보다 작다면 MBR R pruning 가능 (당연한 것)

정리하자면, MINDIST : optimistic best lower bound, MINMAXDIST : pessimistic upper bound

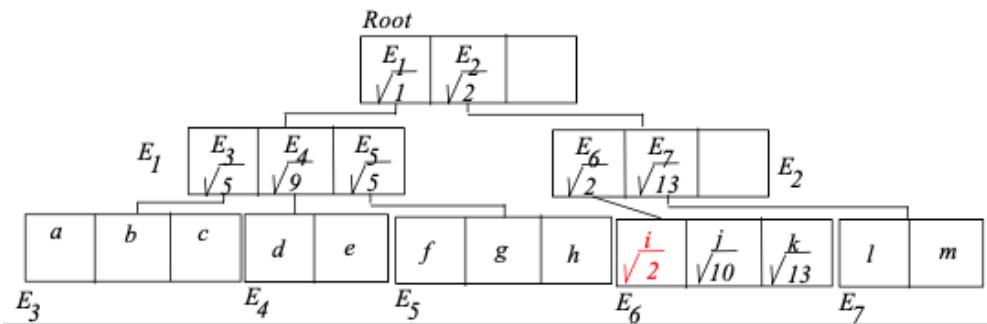
**NN Search Algorithm:** (Priority Queue(Heap)과 MINDIST, MINMAXDIST 활용)

1. Init NN distance = INF
2. Traverse R-tree with DFS, starting from the Root.
  - 각 Index Node에서 MBR들을 정렬해서 Active Branch List (ABL) Heap 구조에 저장한다.
3. ABL에 pruning rule 1,2를 적용
4. ABL에 있는 MBR들을 하나씩 모두 방문.
5. Leaf Node라면 실제 Actual Distance를 계산해서 best NN과 비교해서 update 한다.
6. Recursion에서 돌아오면서 pruning rule 3을 적용
7. ABL이 비어있을 때 NN Search 끝

K-NN Search : k 개의 가장 가까운 Neighbor들을 정렬해놓은 buffer 유지. (k 번째 거리를 이용해 pruning)  
 ex) 다음 예시에서 query point에 대해 NN 구하기



위의 object들의 MBR들을 R-tree로, (q 와의 MINDIST를 같이 저장함)



- ABL에는 각 노드들의 entry들이 MINDIST 순으로 정렬되어 있다. (MINHEAP 구조로)
- 작은 순서대로 DFS 탐색.
- NN 후보는  $C = \{\}$ 에 저장할 예정.

위의 예시 NN Search 순서:

- NN = INF 인 상태에서 Root부터 비교. E1의 MINDIST가 더 작기 때문에 E1부터 탐색
- E1의 ABL에는 {E3, E5, E4} 순서로 저장되어 있음. E3부터 leaf node 확인
  - a부터 actual distance 확인하면서 업데이트.  $a = \sqrt{5}$ ,  $C = \{a\}$
  - $DIST(q, b) = \sqrt{13} > \sqrt{5}$
  - $DIST(q, c) = \sqrt{9} > \sqrt{5}$ .
- 다음으로 E5의 MINDIST는  $\sqrt{5}$ 와 같기 때문에 확인(만약 NN이라면 E5도 볼 필요 없음) -> 변화 없음
- E4의 MINDIST인  $\sqrt{9} > \sqrt{5}$ 이므로 upward pruning 가능 (E4 확인 필요 X)
- 다음 Root의 다음으로 저장된 E2도 같은 방식으로 확인. ABL: E6, E7 순서
- E6는  $\sqrt{2}$ 로 더 작기 때문에 탐색
  - i, j, k의 actual distance를 보면  $\sqrt{2}$ 보다 작은 i로 NN 없데이트.  $C = \{i\}$
- E7은  $\sqrt{2}$ 보다 크기 때문에 pruning.

최종 NN(q) = i

if K-NN 이었다면, Candidate = {}에 K 개 유지. 탐색 횟수도 그거에 맞춰서 진행.

3) 방법 3 : Best-First

R-tree 를 DFS 뿐만 아니라 Best First 로도 가능

Global 한 Heap 을 사용해서 MINDIST 순서로 정렬된 index entry 와 object 를 저장.

노드를 굳이 방문하지 않더라도 Heap 제일 앞 방문을 통해 빠르게 접근 가능...

- element 가 index entry 라면, insert children into H.
- element 가 object 라면 NN 으로 해당 obj return.

Action	Heap
Visit Root	$E_1 \sqrt{1}   E_2 \sqrt{2}   \quad   \quad   \quad   \quad  $
follow $E_1$	$E_2 \sqrt{2}   E_3 \sqrt{5}   E_5 \sqrt{5}   E_4 \sqrt{9}   \quad   \quad  $
follow $E_2$	$E_6 \sqrt{2}   E_3 \sqrt{5}   E_5 \sqrt{5}   E_4 \sqrt{9}   E_7 \sqrt{13}   \quad  $
follow $E_6$	$i \sqrt{2}   E_3 \sqrt{5}   E_5 \sqrt{5}   E_4 \sqrt{9}   j \sqrt{10}   E_7 \sqrt{13}   k \sqrt{13}  $

ex) 위의 예시를 이처럼 Heap 으로 유지.. 마지막에 i 는 object, NN 으로 return 후 terminate

당연히 리소스만 풍부하다면 **DFS << Best First (훨씬 빠름)** : like 서버 환경

하지만 서버가 아닌 스마트폰 같은 곳에서는 리소스 부족 (힘이 너무 커짐) : **DFS** 가 유리..

#### 1. Pruning 1 in NN Query:

object o 를 찾았을때, MBR 중에  $\text{MINDIST} > d(o, q)$  인 MBR 들은 모두 prune

#### 2. Pruning 2 using MINMAXIST:

object 를 찾기도 전이라도,  $\text{MINDIST}(q, E1) > \text{MINMAXDIST}(q, E2)$  인  $E2$  가 존재한다면 MBR  $E1$  도 prune 가능

#### 4) 방법 4 : NN Full-Blown Algorithm

Best First 에 MINMAXDIST 개념을 추가해서 pruning 을 추가하는 것.

ㄱ. MINDIST 순서로 global heap H 를 저장하기.

ㄴ. H 는 처음에는 root 를 저장

ㄷ.  $\delta = +\infty$  로 설정 (첫번째 entry 는 반드시 heap 저장)

ㄹ. While H not empty:

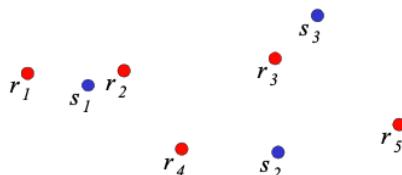
- minimum MINDIST 를 갖는 element e 를 뽑아낸다.
- if object : return as NN
- for every entry se in PAGE(e) whose  $\text{MINDIST} \leq \delta$  :
  - o insert se into H
  - o  $\delta = \text{MINMAXDIST}(q, se)$

#### Best-First vs Branch and Bound

- Best First : optimal 알고리즘 (딱 필요한 노드만 방문한다.)
  - o 하지만 하나의 거대한 우선순위 큐를 저장해야하고, Priority queue 가 커지면 thrashing 문제.
  - o 서버 환경과 같이 리소스 풍부할 때 가능
- Branch and Bound : 각 노드 별로 작은 리스트를 사용함
  - o MINMAXDIST 를 통해 어떤 entry 들을 pruning 함
  - o 리소스가 적은 모바일 환경에서는 더 좋다.

#### 4. Closest Pair Query

두 개의 object set R, S 에 대해 최소 distance 를 갖는 pair of objects (r in R, s in S) 를 구하기.



ex) 이 사진에서 CP = (r2, s1)

실생활 예시 : 가장 가까운 Hotel-Bar pair 를 구하기

CP 도 역시 R, S 를 같은 높이의 R-tree 로 인덱싱 한 후 NN Query 와 비슷하게 적용 가능.

MBR 들의 pair 들끼리 MINDIST 와 MINMAXDIST 를 구한다.

#### CP 기본 알고리즘 :

1) Index entry 의 pair 와 object 의 pair 들이 MINDIST 순서로 정렬되어 저장된 heap H

2) 처음에 root 의 pair 를 H 에 저장

3) while H is not empty:

- minimum MINDIST 값을 갖는  $(e_R, e_S)$  pair 를 추출
- if pair of object : return as CP
- for every entry  $se_R$  in PAGE( $e_R$ ) and every entry  $se_S$  in PAGE( $e_S$ ):
  - o Insert  $(e_R, e_S)$  pair into H

#### CP Full-blown 알고리즘:

1) Index entry 의 pair 와 object 의 pair 들이 MINDIST 순서로 정렬되어 저장된 priority queue H

2) 처음에 root 의 pair 를 H 에 저장

3)  $\delta = +\infty$  로 설정 (첫번째 entry 는 반드시 heap 저장)

4) while H is not empty:

- minimum MINDIST 값을 갖는  $(e_R, e_S)$  pair 를 추출
- if pair of object : return as CP
- for every entry  $se_R$  in PAGE( $e_R$ ) and every entry  $se_S$  in PAGE( $e_S$ ) whose MINDIST  $\leq \delta$ :
  - o Insert  $(se_R, se_S)$  pair into H
  - o Decrease  $\delta$  to MINMAXDIST( $se_R, se_S$ )

### 5. Close Pair Query

Closest Pair Query : 가장 가까운 하나의 pair 찾는 query

Close Pair Query 는 threshold  $\alpha$ 값을 두고 그 이내의 pair 들을 모두 찾는 것.

#### Close Pair Algorithm :

a. Push pair of root nodes into stack

b. while stack is not empty:

- Pop a pair  $(e_R, e_S)$  from stack
- for every entry  $se_R$  in PAGE( $e_R$ ) and  $se_S$  in PAGE( $e_S$ ) where  $MINDIST(se_R, se_S) < \alpha$ :
  - o if  $se_R$  is an index entry (MBR 간의 pair 이면): Push  $(se_R, se_S)$  into stack
  - o else (object 간의 pair 이면): return  $(se_R, se_S)$  as one close pair

## Ch11. M-tree

Curse of Dimensionality : 차원이 높아질수록 부피는 더욱 빠른 속도로 증가함 (data 는 더욱 sparse 해진다..)

→ 고차원에서는 data density 가 sparse 해지기 때문에 grouping 이 어려워짐

→ 기존의 R-tree 는 MBR 로 grouping 해야하는데 고차원에서는 어려움. (index 활용도도 떨어짐)

The M-tree:

- Inherently dynamic structure
- Disk oriented
- 모든 데이터/object 들은 leaf node 에 저장되고,
- Internal Node 들은 Pointers to subtrees + 추가정보들.

