Data Structure and Algorithm Design

ME/MSc (Computer) – Pokhara University

Prepared by:

Assoc. Prof. Madan Kadariya (NCIT)

Contact: madan.kadariya@ncit.edu.np

Chapter 2:
Advanced Data Structures and Algorithms (11 hrs)

Outline

1. Advanced Trees
   - B and B+ Trees
2. Advanced Graph
   - Planar Graphs and its applications in Geographic and Circuit Layout Design
3. Network Flow Algorithms
   - Ford-Fulkerson Algorithm
   - Edmonds-Karp Algorithm
4. Advanced Shortest Path Algorithms
   - Bellman-Ford Algorithm
   - Johnson's algorithm
5. Graphical Data Structures
   - Quadtrees, KD Trees, R-Trees
6. Computational Geometry
   - Convex Hull and Voronoi Diagrams
7. Case Studies in Red-Black Tree and Its Applications, Applications of Directed Acyclic Graph, Applications of Minimum Spanning Tree in Network Design

Graphical/ Spatial Data Structures

1. Quadtrees
2. KD Trees
3. R-Trees

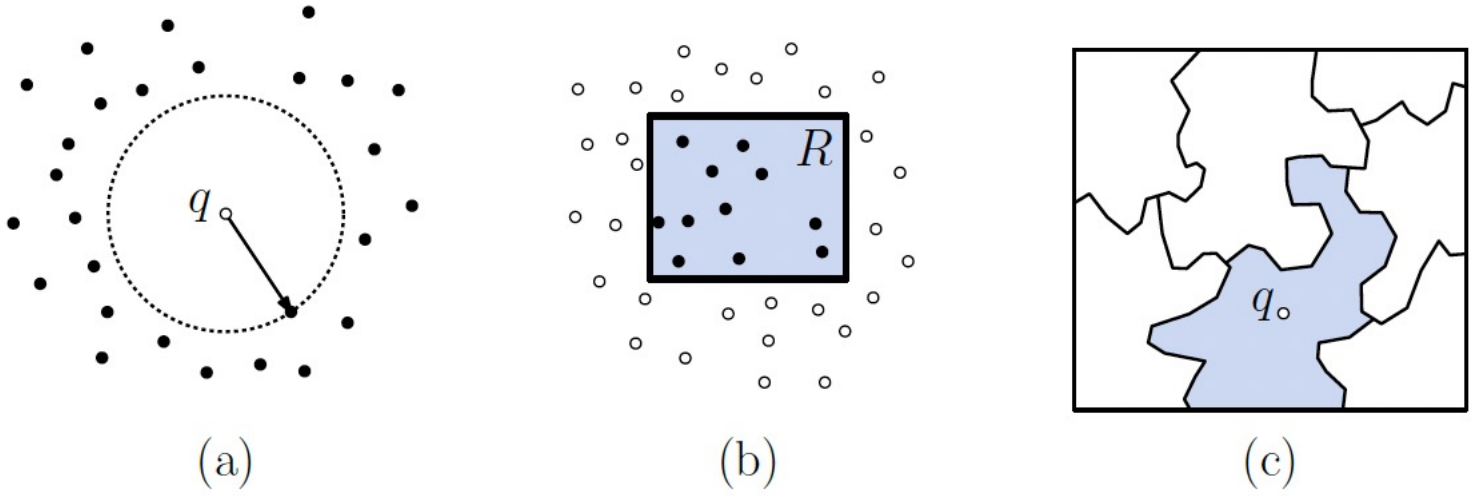
Graphical/Spatial Data Structure

- **Spatial data structures**, are specialized data structures used for representing, organizing, and querying spatial or geometric data.
- Geometric data structures are fundamental to the efficient processing of data sets arising from myriad applications(technology, science, general life, business, engineering etc) including spatial databases, automated cartography (maps) and navigation, computer graphics, robotics and motion planning, solid modeling and industrial engineering, particle and fluid dynamics, molecular dynamics and drug design in computational biology, machine learning, image processing and pattern recognition, computer vision.
- To store a large datasets consisting of geometric objects (e.g.,points, lines and line segments, simple shapes (such as balls, rectangles, triangles), and complex shapes such as surface meshes) in order to answer queries on these data sets efficiently.

Graphical/Spatial Data Structure

Types of Query

- **Nearest-Neighbor Searching**: Store a set of points so that given a query point q, it is possible to find the closest point of the set (or generally the closest k objects) to the query point (see Fig. 1(a)).
- **Range Searching**: Store a set of points so that given a query region R (e.g., a rectangle or circle), it is possible to report (or count) all the points of the set that lie inside this region (see Fig. 1(b)).
- **Point location**: Store the subdivision of space into disjoint regions (e.g., the subdivision of the globe into countries) so that given a query point q, it is possible determine the region of the subdivision containing this point efficiently (see Fig. 1(c)).

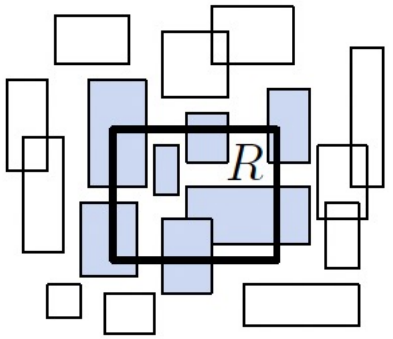


Graphical/Spatial Data Structure

Types of Query

- **Intersection Searching**: Store a collection of geometric objects (e.g., rectangles), so that given a query consisting of an object R of this same type, it is possible to report (or count) all of the objects of the set that intersect the query object (see Fig. 1(d)).
- **Ray Shooting**: Store a collection of object so that given any query ray, it is possible to determine whether the ray hits any object of the set, and if so which object does it hit first.

- If the size n of the set is huge, consisting for example of millions of objects, and the objective is to answer the query in time that is significantly smaller than n, ideally O(log n).
- It is not always possible to achieve efficient query times with storage that grows linearly with n. In such instances, we would like the storage to slowly, for example, O(n log n).



(d)

Graphical/Spatial Data Structure

- Spatial Data Structure are are particularly useful in applications like computer graphics, geographic information systems (GIS), computer-aided design (CAD), gaming, simulations, and collision detection.

1. Quadtrees

- Primarily used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions.
- **Applications:** GIS for region-based data, image compression, and efficient range queries in 2D space.

2. Octrees

- An extension of quadtrees to three-dimensional space, recursively subdividing space into eight octants.
- **Applications:** 3D computer graphics, spatial indexing in 3D environments, and volumetric data representation.

Graphical/Spatial Data Structure

3. KD-Trees (k-dimensional Trees)

- A space-partitioning data structure for organizing points in a k-dimensional space.
- **Applications:** Nearest neighbor search, range queries, and dimensional indexing.

4. R-Trees

- Used to store and query rectangles or multidimensional objects in databases.
- **Applications:** Efficient indexing and querying of spatial data, such as polygons or rectangles in GIS and spatial databases.

5. Voronoi Diagrams

- Partition a plane based on the distance to a specified set of points, resulting in regions known as Voronoi cells.
- **Applications:** Spatial analysis, nearest neighbor search, and cellular networks.

