10. 다차원 공간 화일

▶ 다 차원 공간 화일이란 ?

- 여러 개의 필드를 동시에 키로 사용한 화일
  - k-d 트리(´75)
  - k-d-B 트리(´81)
  - 격자 화일(Grid File) (´84)
  - 사분 트리(Quadtree) (´84)
  - R-트리(´84), R^+-트리(´87), R^*-트리(´90)

- 응용 분야
  - 위치 정보와 같은 다차원 데이타는 단일 키 화일 구조로 처리 불가
    - (x, y) 또는 (x, y, z)는 차원당 하나의 값
    - CAD (Computer Aided Design)
    - GIS (Geographical Information System)

▶ 다차원 공간 화일의 종류

- **PAM (Point Access Method)**
  - 다차원의 점 데이타를 저장, 검색
  - 예) k-d트리, k-d-B 트리, 격자화일

- **SAM (Spatial Access Method)**
  - 선, 면과 같은 다차원 도형 데이타를 저장 검색
  - 예) R-트리, R^*-트리, R^+-트리

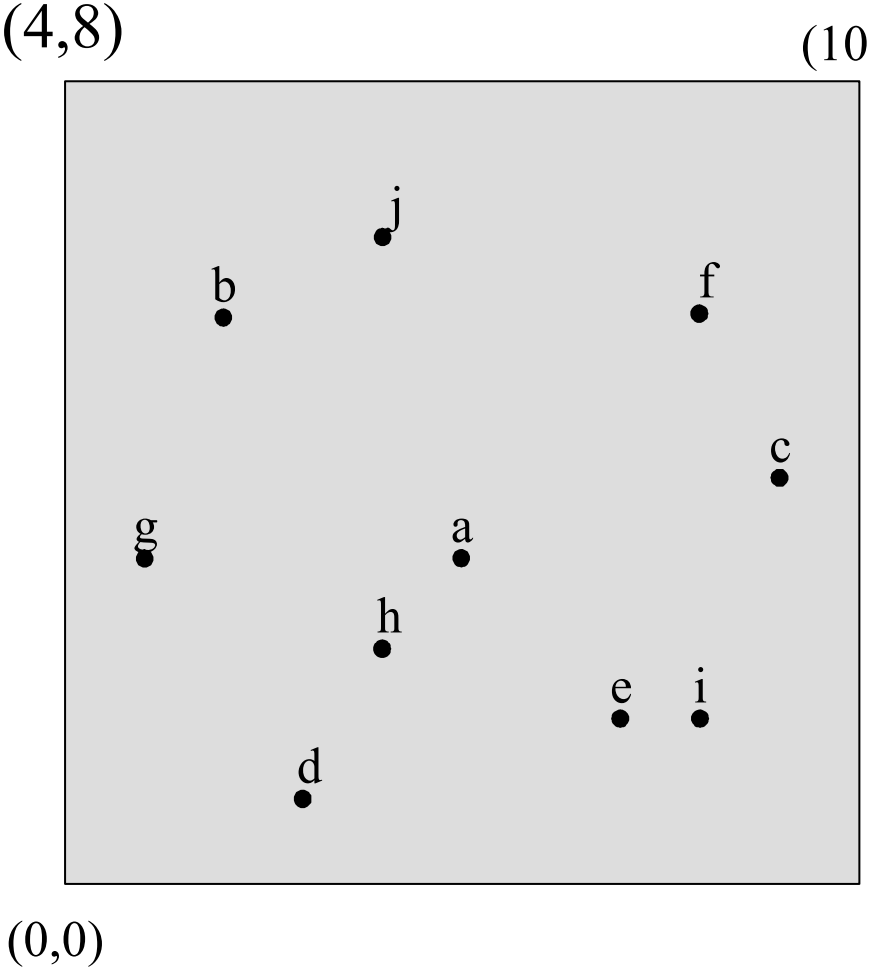
❖ k-d 트리

- **k-d (k-dimensional) 트리**
  - 이원 탐색 트리를 다차원 공간으로 확장한 것
  - 기본 구조와 알고리즘은 이원 탐색 트리와 유사
  - 트리의 레벨에 따라 차원을 번갈아 가며 비교
    - 예) 2차원의 경우: x → y → x → y → „

- **특징**
  - 주기억 장치상에서 동작
  - 소규모의 다차원 점 데이타를 인덱싱할 때 적합(PAM)
  - 균형 트리가 아님

▶ k-d 트리의 삽입

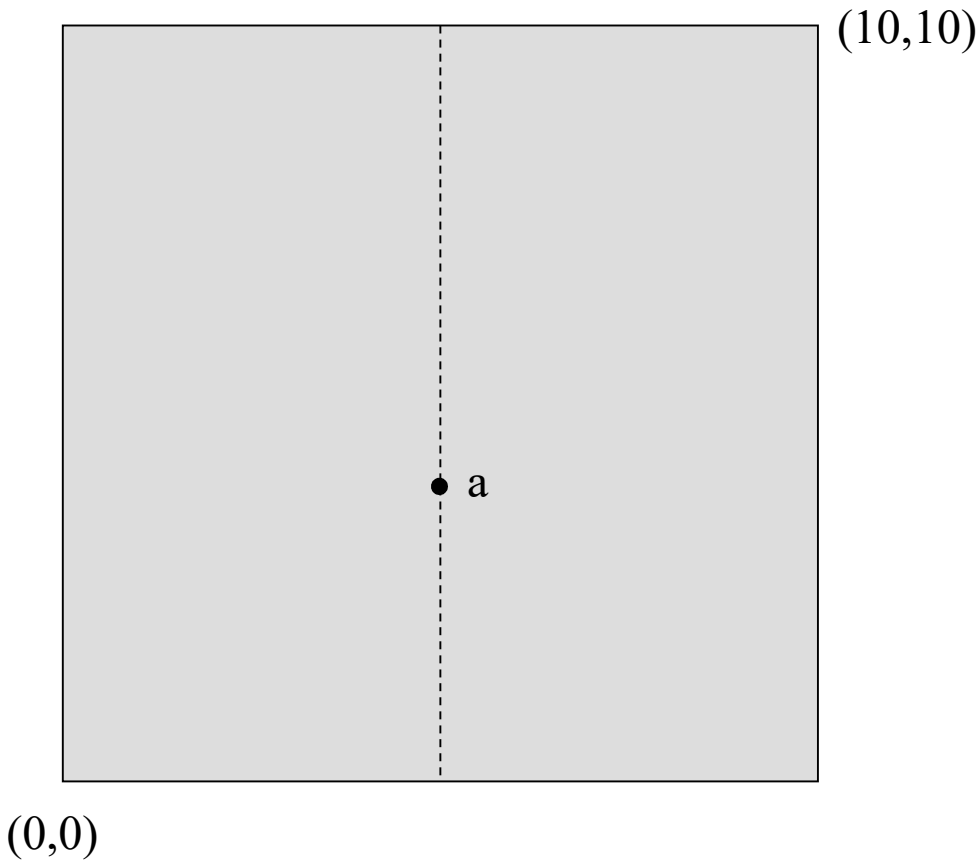
- 다음과 같은 **2**차원의 **10**개의 점을 **a**부터 **j**의 순서로 **k-d**트리에 삽입하는 경우
  - a(5,4), b(2,7), c(9,5), d(3,1), e(7,2), f(8,7), g(1,4), h(4,3), i(8,2), j(4,8)