Dissimilarity measure : distance function  $d(O_i, O_j)$  는 다음과 같은 metric axiom 을 만족:

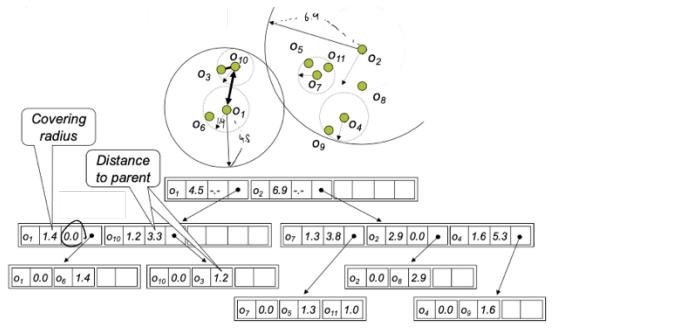
- Reflexivity :  $d(O_i, O_i) = 0$
- Positivity :  $d(O_i, O_j) > 0$  , ( $i \neq j$ )
- Symmetry :  $d(O_i, O_j) = d(O_j, O_i)$
- Triangular Inequality :  $d(O_i, O_k) + d(O_k, O_j) \geq d(O_i, O_j)$

#### Metric Indexing:

- general metric space 로 object indexing (vector space 뿐만 아니라 metric space 도)
- Curse of dimensionality 에 대응

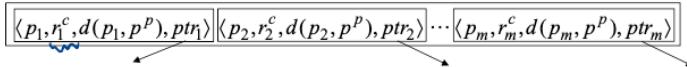
#### M-tree 의 Node :

- 1) Ground Objects (in Leaf Node) : data objects
- 2) Routing Objects (Inner Nodes) : metric regions



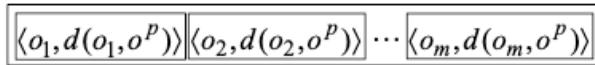
#### Internal Node : entry of each subtree

- Pivot :  $p$  (원의 중심)
- sub-tree 를 cover 하는 radius  $r^c$
- Distance from  $p$  to parent pivot  $p^p$  :  $d(p, p^p)$
- Pointer to sub-tree :  $ptr$



#### Leaf Node : data entry

- Object (identifier) :  $o$
- Distance between  $o$  and its parent pivot :  $d(o, o^p)$



### M-tree Insertion:

#### 1) Single-way Insertion

Object Insertion 을 할 때, object 를 완전히 포함하는 node 하나, 혹은 가장 가까운 (center 가) 노드를 하나 선택해서 process

→  $O(\log N)$

#### 2) Multi-way Insertion

object 를 포함하는 모든 non-full leaf 를 확인

→  $O(n)$

Insert 를 할 때는, 삽입하려는  $O_N$  에 가장 적합한 leaf 를 찾아서 recursively descend the tree.

- 각 step 마다 pivot  $p$  를 가지는 subtree 를 선택할 때, 1) radius  $r^c$  의 증가가 없는 쪽으로(만약 동률이라면 object 와 pivot  $o$  더 가까운 쪽) 2) 혹은 최소화 하는 쪽으로 선택한다.
- leaf node  $N$  에 도달했다면, 1)  $N$  이 아직 남은 공간이 있다면 저장, 2) if  $N$  is FULL ,  $\text{Split}(N, o_N)$

**Split (N, ON):** 삽입하려는 leaf node 가 overflow 인 경우,

- Let  $S : N$  과  $O_N$  을 모두 포함하는 set
- Select pivots  $p_1, p_2$  from  $S$
- Partition  $S$  to  $S_1$  and  $S_2$ , according to the pivots  $p_1, p_2$
- $N$  과  $N'$  (새로 할당하는 노드)를 파티션된  $S_1, S_2$  에 각각 저장
- 1) 만약  $N$  이 root : allocate new root, 거기로 store entries for  $p_1, p_2$
- 2) non-root node : ( $N$  의 parent Node 를  $N^p$ , parent pivot :  $p^p$  라고 가정)
  - o entry  $p_p$  를  $p_1$  과 교체
  - o  $N_p$  가 꽉찼다면, 다시 한번  $\text{Split}(N^p, p_2)$
  - o  $N_p$  에 자리가 있다면,  $p_2$  를  $N^p$  에 저장

### Pivot Selection:

#### Pivot 선택하는 방법:

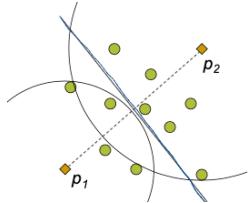
- RANDOM : pivot  $p_1, p_2$  랜덤하게
- m\_RAD : minimum  $(r_1^c + r_2^c)$  를 갖는  $p_1, p_2$  선택
- mM\_RAD : minimum max( $r_1^c, r_2^c$ ) 를 갖는  $p_1, p_2$  선택
- M\_LB\_DIST :  $p_1 = p^p, p_2 = o_i \mid \max \{d(o_i, p^p)\}$ 
  - o pivot  $p^p$  에서 가장 멀리 떨어진  $o_i$  를 선택한다. (이때, 이 거리는 pre-computed, 이미 계산이 되어 있다)

### Pivot selection 두 개의 버전:

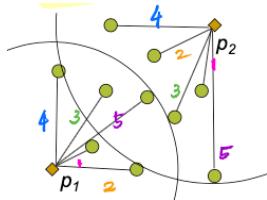
- **Confirmed** : original pivot  $p^p$  를 재사용하고, 한 개만 선택
- **Unconfirmed** : 두 개의 pivot 을 선택하는 방법 (notation : **RANDOM\_2**)
  - o  $p^p$  를 재활용하는 M\_LB\_DIST 방식이 Confirmed, 나머지 방식은 Unconfirmed

### Split Policy :

- **Unbalanced** :  $p_1, p_2$  의 직각 이등분선을 그어서 (generalized hyperplane). 그 선을 기준으로 나누고, 각각  $r^1_c, r^2_c$  구함



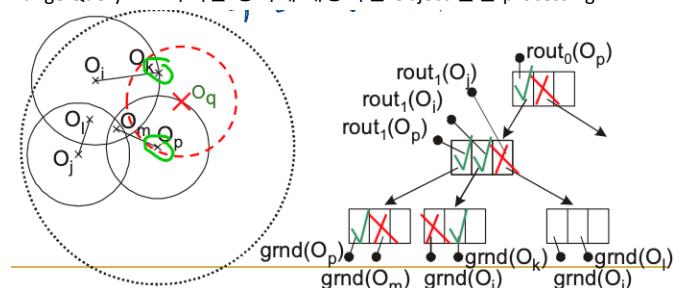
- **Balanced** :  $p_1, p_2$  에서 각각 가까운 것 하나씩 나눠 갖는 (cover 영역이 커짐->overlap 영역도 커짐, unbalanced 보다 별로)



### Similarity Query in M-tree (Range Query에서 Prune 하는 방법 )

(한마디로 안겹치는 것을 어떻게 prune 할것인가?)

Range Query : 교차되는 영역에 해당하는 Object 들만 processing



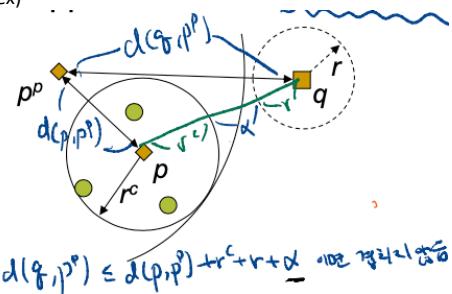
$R(q, r)$ 에 대해, DFS 형태의 tree traversal.

1. Internal Node 에서는, 각 entry  $\langle p, r^c, d(p, p^p), \text{ptr} \rangle$  에 대해,  $(d(p, p^p)$  같은 값은 이미 M-tree 에서 계산되어있음)

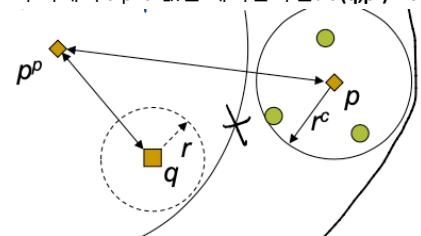
(1) pivot-pivot constraint 를 통한 pruning (parent pivot 이 사용되기 때문에 root 가 아닌 internal 에서)

$|d(q, p^p) - d(p, p^p)| - r^c > r$  인 경우, prune subtree (겹치지 않는 것을 확인할 수 있음)

ex)



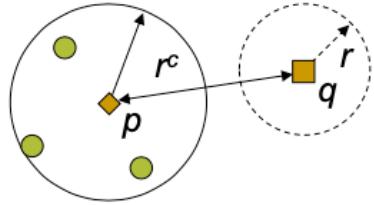
이 식에서 alpha 값을 제거한다면?  $d(q, p^p) > d(p, p^p) + r^c + r$  (겹치지 않음)



이 식에서도  $d(p, p^p) > d(q, p^p) + r + r^c$  (겹치지 않음)

(2) Range-pivot constraint 를 통한 pruning (일반적으로 root node에서)

$|d(q,p) - r^c| > r$  인 경우, Prune Subtree (두 원이 겹치려면 부등호가 반대가 되어야함.)

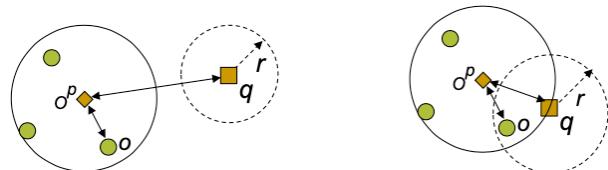


이후 (1), (2)에서 non-pruned entry 들은 search recursively

2. Leaf Node 들에 대해서는, 각 entry  $<o, d(o,o^p)>$ 에 대해

(1) Object-pivot constraint 를 통한 pruning

$|d(q,o^p) - d(o,o^p)| > r$  인 경우, entry 무시. (아닌 경우  $d(q,o)$ 를 계산하여 r 이하인지 확인)



#### M-tree k-NN Search:

k-NN (q)에 대해, priority queue 와 range search 의 pruning 기법을 활용한다.

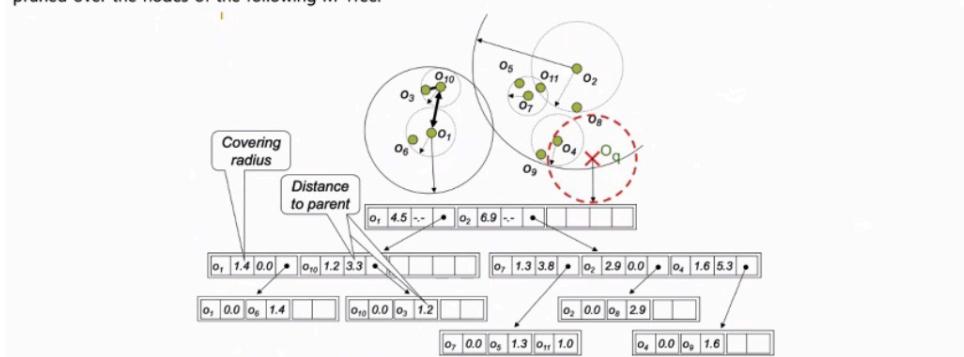
- Priority Queue : object 가 발견될 수 있는 sub-tree 의 pointer 를 저장한다.
- ex) entry E =  $<p, r^c, d(p,p^p), \text{ptr}>$ 에 대해 pair  $<\text{ptr}, d_{\min}(E)>$  를 저장한다.
  - o 이때,  $d_{\min}(E) = \max \{d(p,q) - r^c, 0\}$ . (즉 p 와 q 사이 거리)
- 범위질의때처럼 고정된 반지름 r 이 아닌, k 번째 NN 까지의 distance 를 사용한다.

M-tree 질의 처리에서는 직접적인 거리 계산은 최소로 한다.

- root level 에서만 직접적인 거리 계산이 이루어지고, internal node 들에서는 이 값들을 재활용함

#### ex) M-tree Range Search 예시)

Example: We are going to perform a range search  $R(O_q, 3.0)$  (i.e., a red dotted circle centered at  $O_q$  with radius 3.0) over the following M-tree. Given  $d(O_q, O_1) = 11.5$ ,  $d(O_q, O_2) = 9.7$ ,  $d(O_q, O_3) = 2.8$ , and  $d(O_q, O_9) = 3.1$ , show how the range search are pruned over the nodes of the following M-Tree.