6. Etc.

Spatial Data Structure

- Search trees such as **BSTs**, **AVL trees**, **splay trees**, **2-3 Trees**, **B-trees**, and **tries** are designed for searching on a one-dimensional key.
- A typical example is an integer key, whose one-dimensional range can be visualized as a number line.
- These various tree structures can be viewed as dividing this one-dimensional number line into pieces.
- Some databases require support for multiple keys. In other words, records can be searched for using any one of several key fields, such as name or ID number.
- Typically, each such key has its own one-dimensional index, and any given search query searches one of these independent indices as appropriate.

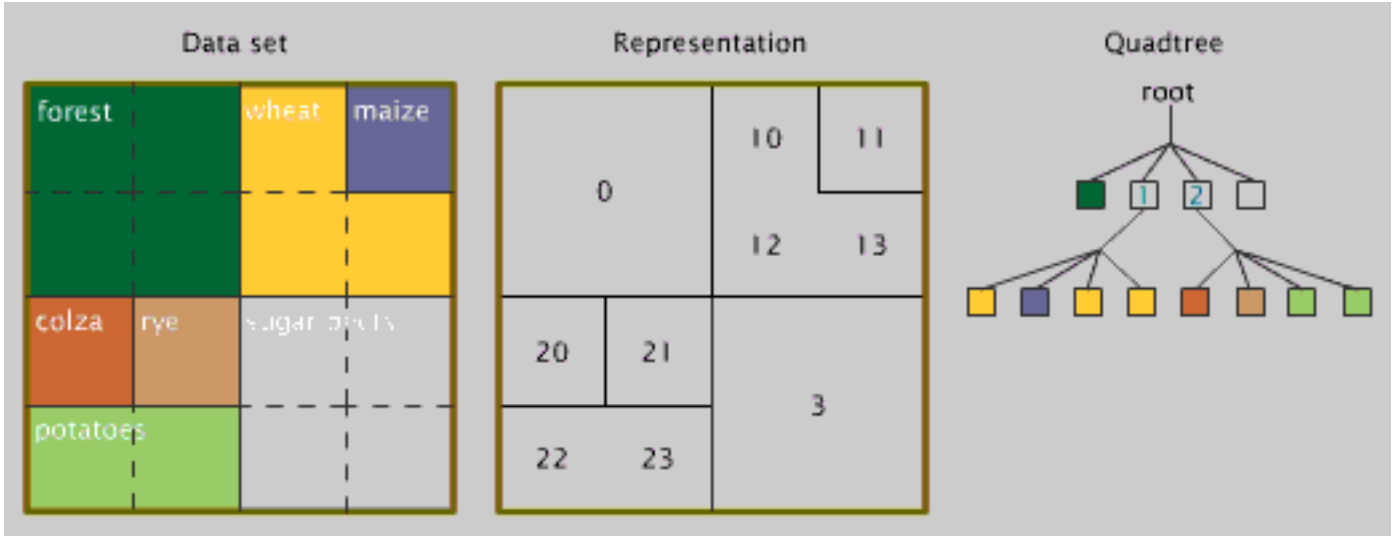
Spatial Data Structure

Multdimensional Keys

- Suppose that we have a database of city records, where each city has a name and an *xy* coordinate. A BST or splay tree provides good performance for searches on city name, which is a one-dimensional key.
- Search on one of the two coordinates is not a natural way to view search in a two-dimensional space.
- Another option is to combine the xy coordinates into a single key, say by concatenating the two coordinates, and index cities by the resulting key in a BST. That would allow search by coordinate, but would not allow for an efficient two-dimensional **range query** such as searching for all cities within a given distance of a specified point.
- The problem is that the BST only works well for one-dimensional keys, while a coordinate is a two-dimensional key where neither dimension is more important than the other.
- To implement spatial applications efficiently requires the use of a **spatial data structure**.
- Spatial data structures store data objects organized by position and are an important class of data structures used in geographic information systems, computer graphics, robotics, and many other fields.
- A number of spatial data structures are used for storing point data in two or more dimensions.
- The **kd tree** is a natural extension of the BST to multiple dimensions.

Quadtree

- A **quadtree** is a tree data structure in which each internal node has exactly four children.
- In Quadtrees two-dimensional space are partitioned by recursively subdividing it into four quadrants or regions.
- The data associated with a leaf cell varies by application, but the leaf cell represents a "unit of interesting spatial information".



Quadtree

Types of Quadtree

1. Point Quadtree

- **Purpose:** Stores individual points in 2D space.
- **Structure:** Each node represents a point, and the space is recursively divided into four quadrants around that point.
- **Applications:**
  - Efficient point location.
  - Nearest neighbor search.
- **Limitation:** Not efficient for uniformly distributed points due to unbalanced tree structures.

Quadtree

Types of Quadtree

2. Region Quadtree

- **Purpose:** Represents a 2D space divided into regions or blocks, often for images or terrains.
- **Structure:** Each node represents a square region, and it is recursively subdivided into quadrants until each leaf is homogeneous.
- **Applications:**
  - Image compression.
  - Region-based queries in GIS.
- **Example:** Used for storing binary images (black and white).

Quadtree

Types of Quadtree

3. PR Quadtree (Point-Region Quadtree)

- **Purpose:** Handles spatial point data by dividing the 2D space into quadrants regardless of point positions.
- **Structure:**
  - Each node corresponds to a specific region of space.
  - Subdivides space into quadrants until each leaf contains at most one point.
- **Applications:**
  - Spatial indexing.
  - Geographic information systems (GIS).
- **Key Feature:** Fixed spatial decomposition regardless of data distribution.

Quadtree

Types of Quadtree

4. MX Quadtree (Matrix Quadtree)

- **Purpose:** Specialized for fixed-grid data, such as pixels in an image.
- **Structure:**
  - Similar to region quadtrees but operates on a matrix-like uniform grid.
  - Subdivision occurs until each region contains homogeneous data.
- **Applications:**
  - Image processing.
  - Terrain modeling.

Quadtree

Types of Quadtree

5. Edge Quadtree

- **Purpose:** Designed for representing linear features, such as lines or curves, within a 2D space.
- **Structure:**
  - The quadtree is used to approximate and store line segments by recursively subdividing space.
- **Applications:**
  - Vector graphics storage.
  - Pathfinding and navigation.
- **Key Feature:** Supports efficient rendering of complex edges.

Quadtree

Types of Quadtree

6. Adaptive Quadtree

- **Purpose:** Dynamically adapts to the density of data in space.
- **Structure:**
  - Regions are subdivided only where necessary (e.g., regions with dense data are further divided).
- **Applications:**
  - Simulation of physical systems.
  - Dynamic terrain or scene representation.
- **Key Feature:** Reduces unnecessary subdivisions and memory usage.

Quadtree

Types of Quadtree

7. Loose Quadtree

- **Purpose:** Allows objects to overlap boundaries by expanding node boundaries slightly.
- **Structure:**
  - Nodes have a tolerance margin that accommodates objects spanning multiple quadrants.
- **Applications:**
  - Collision detection in gaming and simulations.
- **Key Feature:** Reduces the number of unnecessary splits and simplifies spatial relationships.