◆ **a 삽입**

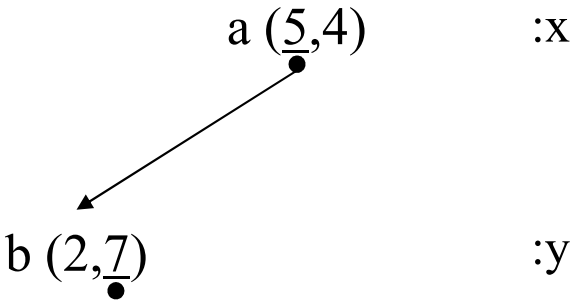
– 루트에 저장

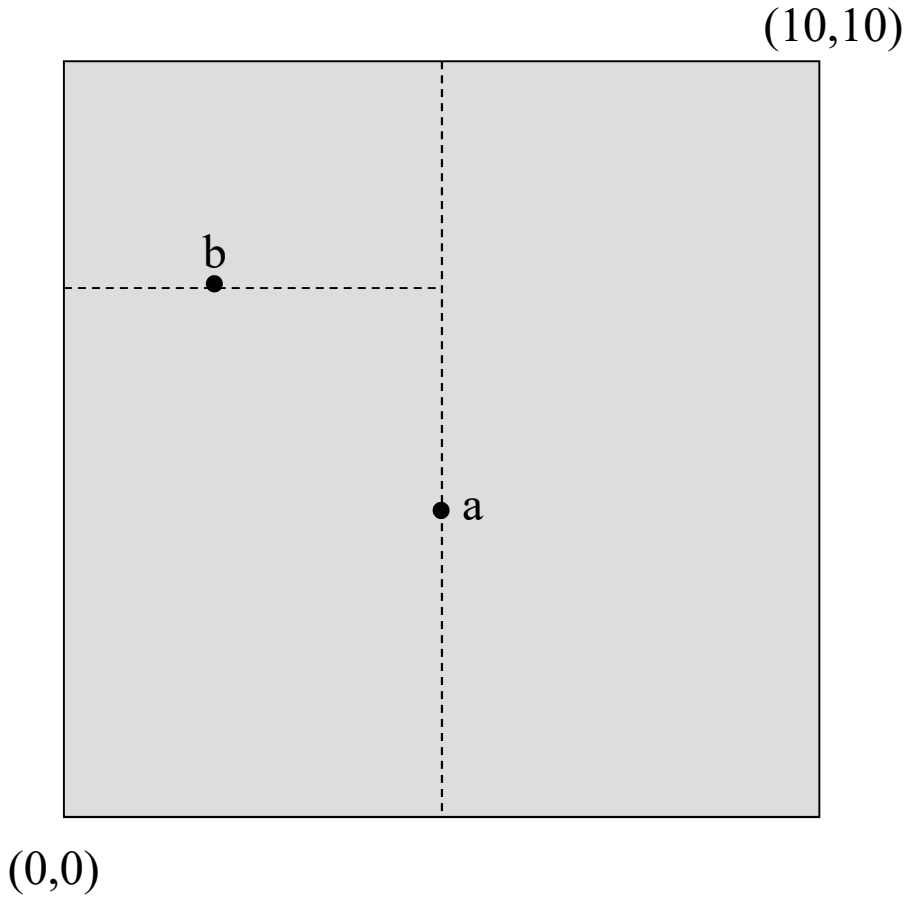
a (5,4) :x



◆ **b 삽입**

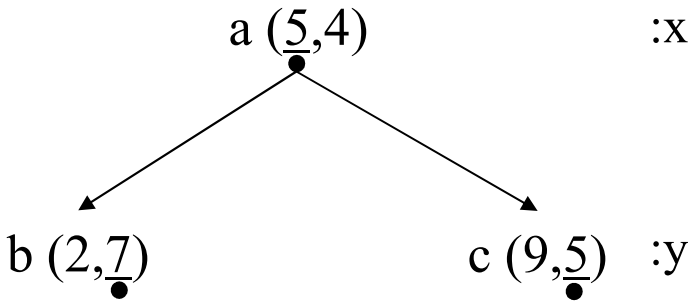
– 루트와 x값 비교, 작으므로 왼쪽 자식 노드에 삽입

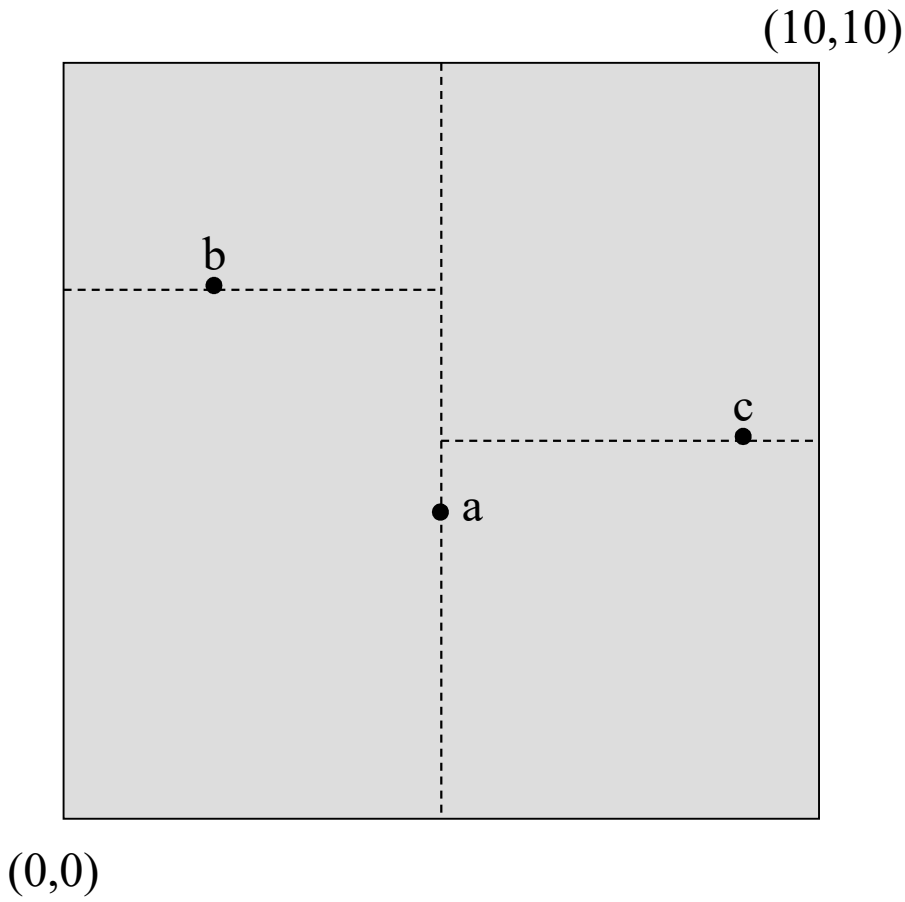




◆ **c 삽입**

– 루트와 x값 비교, 크므로 오른쪽 자식 노드에 삽입

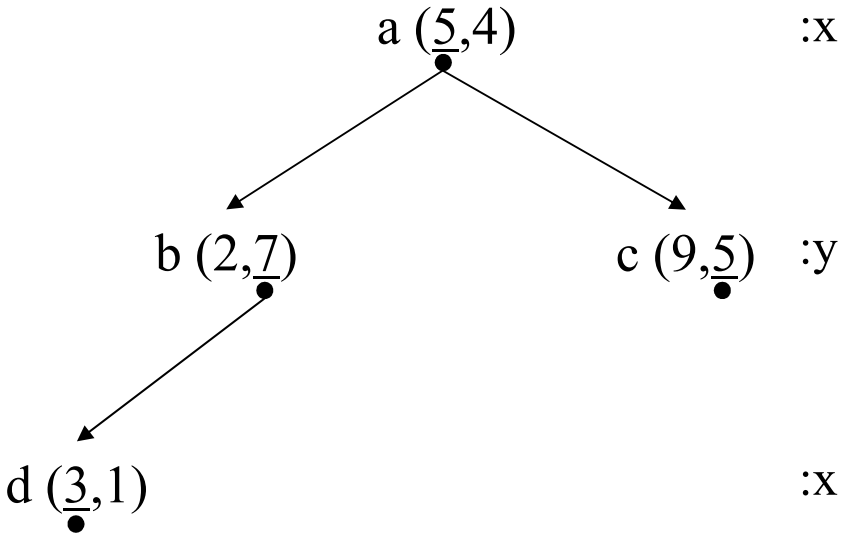


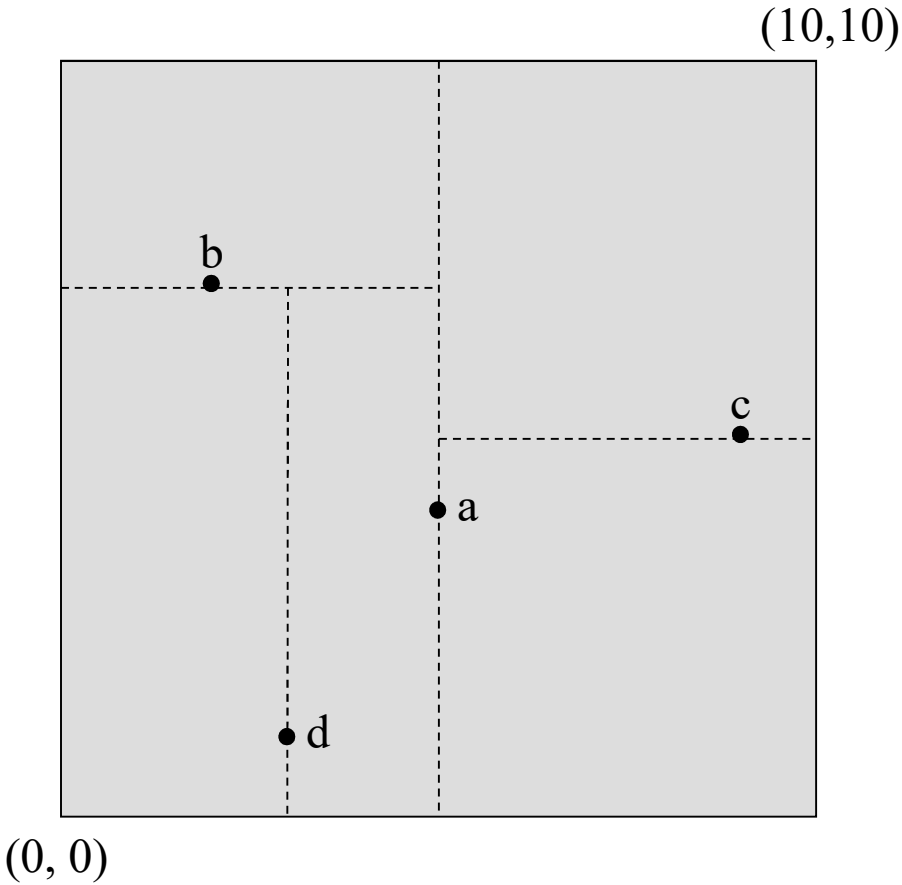


◆ **d 삽입**

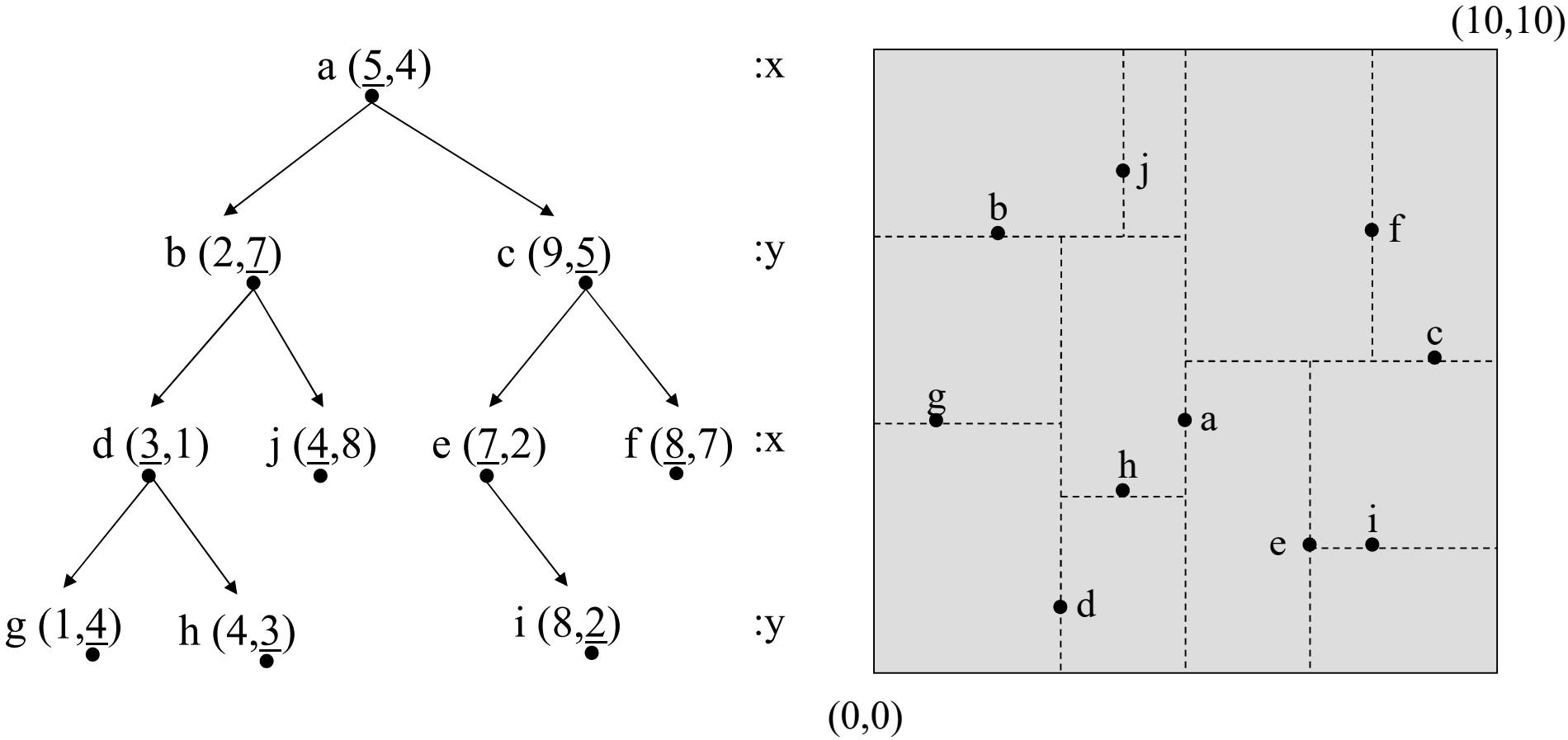
– 루트와 x값 비교, 작으므로 왼쪽 자식 노드로

– b와 y값 비교, 작으므로 왼쪽 자식 노드에 삽입



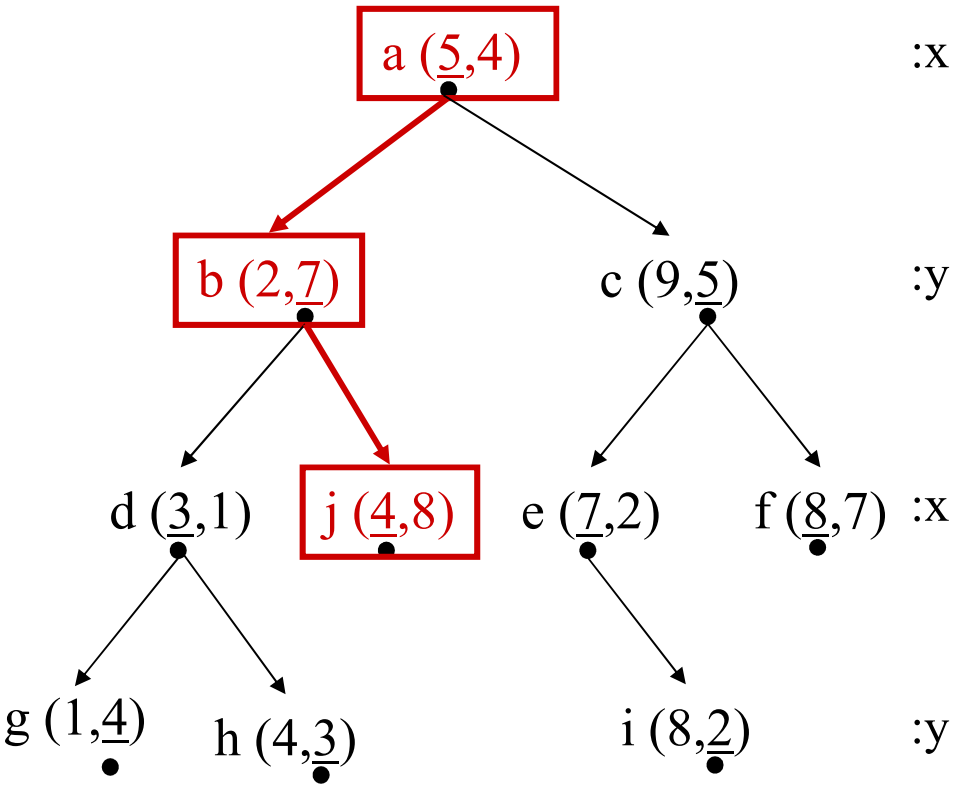


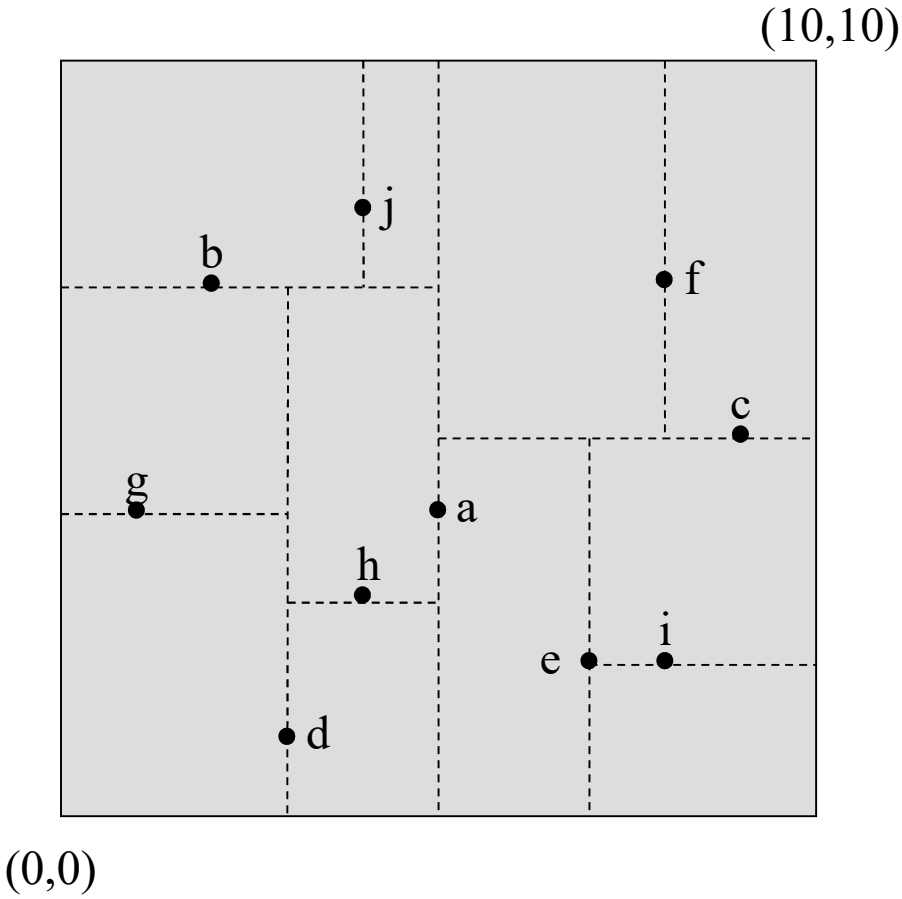
◆ 최종 **k-d** 트리


a (5,4)
:x
b (2,7)
c (9,5)
:y
d (3,1)
j (4,8)
e (7,2)
f (8,7)
:x
g (1,4)
h (4,3)
i (8,2)
:y
(10,10)
j
b
f
c
g
a
h
i
e
d
(0,0)

▶ k-d 트리의 검색

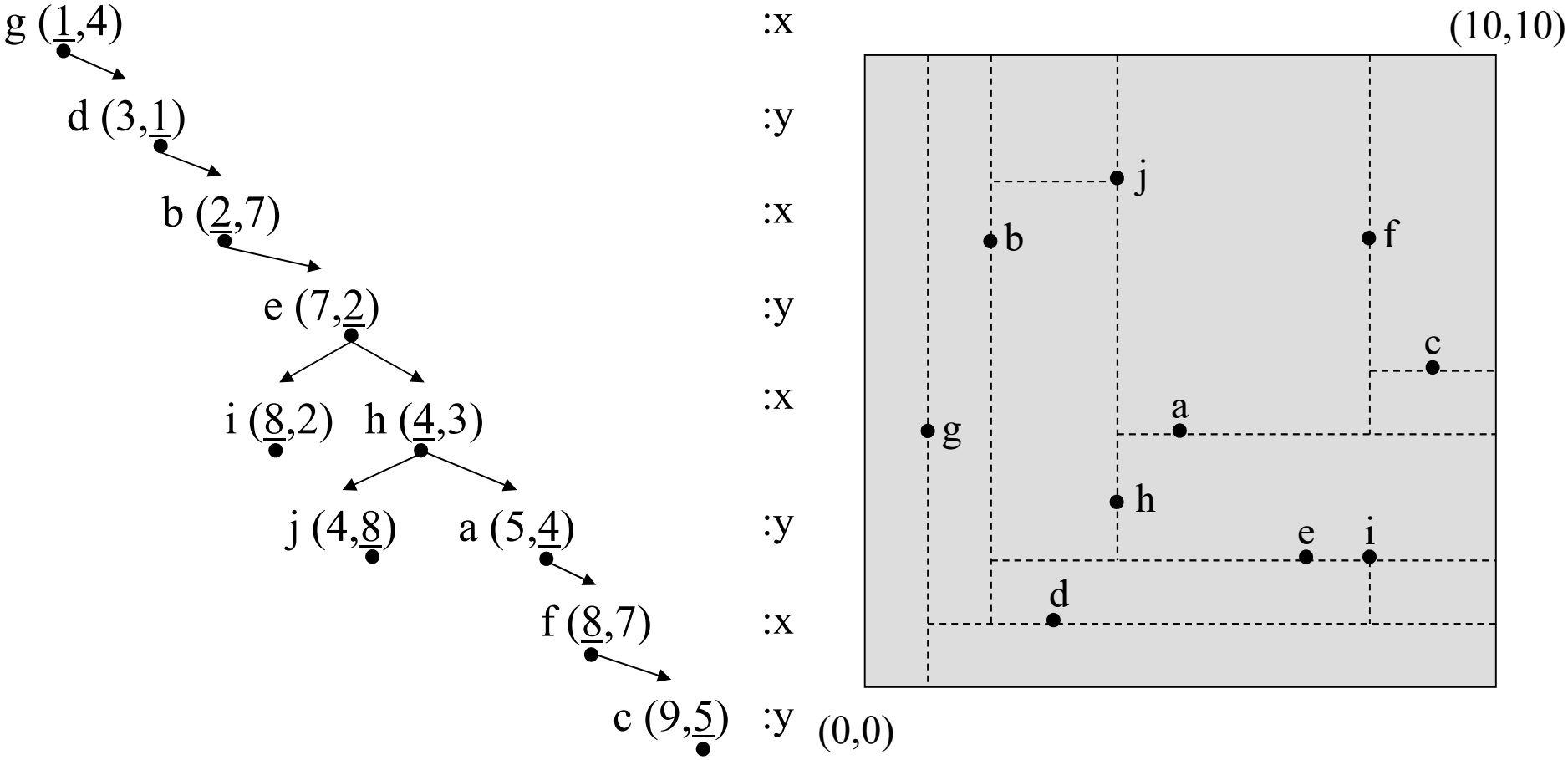
- **(4,8)을 검색**
  - a(루트)와 x값 비교: 4<5이므로 왼쪽 서브 트리로
  - b와 y값 비교: 7<8이므로 오른쪽 서브 트리로
  - j 발견 . 검색 완료





▶ k-d 트리의 단점

- 균형 트리가 아니므로 데이타의 입력 순서나 분포에 따라 트리의 높이가 높아질 수 있음
  → 검색 성능이 떨어짐
- 예) **g, d, b, e, h, a, f, c, i, j** 의 순서로 입력된 예



❖ k-d-B 트리

- **B 트리와 k-d 트리의 결합**
  - 디스크 기반: 디스크 블럭 크기의 노드(페이지)들로 구성
    (블럭은 메인메모리의 페이지와 매핑)
  - 다차원 점 데이타 저장, 검색
  - Multiway search tree, 완전 균형 트리

- **범위 질의(range query) 지원**

▶ 점(point)과 영역(region)

◆ 점(point)과 영역(region)

- 점 : 도메인0×도메인1×…도메인k-1의 한 원소
  - $(x_1, x_2, \ldots, x_{k-1})$, 단 k는 차원(dimension)
- 영역 : 다음 성질을 만족하는 모든 점들의 집합
  - $Min_i \le x_i \le max_i$, $0 \le i \le k-1$, min_i와 max_i는 도메인i의 원소
  - 즉, 각 차원의 인터벌의 카티션 프로덕트, $I_1 \times I_2 \times \ldots \times I_{k-1}$

◆ 점과 영역의 표현

- 점 : k개 필드값을 가진 하나의 레코드 인스턴스
- 영역 : k개 필드값에 대한 범위 조건을 만족하는 점(레코드)의 집합
- 예: 키와 몸무게, 2차원 (그림 10.8)

◆ 범위 질의(range query) 지원

- $min_i \le key_i \le max_i$, $0 \le i \le k-1$

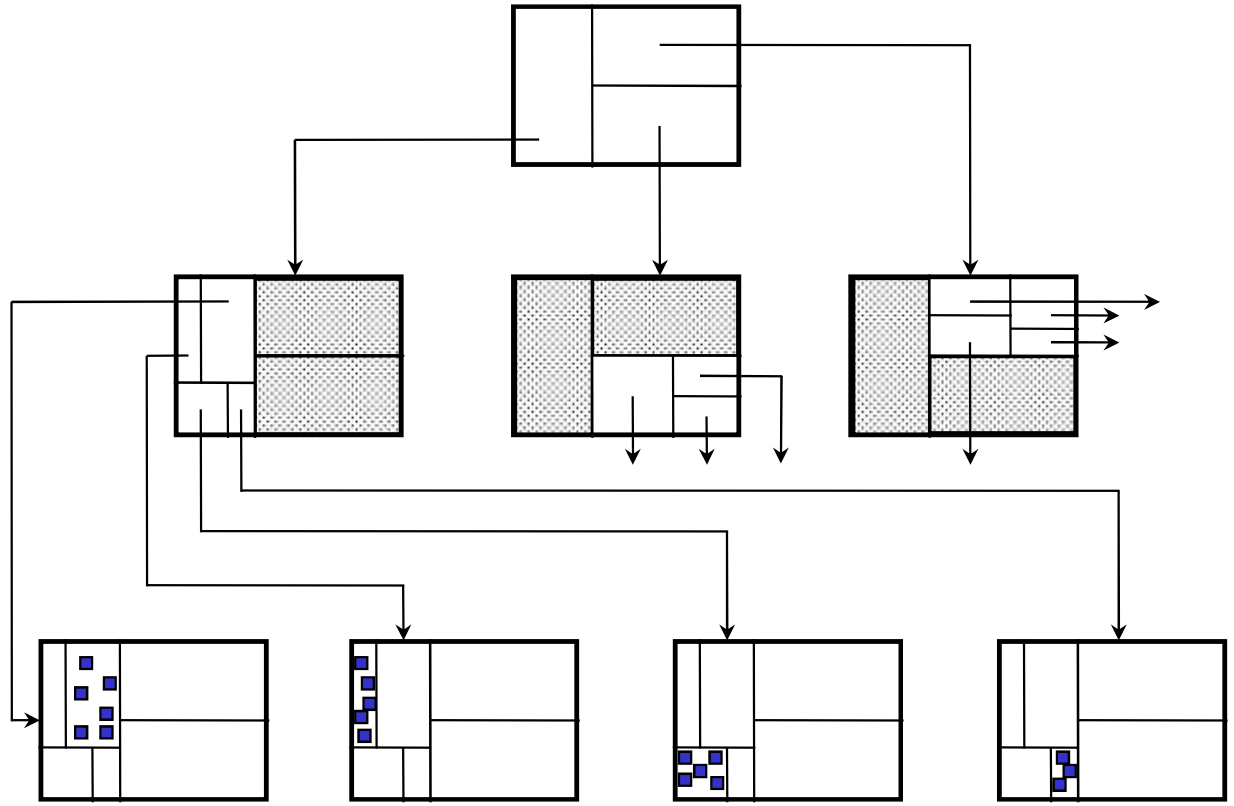
▶ k-d-B 트리의 구조

- 다중키 레코드 검색을 위한 인덱스 레코드의 형식
  - (키0, 키1, …, 키k-1, 주소), 단 k는 차원(dimension)

- **k-d-B** 트리는 페이지의 집합

- 페이지의 유형
  - 영역 페이지(region page) : <영역, page-id> 쌍의 집합
  - 점 페이지(point page) : <점, 주소> 쌍의 집합

▶ 2-d-B 트리의 예



영역 페이지

점
페이지

--- 페이지에 포함된 영역(흰색)

--- 페이지에 포함되지 않은영역 (회색)

--- 점

▶ k-d-B 트리의 특성

① 페이지를 노드로 하고, page-id를 노드 포인터로 하는 다원 탐색 트리

② 루트에서 모든 단말 페이지까지의 경로 길이는 동일

③ 영역 페이지 내의 모든 소영역은 중첩되지 않고 분리(disjoint)

④ 루트 페이지가 영역 페이지이면, 여기에 속한 소영역들의 합은 데이타 전체의 영역. 루트 페이지가 영역 페이지가 아니면, 하나의 점 페이지만 존재.

⑤영역 페이지의 자식이 영역 페이지이면, 자식 영역들의 합은 부모의 영역과 같다.

⑥영역 페이지의 자식이 점 페이지이면, 점 페이지의 모든 점들은 부모의 영역에 속해야 한다.

▶ k-d-B 트리의 연산 : 검색

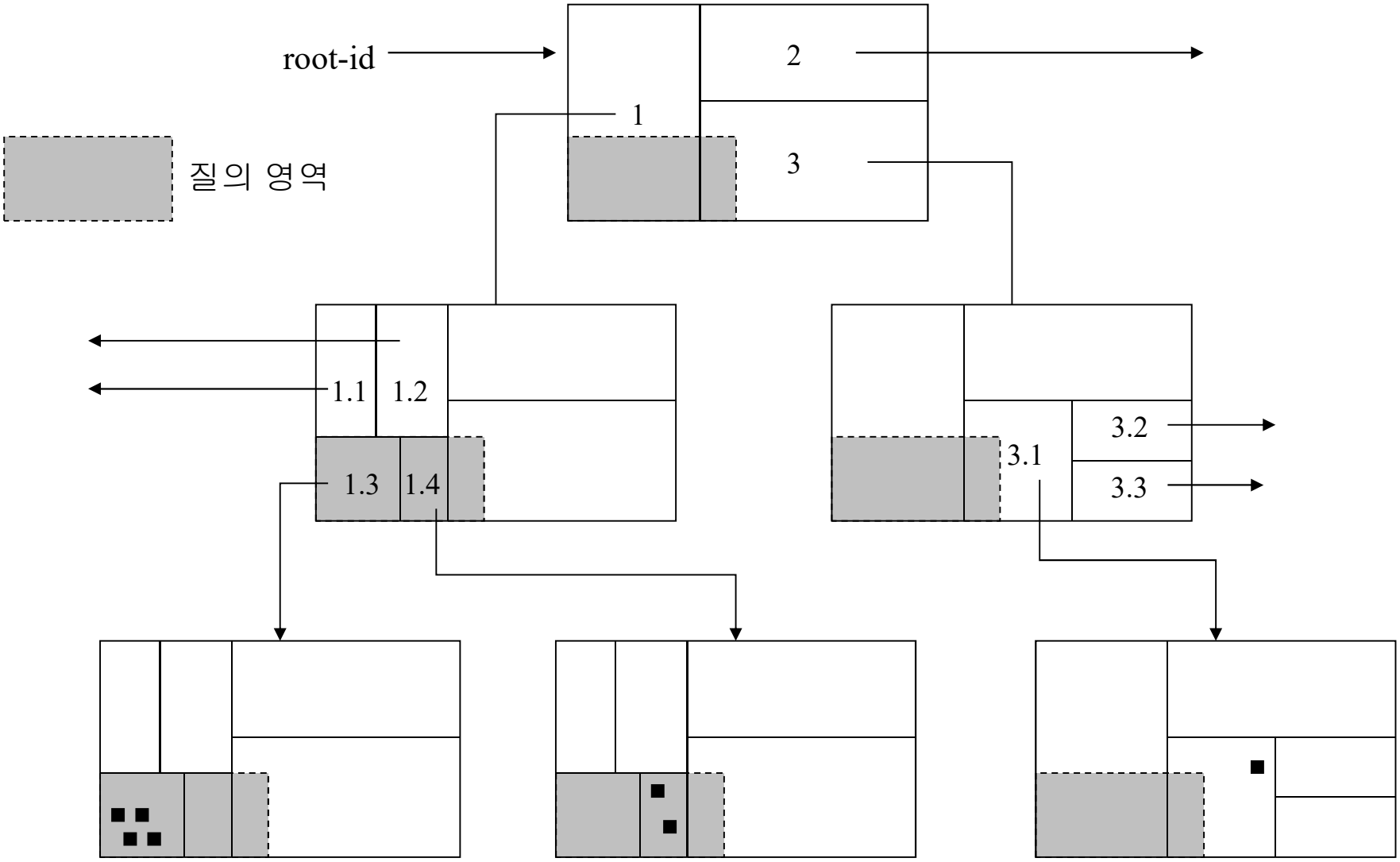
- 범위 질의는 질의 영역(query region)을 명세
  - 영역은 각 차원의 간격의 교차 곱 ($I_1 \times I_2 \times ... \times I_{k-1}$)
- 범위 질의의 종류
  - 부분 범위 질의(partial range query)
    - 모든 차원을 범위로 명세
  - 부분 일치 질의(partial match query)
    - 일부 차원을 점, 나머지는 범위로 명세
  - 완전 일치 질의(exact match) : 모든 차원을 점으로 명세

- 질의 알고리즘

① root-id가 널이면 종료, 그렇지 않으면 변수 page는 루트 페이지를 가리키게 한다.

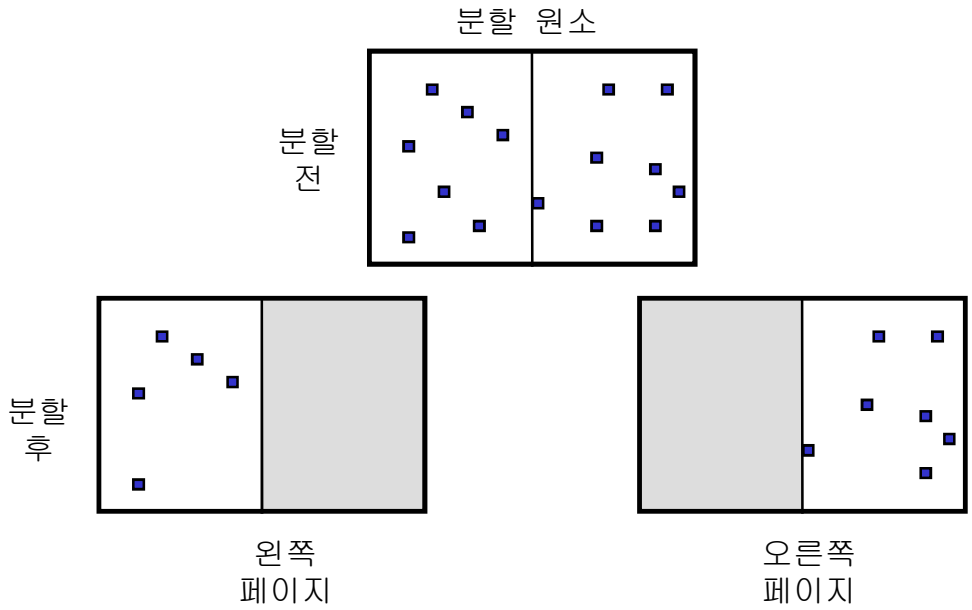
② 변수 page가 점 페이지인 경우, 질의 영역에 속하는 모든 <점, 주소> 쌍에 대해 주소에 있는 레코드를 검색

③ 영역 페이지인 경우, 질의 영역에 중첩하거나 포함되는 모든 <영역, page-id> 쌍에 대해 변수 page가 page-id 값을 갖게 하고, ② 혹은 ③ 을 반복


root-id
1
2
3
질의 영역
1.1
1.2
1.3
1.4
3.1
3.2
3.3

▶ k-d-B 트리의 연산 : 삽입

- 점 페이지의 분할
  - 삽입할 점을 저장할 점 페이지를 검색.
  - 점 페이지에 <점,주소> 쌍을 삽입. 오버플로우가 발생하지 않으면, 종료함.
  - 오버플로우가 발생하면, 적절한 점 x'을 선택하여 점 페이지를 분할
  - 이때 분할된 두 영역이 비슷한 수의 점을 포함하도록 x'을 선택



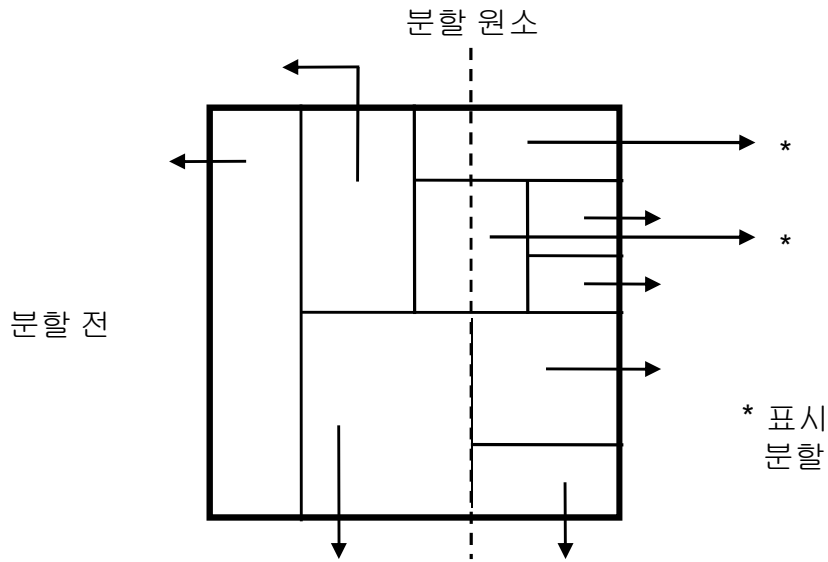
◆ 영역 페이지의 분할

- 부모 페이지의 해당 <영역, page-id> 쌍에서 영역 $I_x \times I_y$ (단, $I_x=[min_x,max_x)$, $I_y=[min_y,max_y)$)를 선택한 x'으로 분할.
  왼쪽 영역(left region) : $[min_x,x') \times I_y$
  오른쪽 영역(right region) : $[x',max_x) \times I_y$
- 생성한 두 분할 영역에 left-id 및 right-id 부여.
- 해당 <영역, page-id> 쌍을 분할한 <왼쪽영역, left-id> 쌍과 <오른쪽영역, right-id> 쌍으로 대체.