#### 1) Root node O1에 Range-pivot constraint 적용

O1 entry :  $d(O_q, O_1) - 4.5 = 7 > 3$  so prune O1's subtree

O2 entry :  $d(O_q, O_2) - 6.9 = 2.8 < 3$  so cannot prune

#### 2) Internal Node O2에 대해 Pivot-pivot constraint 적용

O7 entry :  $|d(O_q, O_2) - d(O_7, O_2)| - r^c = |9.7 - 3.8| - 1.3 = 4.6 > 3.0 \Rightarrow$  so prune O7's subtree

O2 entry :  $|d(O_q, O_2) - d(O_2, O_2)| - r^c = |9.7 - 0| - 2.9 = 6.8 > 3.0 \Rightarrow$  so prune O2's subtree

O4 entry :  $|d(O_q, O_2) - d(O_4, O_2)| - r^c = |9.7 - 5.3| - 1.6 = 2.8 < 3.0 \Rightarrow$  can't prune O4

#### 3) Leaf node O4의 entry 들에 대해 Object-pivot constraint 적용

O4 entry :  $|d(O_q, O_4) - d(O_4, O_4)| = |2.8 - 0.0| = 2.8 < 3.0 \Rightarrow$  cannot prune

→ check  $d(O_q, O_4) = 2.8 < 3.0 \Rightarrow O_4$  is in Range

O9 entry :  $|d(O_q, O_4) - d(O_9, O_4)| = |2.8 - 1.6| = 1.2 < 3.0 \Rightarrow$  can't prune

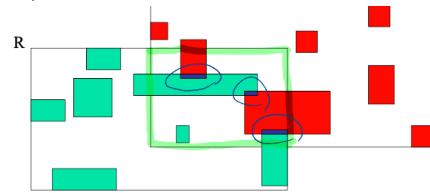
→ check  $d(O_q, O_9) = 3.1 > 3.0 \Rightarrow O_9$  is not in Range

## Ch12. Spatial Join

Spatial Join 은 두 영역간의 overlap 되어있는 MBR 들을 찾기 위한 join .

### 1. R-tree based Join

ex) R 에 포함된 초록색 MBR 들과 S 에 포함된 빨간색 MBR 들 중 overlapping 하는 MBR 은 무엇인가?



#### 1) Join1 (R,S)

가장 단순한 방법 : 모든 pair-wise 확인하는 방법 (R 과 S 의 depth 가 동일하다는 가정하에)

Join(R,S):

Repeat until all pairs are examined :

- R 의 entry E 와 S 의 entry F 중 intersecting 하는 entry pair 를 찾는다
- if R 과 S 가 leaf page: add (E, F) to result
- else : Join1(E,F) (Recursive 호출)

모든 pair 확인하면서 재귀호출이 많다-> 매우 비효율적.

CPU-time 줄일 수 있는 방법?

- Search Space 제한하기 (R,S 의 범위는 아니면 겹치는 부분만 check)
- Spatial Sorting + Plane Sweep

#### 2) Join2(R, S, IntersectedVol)

검색 범위를 줄여서 겹치는 MBR 들에 대해서만 검색하여 비교 횟수를 줄이는 방법

Join2(R, S, IV):

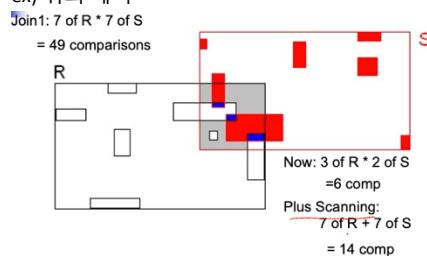
Repeat until all pairs are examined:

- R 의 E 와 S 의 F entry 들에 대한 intersecting pair 중에 IV 와 overlap 되는 pair 찾기
- if R 과 S 가 leaf page 라면: add (E,F) to result
- else : Join2(E, F, CommonEF)

두 MBR R 과 S 의 overlapping 영역과 겹치는 MBR pair 확인 하는 것.

Join2 의 일반적인 비교 횟수 = size(R) + size (S) + relevant (R)\*relevant(S)

ex) 위의 예시



Join1 때는 모든 pair 확인 :  $7 \times 7 = 49$  회 비교

Join2 는 : R 과 S 중에 intersecting 되는 영역에 해당하는 MBR 들 각 R, S 에서 찾기 (7+7 scans)

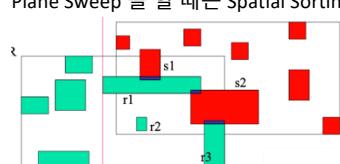
+ 해당 MBR 들에 대한 pair 비교 (R 에서 3 개, S 에서 2 개 :  $3 \times 2 = 6 \text{ comp}$ ) = 20 회 연산 (Join1 보다 훨씬 감소)

여기서 relevant(R) \* relevant(S) 부분, 즉 해당되는 MBR 들을 각각 pairwise 로 (3x2 회) 비교하는 부분을 더 효율적으로

→ Use Plane Sweep

#### 3) Plane Sweep 활용

Plane Sweep 을 할 때는 Spatial Sorting 이 필요함.



intersecting 하는 area 의 lower-left corner 의 x 좌표를 기준으로 영역 내의 MBR 들을 sorting

→ [r1, r2, s1, s2, r3]

- ㄱ. 첫 번째 entry 인  $r_1$  부터 시작해서  $x$  축을 따라서 vertical line 을 sweep. (sweep 범위 :  $r_1$  의  $lower\_x \sim upper\_x$ )
- ㄴ. sweep line 을  $r_1$  의  $lower\_x \sim upper\_x$  까지 움직이면서 lower\_left 겹치는 빨간 MBR ( $s_n$ ) 있는지?
- ㄷ. 있다면 y range 확인,  $r_1$  과 해당  $s_n$  가 intersect 한다면 add to result

다음 entry 들에 대해 계속 (sweep line 다시  $r_2$  의  $lower\_x$  부터 reposition)

위의 그림 예시에 대해서?

$r_1 : s_1$  과 겹침,  $s_2$  와 겹침 (2)

$r_2 : s_1$  과  $x$  겹치지만  $y$  비교 했을 때 실패 ( $s_2$  비교  $X : out of x\_range$ ) (1)

$s_1$  : 비교 대상 없음 (0)

$s_2$  :  $s_2$  의  $x$  부터 시작해서  $r_3$  가 겹친다.  $y$  비교 후 성공  $\rightarrow r_3$  와 겹침 (1)

$r_3$  : 비교 대상 없음 (0)

⇒ 총 네 번의 비교 연산 (Join2 때의  $3 \times 2$  에 비해 줄일 수 있었다.)

정렬이 base 로 있기 때문에 시간복잡도 :  $O(n \log n)$

### Depth 가 다른 두개의 R-tree 를 사용한 Spatial Join Algorithm

```

SJ(R1, R2:R_NODE);
01      BEGIN
02          FOR (all Er2 in R2) DO
03              FOR (all Er1 in R1) DO
04                  IF (overlap(Er1.rect, Er2.rect)) THEN
05                      IF (R1 and R2 are leaf pages) THEN
06                          output(Er1.oid, Er2.oid)
07                      ELSE IF (R1 is a leaf page) THEN
08                          ReadPage(Er2.ptr);
09                          SJ(Er1.ptr, Er2.ptr)
10                      ELSE IF (R2 is a leaf page) THEN
11                          ReadPage(Er1.ptr);
12                          SJ(Er1.ptr, Er2.ptr)
13                      ELSE
14                          ReadPage(Er1.ptr), ReadPage(Er2.ptr);
15                          SJ(Er1.ptr, Er2.ptr)
16                      END-IF
17                  END-IF
18              END-FOR
19          END-FOR;
20      END.

```

## 2. Spatial Hash Join

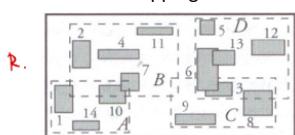
Spatial Hash Join 은 크게 Hash join based on data-driven structure (with overlapping) 과 Hash join based on space-driven structure (with redundancy) 로 나눌 수 있다. 즉, data 를 중심으로 grouping 해서 overlap 은 존재하지만 data 의 중복은 허용 x 하는 경우와 공간을 분할하여 overlap 은 없지만 중복은 존재하는 경우로 나뉜다.

### 1) Hash Join based on Data-driven structures (with overlapping)

#### a. Initial Partitioning (R partitioning) :

각 bucket 은 비슷한 숫자의 rectangle (MBR)을 갖도록 하고, 메모리에 맞도록 한다.

bucket 간의 overlapping 은 최소화 한다.

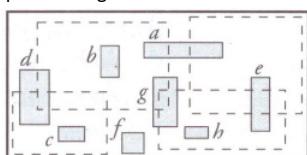


Content of the buckets

$A = \{1, 14, 10\}$ ,  $B = \{2, 4, 7, 11\}$ ,  
 $C = \{3, 8, 9\}$ ,  $D = \{5, 6, 12, 13\}$

#### b. Second Partitioning (S partitioning):

R partitioning 때 만든 bucket 그대로 S partitioning. S 의 rectangle 들과 overlap 하는 bucket 이 있다면, 그 bucket 에 저장. (중복 허용)



$A' = \{c, d\}$ ,  $B' = \{a, b, d, g\}$ ,  $C' = \{e, g, h\}$ ,  $D' = \{a, e\}$

### c. Join phase

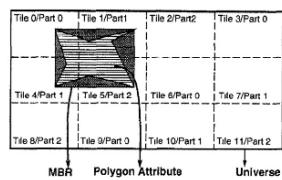
이렇게 얻은 두 bucket set 이 있고, R 과 S 의 각 bucket 은 같은 범위와 위치를 갖는다.

각 bucket 의 rectangle 에서 plane-sweeping 알고리즘을 통해 test. (추가 최적화)

[A, A'], [B, B'], [C, C'], [D, D'] 를 join 하여 각각 겹치는것만 check.

### 2) Hash Join based on Space-driven structures (with redundancy)

R,S 모두 replication 있어 partition 됨. Space is regularly tiled, 이 타일들에 대응하여 파티셔닝이 이루어지거나 이 타일을 기준으로 hashing 하여 partition 함.



결과에 중복이 생기기 때문에 마지막에 결과를 정렬해서 중복 제거하는 작업 필요함

## 3. Z-Ordering Spatial Join

이전까지 R-tree 기반 Spatial Join. Z-ordering tree 를 기반으로한 Join.

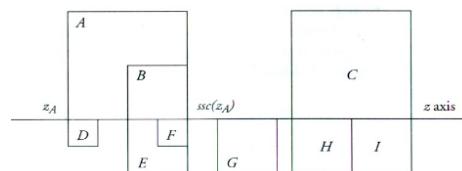
Z-ordering Spatial Join 가정 :

- Z-ordering tree 의 leaf 들은 z 순서로 정렬된 형태의 [z, oid] entry 들의 list L
  - o oid : id of object that contains the cell with z-order value z

Join 알고리즘의 기본 컨셉 :

- 두개의 entry list L1, L2 가 merged 된다.
- 만약 한 개의 key 가 다른 하나의 prefix 라면, entry pair 는 refinement step 을 위한 candidate 으로 keep
- Candidate pair 들은 정렬하여 중복 제거.

ex) 예시를 통한 Z-ordering Spatial Join



- z axis 를 기준으로 리스트 L1 과 L2 의 cell 들을 나타낸다.
- key 값 lowest : z, largest : z'
- c : z' = scc(z) (z 의 lower right corner 의 가장 작은 cell)

### Z Ordering Join 알고리즘

```
ZORDERINGJOIN (L1, L2; list of id): set of pairs of entries
begin
  result = Set of pairs of ids, initially empty
  event = end of file
  while not (eof(L1) and empty(S1) and eof(L2) and empty(S2)) do
    begin
      event = MIN(CURRENT(L1), SCC(top(S1)), CURRENT(L2), SCC(top(S2)))
      if (event = CURRENT(L1)) then // left bound of a rectangle
        ENTRY(L1, S1)
      else if (event = SCC(top(S1))) then // right bound of a rectangle
        result += EXIT(S1, S2)
      else if (event = CURRENT(L2)) then // left bound of a rectangle
        ENTRY(L2, S2)
      else if (event = SCC(top(S2))) then // right bound of a rectangle
        result += EXIT(S2, S1)
    end while
    sort result; remove duplicates;
    return result
  end
```

*수정된 부분*

	C <sub>1</sub>	S <sub>1</sub>	C <sub>2</sub>	S <sub>2</sub>	Event Actions
Step 0	A	()	D	()	
Step 1	B	(A)	D	()	event = current (L <sub>1</sub> ) = A
Step 2	B	(A)	E	(D)	event = current (L <sub>2</sub> ) = D
Step 3	B	(A)	E	()	event = scc(top(S <sub>2</sub> )) = D result = {A,D}
Step 4	C	(B,A)	E	()	event = current (L <sub>1</sub> ) = B
Step 5	C	(B,A)	F	(E)	event = current (L <sub>2</sub> ) = E
Step 6	C	(B,A)	G	(F,E)	event = current (L <sub>2</sub> ) = F
Step 7	C	(A)	G	(F,E)	event = scc(top(S <sub>2</sub> )) = B result = {A,D} + {B,F}
Step 8	C	()	G	(F,E)	event = scc(top(S <sub>1</sub> )) = A result = {A,D}, {B,F}, {B,E} + {A,F}, {A,E}