Comparison of Key Quadtree Types

Type	Primary Focus	Common Applications
Point Quadtree	Storing point data	Nearest neighbor, GIS
Region Quadtree	Dividing 2D space	Image compression, GIS
PR Quadtree	Fixed spatial divisions	Spatial indexing, GIS
MX Quadtree	Matrix-based representation	Image processing
Edge Quadtree	Linear feature representation	Vector graphics, GIS
Adaptive Quadtree	Dynamic data density	Terrain modeling, physics
Loose Quadtree	Flexible boundaries	Collision detection

Point quadtree

- Let us first consider a natural way of generalizing unbalanced binary trees in the 1-dimensional case to a d-dimensional context. Suppose that we wish to store a set $P = \{P_1, P_2, \ldots P_n\}$ of n points in d-dimensional space.
- In binary trees, each point naturally splits the real line in two. In two dimensions if we run a vertical and horizontal line through the point, it naturally subdivides the plane into four quadrants about this point.
- To simplify the presentation, assume that we are working in 2-dimensional space. The resulting data structure is called a point quadtree. (In dimension three, the corresponding structure is naturally called an octree).
- As the dimension grows, it is too complicated to figure out the proper term for the number of children, and so the term quadtree is often used in arbitrary dimensions, even though the outdegree of each node is 2^d, not four.)
- Each node has four (possibly null) children, corresponding to the four quadrants defined by the 4-way subdivision and label these according to the compass directions, as NW, NE, SW, and SE.
- In terms of implementation, we can think of assigning these the values 0, 1, 2, 3, and use them as indices to a 4-element array of children pointers.

Point quadtree

Consider the insertion of the following points: (35; 40); (50; 10); (60; 75); (80; 65); (85; 15); (5; 45); (25; 35); (90; 5)

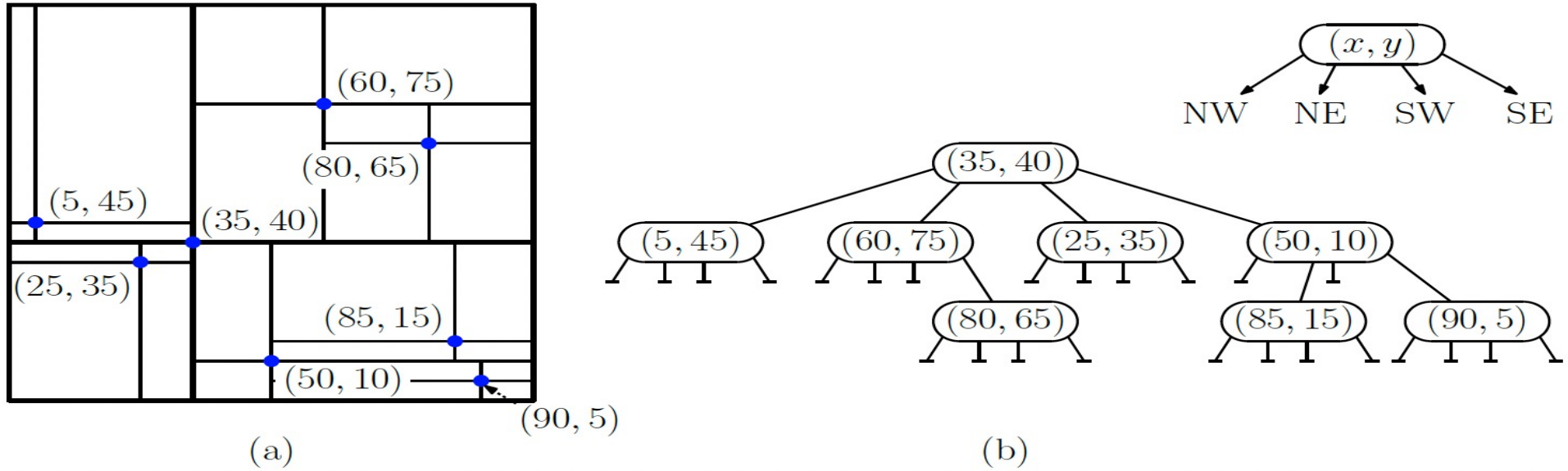


Fig: Point Quadtree (a) Subdivision of space (b) tree structure

- Each node in the tree is naturally associated with a rectangular region of space, which we call its cell.
- Note that some rectangles are special in that they extend to infinity.
- Algorithm will be discuss in **kd-trees** which is a similar algorithm for quadtrees.

Point-Region quadtree

- In the **Point-Region quadtree** (**PR quadtree**) each node either has exactly four children or is a leaf.
- The PR quadtree is a full four-way branching (4-ary) tree in shape.
- The PR quadtree represents a collection of data points in two dimensions by decomposing the region containing the data points into four equal quadrants, subquadrants, and so on, until no leaf node contains more than a single point.
- If a region contains zero or one data points, then it is represented by a PR quadtree consisting of a single leaf node.
- If the region contains more than a single data point, then the region is split into four equal quadrants.
- The corresponding PR quadtree then contains an internal node and four subtrees, each subtree representing a single quadrant of the region, which might in turn be split into subquadrants.
- Each internal node of a PR quadtree represents a single split of the two-dimensional region.
- The four quadrants of the region (or equivalently, the corresponding subtrees) are designated (in order) NW, NE, SW, and SE.
- Each quadrant containing more than a single point would in turn be recursively divided into subquadrants until each leaf of the corresponding PR quadtree contains at most one point.

Point-Region quadtree

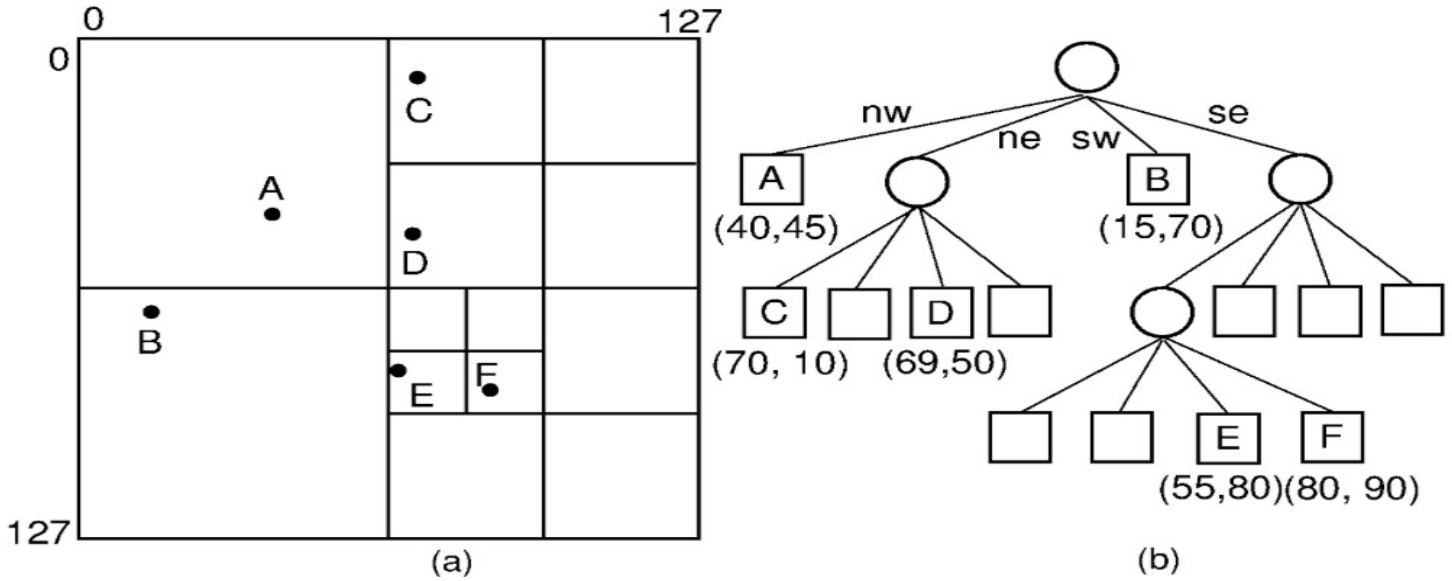


Fig: Example of a PR quadtree.(a) A map of data points. (b) The PR quadtree for the points in (a). (a) also shows the block decomposition imposed by the PR quadtree for this region.