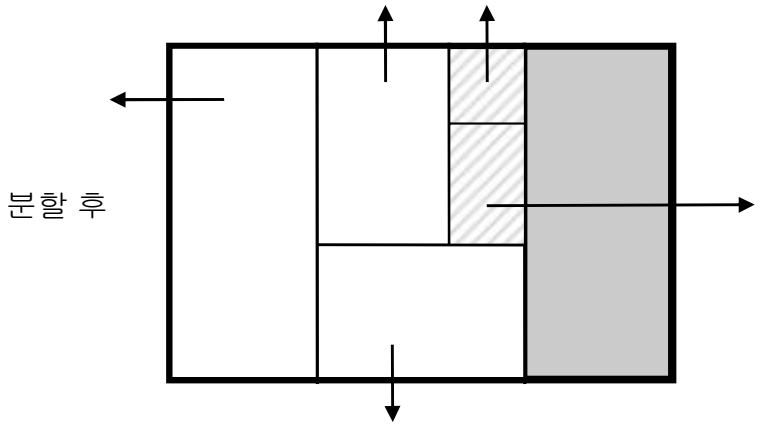
◆ 영역 페이지 분할의 파급

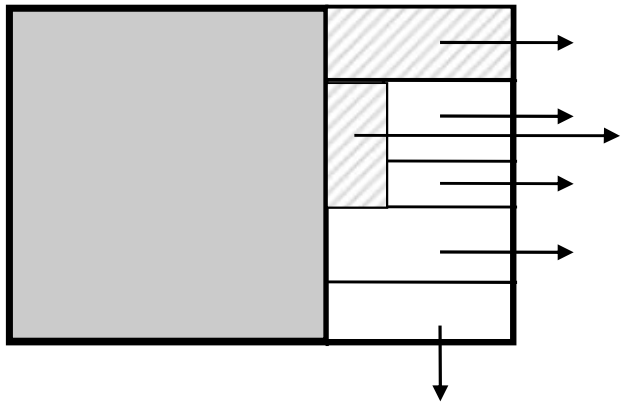
- <영역, page-id> 쌍의 분할로 인하여 영역 페이지에서 오버플로우가 발생하면, 위의 영역 페이지 분할을 반복함.
- 루트에서 오버플로우가 발생하면, 트리의 레벨이 하나 증가함.

– 영역 페이지의 분할



* 표시된 부분이
분할된다



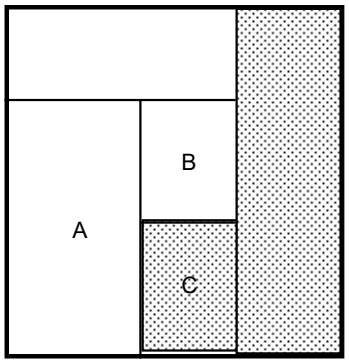


▶ k-d-B 트리의 연산 : 삭제

- 완전 일치 질의로 탐색, 제거
  - 공백 점 페이지를 허용하고, 저장 공간 효율성도 요구하지 않음으로 기본적인 삭제는 매우 단순함.

- **B-트리의 기법을 적용할 수 있음**
  - 언더플로우 발생시, 두 영역간에 재분배
  - 공간 이용률을 높이기 위해, 두 영역을 한 영역으로 합병

- 합병이 가능(joinable)하지 못한 경우
  - A와 B, 혹은 A와 C는 합병 불가능



❖ 격자 화일(Grid file)

- ◆ 격자 화일
  - – 전체 공간을 하나 이상의 격자로 분할
  - – 데이타 추가에 따라 기존 격자를 분할하여 새로운 격자 구성

- ◆ 특징
  - – 디스크 기반
    - ◆ 대용량 데이타 처리 (격자 단위로 저장)
  - – 해시 기반
    - ◆ 일반적으로 두 번의 디스크 접근으로 데이타 검색

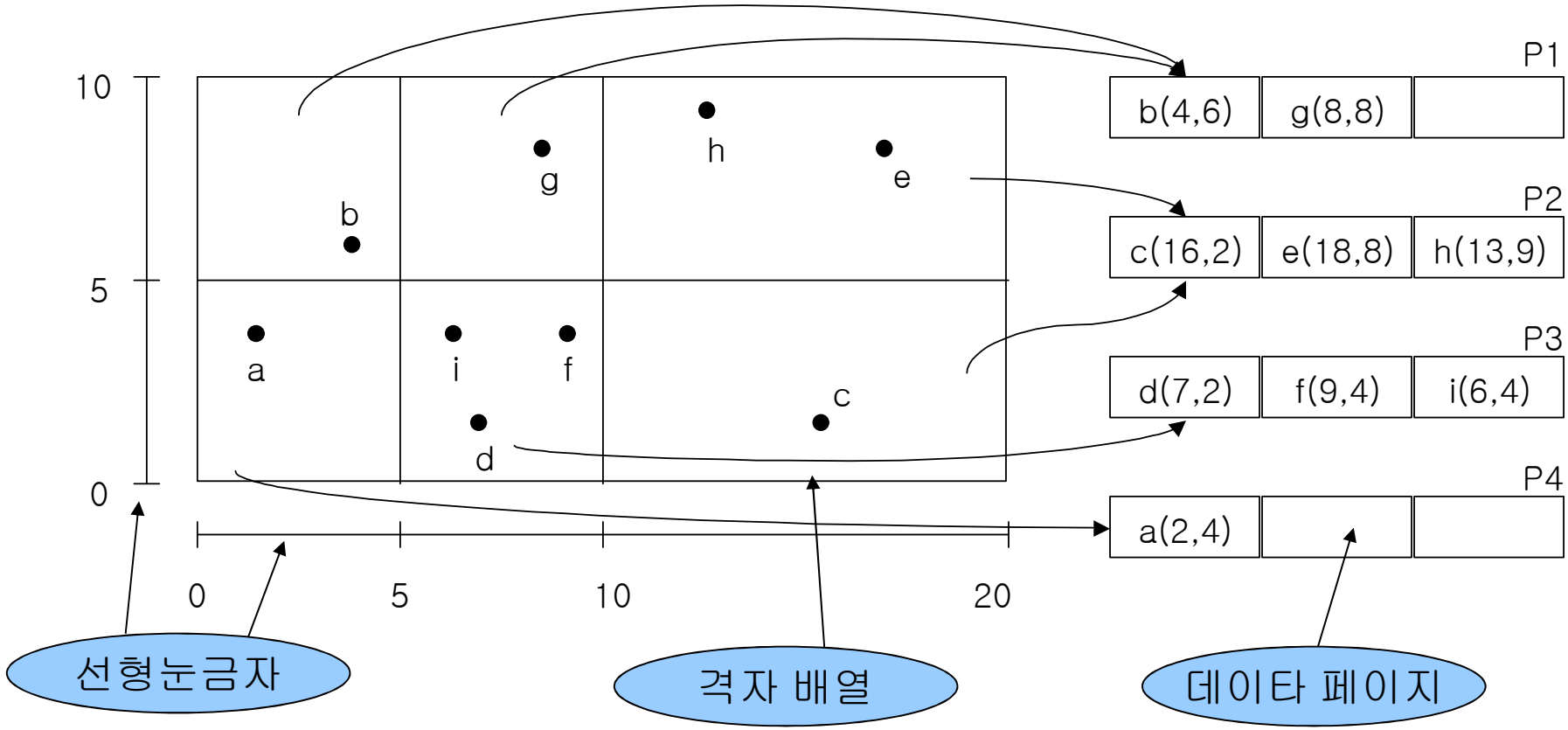
▶ 격자 화일의 구성

- **k-차원 격자 화일**
  - 격자 디렉터리(grid directory)
    - k개의 선형 눈금자(liner scale) : 해싱 역할을 함
      - 격자 디렉토리를 구성하는 각 차원별 눈금 정보
      - 주기억 장치에 유지
    - k-차원의 격자 배열(grid array)
      - 선형 눈금자에 의해 분할된 격자
      - 하나 이상의 격자 블록으로 구성
      - 격자 블록에는 해당 데이타 페이지 번호 저장
      - 디스크에 저장
  - 데이타 페이지
    - 실제 데이타 저장되는 장소
    - 디스크에 저장

- **격자 블록과 데이타 페이지**
  - 기본적으로 하나의 격자 블록당 하나의 데이타 페이지를 가리킴.
  - 두개 이상의 격자 블록이 하나의 데이타 페이지 가리키며 공유 가능

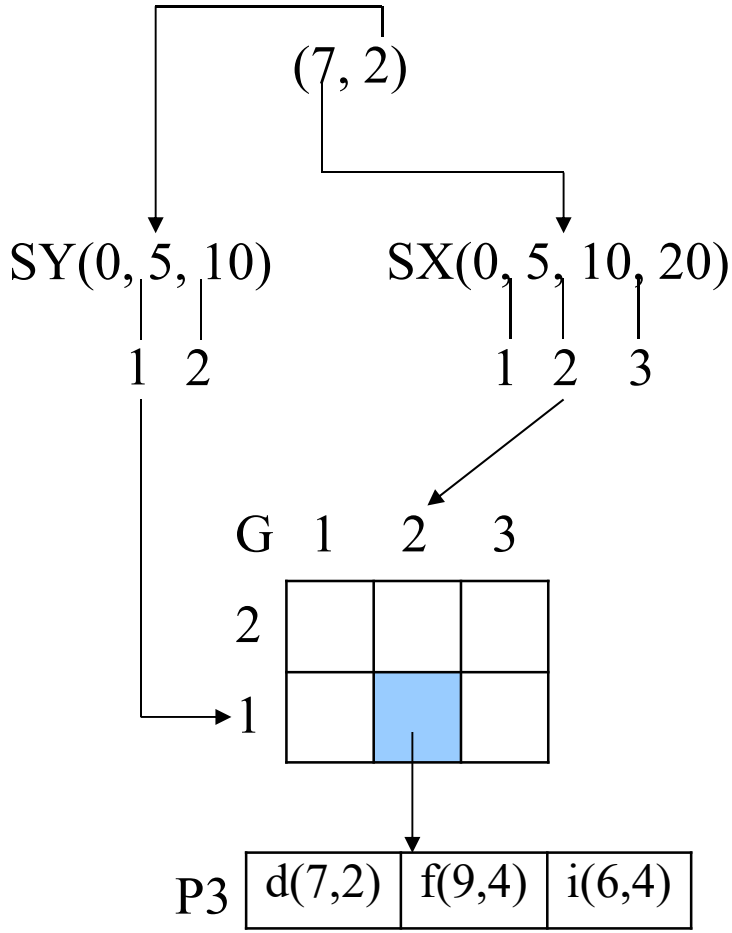
◆ **Example: 2-차원 격자 화일**

– 각 데이터 페이지는 최대 3개의 점을 저장



▶ 격자 화일의 검색

- **x=7, y=2**인 데이타를 검색하라
  - 선형 눈금자(SX, SY) 사용
    - 주기억장치
    - x=7: 두 번째 범위(SX),
      y=2: 첫번째 범위(SY)
    - 격자 배열 인덱스 = (2, 1)
  - 격자 배열(G) 접근
    - 디스크
    - G(2, 1)
    - 데이타 페이지 번호 = 3
  - 데이타 페이지(P) 접근
    - 디스크
    - P3
    - 데이타 d 검색
- 두 번의 디스크 접근

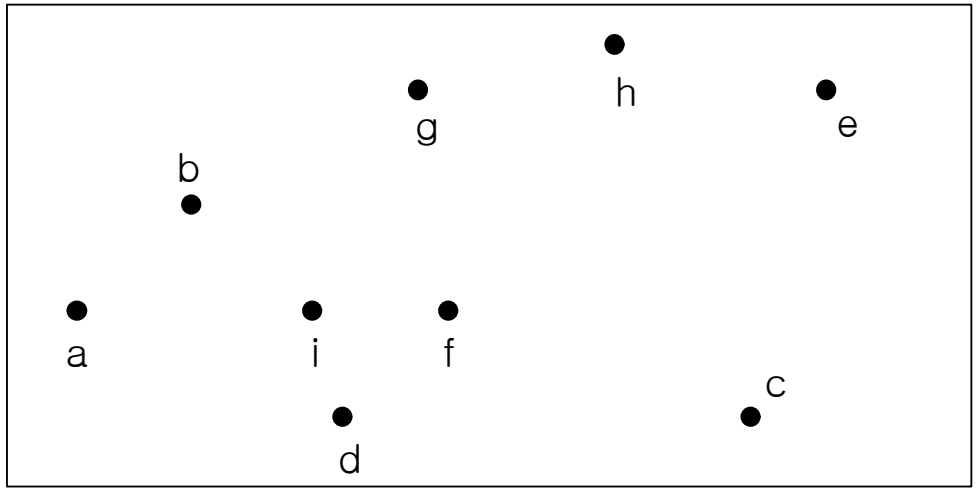


- 해싱 (모조키)
  - x 좌표: h1(x) → grid array의 x 축 인덱스
  - y 좌표: h2(y) → grid array의 y 축 인덱스

▶ 격자 화일의 삽입

◆ 예제 데이타

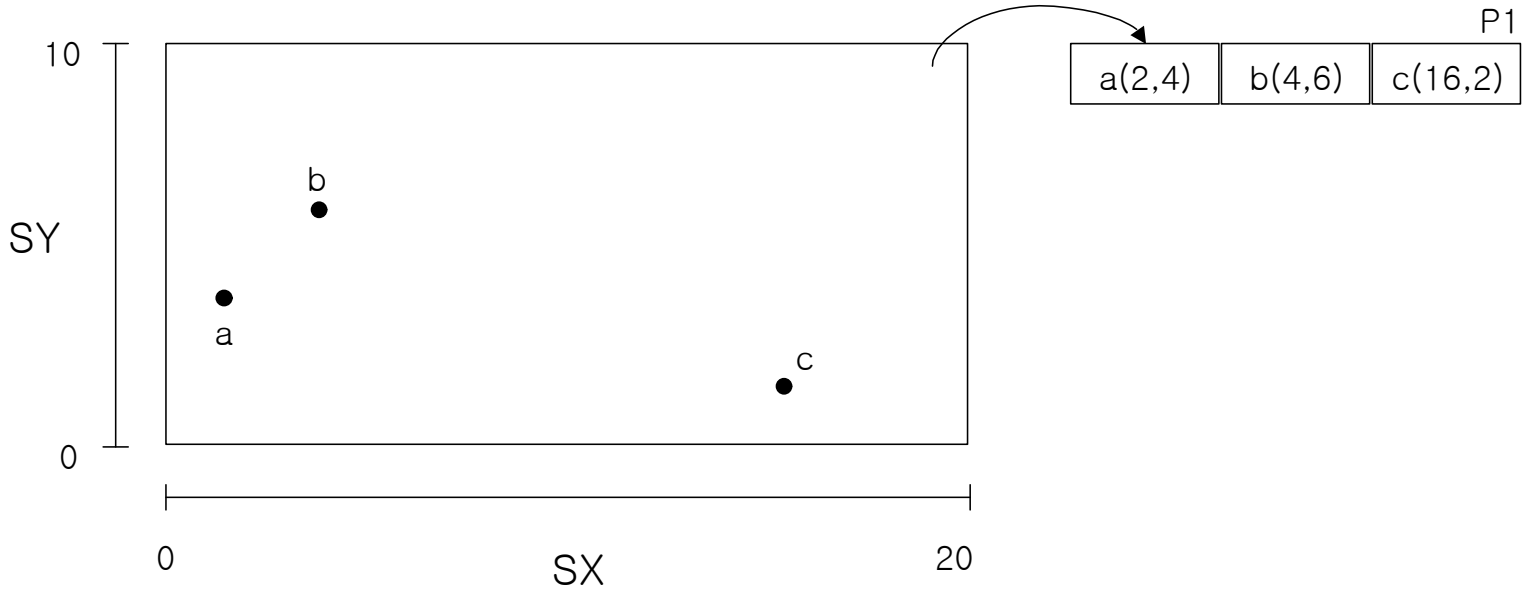
데이타	위치
a	(2, 4)
b	(4, 6)
c	(16, 2)
d	(7, 2)
e	(18, 8)
f	(9, 4)
g	(8, 8)
h	(13, 9)
i	(6, 4)



◆ 데이타 페이지가 최대 3개의 데이타 저장

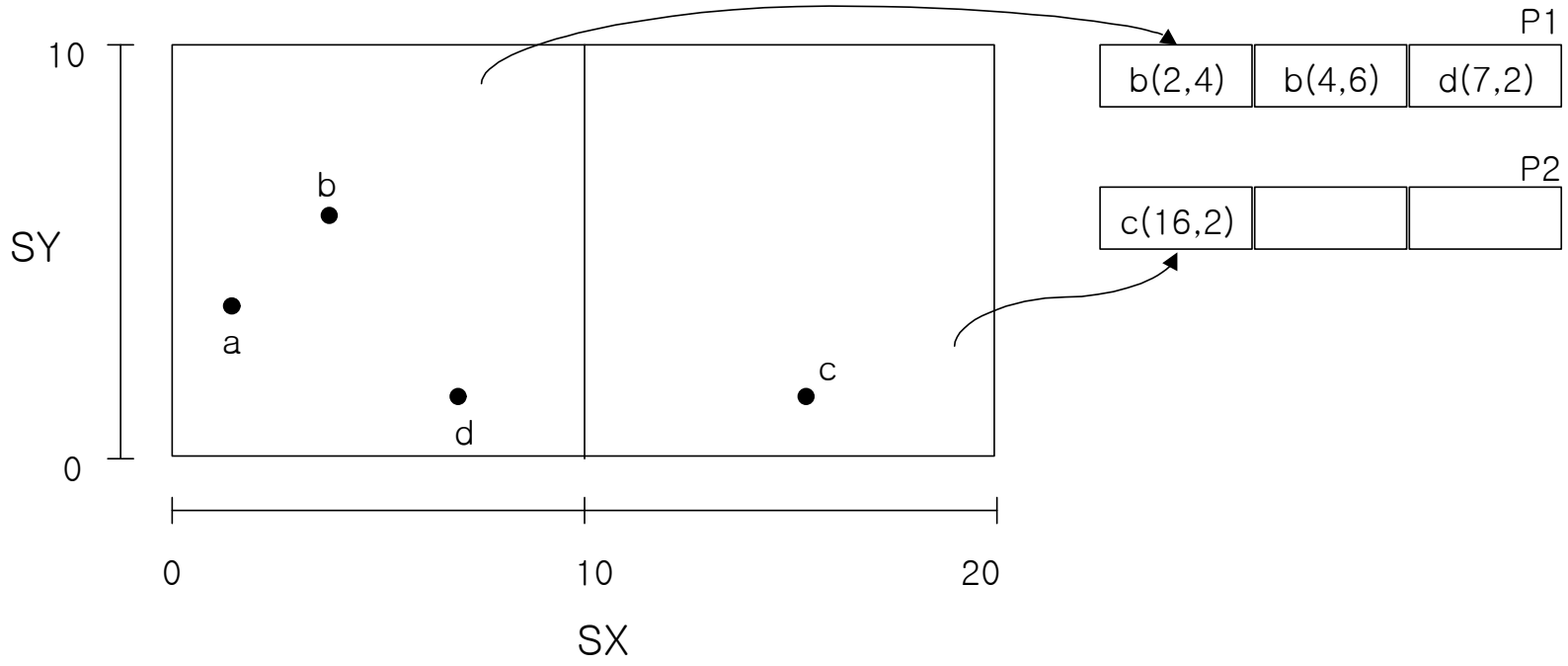
- 격자의 분할 방법
  - 여러 방법이 가능
    - 각 축을 순환적으로 분할
  - 해쉬 함수가 만들어 질 수 있는 방법이어야 함.