예시에 대해 설명:

L1 = [A, B, C], L2 = [D, E, F, G, H, I], S1, S2 는 빈 스택.

event = MIN(CURRENT(L1), SCC(top(S1)), CURRENT(L2), SCC(top(S2)))

- CURRENT 는 현재 가리키는 object 의 left val (z 축 기준으로 위쪽에 있다면 lower left val, 아래에 있다면 upper left val)
  - o ex) CURRENT(A) = CURRENT(D), CURRENT(A) < CURRENT(G)..
- SCC() 는 stack 의 top 값의 right val (z 축 기준으로 위쪽에 있으면 lower right val, 아래에 있다면 upper right val)
  - o ex) SCC(E) = SCC(B)
- MIN 함수는 기본적으로 L1, L2 를 비교할때 만약 같다면 L1 을 더 작게 생각

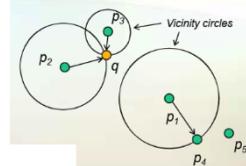
- ENTRY(L, S) : 현재 Current 뽑은 L 값을 S stack에 push
  - EXIT(S1, S2) : S1의 top 값과 S2의 remaining 값을 join 해서 result에 추가. S1에서 top 값을 pop 한다.
- 마지막에 result 정렬 후 중복 제거

## Ch13. RNN Query

RNN : reverse nearest neighbor

P is RNN of query point q, if there is no point  $p'$  such that  $\text{dist}(p', p) < \text{dist}(q, p)$

ex) 신촌에 스벅 개장하려고 하는데, point들을 기준으로 RNN 진행해서 RNN이 많이 존재한다면, 경쟁 높다는 뜻..



위 그림에서  $\text{NN}(p2) = \text{NN}(p3) = q$ . 역으로,  $\text{RNN}(q) = \{p2, p3\}$

→ 다시 말해 RNN은 자기 자신을 NN으로 갖는 point들

$\text{RNN}(p2) = \text{공집합}, \text{RNN}(p3) = \{q\} \Rightarrow \text{RNN is not a symmetric relation}$

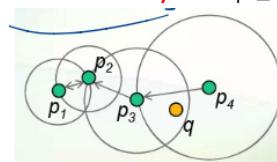
RNN 알고리즘은 Precomputing 기반 방식과 Filter/Refinement 기반 방식으로 나뉜다.

### 1. Original RNN (KM00) – Pre-computing

a) 모든 p 점들에 대해 NN(p)를 계산 (Precomputing)

b) p를 vicinity circle로 표현

- **vicinity circle** : p를 원의 중심, p와 NN(p)까지의 거리를 반지름으로 하는 원

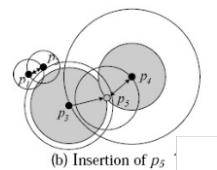


c) R-tree를 통해 이 모든 원들의 MBR를 indexing 한다. (RNN-tree)

d)  $\text{RNN}(q) = q$ 를 포함하는 모든 원 (ex) 위에 그림에서는  $\{p3, p4\}$

KM 알고리즘은 R-tree와 RNN-tree 두 개가 필요하다.

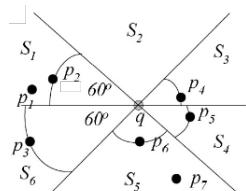
- 별도의 RNN-tree overhead!!



또, 위 사진과 같이 Insertion과 같은 update 때 원과 MBR이 모두 변해 RNN tree cost overhead가 심하다.

### 2. SAA – Filter/Refinement

NN을 모두 계산해야하는 precomputing 과정을 제거.



- 그림과 같이 query point q를 기준으로 주변 공간을 6개의 동일한 구역으로 나눔
- S1부터 차례대로 각 region에서 NN(q)인 점을 찾으면 candidate. (Refine)

o 해당 점 (candidate)p의 NN이 q인가? ( $\text{RNN}(q) = p?$ ) 맞다면 result에 추가 가능. (Filter)

위의 예시에서는  $S1 : \text{NN}(q) = p2$  but  $\text{NN}(p2) = p1$ .  $S2$ 는 없고,  $S3, S4$ 는  $\text{NN}(p4) = p5$ , vice versa.  $S5$  영역에서  $\text{NN}(q) = p6$ 의 NN은 q

⇒  $\text{RNN}(q) = \{p6\}$

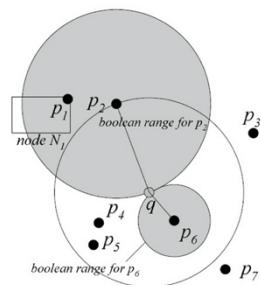
문제 : 차원이 증가할수록 region이 exponential하게 커짐. 2 차원 이상에서는 거의 사용되지 않음.

### 3. SFT – Filter/Refinement

역시 Precomputing 안함.

- a) query  $q$  의 kNN 들을 찾는다. k 개의 candidate. (Filter)
- b) 이 k 개의 candidate 들 중  $q$  와의 거리보다 서로 거리가 더 가까운 점들 제거 (Refine)
- c) 남은 점들에 대해 Boolean range query 를 통해 실제 RNN 을 찾는다 . (Refine)
  - 해당  $p$  원 안에 다른 object 걸리는게 존재하는가? 있으면 제거.
    - o  $\text{minmaxdist}(N_i, p_i) \leq \text{dist}(p_i, q)$

ex) SFT 예시,  $k = 4$  로 설정



Step1 : query point  $q$  를 기준으로 4NN 찾기. Candidate Set = { $p_6, p_4, p_5, p_2$ }

Step2 :  $p$  들 중  $q$  와의 거리보다  $p$  가 존재하는  $p_4, p_5$  를 각각 제거

Step3 : Boolean Range Query 를 진행 ( $p_2, \text{dist}(p_2, q)$ ) and ( $p_6, \text{dist}(p_6, q)$ )

Step4 :  $\text{minmaxdist}(N_1, p_2) \leq \text{dist}(p_2, q)$  (즉,  $p_2$  원안에는 node  $N_1$  이 존재) 으로,  $p_2$  제거

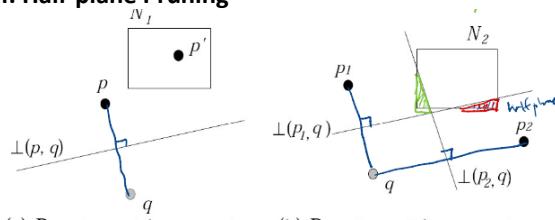
$\text{RNN}(q) = p_6$

하지만!!  $k = 4$  로 선택했기 때문에  $p_3$  는 고려 대상에 들지 못했는데,  $p_3$  는 사실  $\text{RNN}(q)$ 에 해당한다. (False Miss)

단점 :

- 1) **False Misses** :  $k$  값을 처음에 설정하는 것에 따라 값이 달라진다. (위의 예시에서  $k = 6$  으로 선택했다면,  $p_3, p_7$  이 처음 후보에 추가.  $p_7$  은 step2 에서  $p_6$  에 의해 제거되었겠지만,  $p_3$  는 끝까지 남는다.)
- 2)  $k$  값을 어떻게 설정할 것인가?

### 4. Half-plane Pruning



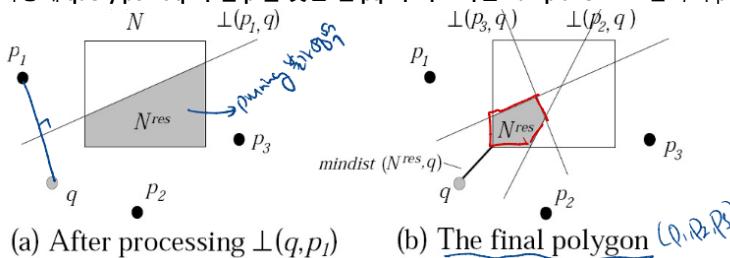
(a) Pruning with one point (b) Pruning with two points

(a)에서는 Node  $N_1$  안의 그 어느 점도  $q$  의 RNN 이 될 수 없다.  $PQ$  에 직교하는 half plane 에 의해 prune

(b)에서는  $p_1$  점 하나만으로는 아직 빨간색 부분의 점들에 대해서는 알 수 없다.  $\rightarrow N_2$  node 를 아직 prune 할 수 없다.

추가로  $p_2$  점이 발견  $\rightarrow$  이를 활용해서 Node  $N_2$  를 완전히 prune 가능. (초록색 영역은 무조건 NN 이  $p_1$ , 빨간색 무조건 NN 이  $p_2$ )

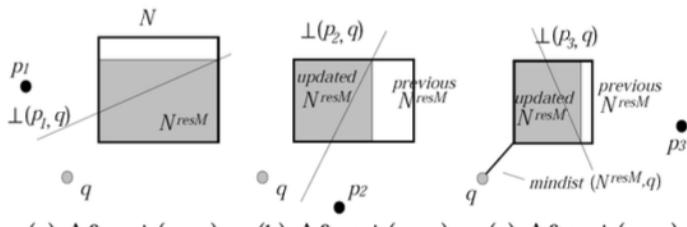
이렇게 query point  $q$  와 점  $p$  를 잇는 선  $pq$  와 직교하는 half plane 으로 잘라서 pruning..



- $N^{\text{res}}$  를 구하기 위해 Bisector trimming (자르는 것) 의 과정이  $O(N^2)$  processing time. (계산 cost 가 높음)
- 또, 저 모양을 관리하는 것도 irregular shape 이라서 훨씬 더 오래걸림.
- 차원이 증가할 수록 더 복잡해짐..

그래서  $N^{res}$  를 irregular shape 이 아니라 관리하기 편한 직사각형 MBR 화시키면?

### Approximating the residual MBR:



(a) After  $\perp(p_1, q)$    (b) After  $\perp(p_2, q)$    (c) After  $\perp(p_3, q)$

After  $\perp(p_2, q)$

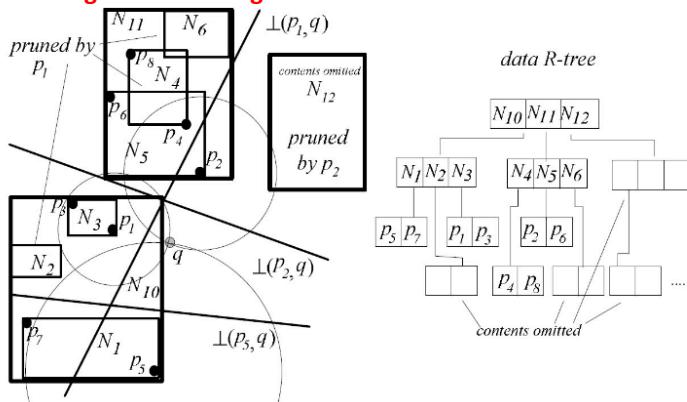
er  $\perp(p_3, \alpha)$

- 같은 방식으로 residual region 이 empty 이면 MBR pruning 이 가능함.
  - $N^{\text{resM}}$  영역을 실제 residual region 의 superset 으로 직사각형 MBR 의 형태로 관리한다.

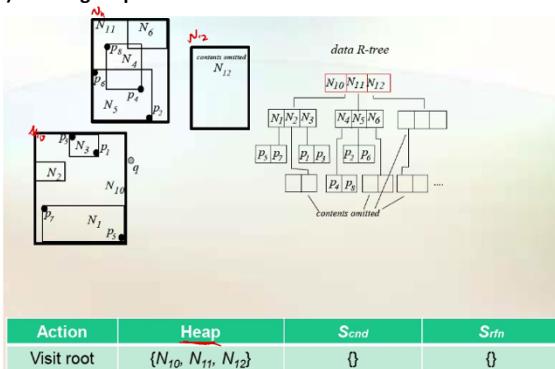
## 장점 :

- **O(n) processing time for computing  $N^{\text{resM}}$**
  - 더 이상 hyper-polyhedron 필요없음. 직사각형 관리도 편함 (두개의 꼭짓점 좌표만 있으면 됨!)