- Consider the region of Fig (a) and the corresponding PR quadtree in Fig (b). The decomposition process demands a fixed key range. In eg,the region is assumed to be of size 128×128. Note that the internal nodes of the PR quadtree are used solely to indicate decomposition of the region; internal nodes do not store data records.

Point-Region quadtree

- Search for a record matching point Q in the PR quadtree is straightforward. Beginning at the root, we continuously branch to the quadrant that contains Q until our search reaches a leaf node. If the root is a leaf, then just check to see if the node's data record matches point Q. If the root is an internal node, proceed to the child that contains the search coordinate.
- For example, the NW quadrant of Fig(a) contains points whose x and y values each fall in the range 0 to 63. The NE quadrant contains points whose x value falls in the range 64 to 127, and whose y value falls in the range 0 to 63.
- If the root's child is a leaf node, then that child is checked to see if Q has been found.
- If the child is another internal node, the search process continues through the tree until a leaf node is found.
- If this leaf node stores a record whose position matches Q then the query is successful; otherwise Q is not in the tree.

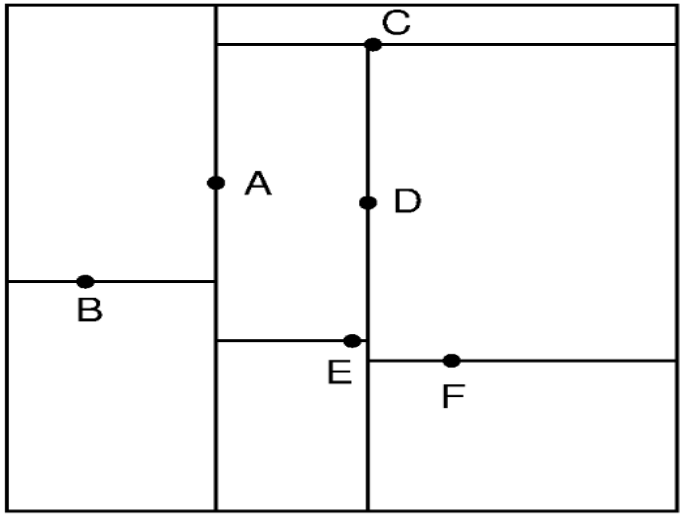
KD Trees

KD Trees

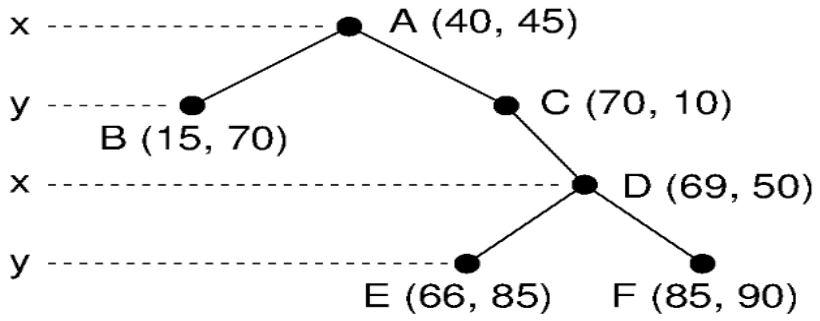
- The **kd tree** is a modification to the **BST** that allows for efficient processing of **multi-dimensional search keys**.
- The **kd tree** differs from the BST in that each level of the **kd tree** makes branching decisions based on a particular search key associated with that level, called the **discriminator or cutting dimension**.
- In principle, the **kd tree** could be used to unify key searching across any arbitrary set of keys (like name and zipcode).
- But in practice, it is nearly always used to support search on multi-dimensional coordinates, such as locations in 2D or 3D space.
- **Discriminator/Cutting Dimension** at level i: $i \bmod k$ for k dimensions.
- Eg: If data organized by xy-coordinates (2D): $k = 2$.
  - with the x-coordinate field arbitrarily designated ***key 0***, and the y-coordinate field designated ***key 1***.

KD Trees

- At each level, the discriminator/cutting dimension alternates between *x* and *y*.
- A node N at *level 0* (the root)
  - Its left subtree only nodes whose *x* values are less than N_x (because *x* is search key 0, and *0 mod 2=0*).
  - The right subtree would contain nodes whose *x* values are greater than N_x.
- A node M at level 1 would have in its left subtree only nodes whose *y* values are less than M_y. (key = 1, *1 mod 2=1*)
- There is no restriction on the relative values of M_x and the *x* values of M's descendants, because branching decisions made at M are based solely on the *y* coordinate.



(a)



(b)

Fig KD(1): Example of a kd tree. (a) The kd tree decomposition for a 128×128-unit region (b) The kd tree for the region of (a).

KD Trees

- In above figure, the region containing the points is restricted to a 128×128 square, and each internal node splits the search space.
- Each split is shown by a line, vertical for nodes with *x* discriminators and horizontal for nodes with *y* discriminators.
- The root node splits the space into two parts; its children further subdivide the space into smaller parts.
- The children's split lines do not cross the root's split line. Thus, each node in the ***kd tree*** helps to decompose the space into rectangles that show the extent of where nodes can fall in the various subtrees.

KD Trees

```
class KDNode { // node in a kd-tree
        Point point; //splitting point
        int cutDim; //cutting dimension
        KDNode left; // children
        KDNode right;

KDNode(Point point, int cutDim){//
        constructor
        this.point = point;
        this.cutDim = cutDim;
        left = right = null;
}
        boolean inLeftSubtree(Point x){//is
                x in left subtree?
        return x[cutDim] < point[cutDim];
        }
}
```