◆ **a, b, c 삽입**



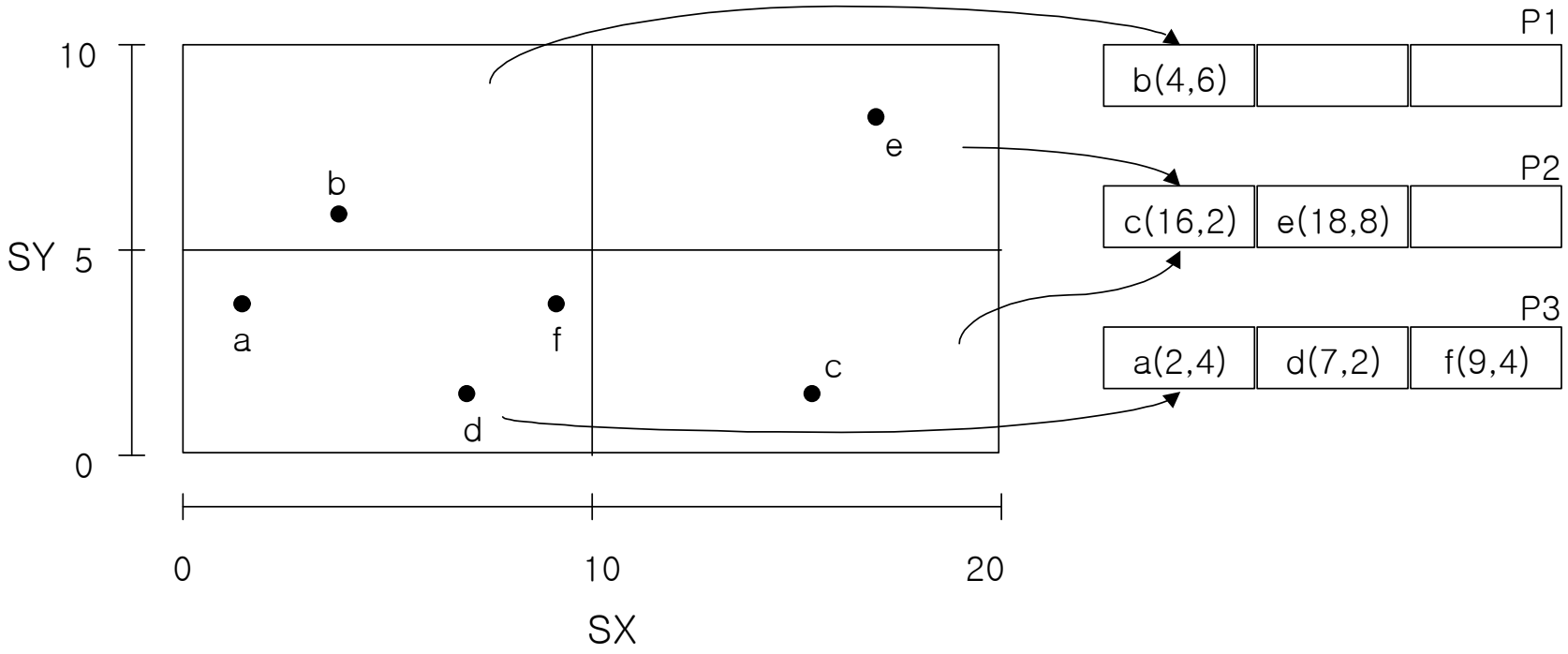
◆ **d 삽입**

– 격자 분할



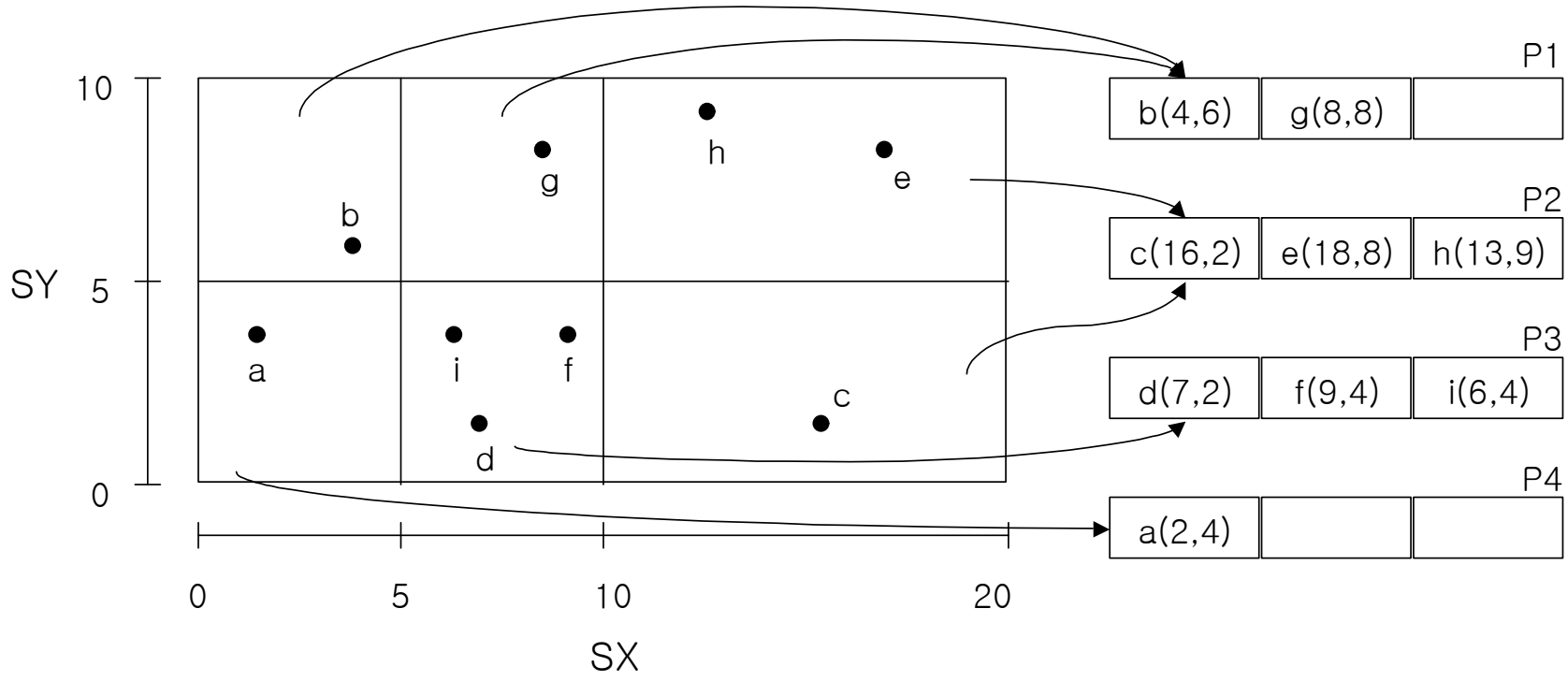
◆ **e, f 삽입**

– f 삽입 시 격자 분할



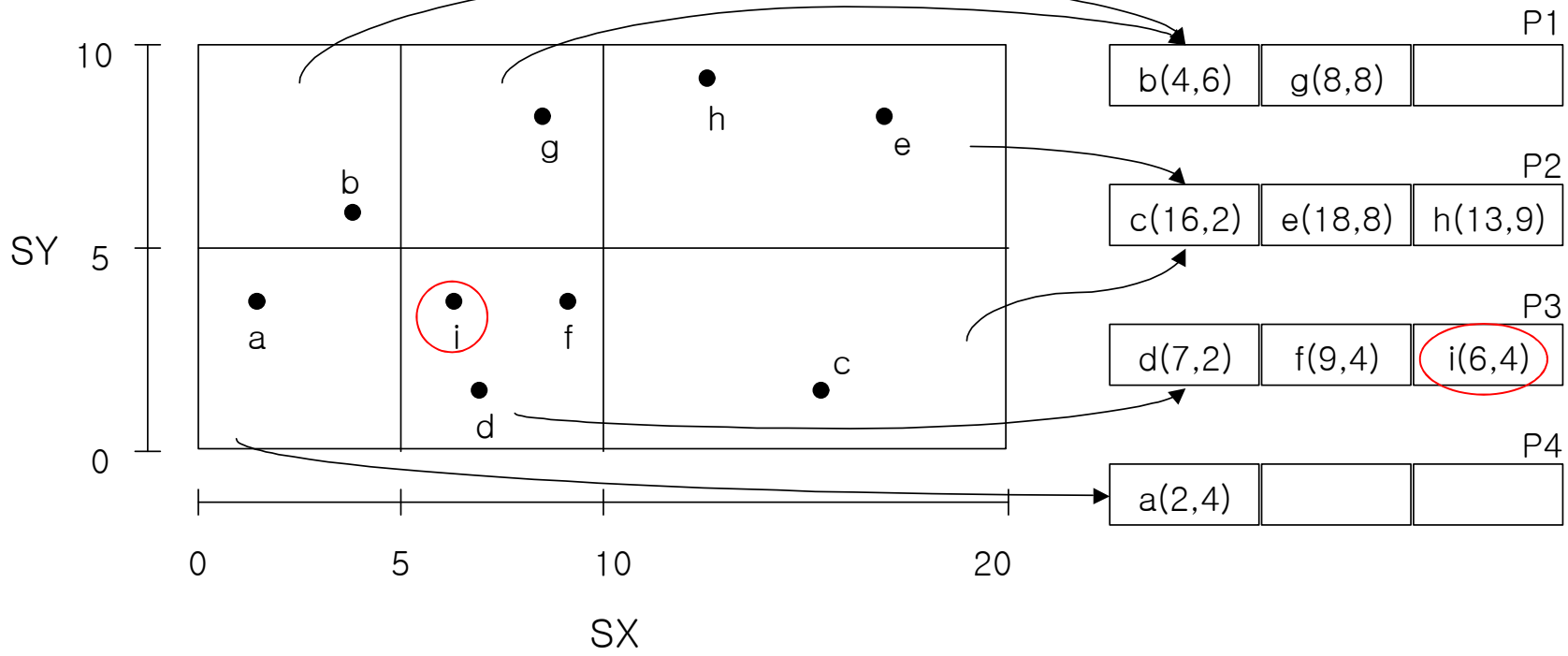
◆ **g, h, i 삽입**

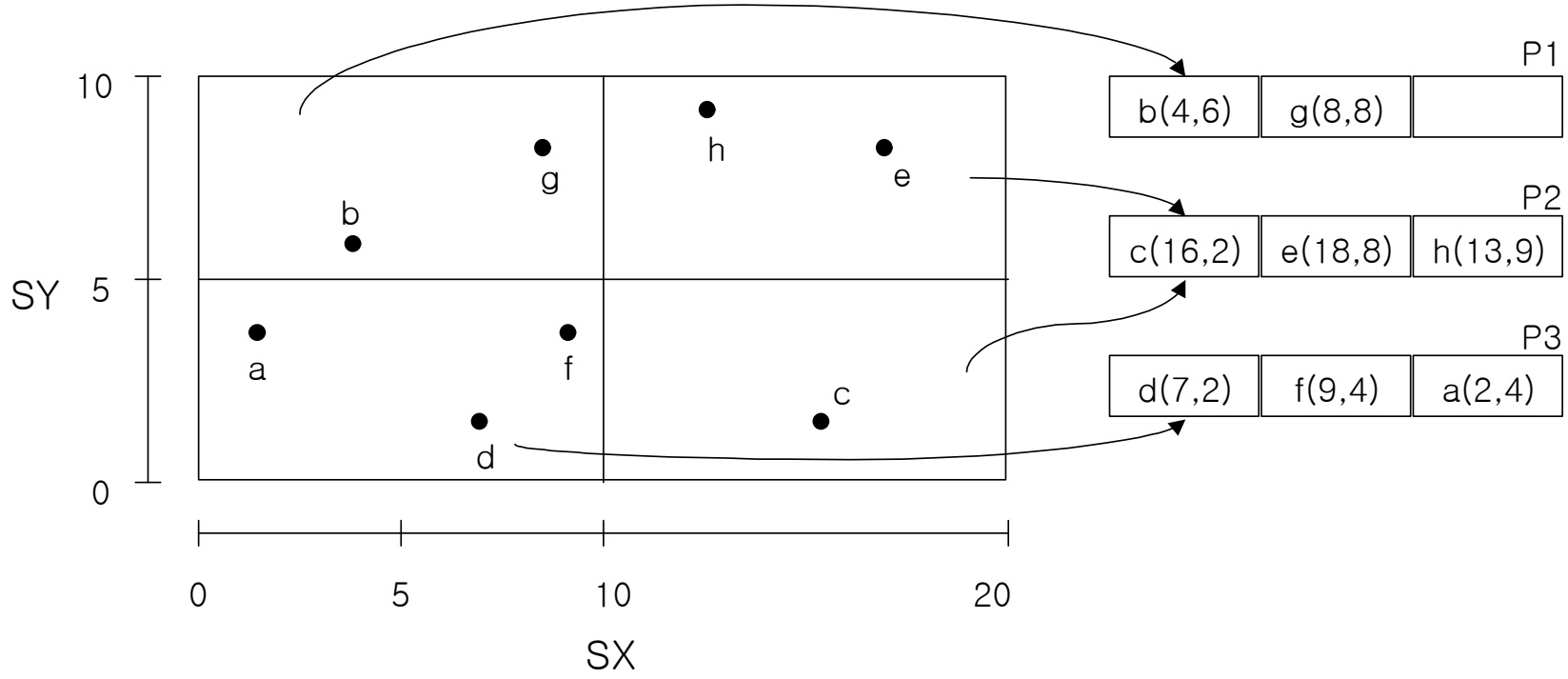
– i 삽입 시 격자 분할



▶ 격자 화일의 삭제

- 점 i의 삭제
  - 데이타 페이지 P3, P4는 하나의 페이지 P3로 합병 가능
  - P1은 두 개의 격자 블록이 공용 하다가 합병된 격자 블록이 사용
  - x=5 분할 제거
    - 격자 블록 합병
    - 선형 눈금자 수정




10
5
0
SY
0
5
10
20
SX
a
b
c
d
e
f
g
h
P1
b(4,6)
g(8,8)
P2
c(16,2)
e(18,8)
h(13,9)
P3
d(7,2)
f(9,4)
a(2,4)

▶ 격자화일의 성능

- 장점
  - 키가 균등하게 분포되어 있을 때 효율적

- 단점
  - 격자 분할시 분할할 필요가 없는 격자 블록까지 분할하므로, 불필요하게 격자 수가 증가.
  - 격지 합병시 비용이 많이 듬.

❖ 사분트리 (Quadtree)

- 사분트리
  - 공간을 순환적으로 분해하는 계층적 자료구조(hierarchical data structure)
  - 다양한 유형이 있으며, 다음 기준에 따라 분류함 (사분트리의 분류 기준)
    - ① 표현하고자 하는 자료의 유형
    - ② 공간 분해 과정의 원칙
    - ③ 해상도(resolution) – 분해 과정의 적용 횟수를 고정, 또는 입력 자료 값에 따라 가변

- 사분트리로 표현하는 자료의 유형
  - 점(point), 영역(region), 곡선(curve), 표면(surface), 볼륨(volume) 등을 표현
  - 개체의 경계/내부를 표현하는 경우에 따라 자료구조가 달라짐.
    - 개체의 경계를 표현하는 경우 : 곡선, 표면 데이타
    - 개체의 내부를 표현하는 경우 : 영역, 볼륨 데이타

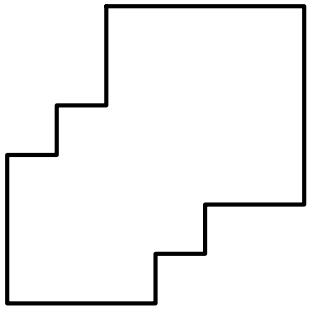
- 공간의 분해 원칙
  - Image space hierarchy : 각 레벨마다 공간을 일정 크기의 동일한 부분들로 분해
  - Object space hierarchy : 입력 자료 값에 따라 서로 다른 크기의 공간으로 분해

▶ 영역 사분트리(region quadtree)

- 이차원 영역 데이타 표현에 많이 사용
- 영역을 크기가 동일한 **4개 소영역(subregion), 즉 사분면(quadrant)으로 연속적으로 분할**
  - 이미지를 표현하는 2진수의 배열을 연속적으로 동일한 크기의 사분면들로 분할 : 가변 해상도의 자료 구조
  - 루트 노드는 전체 배열에 대응
  - 자식 노드들은 각 소영역의 사분면(quadrant) 표현
    - NW, NE, SW, SE순
- 영역 사분트리
  - 리프 노드
    - 흑색노드(black node): 대응되는 블럭이 모두 1
    - 백색노드(white node): 대응되는 블럭이 모두 0
  - 비단말 노드
    - 회색 노드(gray node): 0과 1 모두 가짐

◆ 예: 이미지의 표현

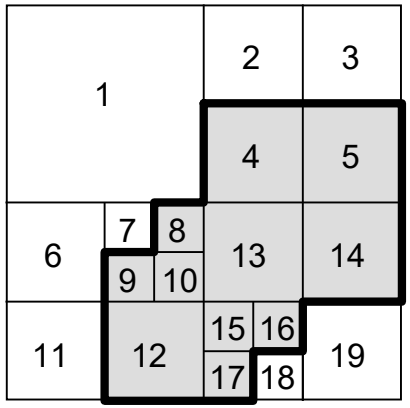
– 1은 그림 영역, 0은 그림의 외부 영역



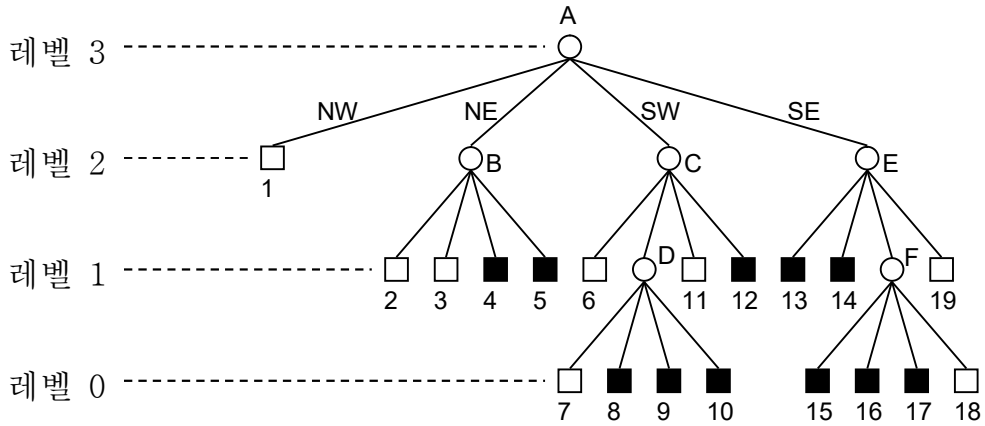
(a)

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	1	1	1	1	1
0	0	1	1	1	1	1	1
0	0	1	1	1	1	0	0
0	0	1	1	1	0	0	0

(b)



(c)



(d)

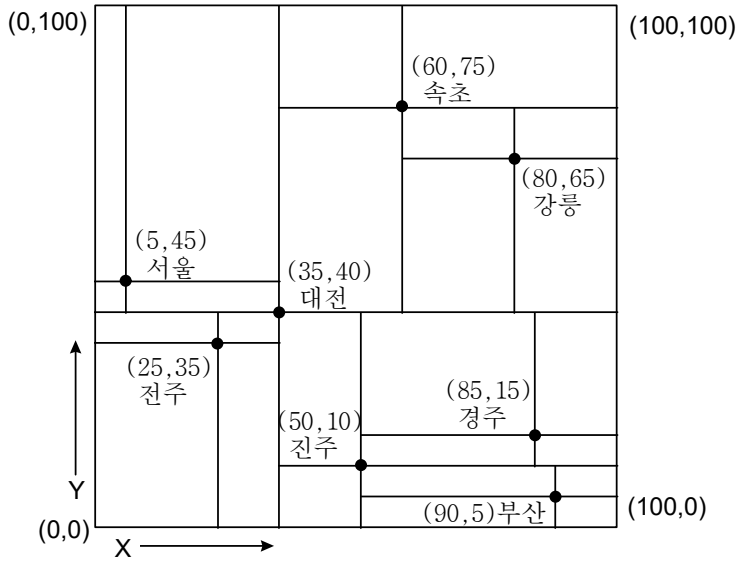
해상도: 차수 3
(3회 분해)

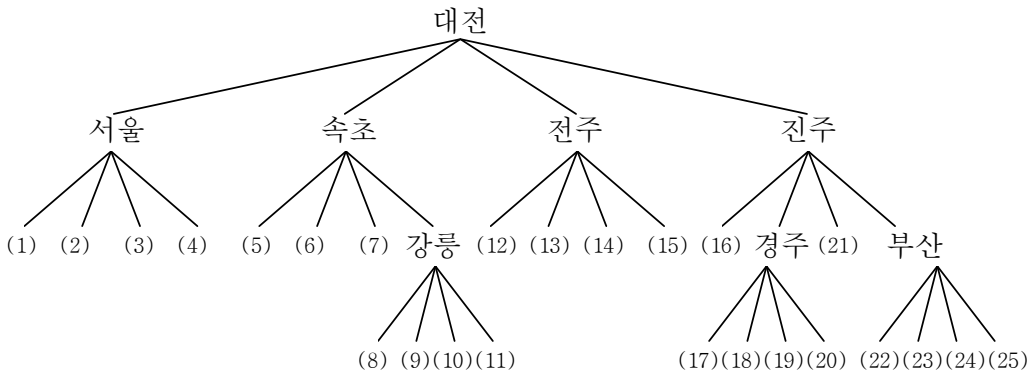
▶ 점 사분트리 (point quadtree)

- 영역을 크기가 동일하지 않은 **4**개의 소영역**(subregion)**, 즉 사분면**(quadrant)**으로 연속적으로 분할
  - 루트 노드는 전체 영역에 대응
  - 자식 노드들은 각 소영역의 사분면(quadrant) 표현
    - NW, NE, SW, SE순
- 다차원 점 데이타 표현에 많이 사용
  - 다차원 데이타를 위한 이진 탐색 트리의 일반화
- 이차원 점 데이타는
  - x 좌표, y 좌표, 네 개(NW, NE, SW, SE)의 포인터 필드, 데이타 필드를 가지는 노드로 표현
  - 예: 서울의 경우 <5, 45, q_1, q_2, q_3, q_4, 서울>

◆ 예: 도시 좌표의 표현

도시명	X	Y	기타 정보
대전	35	40	
진주	50	10	
속초	60	75	
강릉	80	65	
서울	5	45	…
전주	25	35	
경주	85	15	
부산	90	5	





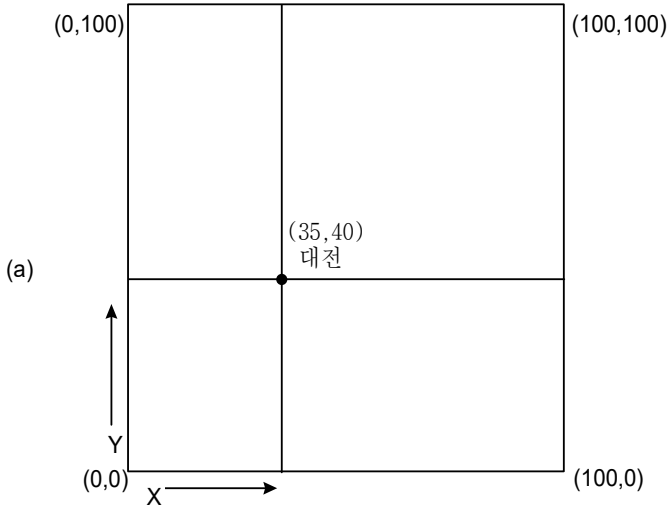
- 단말 노드가 버켓 포인터를 가진다면 인덱스 역할 가능
- 예: 버켓 1에는 $0 \le x < 5$와 $45 \le y < 100$의 값을 갖는 점 데이타들을 저장

▶ 점 사분트리의 삽입

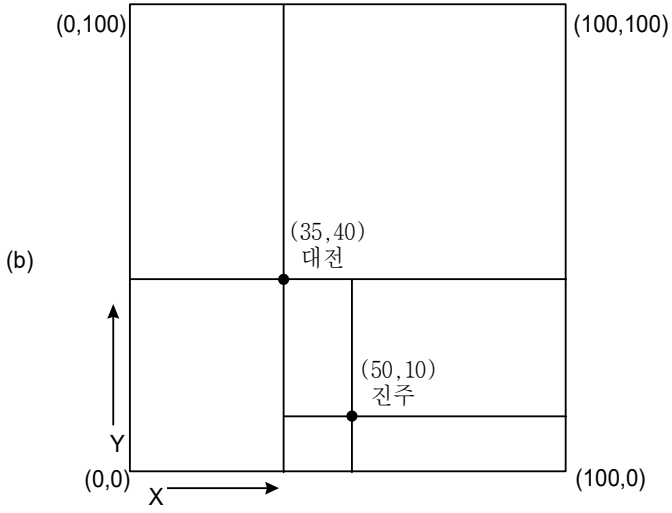
- 이진 탐색 트리에 대한 삽입과 유사한 방법
  - 삽입할 레코드의 위치를 x, y 좌표 값을 바탕으로 탐색
  - 노드의 좌표 값과 삽입할 데이타의 좌표 값을 비교
  - 이 과정을 반복한 후, 도착한 리프 노드에 레코드 삽입