## 5. TPL Algorithm for Single RNN



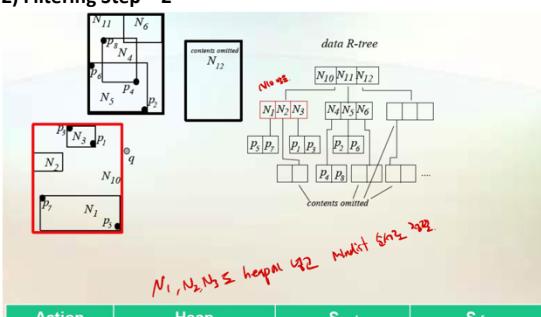
### 1) Filtering Step -1



Action	Heap	$S_{cnd}$	$S_{rfn}$
Visit root	$\{N_{10}, N_{11}, N_{12}\}$	$\emptyset$	$\emptyset$

R-tree 를 root 에서부터 방문해서 query point  $q$  를 기준으로 mindist 가 가까운 MBR 순서로 Heap 에 저장.

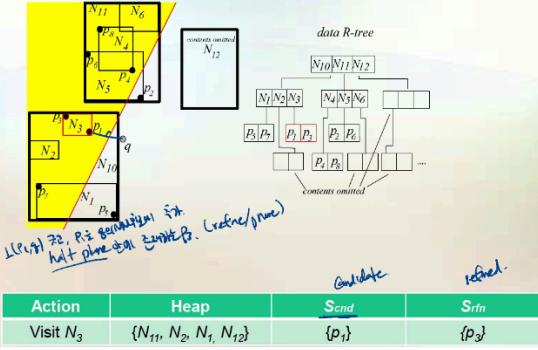
## 2) Filtering Step – 2



Action	Heap	$S_{cnd}$	$S_{rfn}$
Visit $N_1$	$\{N_1, N_2, N_3, N_4, N_5\}$	$\emptyset$	$\emptyset$

N10 브터 반문 Head에 해당 노드의 N1, N2, N3 등 추가 (MINDIST 순서로 정렬)

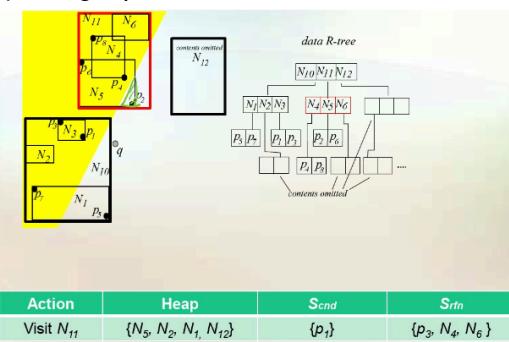
### 3) Filtering Step - 3



N3 방문. -> 실제 데이터인 p1, p3.

p<sub>1</sub>, q에 직교하는 half plane 밖에 존재하는 p<sub>3</sub>는 refine/pruned (p<sub>1</sub>은 후보에 추가, p<sub>3</sub>는 refine에 추가)

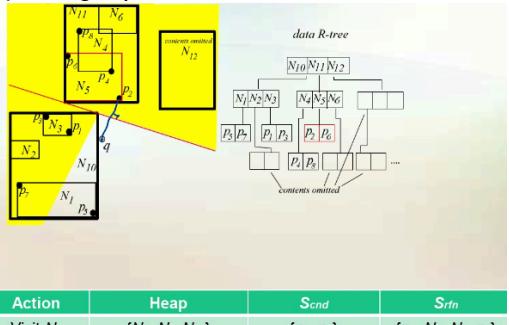
### 4) Filtering Step - 4



N11 방문 -> N4, N5, N6 중 N4, N6는 이전 half plane 으로 prune 가능 (refine)

이전 half plane에서 저 초록부분이 남기 때문에 아직 prune 불가 -> Heap에 N5 추가

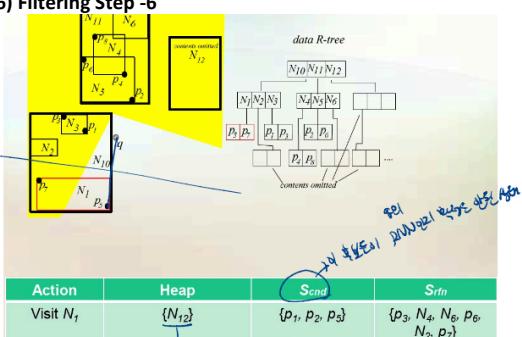
### 5) Filtering Step - 5



N5 방문 -> 실제 데이터 p2, p6 중 p2 부터.

p2, q에 직교하는 half plane 을 그어서 그 밖에 존재하는 영역 prune 가능해짐. p6는 refine 으로 추가.

### 6) Filtering Step - 6



N2 : prune 영역 (refine). N1 방문 -> 실제 데이터 p5, p7

p5, q에 직교하는 half plane 을 그어서 그 밖에 존재하는 영역 prune 가능해짐. p5는 candidate, p6는 refine 으로 추가

마지막으로 heap에 남은 N12는 prune 영역이므로 refine에 추가. (heap empty)

결국 Heap 이 비었을 때, 남는 후보 :  $S_{cnd} = \{p1, p2, p5\}$ ,  $S_{rfn} = \{p3, N4, N6, p6, N2, p7, N12\}$

하지만  $S_{cnd}$  는 아직 최종 q 의 RNN 이 아니라, q 의 RNN 후보!!

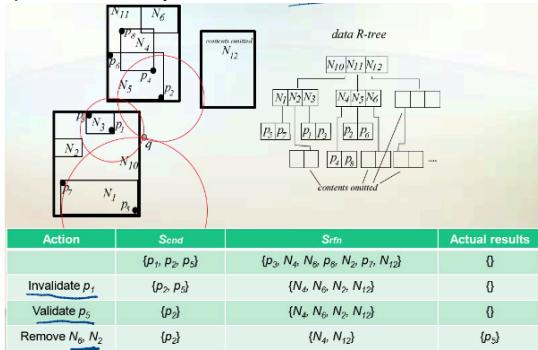
- ex) p1 의 경우 q 보다 p3 과의 거리가 짧다. => q 의 RNN 으로 p3 불가..
- 정확한 최종 검증으로 RNN 구해야함. (추가적인 Refine 필요 ->Refinement Heuristic)

Refinement Heuristic: 최종적으로 나온 q 의 RNN 후보들을  $S_{rfn}$  의 비교를 통해 최종 refining

- $S_{rfn}$  의 set of points :  $P_{rfn}$ , set of nodes :  $N_{rfn}$
- 1)  $S_{cnd}$  의 point p 는 false hit 으로 제거 가능 if there is point  $p'$  in  $P_{rfn}$  such that either of following hold:
  - (i)  $dist(p, p') < dist(p, q)$  를 만족하거나
  - (ii)  $minmaxdist(p, N) < dist(p, q)$  를 만족하는 node MBR N 이  $N_{rfn}$  에 존재하는 경우
- 2) Candidate point 는 q 보다 다른 candidate point 와 더 가까이 있다면 당연히 제거 가능
- 3)  $S_{cnd}$  의 point p 는 다음 두 조건을 만족하면 actual result:
  - (i)  $dist(p, p') < dist(p, q)$  를 만족하는  $p'$  이  $P_{rfn}$ 에 존재하지 않고
  - (ii) N 의 모든  $N_{rfn}$  이  $mindist(p, N) \geq dist(p, q)$  를 만족할 경우
- 4) 위의 모든 것들이 적용하지 않는다면,  $N_{rfn}$ 에 있는  $mindist(p, N) < dist(p, q)$  인 노드 MBR N 들을 모두 방문해서 다시 heuristic 적용

위의 예시서 Refinement step:

### 1) Refinement Step - 1



$P_{rfn} = \{p3, p6, p7\}$ ,  $N_{rfn} = \{N4, N6, N2, N12\}$ ,  $S_{cnd} = \{p1, p2, p5\}$

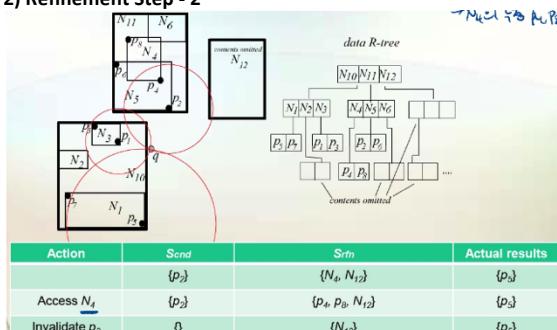
a)  $S_{cnd}$  의 점들을 중심으로 하는 원에 p 점들이 들어가는지 부터 확인.

- $p1$  을 중심으로 하는 원에  $p3$  가 들어있음 -> Invalidate  $p1$ .
- $p2, p5$  원에  $P_{rfn}$  의 어느 점도 안들어감 (이걸로는 invalidate 불가) ->  $S_{rfn}$  에서 p 점들은 제거

b)  $S_{cnd}$  의 점들을 중심으로 하는 원들에 Node 들이 들어가는지 확인

- $p5$  원 내부에는  $N_{rfn}$  에도 들어가는것이 없다. ( $p$  는 validate 가능)
- $p2$  원 내부에도  $N2, N6$  없음 ( $S_{rfn}$  에서  $N2, N6$  제거- prune) but  $N4, N12$  는 overlap

### 2) Refinement Step - 2



c)  $P2$  는  $N4$  와  $N12$  중 minmaxdist 가  $N4$  가 더 작음.

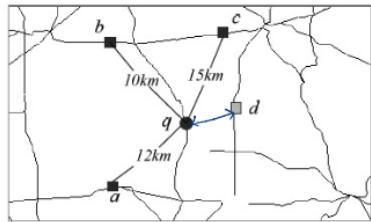
- $N4$  확인 (두 점  $p4, p8$  을  $S_{rfn}$  에 추가)
- $dist(p2, p4) < dist(p2, q)$  임으로  $p2$  제거 가능.

최종 Actual Results : {p5}

## Ch14. RoadNetwork LBS Query

이전까지는 직선 거리로 판단 (Euclid. distance)

이제는 도로망 기준.



위의 예시에서 q의 euclidean NN은 d이지만, 네트워크(도로망) 거리는 가장 멀리 떨어져있음. Network NN은 b.

네트워크형 구조가 graph 형태로 (adjacent list) 저장됨.

Graph Node :

- Black Points : Network junction(교차로)
- White Points : Start/end of road
- Gray : 휘어지는 부분, curvature or 속도제한변화..

Network distance :  $d_N(n_i, n_j)$  (직선거리가 아닌 네트워크 거리) (연결된 connecting nodes에 대해. not connecting?  $d_N = \text{shortest path}$ )

- Unidirectional 이라면,  $d_N$ 은 asymmetric. ( $d_N(n_i, n_j) \neq d_N(n_j, n_i)$ ). 오히려 양방향이면 symmetric
- Euclidean lower-bound property. (Euclid. 거리는 직선거리이기 때문에 반드시 최소 거리 (lower bound))

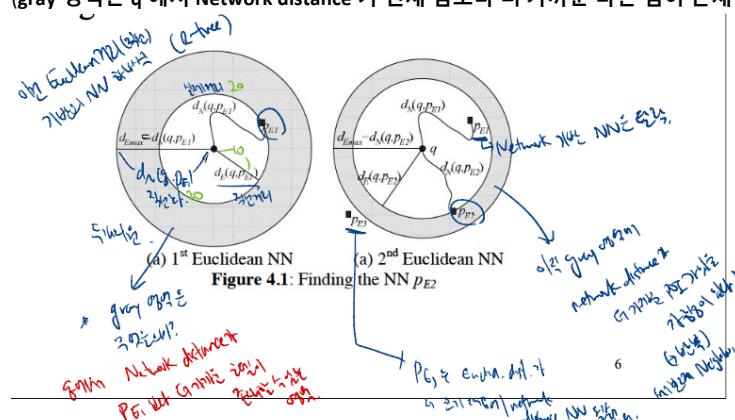
### NN Query in Spatial Network DB.