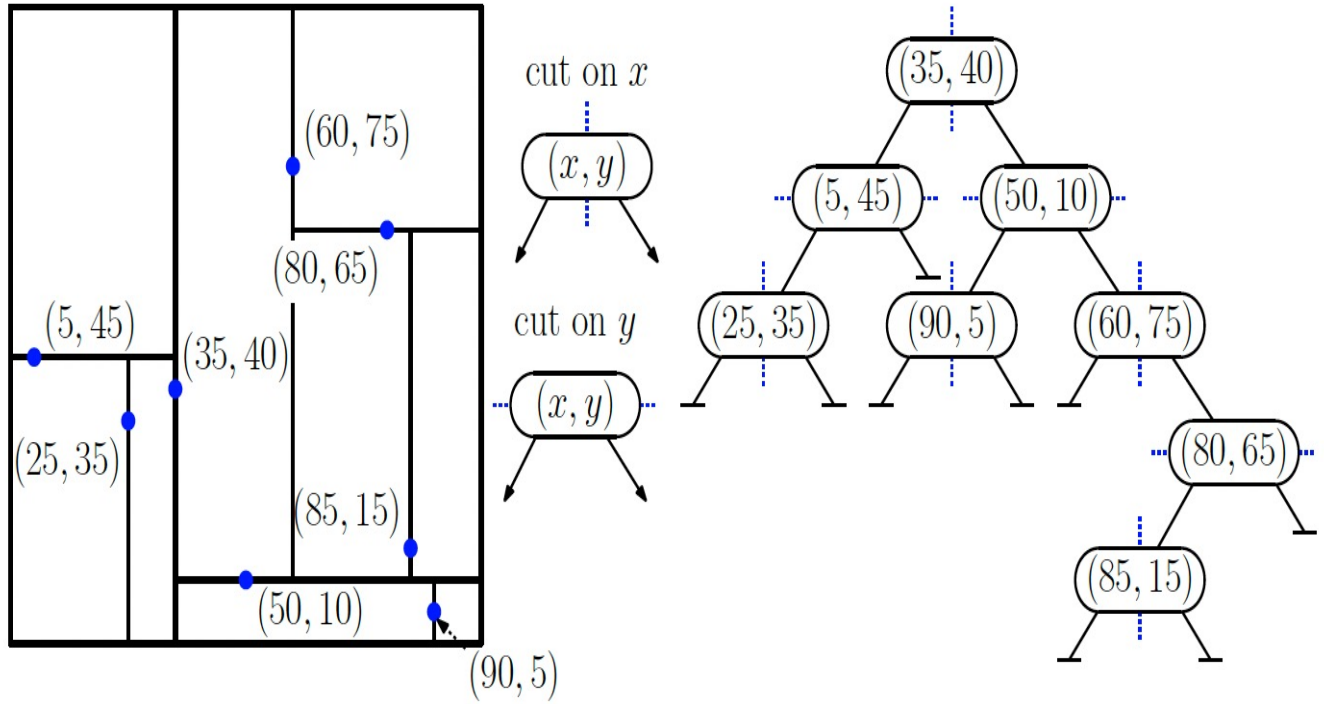
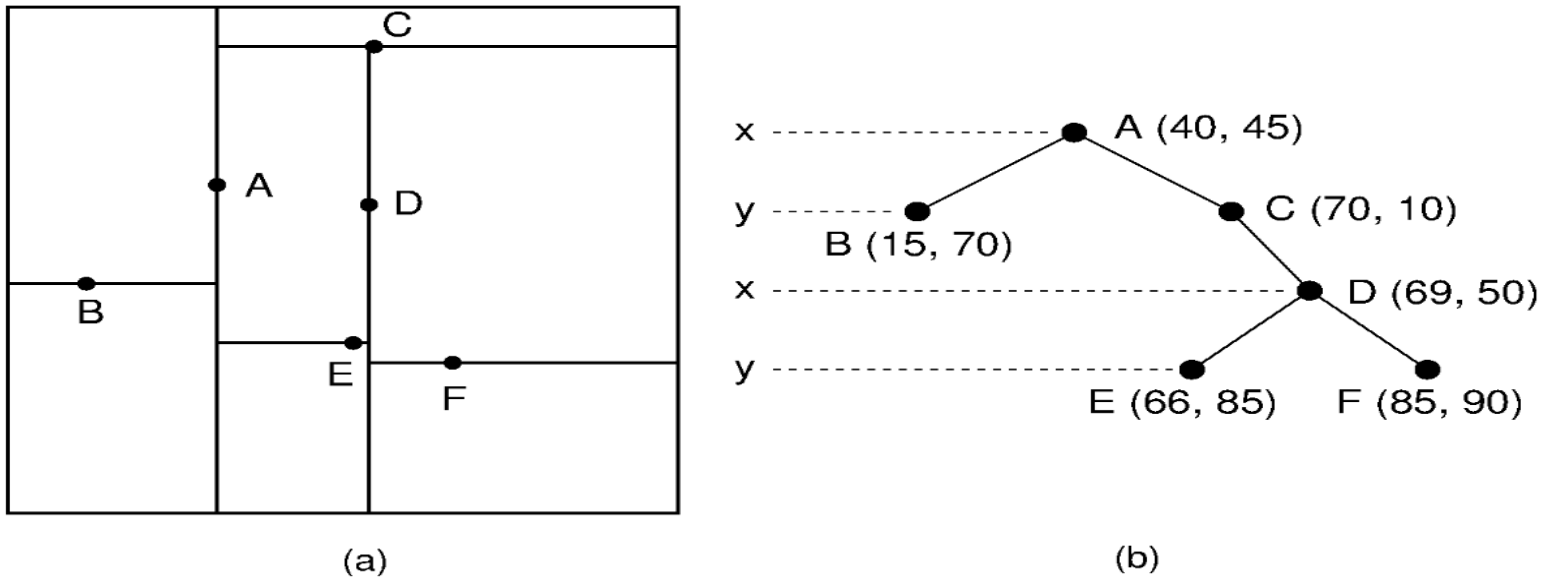


Fig KD(2): Point kd-tree decomposition.

Code: Representation of kd tree

The contents of the left child of a node contain the points x such that *x[cutDim]* < *point[cutDim]* and for the right child *x[cutDim]* >= *point[cutDim]*.



Search Demo (Next Slide)

KD Trees (Searching)

- Consider searching the **kd tree** for a record located at P=(69,50).
- First compare P with the point stored at the root (record A in Fig KD(1),previous slide). If P matches the location of A, then the search is successful.
- In example: it do not match (A(40,45) is not same as (69, 50),so the search must continue.
- The x value of A is compared with that of P to determine in which direction to branch. Because A_x's value of 40 is less than P's x value of 69, we branch to the right subtree.
- A_y does not affect the decision on which way to branch at this level.
- At the second level, P does not match record C's position, so another branch must be taken.
- At this level we branch based on the relative y values of point P and record C(since $1\%2=1$, which corresponds to the y-coordinate).
- C_y's value of 10 is less than P_y's value of 50, we branch to the right.
- At this point, P is compared against the position of D. A match is made and the search is successful.
- If the search process reaches a NULL pointer, then that point is not contained in the tree.

KD Trees

```
Point findMin(KDNode p, int i) { // get min point along dim i
    if (p == null) { // fell out of tree?
        return null;
    }
    if (p.cutDim == i) { // cutting dimension matches i?
        if (p.left == null) // no left child?
                return p.point; // use this point
        else
                return findMin(p.left, i); // get min from left subtree
    }
    else { // it may be in either side
        Point q = minAlongDim(p.point, findMin(p.left, i), i);
        return minAlongDim(q, findMin(p.right, i), i);
    }
}
Point minAlongDim(Point p1, Point p2, int i) { // return smaller point on dim i
    if (p2 == null || p1[i] <= p2[i]) // p1[i] is short for p1.get(i)
        return p1;
    else
        return p2;
}
```

Code: Find the minimum point in subtree along i^{th} coordinate (same function can use for deletion as well)

KD Trees (Insertion)

- Insertion operates as it would for a regular binary search tree.
- We descend the tree until falling out, and then we create a node containing the point and assign its cutting dimension by whatever policy is used by the tree.
- The principal utility function is presented in the below.
- The function takes three arguments, the point *x* being inserted, the current node *p*, and the cutting dimension of the newly created node.
- The initial call is *root = insert(x, root, 0)*.
- An example is shown in ***Fig KD(3)***, where we insert the point (50, 90) into the kd-tree of **Fig KD(2).**
- We descend the tree until we fall out on the left subtree of node (60, 75).
- We create a new node at this point, and the cutting dimension cycles from the parent's x-cutting dimension (cutDim = 0) to a y-cutting dimension (cutDim = 1).