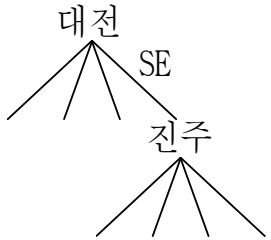
- 점 사분트리의 구축 비용 = 트리의 총 경로 길이
  - 평균 삽입 비용(실험적) : $\mathbf{O(N\log_4 N)}$
  - 한 노드의 탐색 비용 : $\mathbf{O(\log_4 N)}$

- 최적 점 사분트리 구성 방법
  - 임의 노드의 어떤 서브트리도 전체 노드 수의 반 이상을 갖지 않는 트리로 정의한다.
  - 이를 위해, 모든 점 데이타들을 하나의 좌표축(x) 값으로 정렬하고 다른 좌표축(y) 값은 보조 키로 사용한다
  - 루트는 정렬 화일의 중간 값을 갖고, 나머지는 4개 부속 그룹으로 나누어 루트의 네 서브트리가 되도록 한다

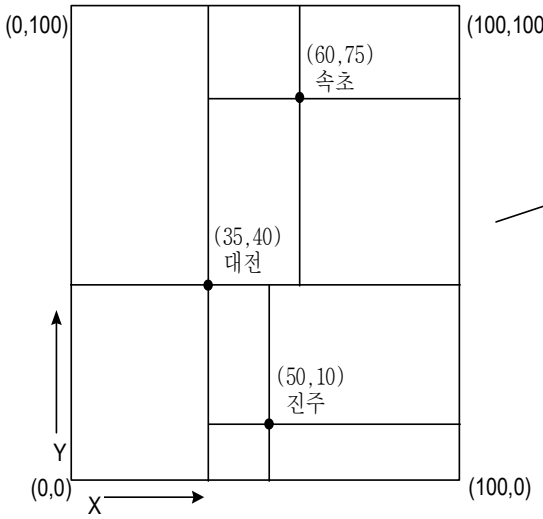

(0,100)
(100,100)
(35,40)
대전
(a)
Y
(0,0)
X
(100,0)

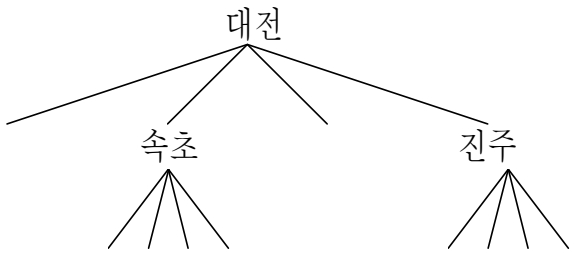

대전


(0,100)
(100,100)
(35,40)
대전
(b)
(50,10)
진주
Y
(0,0)
X
(100,0)

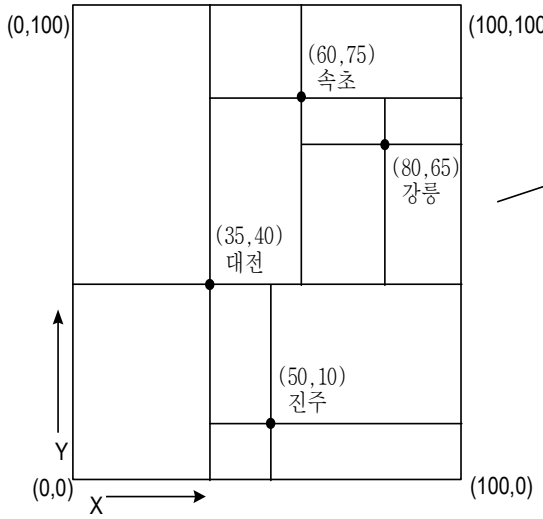

대전
SE
진주

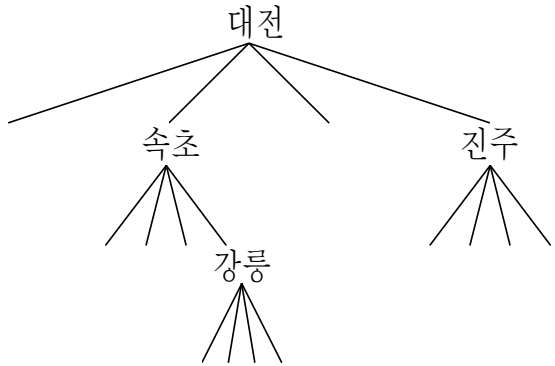
(c)


(0,100)
(100,100)
(60,75)
속초
(35,40)
대전
(50,10)
진주
Y
(0,0)
X
(100,0)


대전
속초
진주

(d)


(0,100)
(100,100)
(60,75)
속초
(80,65)
강릉
(35,40)
대전
(50,10)
진주
Y
(0,0)
X
(100,0)


대전
속초
진주
강릉

▶ 점 사분트리의 검색

- 탐색 공간을 좁혀 나가는 기법
  - 한 레벨 아래로 갈수록 탐색 공간은 1/4로 감소
  - 리프 노드가 가리키는 버킷에서 원하는 데이타 검사
  - (예) “(95, 8)에 위치한 도시를 검색하라”
    - 대전(35, 40)의 SE, 진주(85, 15)의 SE,
    - 부산(90, 5)의 NE에 속하므로 버킷 23을 조사

- 범위 탐색, 근접 탐색에도 적합
  - (예) “좌표 값 (83, 10)에서 거리 8 이내에 존재하는 모든 도시를 검색하라”
    - 대전(35, 40)의 SE를 검색하고
    - 진주 (50, 10)의 NE와 SE만 검색하면 됨

▶ 점 사분트리의 삭제

- 삭제는 매우 복잡
  - 삭제할 노드를 루트로 하는 서브 트리에 있는 모든 노드들이 재삽입 되어야 함

- 이상적인 삭제 방법
  - 삭제할 노드 A를 어떤 노드 B로 대치한 후 삭제
  - 이때, 노드 B는 노드 A를 루트로 하는 서브트리의 노드들이 속해 있는 사분면에 영향을 주지 않아야 한다.
  - 이런 B가 존재 않을 경우 가장 근접한 것으로 대체하고 영역에 존재하는 점 데이타들을 재삽입

❖ R-트리

◆ R-트리

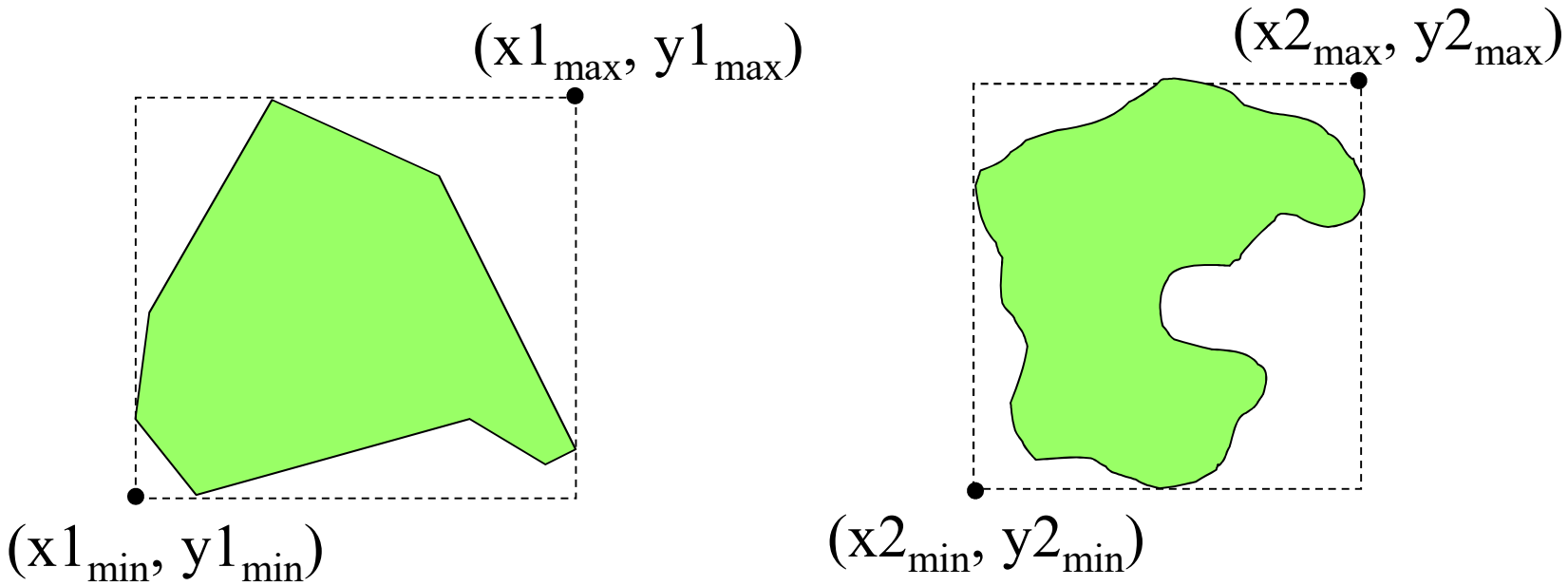
- B-트리를 다차원으로 확장시킨 완전 균형 트리
- 선, 면, 도형 등 다양한 다차원 공간 데이타 저장 가능(SAM)

◆ 특징

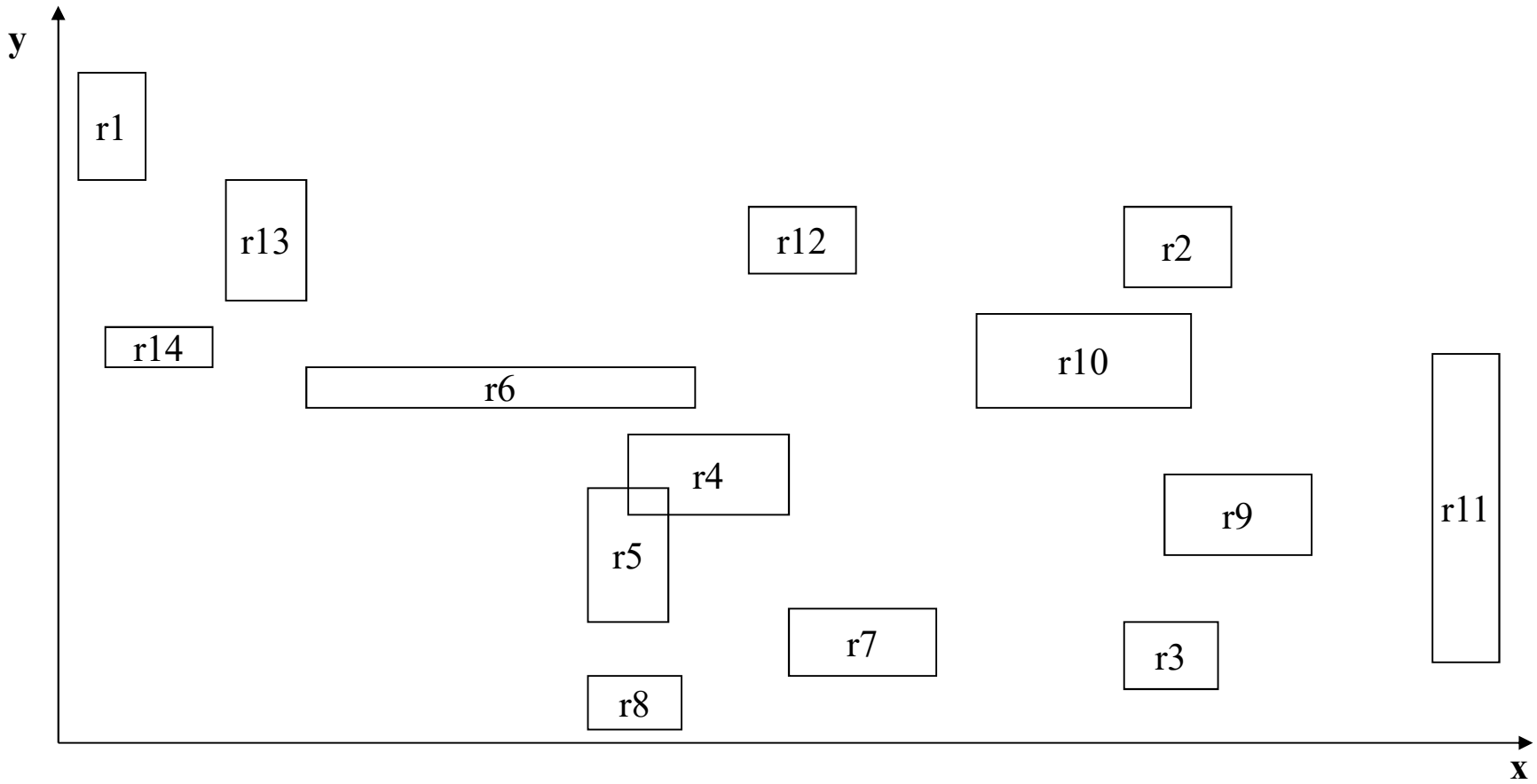
- 루트노드가 아닌 노드는 최소 m, 최대 M개의 엔트리를 포함한다.(m<=M/2)
- 루트노드는 리프가 아닌 경우 최소 2개의 엔트리를 포함한다.
- 완전 균형트리(모든 리프노드는 같은 레벨)

▶ MBR

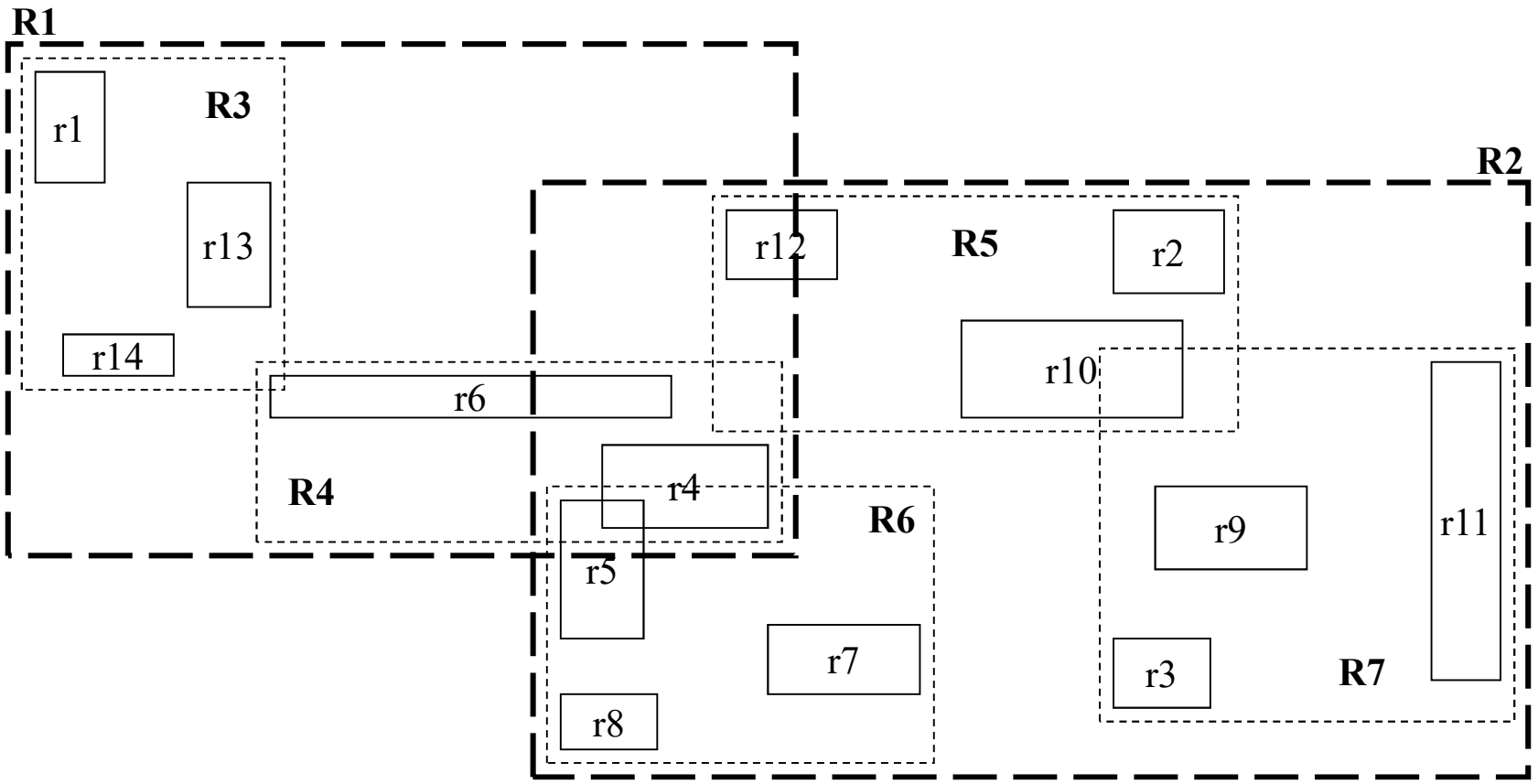
- 복잡한 공간 도형을 저장하기 위해 **MBR(Minimum Bounding Rectangle)** 을 이용



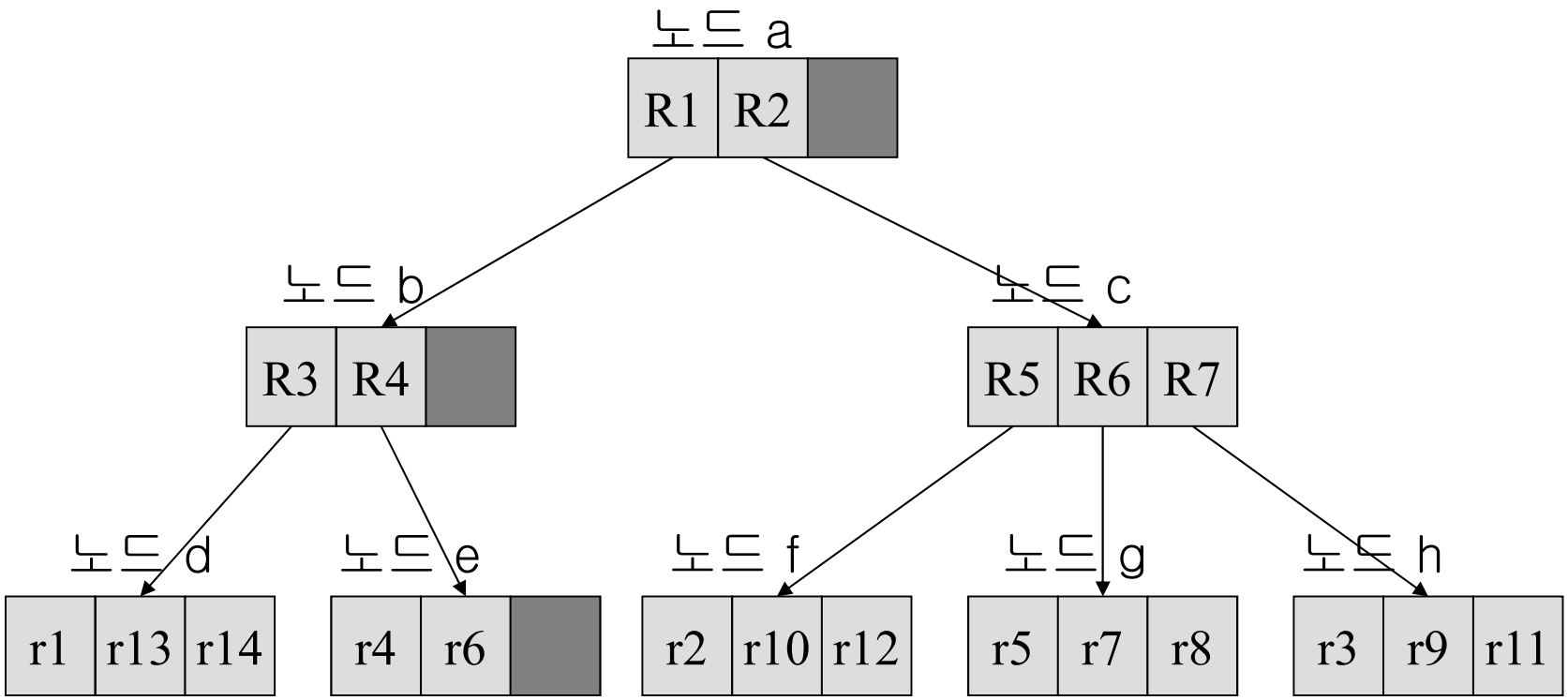
▶ 데이타 예



▶ R-트리 구성 예



▶ R-트리 노드 구성 예

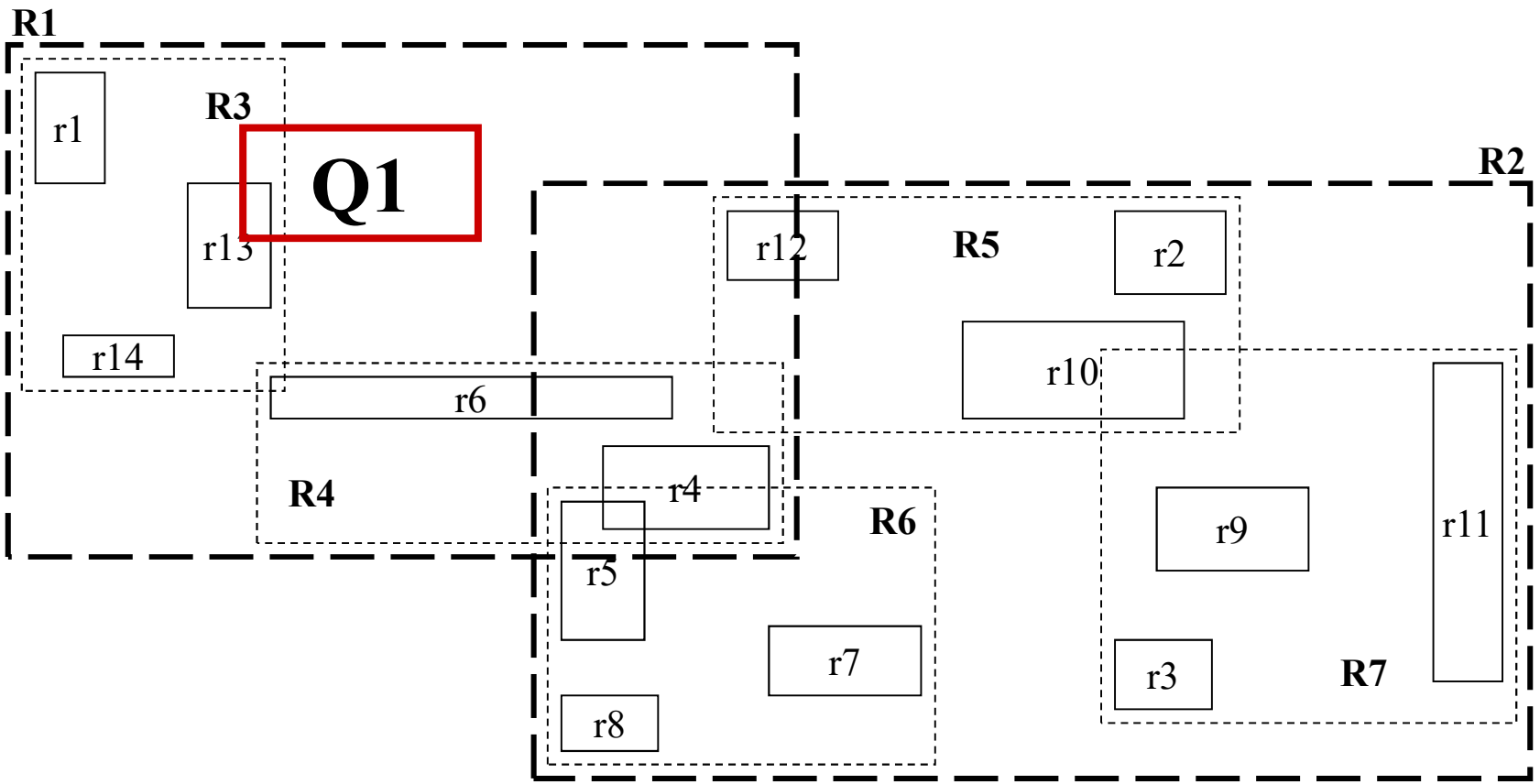

노드 a
R1
R2
노드 b
R3
R4
노드 c
R5
R6
R7
노드 d
r1
r13
r14
노드 e
r4
r6
노드 f
r2
r10
r12
노드 g
r5
r7
r8
노드 h
r3
r9
r11