#### 1. IER (Incremental Euclidean Restrictions) Algorithm

방법 : Euclidean 직선거리를 기반으로 최소 점을 고르고, 해당 점과의 Network 기반 거리를 구한다.

이점과의 직선거리이상, Network 기반 거리 이하인 영역

(gray 영역은 q에서 Network distance 가 현재 점보다 더 가까운 다른 점이 존재할 수 있는 영역)



2<sup>nd</sup> Euclidean NN에서 PE3는 Euclid dist 보다 더 크기 때문에 network distance NN 될 수 없음.

Gray 영역에 network dist. 가 더 가까운 POI가 있을 가능성은 있다!!!

#### IER Algorithm to solve Network DB KNN Query?

##### Algorithm IER ( $q, k$ )

/\*  $q$  is the query point \*/

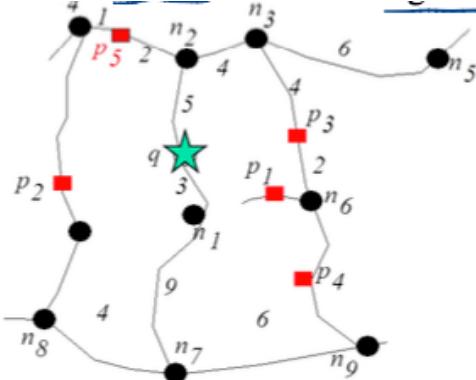
1.  $\{p_1, \dots, p_k\} = \text{Euclidean\_NN}(q, k);$
2. for each entity  $p_i$
3.      $d_N(q, p_i) = \text{compute\_ND}(q, p_i)$  ) Network distance는 2<sup>nd</sup> 2<sup>nd</sup>
4.     sort  $\{p_1, \dots, p_k\}$  in ascending order of  $d_N(q, p_i)$   $\rightarrow$  network distance가 빠른 것부터 정렬
5.      $d_{E_{\max}} = d_N(q, p_k)$
6.     repeat
7.          $(p, d_E(q, p)) = \text{next\_Euclidean\_NN}(q);$
8.         if  $(d_N(q, p) < d_N(q, p_k))$  //  $p$  closer than the  $k^{\text{th}}$  NN
9.             insert  $p$  in  $\{p_1, \dots, p_k\}$  // remove ex- $k^{\text{th}}$  NN
10.           $d_{E_{\max}} = d_N(q, p_k)$
11. until  $d_E(q, p) > d_{E_{\max}}$

End IER

## 2. INE (Incremental Network Expansion)

IER 은 gray 영역이 너무 크게 나올 수도 있는데, 이 경우 비효율적이다.

실제로 Network DB 상에서 연결되는 것부터 하나씩 따라가면서 NN 질의 처리 되는 것이 INE  
ex)



이 사진과 같은 Network DB 에서 IER 에 의하면, **q 와 Euclidean distance 가 가장 멀리 떨어져있는 p5** 는 마지막에 retriive 된다.  
하지만 실제로는 Network distance 가 NN 인 점이 p5 이다. (IER 에서는 원이 줄어들면서 gray 영역을 줄여나가는 것이 오래걸림)

INE algorithm pseudo-code:

```

Algorithm INE (q, k)  // n1n2
1.  $n_i, n_j = \text{find\_segment}(q)$  → 끊어진 line segment 찾기
2.  $S_{\text{cover}} = \text{find\_entities}(n_i, n_j)$ ; //  $S_{\text{cover}}$  is the set of entities
   covered by  $n_i, n_j$  → Scover는 n1n2를 포함하는 모든 노드의 집합
3.  $\{p_1, \dots, p_k\} =$  the  $k$  (network) nearest entities in  $S_{\text{cover}}$ 
   sorted in ascending order of their network distance
   ( $p_m, p_{m+1}, \dots, p_k$  may be  $\emptyset$  if  $S_{\text{cover}}$  contains  $< k$  points)
4.  $d_{N_{\text{max}}} = d_N(q, p_k)$  // if  $p_k = \emptyset$ ,  $d_{N_{\text{max}}} = \infty$ 
5.  $Q = \langle (n_i, d_N(q, n_i)), (n_j, d_N(q, n_j)) \rangle$  // sorted on  $d_N$ 
6. de-queue the node  $n$  in  $Q$  with the smallest  $d_N(q, n)$ 
7. while ( $d_N(q, n) < d_{N_{\text{max}}}$ )
8.   for each non-visited adjacent node  $n_x$  of  $n$ 
9.      $S_{\text{cover}} = \text{find\_entities}(n_x, n)$ ;
10.    update  $\{p_1, \dots, p_k\}$  from  $\{p_1, \dots, p_k\} \cup S_{\text{cover}}$ 
11.     $d_{N_{\text{max}}} = d_N(q, p_k)$ 
12.    en-queue  $(n_x, d_N(q, n_x))$ 
13. de-queue the next node  $n$  in  $Q$ 

```

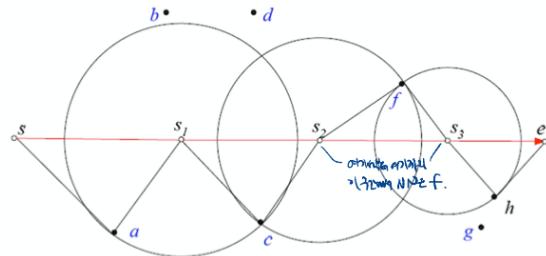
End INE

- 1) q 가 있는 line segment 를 찾는다. 위 사진에서는 n1n2
- 2) n1n2 사이에 있는 entity p 를 찾는다. 없으면  $S_{\text{cover}} = \text{공집합}$
- 3)  $S_{\text{cover}} < k$  so  $p_k = \text{공집합}$
- 4)  $d_{N_{\text{max}}} = \text{INF}$   
 $Q = \langle (n_1, 3), (n_2, 5) \rangle$   
de-queue  $(n_1, 3)$  from Q and  $(n_1, n_2)$   
 $Q = \langle (n_2, 5), (n_1, 12) \rangle$  *→ p5는 (n1, n2)에 가깝다*  
de-queue  $(n_2, 5)$  from Q and  $(n_2, n_4)$   
 $S_{\text{cover}} = \{p_5\}$ ,  $1NN = \{p_5\}$ ,  $d_{N_{\text{max}}} = d_N(q, p_5) = 7$   
 $Q = \langle (n_4, 8), (n_1, 12) \rangle$  and  $(n_2, n_3)$   
 $Q = \langle (n_4, 8), (n_1, 9), (n_7, 12) \rangle$   
de-queue  $(n_4, 8)$  from Q,  $d_N(q, n_4) = 8 > d_{N_{\text{max}}} = 7$ , and Stop

Roadnetwork Range Query (생략)

## Ch15. CNN

연속된 질의처리 (Continuous NN) Query Processing using R-tree

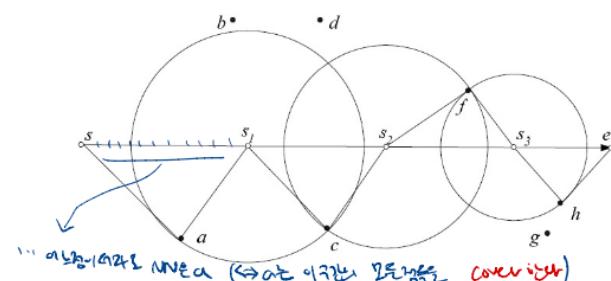


Query : line segment  $q = [s, e]$  ( $s$  부터  $e$  까지 이동하면서 연속적으로 NN 구하기)

Data points :  $a \sim h$

Result : Query  $q$  위의 모든 점에서 NN

→ 표현 :  $\{s(\text{NN} = a), s1(\text{NN} = c), s2(\text{NN} = f), s3(\text{NN} = h), e\}$



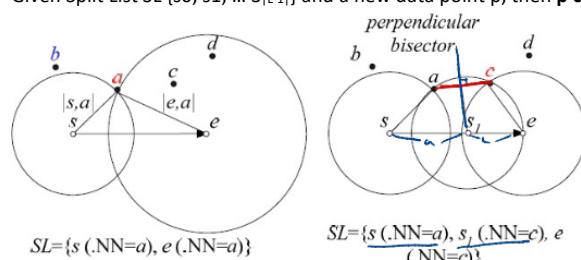
- Split points :  $s, e$  사이의  $s1, s2, s3$  찾아서 **split list** =  $\{s, s1, s2, s3, e\}$

- **Vicinity Circle** of  $s_i$  = split point  $s_i$ 를 원의 중심으로, radius :  $\text{dist}(s_i, s_i.\text{NN})$  인 원

- Data point  $u$  **covers** a point  $s$  if  $u = s.\text{NN}$ . (ex) point 'a' covers segment  $[s, s1]$ , point 'b' covers segment  $[s1, s2]$ )

### Lemma 1:

Given Split List  $SL = \{s_0, s_1, \dots, s_{|L|-1}\}$  and a new data point  $p$ , then  $p$  **covers** some point on  $q$  if and only if,  $p$  covers a split point.



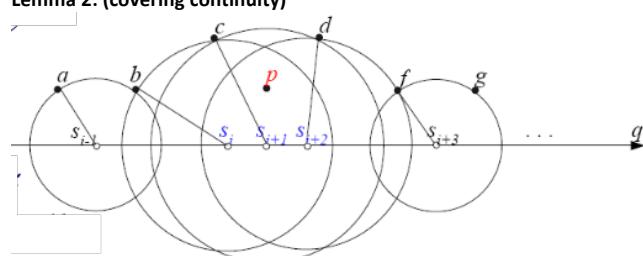
After processing  $a$

After processing  $c$

- a, c 모두 split point 를 cover 하기 때문에  $q$  위의 점들을 cover 한다고 볼 수 있다.

- c 를 processing 하면서 (추가하면서) ac 선에 직각 이등분선을 그려서 se에 닿는 점이 새로운 split point.

### Lemma 2: (covering continuity)



-  $p$  가 cover 하는 split point 들은 연속적이다.

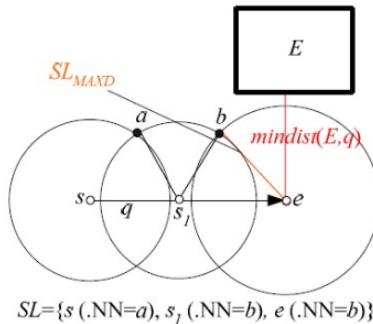
- ex)  $p$  가 split point  $s_i$ 를 cover 하지만  $s_{i-1}$  (혹은  $s_{i+1}$ )를 cover 하지 않는다면,  $p$  는 절대  $s_{i-1}$  (혹은  $s_{i+1}$ )를 cover 할 수 없다.

o 위 그림과 같이 새로운 data point 가 들어와도 모든 split point 를 확인할 필요없다. (Binary Search 활용가능)

### Heuristic 1:

entry E 와 query segment q 에 대해, E 의 subtree 는  $\text{mindist}(E, q) < \text{SL}_{\text{MAXD}}$  인 경우에만 qualify 한다. (vicinity 원안에 있느냐?)

- 다시 말해,  $\text{mindist}(E, q) > \text{SL}_{\text{MAXD}}$  인 경우에는 절대 E 안에 어느점도 NN 이 될 수 없다. (탐색 필요 X)



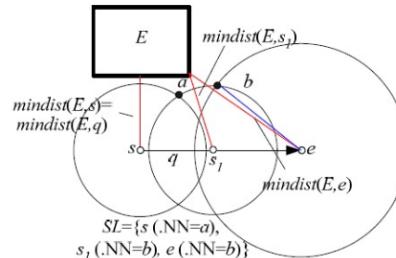
- $\text{mindist}(E, q)$  : MBR of E 와 q 사이의 최소거리,  $\text{SL}_{\text{MAXD}}$  : max distance btw split point and its NN.