KD Trees (Insertion)

```
KDNode insert(Point x, KDNode p, int cutDim) {
    if (p == null) { // fell out of tree
        p = new KDNode(x, cutDim); // create new leaf
    }
    else if (p.point.equals(x)) {
        throw Exception("duplicate point"); // duplicate data point!
    }
    else if (p.inLeftSubtree(x)) { // insert into left subtree
        p.left = insert(x, p.left, (p.cutDim + 1) % x.getDim());
    }
    else { // insert into right subtree
        p.right = insert(x, p.right, (p.cutDim + 1) % x.getDim());
    }
        return p;
}
```

Note: In some references, we can replace with new point if p.point.equals(x)

KD Trees (Insertion)

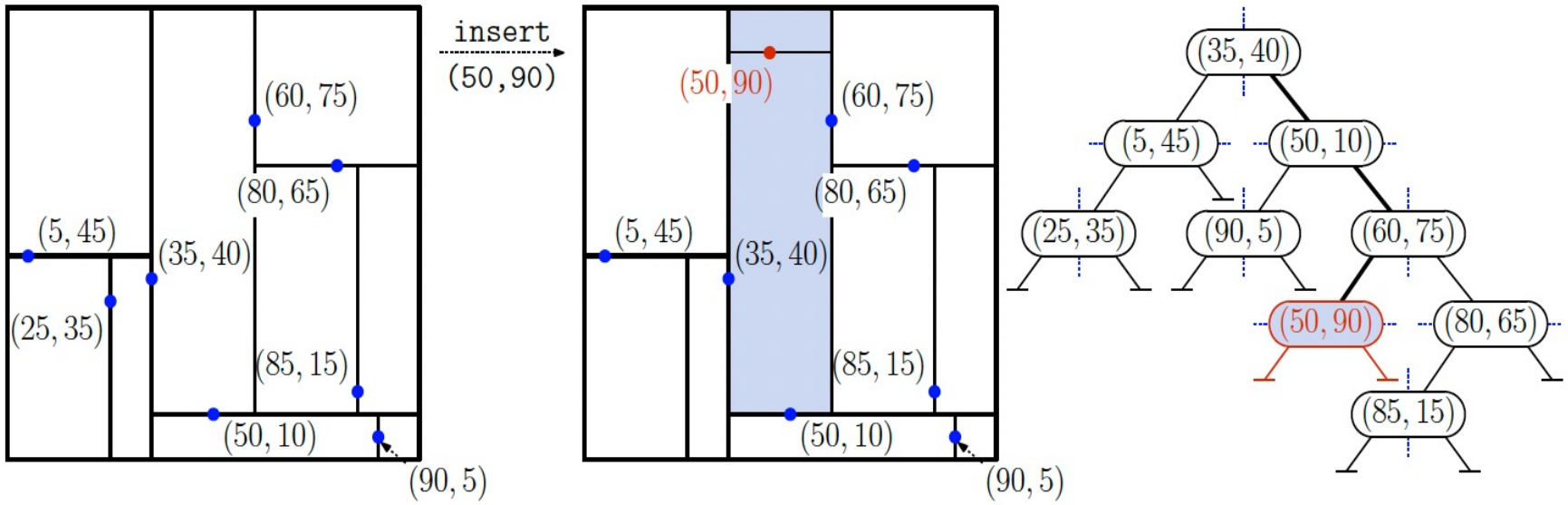


Fig KD(3): Insertion into the point kd-tree of **Fig KD(2)**

Another Example KD tree Insertion Demo from Professor Joshua Hug (Berkeley, California, United States)

KD Trees (Deletion and Replacement)

- With deletion in standard binary search trees, an issue that will arise when deleting a point from the middle of the tree is what to use in place of this node, that is, the replacement point.
- It is not as simple as selecting the next point in an inorder traversal of the tree, since we need a point that satisfies the necessary geometric conditions.
- Suppose that we wish to delete a point in some node p, and suppose further that $p.cutDim == 0$, that is, the x-coordinate. An appropriate choice for the replacement point is the point of $p.right$ that has the smallest x-coordinate.
- In a procedure *findMin(p, i, cutDim)* that computes the point in the subtree rooted at p that has the smallest i^{th} coordinate. The procedure operates recursively. When it arrive at a node p, if the cutting dimension matches i, then it is known that the subtrees are ordered according the the i^{th} coordinate.
- If the left subtree is nonempty, then the desired point is the minimum from this subtree. Otherwise, it return p's associated point.

KD Trees (Deletion and Replacement)

- If *p*'s cutting dimension differs from *i* then it cannot infer which subtree may contain the point with the minimum i^{th} coordinate. In that case try the right subtree, the left subtree, and the point at this node.
- A function *minAlongDim(p1,p2, i)*, which returns whichever point p_1 or p_2 is smallest along coordinate *i*.
- `Function : Point findMin(KDNode p, int i)`
- `Function : Point minAlongDim(Point p1, Point p2, int i)`
- Figure below (next slide) presents an example of the execution of this algorithm to find the point with the minimum *x*-coordinate in the subtree rooted at (55, 40).
- Since this node splits horizontally, we need to visit both of its subtrees to find their minimum *x* values. (These will be (15, 10) for the left subtree and (10, 65) for the right subtree.)
- These values are then compared with the point at the root to obtain the overall *x*-minimum point, namely (10, 65).
- The subtrees at (45, 20) and (35, 75) both split on the *x*-coordinate, and we are looking for the point with the minimum *x*-coordinate, we do not need to search their right subtrees.
- The nodes visited in the search are shaded in blue.

KD Trees (Deletion and Replacement)