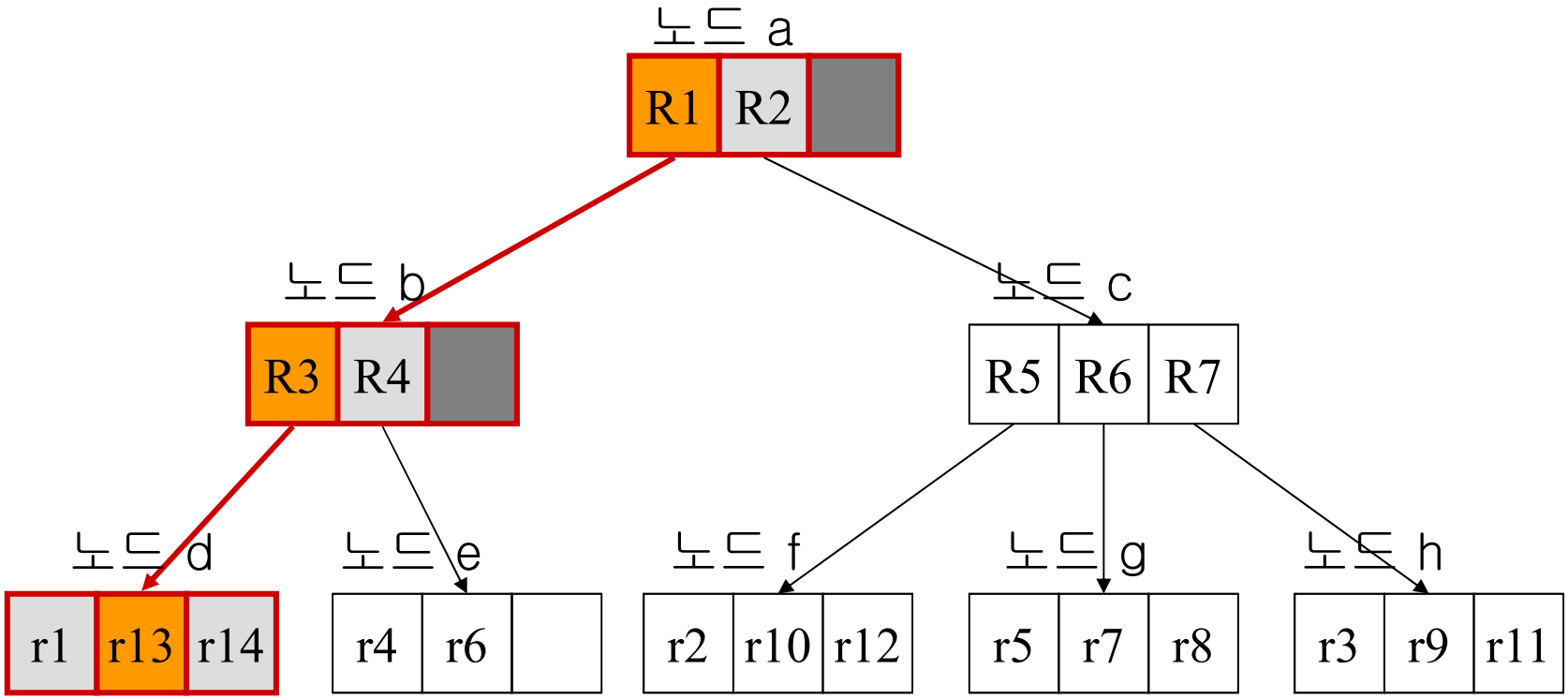
▶ R-트리의 연산

- 검색
  - 영역 검색 질의
  - k-NN 질의
- 삽입
- 삭제

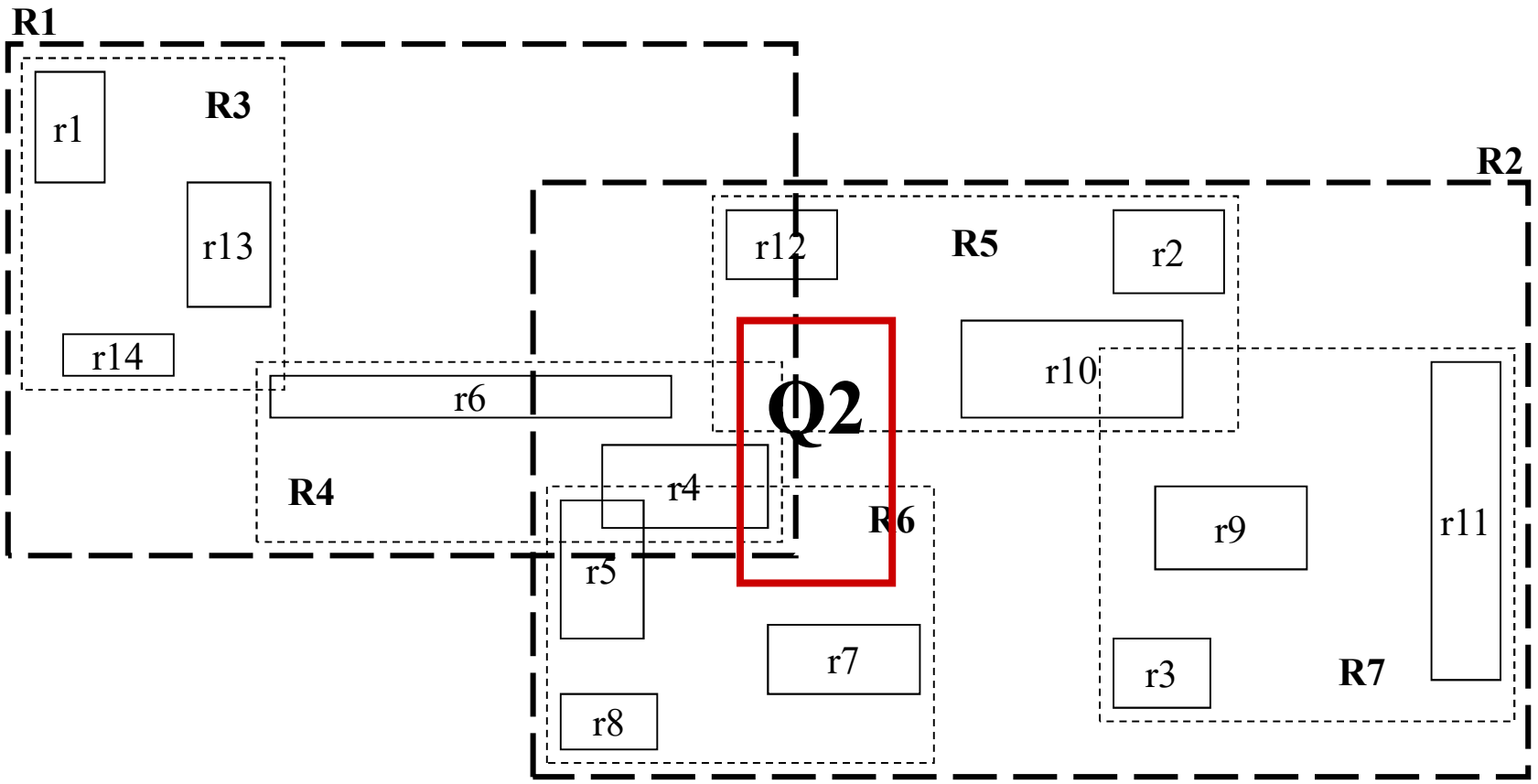
▶ R-트리의 영역 검색 질의

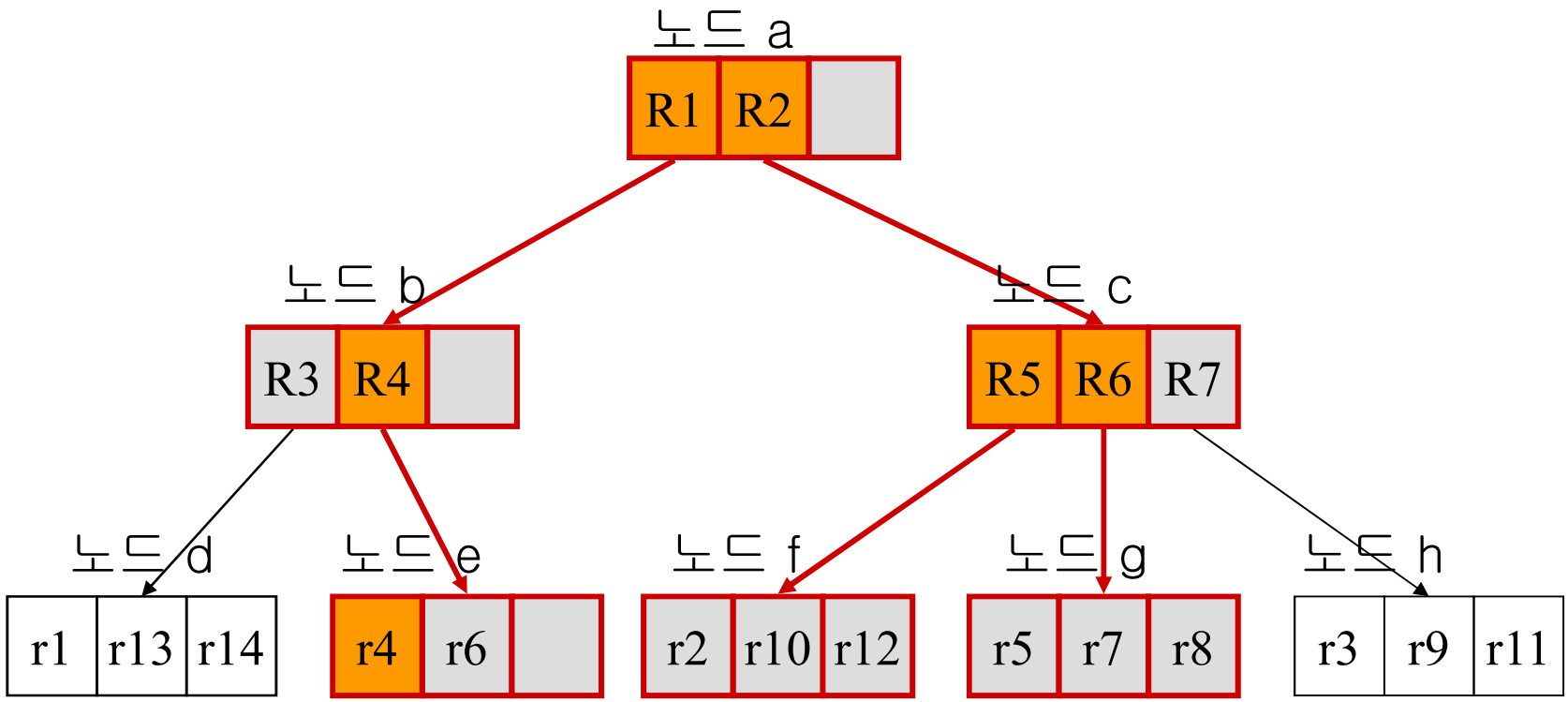
◆ 영역 검색 예 1




노드 a
R1
R2
노드 b
R3
R4
노드 c
R5
R6
R7
노드 d
r1
r13
r14
노드 e
r4
r6
노드 f
r2
r10
r12
노드 g
r5
r7
r8
노드 h
r3
r9
r11

◆ 영역 검색 예 2





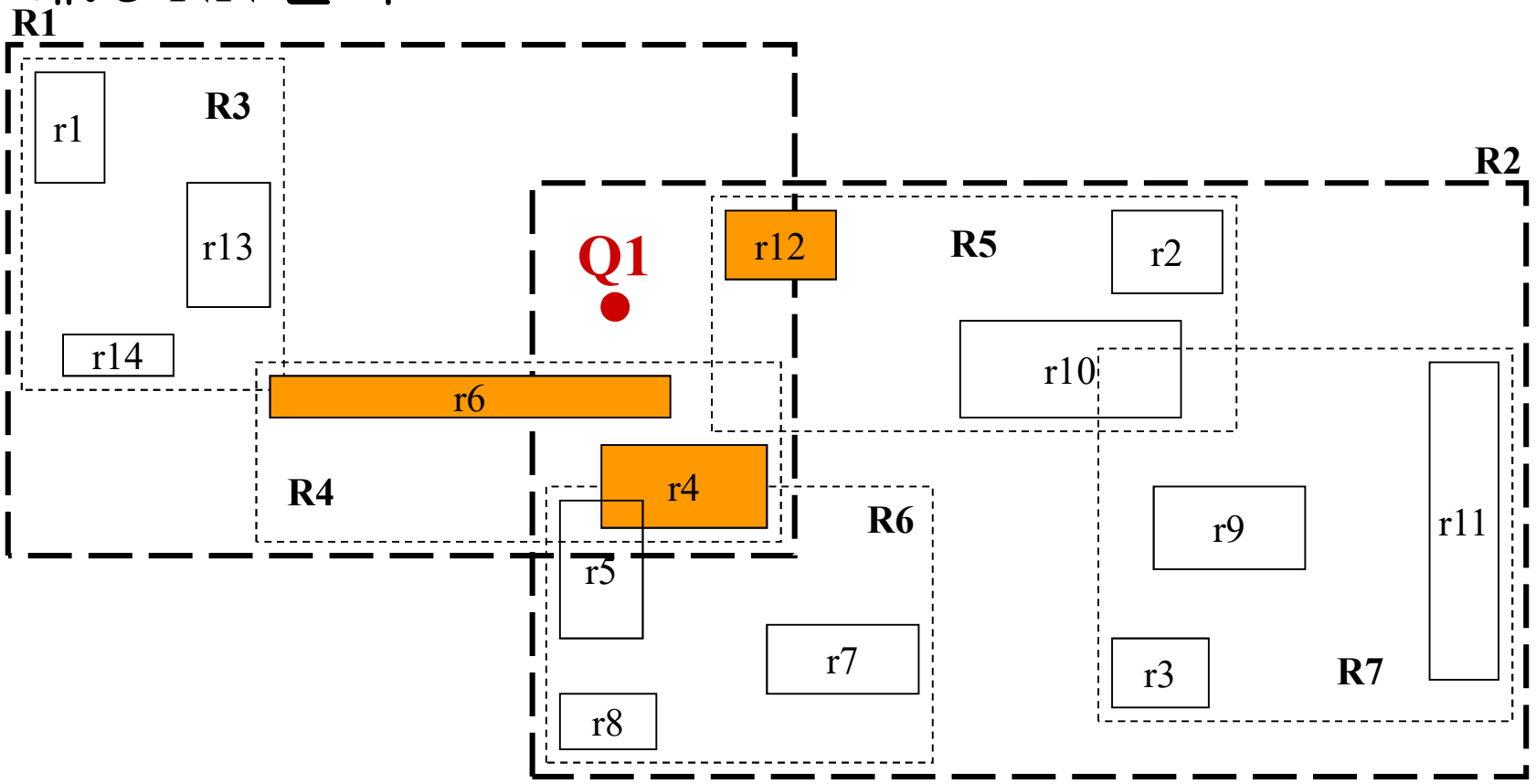
Note: MBR이 겹치는 부분이 클수록, 검색 성능이 떨어짐.
(겹치는 MBR이 최대한 작게, 노드 삽입이 이루어져야 함.)

▶ 영역 검색 질의 알고리즘

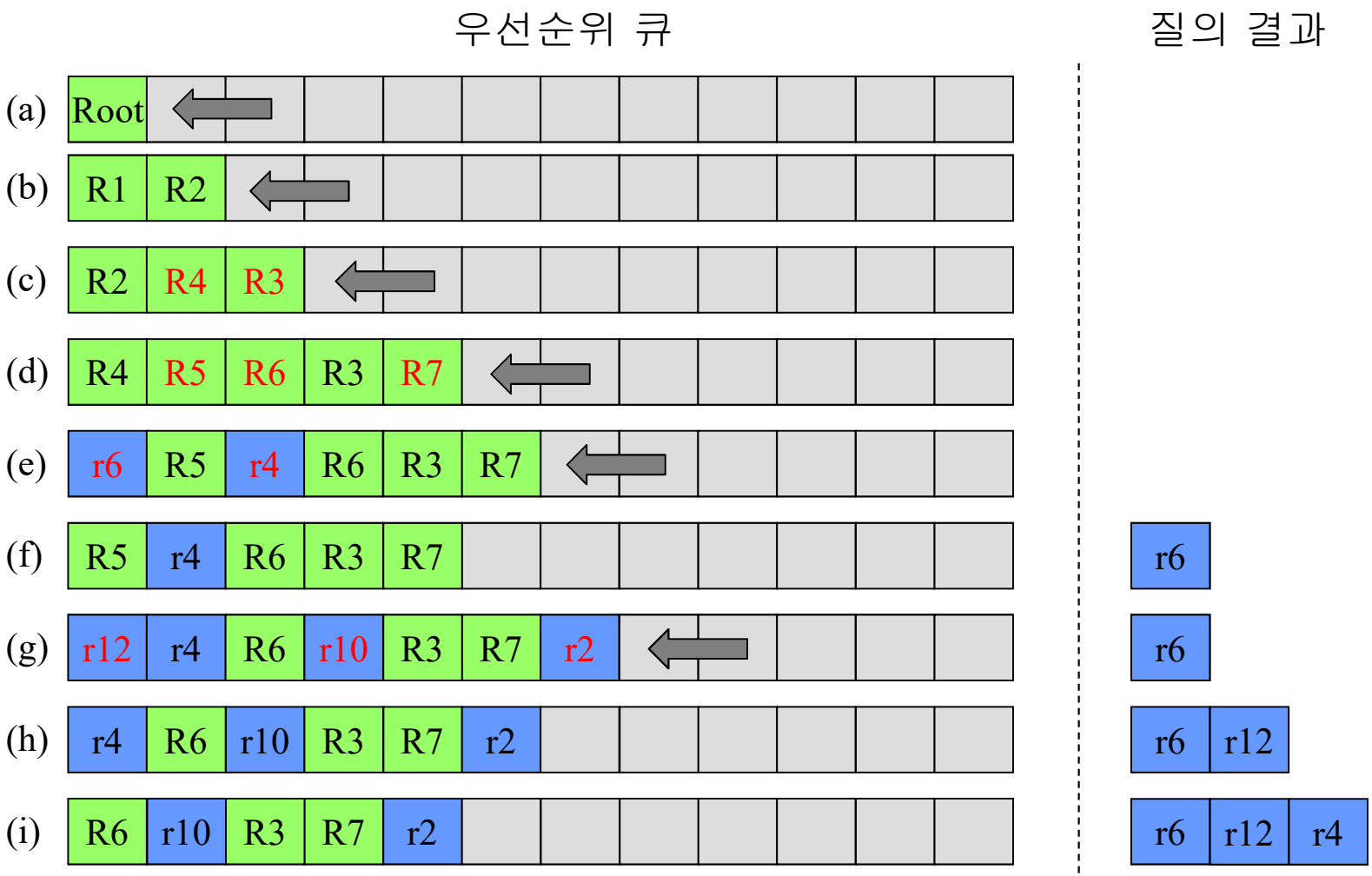
```
void Search(Node node, Region S)
{
    if (node.level > 0) // 리프노드가 아닌경우
    {
        for (i=0; i<node.NumOfEntry; i++)
        {
            if (IsOverlap(node.entry[i].MBR, S))
            {
                Search(node.entry[i].child, S); // 하위노드 검사
            }
        }
    }else {   // 리프노드인경우
        for (i=0; i<node.NumOfEntry; i++)
        {
            if (IsOverlap(node.entry[i].MBR, S))
            {
                PrintResult(node.entry[i].child);  // 결과 발견
            }
        }
    }
}
```

▶ R-트리의 k-NN (k Nearest Neighbor) 질의

- 질의점으로부터 가장 가까운 k개의 데이타를 검색
  - Q1이 MBR 내부에 있을 경우: 중심점과의 거리
  - Q1이 MBR 외부에 있을 경우: 가까운 x/y 축과의 직선 거리, x/y축과 만나지 않으면 가장 가까운 모서리와의 거리
- 예: 3-NN 질의



– 우선순위 큐를 이용한 구현



▶ k-NN 질의 알고리즘

```
kNNSearch(Node Root, Point P, int k)
{
        NumResult = 0;
        PQ.clear(); // 우선순위 큐 초기화
        distance = CaculateDistance(Root, P);
        PQ.insert(distance, Root, NODE_TYPE);  // 우선순위 큐에 루트 삽입


        while(NumResult<k, && !PQ.isEmpty()) {
                node = PQ.delete();
                if (IsNodeType(p)) {
                        // 노드일 경우 하위 엔트리를 우선순위 큐에 모두 추가
                        for (i=0; i<node.NumberOfEntry(); i++) {
                                distance = CaculateDistance(node.entry[i], P);
                                if (node.level==0) {
                                        // 리프노드의 엔트리는 데이타 객체
                                        PQ.insert(distance, node.entry[i], OBJECT_TYPE);
                                }else
                                        PQ.insert(distance, node.entry[i], NODE_TYPE);
                                }
                        }
          }else {
                        // 결과 발견
                        PrintResult(node);
                        NumResult ++;
          }
        }
}
```

▶ R-트리의 삽입

① 리프노드 선택: 데이타를 삽입할 리프 선택. **ChooseLeaf** 알고리즘 사용

② 리프노드에 데이타 저장 : 선택된 리프노드에 빈 공간이 있으면 새로운 엔트리 **E**를 저장. 빈 공간이 없는 경우 노드를 분할, **E**를 저장

③ 트리 재조정: 데이타가 삽입된 경로를 따라 새로운 데이타 삽입에 따라 변화된 **MBR**들을 조정. 새로 분할 된 노드 삽입. 중간 노드의 분할이 일어날 수 있음

④ 트리 높이 증가 : 만일 루트 노드가 분할되어야 하면 루트 노드를 분할하고 분할 된 두 노드를 새로 루트 노드를 만들어 저장. 이 경우 트리의 높이가 **1**증가한다.