- o  $\text{mindist}(E, q)$  구하는 방법 (MBR 의 꼭짓점 네개 (a,b,c,d) 와 line segment (s,e))
  - a,b,c,d 에서 line segment 에 수직으로 (4 개의 선)
  - s,e 각 각 MBR 과  $\text{mindist}$  (2 개의 선)
  - 이 6 개의 선을 비교.

### Heuristic 2:

Heuristic 1 과 더불어, subtree of E 가 탐색되는 경우는 **if and only if** split point  $s_i$  (in SL) such that  $\text{dist}(s_i, s_i.NN) > \text{mindist}(s_i, E)$

- 즉,  $\text{mindist}$  가 더 작아서 원 안에 있는 경우만 subtree E 를 탐색한다.

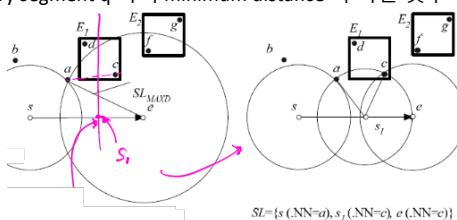


Heuristic 2 의 경우에서 E 와 모든 split point 와의  $\text{mindist}$  연산이 필요하다.

- 이것은 Heuristic 1 을 통과한 사례들에 대해서만 확인 : only one computation

### Heuristic 3: (접근 순서에 따라서 prune 이 되고 안되고..)

Heuristic 1,2 를 만족하는 entry 들은 query segment q 와의 minimum distance 가 작은 것부터 access 한다. (불필요한 탐색 줄이기)



- Heuristic 1,2 를 통과한 E1,E2. (E1 0| mindist 가 더 작으니, E1 부터 탐색)
- E1 내부의 점 중 c 가 mindist 가 더 작음. (c check) -> c covers e
- c 의 NN 인 a 와 c 를 이어서 직각 이등분선 -> 새로운 split point  $s_1$
- split point  $s_1$  에 의해 새로운 vicinity circle update, d 는 pruning (d covers no point), E2 도 heuristic 1 에 의해 pruning

만약 위의 예시에서 E2 부터 탐색했다면?

- af 의 직각 이등분선으로 split point 가 e 에서 살짝 왼쪽 인 곳에 생기고, E1 이 prune 되지 않음 (한 번 더 작업)
- heuristic 3 은 이렇게 query segment 까지의 거리가 작은 순서대로 접근하게 하여 탐색횟수를 줄인다

### R-tree 를 활용한 CNN Algorithm:

leaf entry(새로운 점) p 가 추가되었을때 어떤 부분을 cover 하는가?

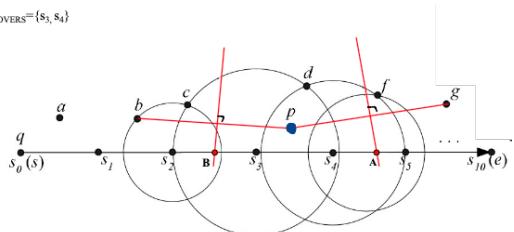
- 1) p 가 cover 하는 split point 집합  $S_{\text{COVERS}} = \{s_i, s_{i+1}, \dots, s_j\}$  return

## 2) cover 하는 split point이 있다면, SL 업데이트

Binary Search 를 활용하여 모든 split point 의 NN 들과 비교하는 것을 방지함 (연속성)

ex)

$S_{COVERS} = \{s_3, s_4\}$



1) 위의 예시에서는 p를 추가했을 때 cover 하는 split point들을 찾는 것.

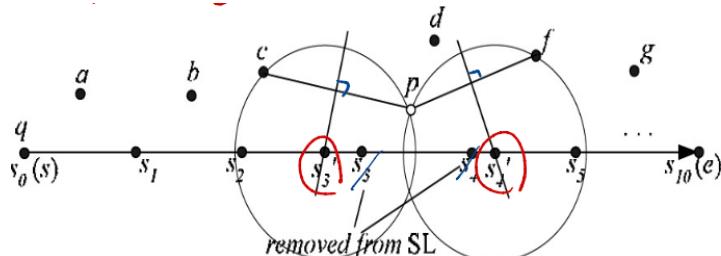
이때 모든 split point를 찾는 것이 아니라,

- $s_1 \sim s_{10}$  까지 중에서 median인  $s_5$  먼저 확인 ( $NN(s_5) \neq p$ )
- $s_5$ 의 NN인  $g$  와  $p$ 를 이은 후 수직이등분선을 그은 점  $A$ 를  $s_5$ 와 비교 (왼쪽에 있음)  $\rightarrow s_1 \sim s_4$  확인
- $s_1 \sim s_4$ 의 중간점  $s_2$  확인. ( $NN(s_2) \neq p$ )  $\Rightarrow$  같은 작업. 점  $B$ 는  $s_2$  오른쪽에 위치  $\rightarrow s_3 \sim s_4$  확인
- **s3 확보 ( $NN(s_3) = p$ )**

이때, covering continuity에 의해 연속되는 점  $s_2$ 와  $s_4$ 만 추가로 확인하면 된다. ( $s_2$ 는 이미 확인함,  $s_4$  확인  $\rightarrow s_5$  이미 확인)

$\Rightarrow S_{cover} = \{s_3, s_4\}$

## 2) $S_{cover}$ 를 구했으니, update SL

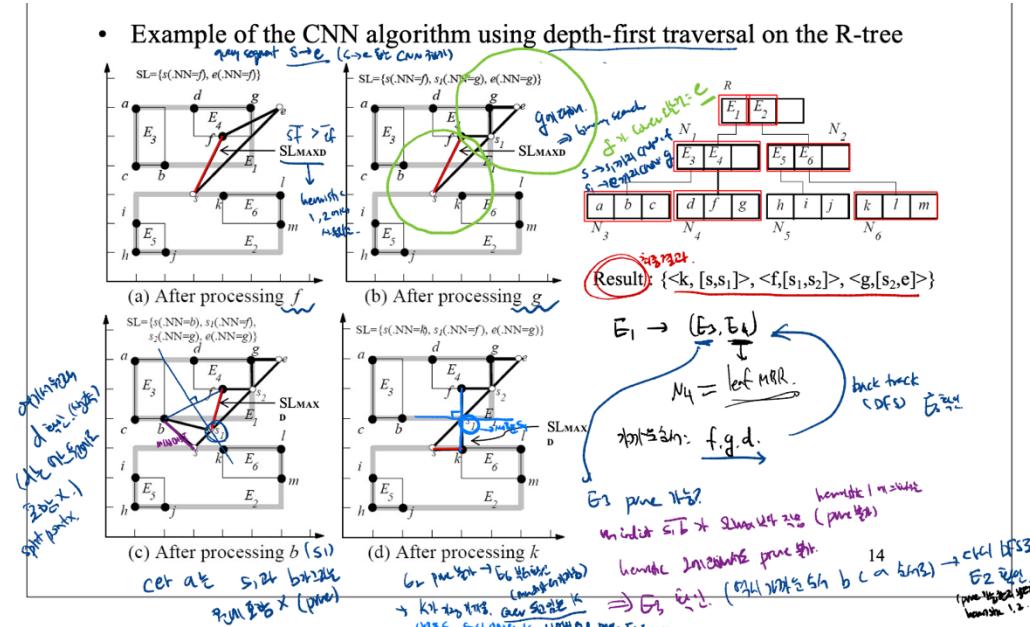


이전  $s_3$ 와  $s_4$ 를 SL에서 삭제하고  $s_3'$ 와  $s_4'$  추가. (추가하는 알고리즘 pseudo-code)

```

Algorithm Handle_Leaf_Entry
/* p: the leaf entry being handled, SL: the split list*/
1. apply binary search to retrieve all split points covered
   by p:  $S_{COVER} = \{s_1, s_2, \dots, s_5\}$  Scover = {s3, s4}
2. let  $u = s_1.NN$  and  $v = s_5.NN$  u = s2.NN = c, v = s4.NN = f
3. remove all split points in  $S_{COVER}$  from SL Remove s3, s4 from SL
4. add a split point  $s'_1$  at the intersection of  $q$  and  $\perp(u, p)$  Add s3' at the intersection of q and ⊥(c, p)
   with  $s'_1.NN = p$ ,  $dist(s'_1, s'_1.NN) = |s'_1, p|$ 
5. add a split point  $s'_{i+1}$  at the intersection of  $q$  and  $\perp(v, p)$  Add s4' at the intersection of q and ⊥(f, p)
   with  $s'_{i+1}.NN = p$ ,  $dist(s'_{i+1}, s'_{i+1}.NN) = |s'_{i+1}, p|$ 

```



## Ch16. Spatial Keyword Search Query

Geospatial and Textual (Geo-Textual) Data :  $p = \langle \text{location}, \text{text description} \rangle$

Geo-textual data : Static 과 Streaming

- Static geo-textual data : POI data. 정해진 데이터 (ex) Web page with location)
- Streaming geo-textual data : 계속 변하는 geo textual data (ex) Geo-tagged micro-blog posts.. )

**Boolean Range Query : Keyword 를 갖는가?**

Data 들은 Query Region 과 Set of Keywords 를 갖는다.

**Top-k kNN Query (TkQ): Combination of spatial proximity and text relevancy**

**Boolean kNN Query :** 사용자의 현재 위치 중심으로 가장 가까운 object 를 뽑으면서 keyword 포함하는 k 개 찾기

- Boolean Range Query 도 주어진 range 에 대해 boolean query. (Top-K 와는 달리 Spatial Distance/Textual Relevance 고려한다고 보기 힘듬)

### 1. Top-k Spatial Keyword Query:

Objects :  $p = \langle \lambda, \varphi \rangle$  (location, text description)

Query :  $q = \langle \lambda, \varphi, k \rangle$  (location, keywords, # of objects)

**Ranking Function**

**Ranking function**

$$rank_q(p) = \alpha \frac{\|q.\lambda, p.\lambda\|}{\max D} + (1-\alpha) \left( 1 - \frac{tr_{q.\varphi}(p.\varphi)}{\max P} \right) \quad 0 \leq \alpha \leq 1$$

Distance (공간적) :  $\|q.\lambda, p.\lambda\|$

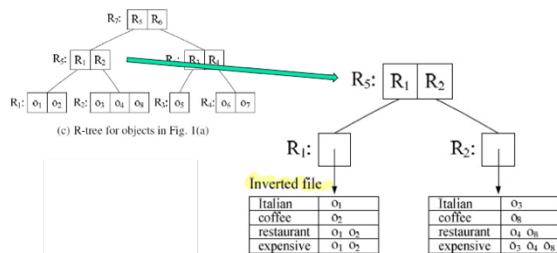
Text Relevancy (Textual 유사도, 얼마나 유사한지) :  $tr_{q.\varphi}(p.\varphi)$

$\alpha$  : 선호도 (가중치)

### 2. R\* - IF Index

$R^* - IF = R^* \text{tree} - \text{Inverted File}$  (Spatial First index)

- $R^*$  Tree 를 먼저 build
- leaf node 에 inverted file 을 만들어 text component 를 저장
  - o Inverted File 에는 keyword 에 의해 정렬되어 있음



이후에 query processing 에서는 R-tree 범위 질의처리 (kNN) + inverted file 탐색

### 3. IF - R\* Index

$IF - R^* = \text{Inverted File} - R^* \text{-tree}$  (Text First Geotextual index, counterpart of  $R^* - IF$ )

- 각 Distinct term  $t$  in  $D$  에 대해 separate  $R^*$ -tree is built (각 keyword 에 대한 object 들에 대해  $R^*$ -tree 를 build)

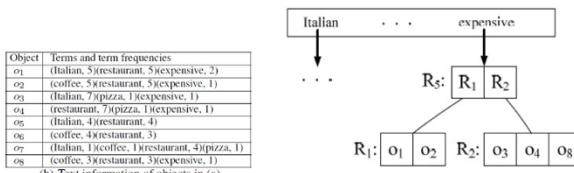


Figure 4: R-tree under the word  
expensive

Boolean 질의처리에서 : 각 keyword 별로 질의처리 => 각 결과의 intersection

### R\*-IF Index & IF-R\* Index:

- 둘 다 BkQ, BRQ (Boolean kNN Query, Boolean Range Query)
  - 일반적으로 BRQ 에 대해서는 **IF- $R^*$  성능 >>  $R^*$ -IF**
    - o maintenance 면에서는 중복이 없는  $R^*$ -IF 가 나음
  - Top -k Query 는 지원하지 않는다. (internal node 에 frequency 정보가 없음)

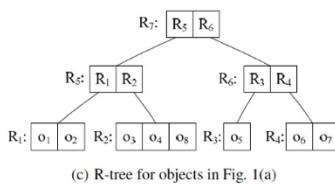
## IR<sup>2</sup> – Tree

R\*-IF, IF-R\* 개선. But 역시 Boolean Range, kNN query 만 지원.