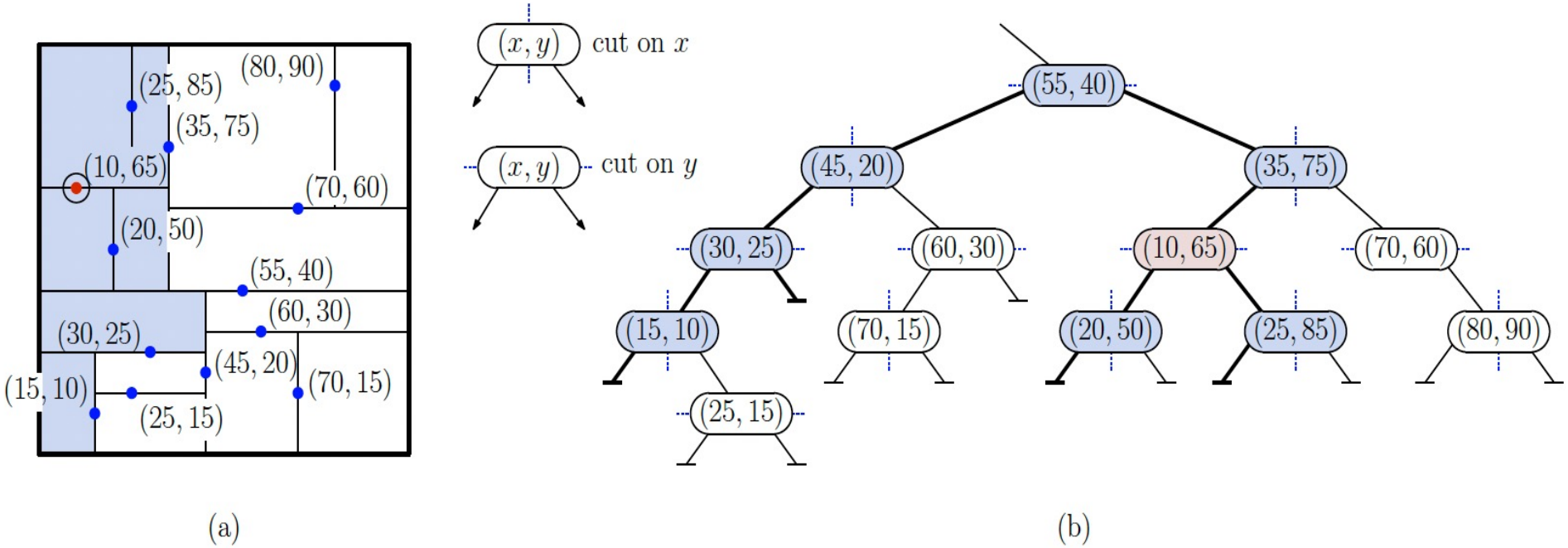


Fig KD(4) : Example of findMin when i = 0 (the x-coordinate) on the subtree rooted at (55, 40). The function returns (10, 65).

KD Trees (Deletion)

Deletion from a kd-tree:

- In 1D BST case we needed to consider a number of different cases.
- If the node is a leaf we just delete the node. Otherwise, its deletion would result in a hole in the tree.
- We need to find an appropriate replacement element. In 1D case, we were able to simplify this if the node has a single child (by making this child the new child of our parent).
- This would move the child from an even level to an odd level, or vice versa (this would violate the assumption of cutting dimensions cycle among the coordinates. (Given procedure will not alter a node's cutting dimension)
- Assume that the right subtree is non-empty. In 1D case, the replacement key was taken to be the smallest key from the right child, and after using the replacement to fill the hole, it recursively deleted the replacement.
- In Multi-dimentional case ,the thing to do is to find the point whose coordinate along the current cutting dimension is minimum. Thus, if the cutting dimension is the x-axis, then the replacement key is the point with the smallest x-coordinate in the right subtree.
- We use the findMin() function to do this.

KD Trees (Deletion)

- What if the right subtree is empty?
  - At first, it might seem that the right thing to do is to select the maximum node from the left subtree. (It should be noted that the points whose coordinates are equal to the cutting dimension are stored in the right subtree. If we select the replacement point to be the point with the maximum coordinate from the left subtree, and if there are other points with the same coordinate value in this subtree, then it violates our invariant).
  - Trick: For the replacement element we will select the minimum (not maximum) point from the left subtree, and we move the left subtree over and becomes the new right subtree.
  - The left child pointer is set to null.

KD Trees (Deletion)

```
KDNode delete(Point x, KDNode p) {
    if (p == null) { // fell out of tree?
        throw Exception("point does not exist");
    }
    else if (p.point.equals(x)) { // found it
        if (p.right != null) { // take replacement from right
                p.point = findMin(p.right, p.cutDim);
                p.right = delete(p.point, p.right);
        }
        else if (p.left != null) { // take replacement from left
                p.point = findMin(p.left, p.cutDim);
                p.right = delete(p.point, p.left); // move left subtree to right!
                p.left = null; // left subtree is now empty
        }
        else { // deleted point in leaf
                p = null; // remove this leaf
        }
    } else if (p.inLeftSubtree(x)) {
        p.left = delete(x, p.left); // delete from left subtree
    }
    else { // delete from right subtree
        p.right = delete(x, p.right);
    }
    return p;
}
```

KD Trees

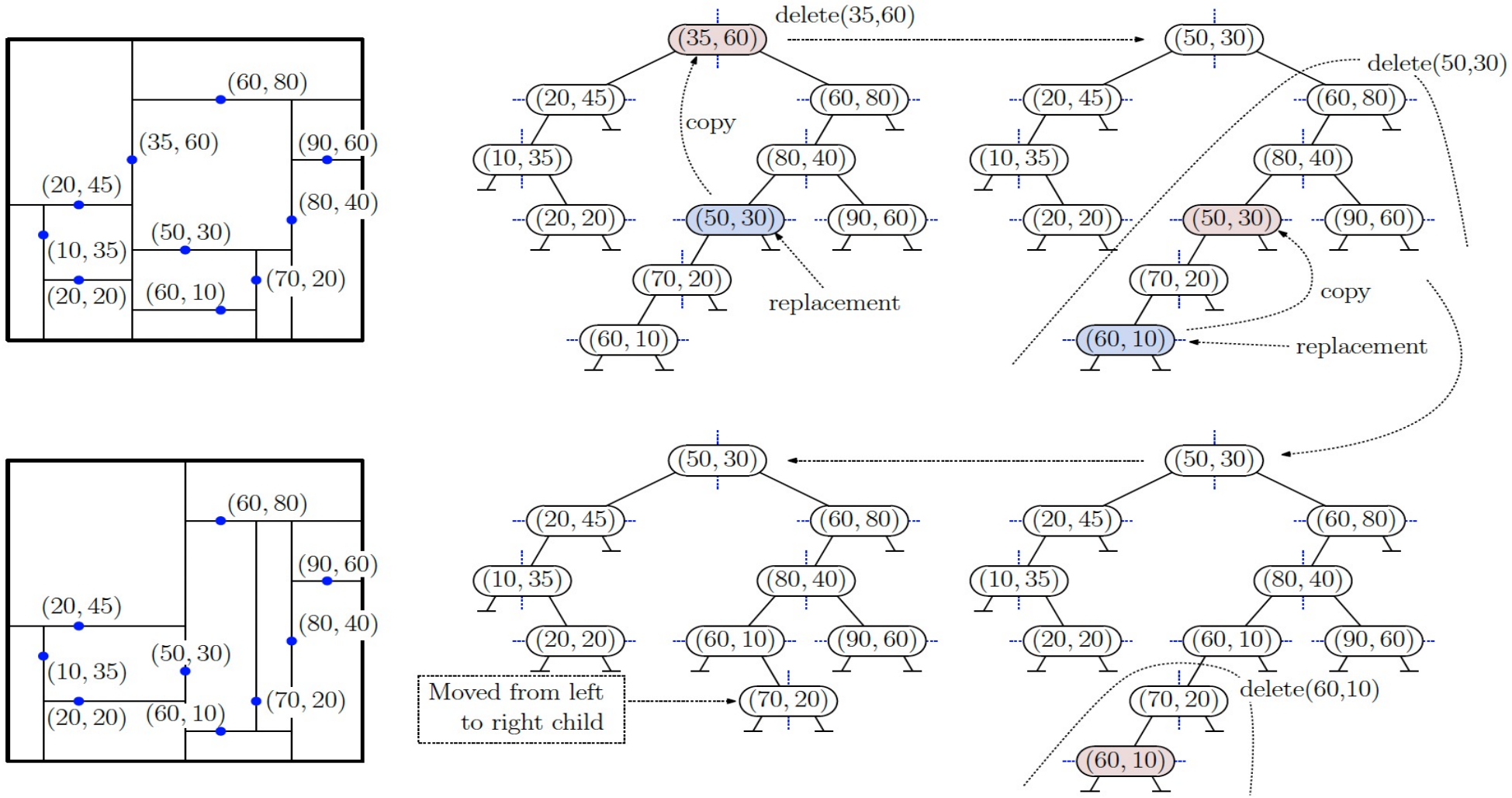


Fig KD(5): Deletion from a kd-tree. **Delete (35,60)**

KD Trees

Description: Objective is to delete the point (35,60)

- It is at the root of the tree. Because the cutting dimension is vertical, search its right subtree to find the point with the minimum *x*-coordinate, which is (50, 30).
- The point is copied to the root, and then recursively delete (50, 30) from the root's right subtree. This recursive call then seeks the node p containing (50, 30).
- This node has no right child, unlike standard binary search trees, we cannot simply unlink it from the tree. It is observe that its cutting dimension is horizontal, and we search for the point with the minimum *y*-coordinate in *p*'s left subtree, which is (60, 10).
- Copy (60, 10) to *p*, and then recursively delete (60,10) from *p*'s left subtree.
- It is a leaf, so it may simply be unlinked from the tree.
- Finally move this left subtree *p* over to become *p*'s right subtree.