◆ **ChooseLeaf 알고리즘 (리프 노드 선택)**

① N은 루트노드 T에서 시작

② N이 리프노드이면 N을 반환

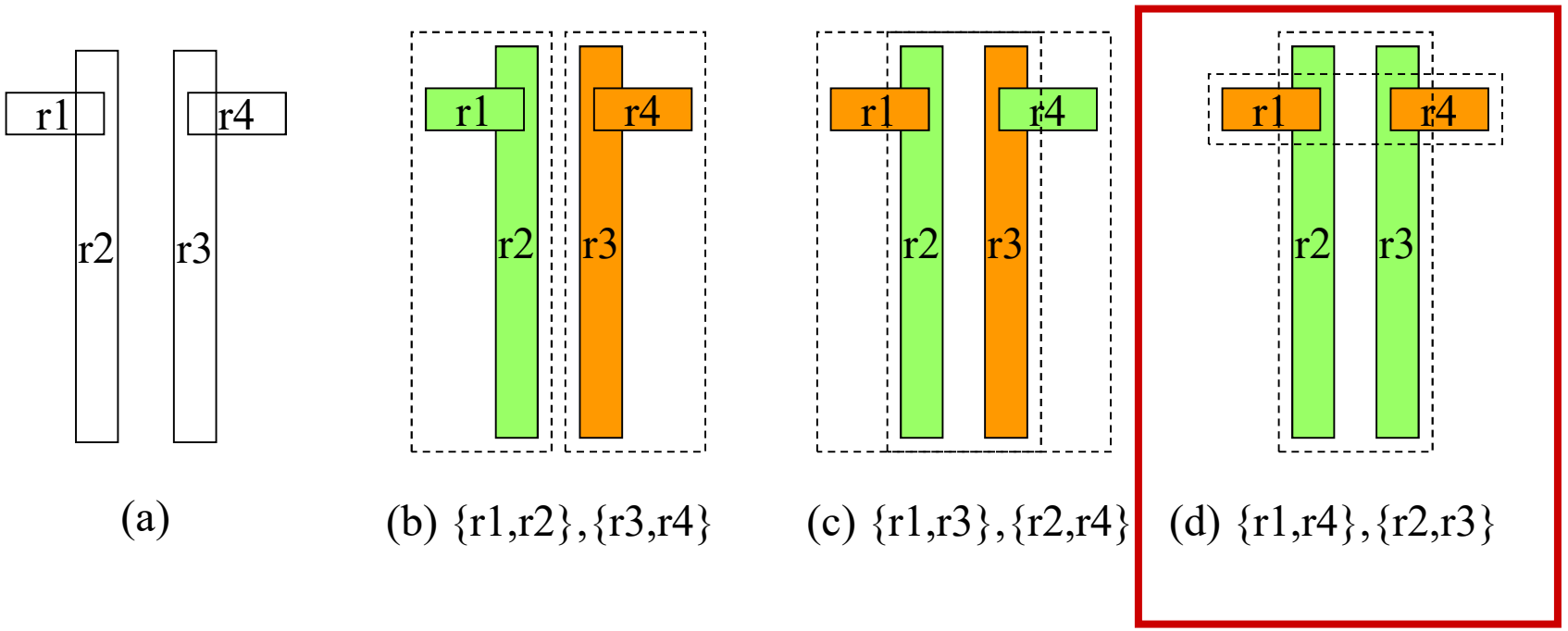
③ N이 리프노드가 아니면 N의 엔트리 가운데 E가 삽입되었을 경우 MBR의 크기가 가장 작게 증가하는 엔트리 F를 선택, 만일 증가하는 크기가 같으면 MBR의 크기가 더 작은 엔트리를 선택 (면적 최소)
→ 겹치는 MBR의 크기가 작을 확률이 최대인 방법.

④ 선택된 엔트리F가 가리키는 노드를 N으로 하여 알고리즘 ②부터 반복 수행

◆ 노드의 분할 방법

– 분할 된 두 노드 MBR의 합을 최소가 되도록 분할을 선택
→ 상위노드 MBR의 크기를 최소화함으로써, 검색시 불필요한 노드의 탐색을 줄일 수 있음

– (a)를 2개의 노드에 2개씩 분할하는 3가지 방법 예



◆ R-트리 삽입의 예 (r15를 삽입)

① 리프노드 선택 :

- MBR R7에 추가할 때가 MBR의 크기가 가장 작게 커지므로, 노드 h가 새로운 데이타를 삽입할 리프노드로 선택된다.

② 리프노드 h에 데이타 저장 :

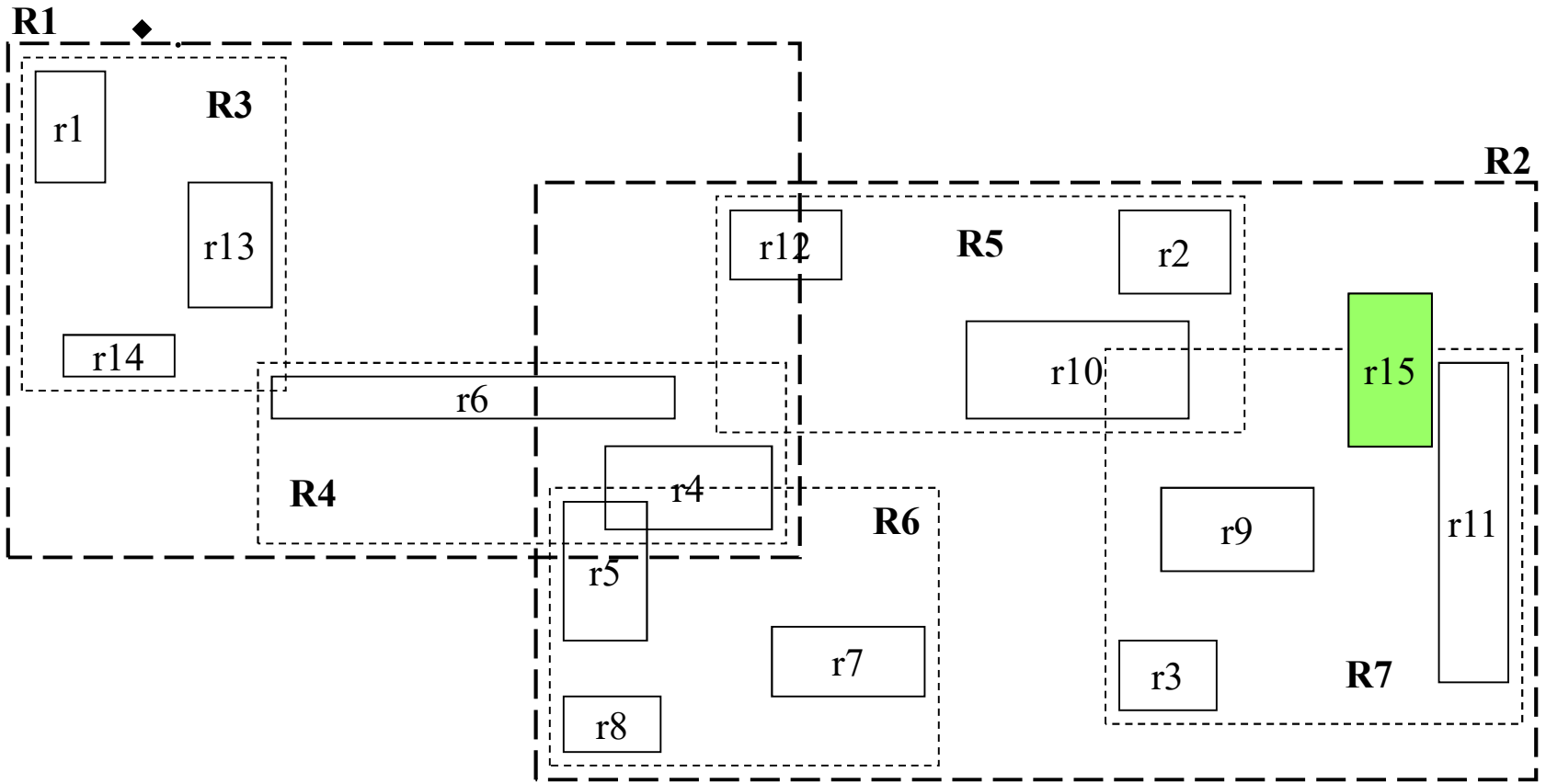
- 노드 h를 MBR R8을 포함하는 노드 h와 MBR R9를 포함하는 노드 i로 분할된다. 노드 i에 데이타를 저장

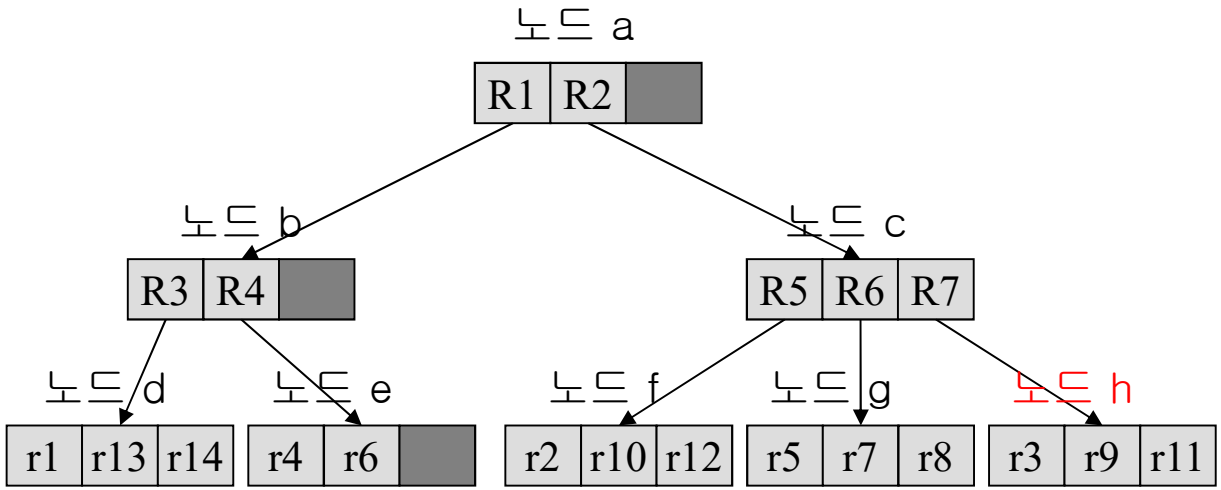
③ 트리 재조정 :

- 부모 노드에 저장된 h노드의 MBR R7을 새로운 MBR R8로 변경하고 새로 삽입된 노드 i에 관한 엔트리를 추가.
- 노드 c역시 빈 공간이 없으므로 다시 분할, 그 결과 R5와 R6을 가진 노드 c와 R8과 R9를 가진 노드 j로 분할됨.
- 다시 트리의 재조정을 하면 마지막으로 분할된 노드 c의 부모인 루트 노드 a에 노드 c의 MBR을 R10으로 변경하고, 빈 공간에 노드 j의 MBR인 R11을 저장하면 모든 트리의 조정이 끝난다.

– 삽입 전: r15를 새로 삽입하는 경우

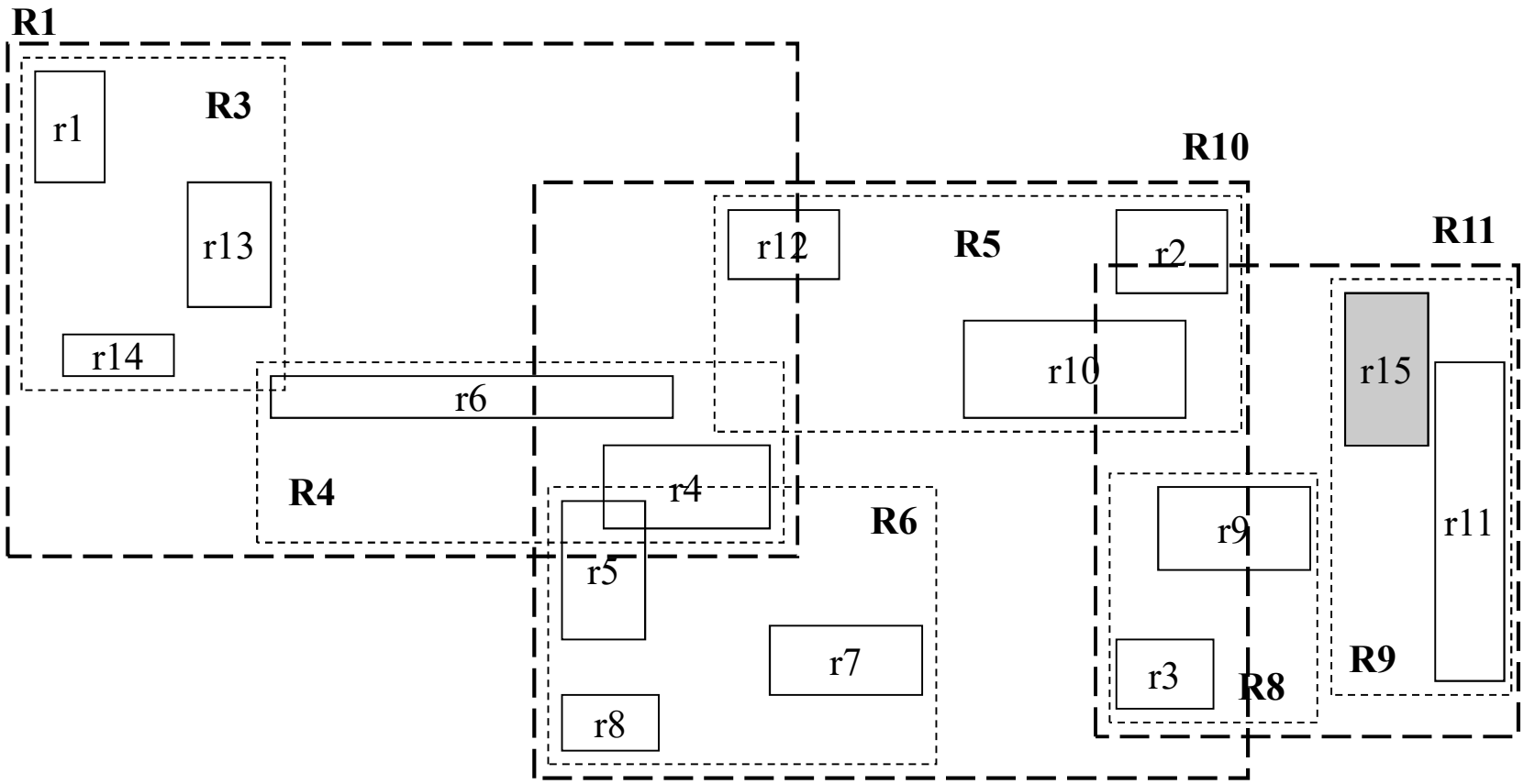
◆ ChooseLeaf 알고리즘: R2의 부속영역 중에, R7에 삽입할 때 삽입된 MBR의 크기 증가가 최소로 됨



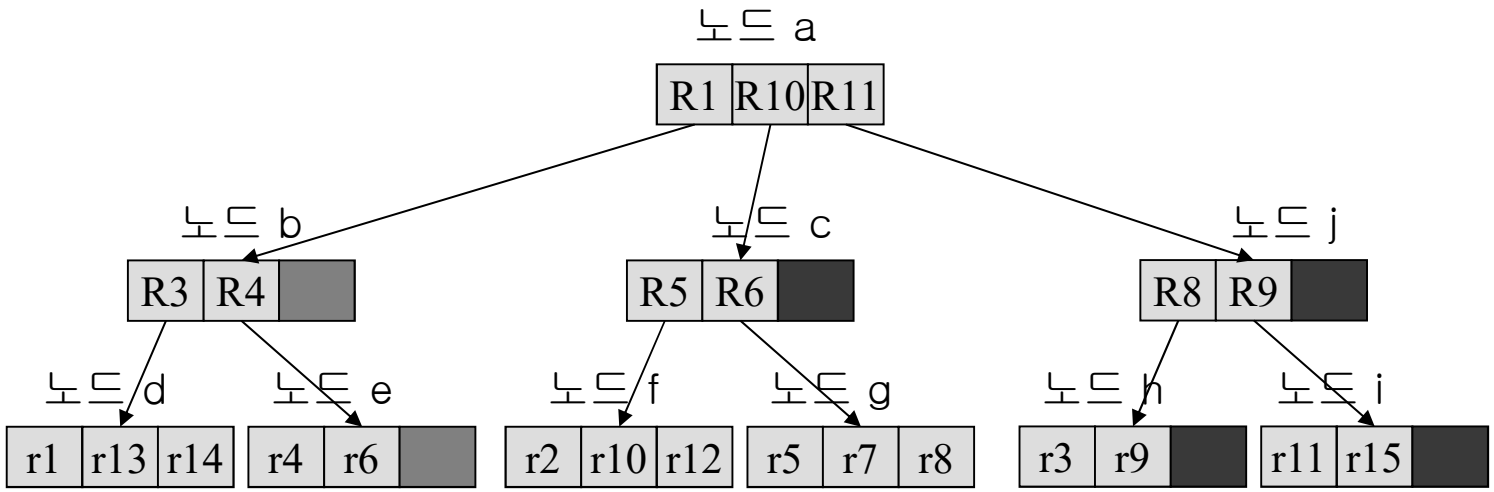

노드 a
R1
R2
노드 b
노드 c
R3
R4
R5
R6
R7
노드 d
노드 e
노드 f
노드 g
노드 h
r1
r13
r14
r4
r6
r2
r10
r12
r5
r7
r8
r3
r9
r11

– 삽입 후

- R7 노드 분할: r15 삽입으로 인해 오버플로우된 R7 (새로운 R11)을 R8와 R9으로 묶을 때, 분할된 두 MBR의 합이 최소가 됨.



– 삽입 후 (R-트리)


노드 a
R1 R10 R11
노드 b
R3 R4
노드 c
R5 R6
노드 j
R8 R9
노드 d
r1 r13 r14
노드 e
r4 r6
노드 f
r2 r10 r12
노드 g
r5 r7 r8
노드 h
r3 r9
노드 i
r11 r15

▶ R-트리의 삭제

① 리프노드 탐색:삭제될 데이타를 가지는 엔트리를 가지는 리프노드를 찾는다.

② 해당 엔트리 삭제: 리프노드에서 해당 엔트리를 삭제한다.

③ 언더플로 처리: 리프노드의 엔트리 개수가 **m**보다 작으면 언더플로가 발생하게 된다.

- 언더플로의 발생: 해당 노드 삭제 후, 노드의 모든 엔트리를 트리에 재삽입
- 언더플로의 전파: m개 이하의 엔트리를 가지는 중간 노드 역시 노드를 삭제하고, 트리에 재삽입한다. 중간 노드를 재삽입할 경우, 이 노드는 원래의 레벨에 삽입되어야 한다.
- MBR이 변하게 될 경우 트리의 루트까지 경로를 따라 MBR을 조정한다.
- 트리가 모두 조정된 후에도 트리의 루트가 단지 하나의 자식만을 가질 경우: 루트의 자식 노드를 새로운 루트 노드로 만들고, 기존의 루트 노드를 삭제한다. 이 경우 트리의 높이가 1 낮아진다.