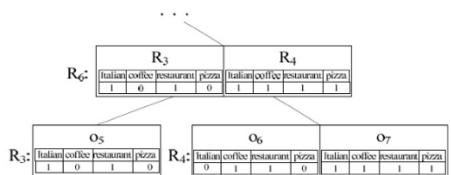
- **Bitmap** 을 사용한다
  - **fan-out of tree** : length of signature file 에 의존적
  - signature file of node : union of all signature of its entry

Object	Terms and term frequencies
<i>o<sub>1</sub></i>	(Italian, 5)(restaurant, 5)(expensive, 2)
<i>o<sub>2</sub></i>	(coffee, 5)(restaurant, 5)(expensive, 1)
<i>o<sub>3</sub></i>	(Italian, 7)(pizza, 1)(expensive, 1)
<i>o<sub>4</sub></i>	(restaurant, 7)(pizza, 1)(expensive, 1)
<i>o<sub>5</sub></i>	(Italian, 4)(restaurant, 4)
<i>o<sub>6</sub></i>	(coffee, 4)(restaurant, 3)
<i>o<sub>7</sub></i>	(Italian, 1)(coffee, 1)(restaurant, 4)(pizza, 1)
<i>o<sub>8</sub></i>	(coffee, 3)(restaurant, 3)(expensive, 1)

(b) Text information of objects in (a)



**Figure 1: Example**

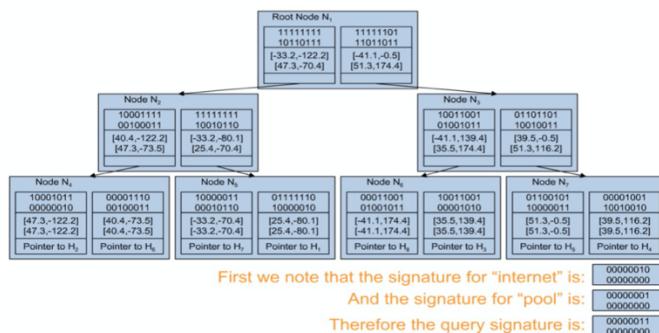


IR<sup>2</sup> – tree : keyword 가 들어가면 1, 아니면 0 으로 표현

- 자식 노드가 있다면 (R3, R4) 해당 자식 노드 (object)들의 bitmap 의 union 결과 (logical OR)

## IR<sup>2</sup>-tree and Algorithm:

Example: Execution of the IR<sup>2</sup>-Tree Algorithm on Distance-First Top-2 Spatial Keyword Query [30.5, 100.0] with keyword "internet" and "pool"



- Top2 keyword spatial Query with keyword “internet” and “pool”
  - signature for “internet”
  - signature for “pool” 확인
  - **query signature** : keyword query 의 keyword 들의 signature bitmap Logical OR 연산 한 결과. (Internet/Pool bitmap : IPB)
  - Priority Queue (거리 순으로 자동으로 정렬되는) : (Node/Pointer, query point 까지의 거리) 가 저장된

### a) Root Node N1 화이



1) Point 6.에서 회연 반복

- Priority Queue에서 pop하여 만족하는 경우에 대해서는 그 이후의 경우는 틀린 경우이다.

1.N2 끝나니 popped. N2 에 대해서 XOR 와 AND => 풀니 OK

2. 4~4 N4, N3 ၏ query distance

1. NFA 머신 확인 (그만 가는) → 깨끗이 가고 계산 후 ... → 정작 (1) ...

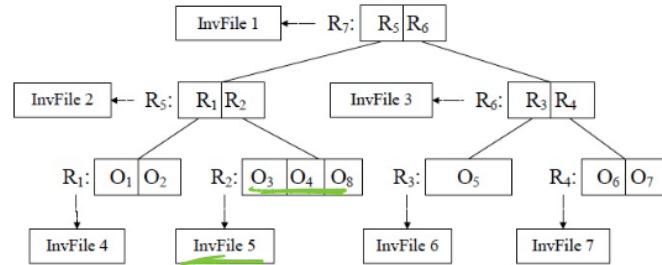
I.NS 단지 목적(H) 단기(H) -> H

1. H7 보다 N4 의 거리가 작기 때문에 N4 에 더 가까운 게 있을 가능성 있음
2. N4 탐색 -> IPB 와 AND 연산 결과 : H2 (가능), H6 (pruned)
- e) Priority Queue 에 남아있는 H 들이 k 개 있다면? 종료 가능
  - Priority Queue 에 남아있는 것 : (H7, 181.9), (H2, 222.8). 2NN 이기 때문에 stop

IR<sup>2</sup>-Tree 역시 BkQ 와 BRQ 는 processing 가능.. 하지만 signature file doesn't have frequency info : TkQ 는 불가.

## IR-Tree Index

- Top-K 지원 가능!!
- 각 노드가 subtree 의 object 들을 표현하는 inverted file 로 pointer 가 있음
  - o inverted file : vocab of distinct terms, set of posting lists (relates to term t)



(이런식으로 InvFile 이 각 노드에 연결되어있음)

	$O_1.doc$	$O_2.doc$	$O_3.doc$	$O_4.doc$	$O_5.doc$	$O_6.doc$	$O_7.doc$	$O_8.doc$
Chinese	5	0	7	0	4	0	1	0
Spanish	0	5	0	0	0	4	1	3
restaurant	5	5	0	7	4	3	4	3
food	0	0	1	1	0	0	1	0

Vocabulary	InvFile 4	InvFile 5	InvFile 6	InvFile 7
Chinese	$\langle O_1.doc, 5 \rangle$	$\langle O_3.doc, 7 \rangle$	$\langle O_5.doc, 4 \rangle$	$\langle O_7.doc, 1 \rangle$
Spanish	$\langle O_2.doc, 5 \rangle$	$\langle O_8.doc, 3 \rangle$	$\langle O_6.doc, 4 \rangle, \langle O_7.doc, 1 \rangle$	
restaurant	$\langle O_1.doc, 5 \rangle, \langle O_2.doc, 5 \rangle$	$\langle O_4.doc, 7 \rangle, \langle O_8.doc, 3 \rangle$	$\langle O_5.doc, 4 \rangle$	$\langle O_6.doc, 3 \rangle, \langle O_7.doc, 4 \rangle$
food		$\langle O_3.doc, 1 \rangle, \langle O_4.doc, 1 \rangle$		$\langle O_7.doc, 1 \rangle$

Vocabulary	InvFile 2	InvFile 3	InvFile 1
Chinese	$\langle R_1.doc, 5 \rangle, \langle R_2.doc, 7 \rangle$	$\langle R_3.doc, 4 \rangle, \langle R_4.doc, 1 \rangle$	$\langle R_5.doc, 7 \rangle, \langle R_6.doc, 4 \rangle$
Spanish	$\langle R_1.doc, 5 \rangle, \langle R_2.doc, 3 \rangle$	$\langle R_4.doc, 4 \rangle$	$\langle R_5.doc, 5 \rangle, \langle R_6.doc, 4 \rangle$
restaurant	$\langle R_1.doc, 5 \rangle, \langle R_2.doc, 7 \rangle$	$\langle R_3.doc, 4 \rangle, \langle R_4.doc, 4 \rangle$	$\langle R_5.doc, 7 \rangle, \langle R_6.doc, 4 \rangle$
food	$\langle R_2.doc, 1 \rangle$	$\langle R_4.doc, 1 \rangle$	$\langle R_5.doc, 1 \rangle, \langle R_6.doc, 1 \rangle$

- Object 가 저장된 Leaf Node 들의 InvFile 에는 해당 object (이름)와 빈도수가 저장되어있다. (ex) InvFile4 )
- Child node 를 가리키는 internal node 들의 InvFile 에는 child node MBR (이름) 과 child node union 빈도수 (더 큰값) (ex) InvFile2)

## R-tree Index 활용한 LkT (Query, Index, k) : Priority Queue

### Algorithm 2 LkT( $Query, Index, k$ )

```

1: Queue ← NewPriorityQueue();
2: Queue.Enqueue(Index.RootNode, 0);
3: while not Queue.IsEmpty() do
4:   Element ← Queue.Dequeue();
5:   if Element is an object then
6:     if not Queue.IsEmpty() and  $DST(Query, Object) > Queue.First().Key$  then
7:       Queue.Enqueue(Object,  $DST(Query, Object)$ );
8:     else
9:       Report Element as the next nearest object;
10:      if k nearest objects have been found then
11:        break;
12:      else if Element is a leaf node then
13:        for each entry(Object) in leaf node Element do
14:          Queue.Enqueue(Object,  $DST(Query, Object)$ );
15:        else
16:          for each entry(Node) in node Element do
17:            Queue.Enqueue(Node,  $MINDST(Query, Node)$ );
  
```

$$D_{ST}(Q, O) = \alpha \frac{D_{\varepsilon}(Q.loc, O.loc)}{\max D} + (1 - \alpha) \left( 1 - \frac{P(Q.keywords | O.doc)}{\max P} \right),$$

$$MIND_{ST}(Q, N) = \alpha \frac{MIND_{\varepsilon}(Q.loc, N.rectangle)}{\max D} + (1 - \alpha) \left( 1 - \frac{P(Q.keywords | N.doc)}{\max P} \right),$$

$$P(Q.keywords | O.doc) = \prod_{t \in Q.keywords} \hat{p}(t | \theta_{O.doc})$$

$$\hat{p}(t | \theta_{O.doc}) = (1 - \lambda) \frac{tf(t, O.doc)}{|O.doc|} + \lambda \frac{tf(t, Coll)}{|Coll|}$$

$$\max P = \prod_{t \in Q.keywords} \max_{O' \in D} \hat{p}(t | O'.doc)$$

**DST(Q,O)** = Top-K Query Ranking function 과 유사 (leaf node)

- 노란색으로 하이라이팅 친곳 : euclidean 거리 구하고 normalize

- 초록색으로 하이라이팅 친곳 : keyword 유사성 normalize (

o  $P(Q.keywords | O.doc)$  = 각 keyword 별로  $\hat{p}(t | \theta_{O.doc})$  유사성 구해서 곱해준 것.

**MIND<sub>ST</sub>(Q,N)** = entry node 에 대해.

-  $tf(t, Coll)$  : term frequency of t in Coll

-  $tf(t, O.doc) \Rightarrow$  ex) chines, doc3 일때 chinese 의 freq : 7/8

**ex) Q.Keyword : Spanish. O.doc : 6,  $\lambda = 0.5$  일때**

$\max P = \hat{p}(t | \theta_{O.doc})$  의 max 의 곱

이 경우 키워드 하나이므로 : 하나만 구하면 됨.

$$\frac{tf(t, O.doc)}{|O.doc|} = 4/7 \text{ ( O6.doc 0|| Spanish / O6.doc 0에 모든 term 갯수)}$$

$$\frac{tf(t, Coll)}{|Coll|} = 4/8 \text{ (전체 doc 에서 spanish 몇번 등장? (O2, O6, O7, O8 : 4)) / (전체 document : 8)}$$

$$\Rightarrow 0.5 * 4/7 + 0.5 * 4/8$$

만약 keyword 가 spanish, restaurant 둘다? 둘다에 대해서 하고 곱하기