▶ R-트리의 분석

- **d 차원의 데이타를 처리하기 위한 B 트리의 확장으로서 중간 노드와 리프 노드로 구성된 높이 균형 트리**
  - 리프 노드 : 공간 데이타에 대한 인덱스 엔트리 저장
  - 중간 노드 : 하위 노드의 각 엔트리의 사각형을 포함하는 사각형 엔트리들로 구성

- **탐색 : 포함과 겹칩 관계가 중요**
  - 포함과 겹침 관계가 최소일 때 가장 효율적임
  - 최소 포함은 죽은 공간(dead space) 즉, 빈 공간의 양을 감소시킴
  - 겹침 : 높이 h, 레벨 h-l에서 k개의 노드가 겹치면, 최악은 k 노드 접근

- **N개의 인덱스를 갖는 R-트리의 높이는 최대 $\lceil \log_m N \rceil - 1$ (각 노드의 분기율이 적어도 m)**

- **루트를 제외한 모든 노드에서 최악의 공간 활용도 (space utilization)는 m/M**
  - 트리의 높이를 감소시키면 공간 활용도는 증가

❖ R-트리의 변형

- **R-**트리의 성능 향상이나 다양한 응용 지원을 위한 여러가지 **R-**트리의 변형 트리들이 있음

- 대표적 **R-**트리 변형 트리
  - R^+-트리
    - R-트리에서 노드간의 겹치는 영역을 없앰
    - k-d-B 트리의 특징이 접목
  - R^*-트리
    - R-트리의 삽입, 삭제 알고리즘 개선

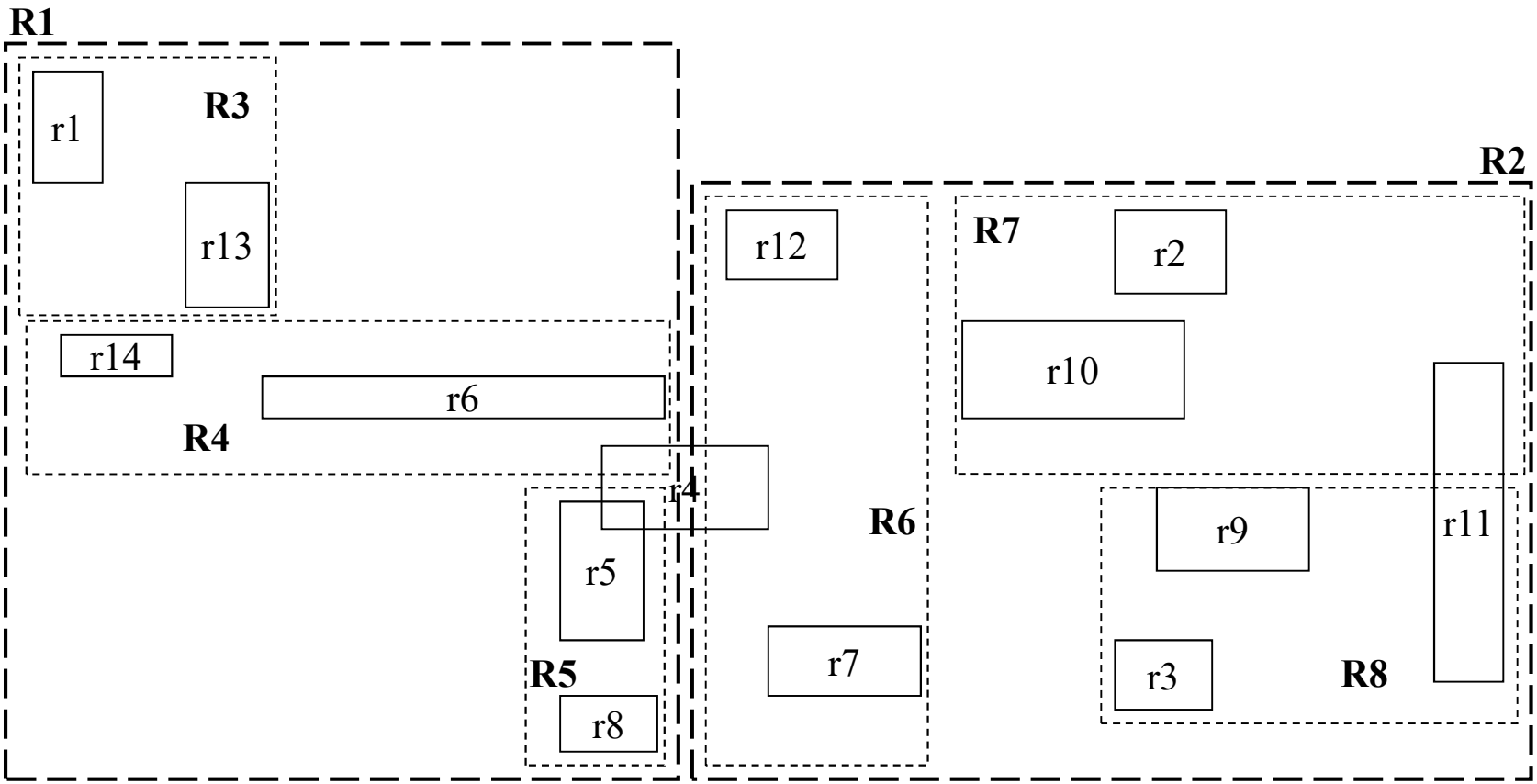
❖ R^+-트리

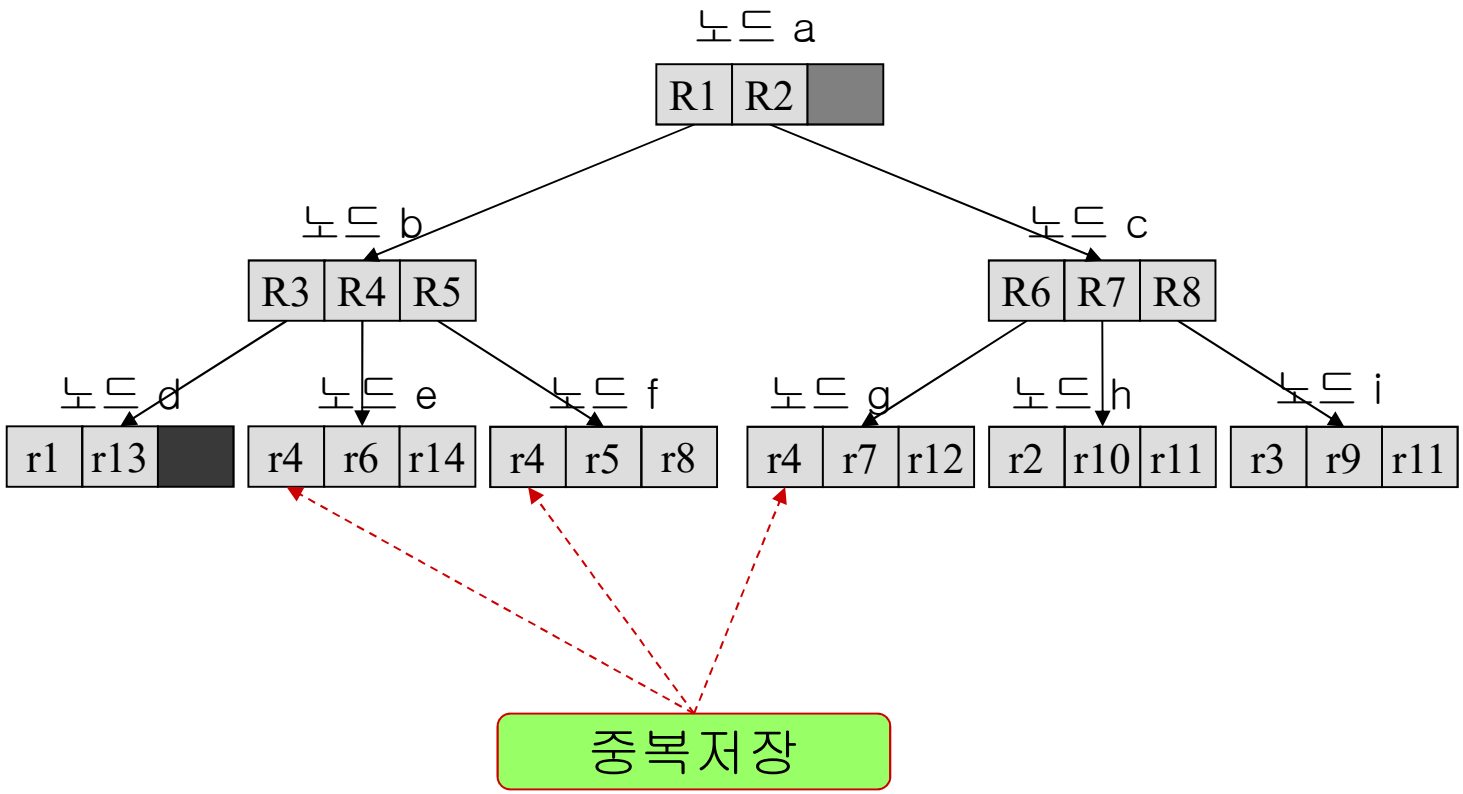
- 겹치는 데이타는 여러노드에 중복 저장
  → 중간 노드간의 겹침을 없앰
  → 검색 시 불필요한 노드 탐색을 줄임

- **R-트리와의 차이점**
  ① R^+-트리의 노드는 적어도 1/2이상의 엔트리가 차있다는 보장을 해주지 않는다.
  ② R^+-트리의 중간 노드의 엔트리들의 MBR은 겹치지 않는다.
  ③ R^+-트리의 데이타 객체는 하나 이상의 리프 노드에 저장될 수 있다.

▶ R^+-트리의

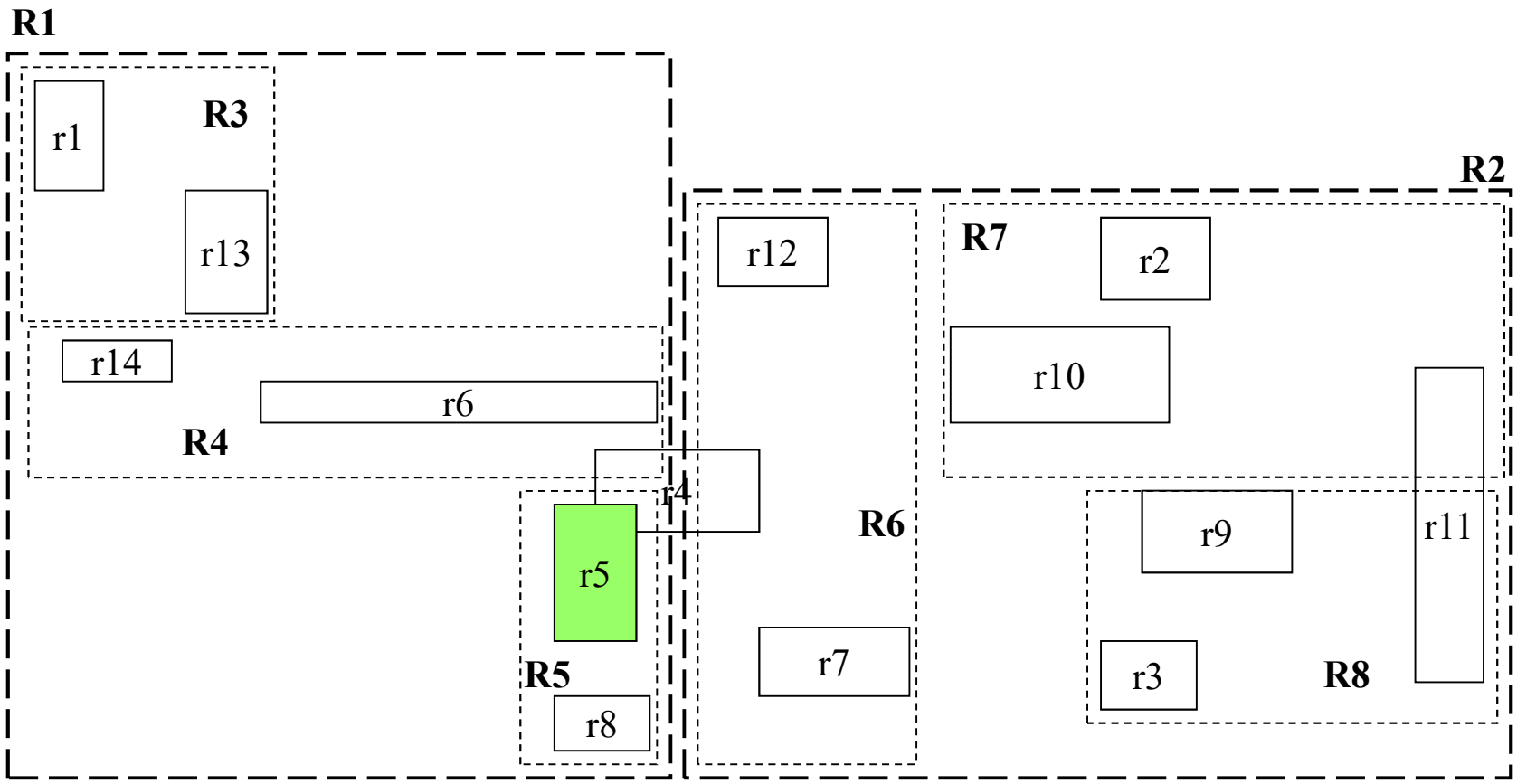
◆ R+-트리의 예

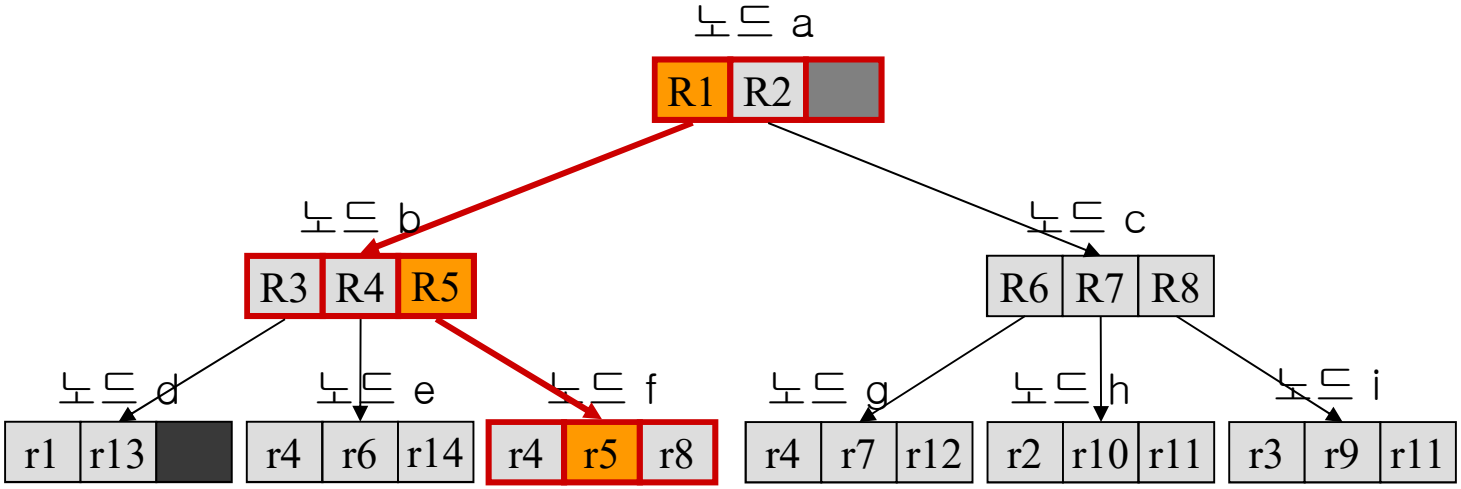



노드 a
R1
R2
노드 b
R3
R4
R5
노드 c
R6
R7
R8
노드 d
r1
r13
노드 e
r4
r6
r14
노드 f
r4
r5
r8
노드 g
r4
r7
r12
노드 h
r2
r10
r11
노드 i
r3
r9
r11
중복저장

▶ R^+-트리의 검색

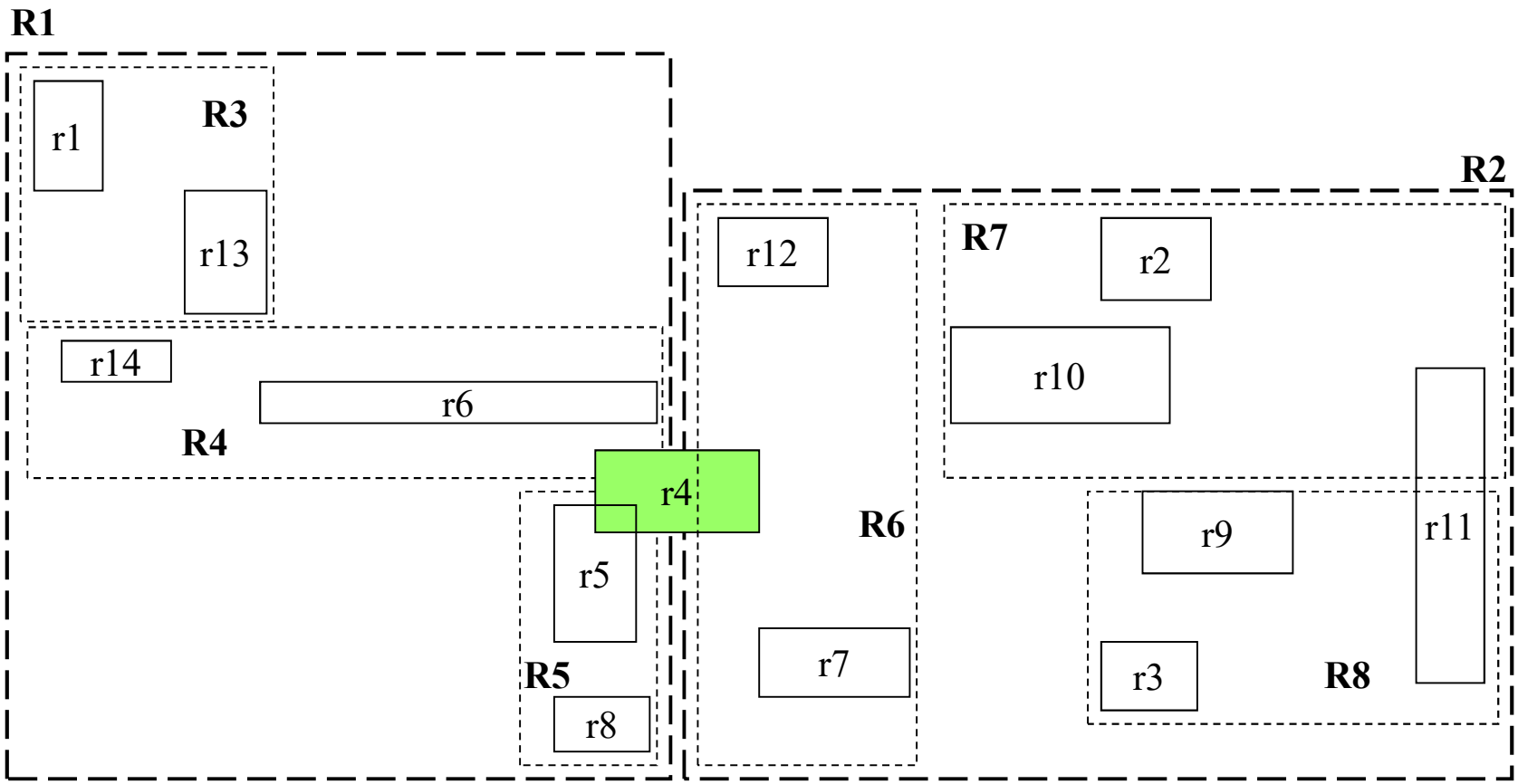
◆ R+-트리의 검색 예 1

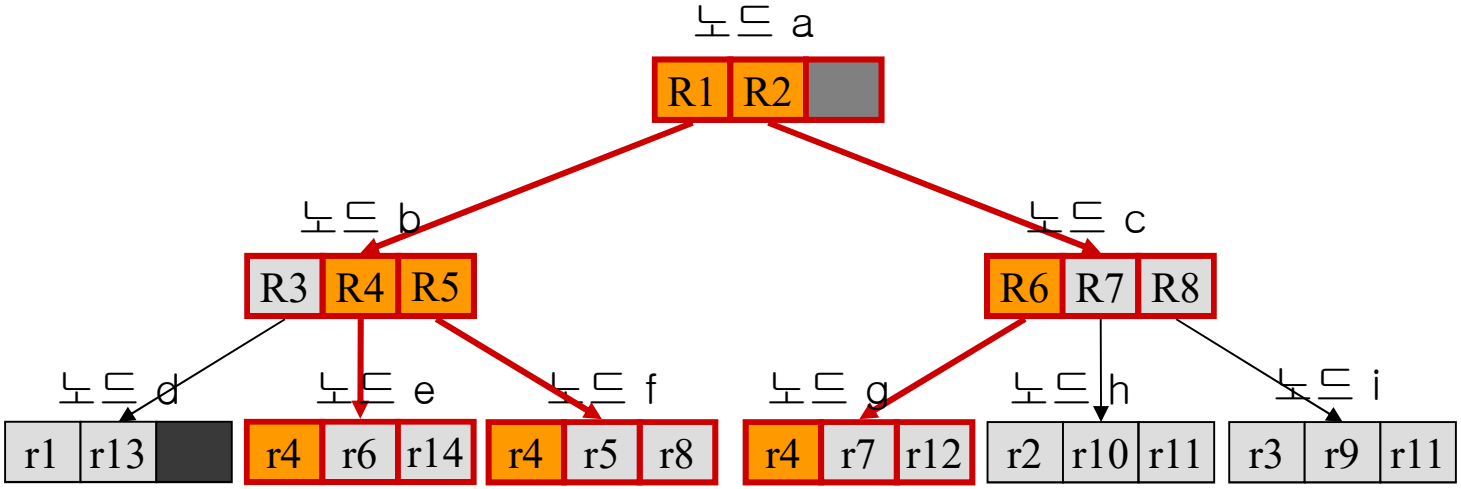



노드 a
R1
R2
노드 b
R3
R4
R5
노드 c
R6
R7
R8
노드 d
노드 e
노드 f
노드 g
노드 h
노드 i
r1
r13
r4
r6
r14
r4
r5
r8
r4
r7
r12
r2
r10
r11
r3
r9
r11

3개의 노드 방문

◆ **R+-트리의 검색 예 2**




노드 a
R1
R2
노드 b
R3
R4
R5
노드 c
R6
R7
R8
노드 d
노드 e
노드 f
노드 g
노드 h
노드 i
r1
r13
r4
r6
r14
r4
r5
r8
r4
r7
r12
r2
r10
r11
r3
r9
r11

6개의 노드 방문

▶ R^+-트리의 삽입 및 삭제

- 일반적으로 R-트리보다 복잡
  - 하나의 데이타가 여러 리프 노드에 중복될 수 있기 때문
- 데이타를 삽입
  - 여러 패스를 따라 데이타를 삽입
  - 노드를 분할하는 경우, 겹치는 영역없이 노드를 분할할 수 있는 축을 찾아야함
  - 겹치는 영역이 없이 분할하는 것이 불가능할 경우 하부 엔트리를 중복하여 저장
- 데이타 삭제
  - 해당 객체가 하나 이상의 리프 노드에 저장될 수 있으므로 해당하는 모든 리프노드에서 해당 객체를 삭제하여야 한다.
  - 삭제가 많이 발생한 뒤에는 공간 활용도가 매우 나빠지게 되는데, 이와 비슷한 현상은 k-d-B 트리에서도 발생한다.
  - 이런 경우에는 성능을 고려하여 주기적으로 서브트리들을 재구성해야만 한다.

▶ R^+-트리의 장단점

◆ 장점
- MBR이 겹치는 문제점을 해결
- 불필요한 노드 탐색을 줄임

◆ 단점
- R^+-트리의 경우 노드의 분할이 하위 노드로 파급될 수 있다. (상위 노드의 분할이 다시 하위 노드의 분할을 야기할 수 있음) → 노드의 공간 활용도가 나쁨
- 리프노드에 중복되어 저장되는 데이타의 수는 데이타의 분포나 데이타 개수의 영향을 받으므로 데이타의 분포나 데이타의 개수에 따라 R^+-트리의 성능이 저하될 수 있다.

❖ R*-트리

- **R-**트리와 기본적인 구조 및 연산이 동일
- 삽입, 삭제 시 몇가지 알고리즘 개선

- **R-**트리 질의 처리시 고려해야 할 사항
  - ① 면적(area) 최소화
  - ② 겹침영역(overlap) 최소화
  - ③ 둘레 길이(margin) 최소화: 정사각형에 가까운 모양의 MBR이 좋음
  - ④ 저장장소 이용률(storage utilization) 최소화 : 트리의 노드 수를 줄일 수 있고, 트리의 높이를 낮은 상태로 유지할 수 있음

▶ R*-트리의 변경 사항

- **ChooseLeaf 알고리즘 변경**
  - 중간 노드 선택 : 면적최소화
  - 리프 노드 선택 : 겹침 영역 최소화
- **노드 분할 알고리즘 변경: 2단계 heuristic 사용**
  - 분할 축 선택 : 둘레길이 최소화
  - 분할 선택 : 겹침 영역 최소화 (같을 경우 면적 최소화)
- **강제적 재삽입 전략**
  - 오버플로 발생시 노드의 M+1개의 데이타 중 트리의 중심에서 멀리 떨어진 p개(보통 30%)를 강제로 트리에 재삽입
  - 이점
    ① 겹침 영역이 줄어듬
    ② 분할 비용을 줄일 수 있음
    ③ MBR 모양이 정사각형에 가깝게 변함

▶ R*-트리의 장점

- **R-**트리의 기본 구조나 연산을 유지하고 있으므로 구현이 비교적 간단
- 삽입, 삭제 알고리즘 개선으로 질의 성능이 우수