KD Trees (Nearest Neighbor Search)

Steps:

1. **Start at the Root Node**
   - Begin with the root of the KD-tree and compare the query point with the splitting dimension (e.g., x, y, etc.) of the current node.
   - Decide whether to go to the **left** or **right** child based on the comparison .

2. **Traverse Down the Tree**
   - Recursively move down the tree following these rules:
     - If the query point's value in the splitting dimension is less than the node's value, go to the left child.
     - Otherwise, go to the right child.
   - Keep traversing until you reach a leaf node (a node with no children).

KD Trees (Nearest Neighbor Search)

Steps:

3. Assume the Leaf Node is the Nearest Neighbor

- Set the current nearest neighbor (NN) as the point at the leaf node.
- Calculate the distance between the query point and this leaf node (using Euclidean distance : $\mathbf{sqrt((x_2-x_1)^2+(y_2-y_1)^2))}$
- Store this distance as the **current best distance**.

4. Backtrack and Explore Other Branches

- During backtrack up the tree, check if the splitting hyperplane (the dividing line or plane for the current node) could have points closer to the query point.
- **How to check:** Compare the best distance so far with the distance to the splitting hyperplane. If the hyperplane is closer, the other branch might contain a closer point.
- If the other branch could contain a closer point, **explore it recursively**.

KD Trees (Nearest Neighbor Search)

Steps:

5. Update the Nearest Neighbor

- For each node visited (while backtracking or exploring the other branch):
  - Calculate the distance between the query point and the current node.
  - If this distance is smaller than the current best distance, update the nearest neighbor and the best distance.

6. Continue Until the Root is Revisited

- Keep backtracking and checking all possible branches that might contain closer points.
- When returning to the root node and finish exploring all branches, the **current nearest neighbor** will be the final result.

KD Trees (Nearest Neighbor Search)

Correction (Another Version):

- Many implementations of **KD-Tree Nearest Neighbor Search** do calculate the distance to the root and intermediate nodes **while traversing to the leaf node**. This approach is perfectly valid and can provide some advantages.

Advantages:

- **More Pruning:**
  - Early distance calculations allow branches to be pruned earlier, reducing unnecessary computations.
- **Faster Nearest Neighbor Detection:**
  - A good candidate may be found early, potentially avoiding deeper backtracking.
- **Flexibility in Implementation:**
  - This approach often feels more intuitive and aligns well with depth-first traversal logic.

KD Trees (Nearest Neighbor Search)

Simple Example:

KD-Tree Structure

- Root: A=(2,3)
- Left Child: B=(1,5)
- Right Child: C=(5,4)
- Query Point: Q=(3,4)

Step-by-Step Nearest Neighbor Search

1. Start at the Root Node A(2,3)

- Calculate the distance between Q and A: $d(Q,A)=\text{sqrt}((3-2)^2+(4-3)^2)=1+1=\text{sqrt}(2)$
- Current nearest neighbor = A(2,3), best distance = sqrt(2) .

2. Traverse to the Right Child C(5,4)

- Compare Q[0](3) with A[0](2). Since 3>2, move to the right child.
- Calculate the distance between Q and C(5,4) :$d(Q,C)=\text{sqrt}((5-3)^2+(4-4)2)=\text{sqrt}(4+0)=2$
- Compare d(Q,C)=2 with the current best distance sqrt(2):
  - Sqrt(2)<2, so A(2,3) remains the nearest neighbor.

KD Trees (Nearest Neighbor Search)

3. No Further Right Subtree (Leaf Node Reached)

- Since C(5,4) is a leaf, stop downward traversal.

4. Backtrack to Check the Left Child B(1,5)

- The left child of A(2,3) is B(1,5).
- Calculate the distance between Q and B :$d(Q,B)=\text{sqrt}((3-1)^2+(4-5)^2)=\text{sqrt}(4+1)=\text{sqrt}(5)$
  $d(Q,B)=\text{sqrt}(5)$

- Compare $d(Q,B)=\text{sqrt}(5)$ with the current best distance sqrt(2):
  - Sqrt(2<5) , so A(2,3) remains the nearest neighbor.

- **Conclusion**: The nearest neighbor to Q(3,4) is: A(2,3)
- Distance to the nearest neighbor: sqrt(2) ≈1.412

- Another Example KD tree Nearest Search Demo from Professor Joshua Hug (Berkeley, California, United States)

Complexities in KD tree

Operation	Average Case	Worst Case
Construction	O(nlogn)	$O(n^2)$
Insertion	O(logn)	O(n)
Deletion	O(logn)	O(n)
Exact Match Search	O(logn)	O(n)
Nearest Neighbor Search	O(logn)	O(n)
Range Search	$O(n^{1-1/k}+m)$	O(n)
Space Complexity	O(n)	O(n)

R-Trees

R-Tree

- Efficient indexing and searching of spatial data, such as counties or map objects, pose challenges due to their multi-dimensional nature and non-zero size in space.
- 1D indexing structures like B-trees and hash tables are unsuitable for spatial searches, as they cannot handle the complexity of multi-dimensional range queries.
- This limitation is significant in applications like CAD and geo-data systems, where operations such as finding all objects within a specified distance are common.
- Structures like quad trees, k-d trees, and grid files attempt to address these needs, they often fall short in handling dynamic datasets or large-scale collections efficiently.
- To overcome these challenges, the R-tree is introduced as a robust solution, representing spatial objects using intervals in multiple dimensions and supporting efficient search, insertion, deletion, and updates.

R-Tree

- An R-tree is a height-balanced tree similar to a B-tree with index records in its leaf nodes containing pointers to data object.
- Nodes correspond to disk pages if the index is disk-resident, and the structure is designed so that a spatial search requires visiting only a small number of nodes
- The index is completely dynamic; inserts and deletes can be mixed with searches and no periodic reorganization is required.
- A spatial database consists of a collection of tuples representing spatial objects, and each tuple has a unique identifier which can be used to retrieve it.

R-Tree Structure

- Let P be a set of points. In R-Tree
  - Actual data point is store in the leaf node. (P points).
  - Each leaf node can store up to B points, where B is the size of a disk page (a fixed number of entries that can fit in one page of disk memory).
  - Non-leaf nodes in the tree represent bounding boxes (minimum bounding rectangles, or **MBR**s) that encapsulate the data points or child nodes beneath them.
  - Each non-leaf node can have up to B children, ensuring efficient organization and a balanced tree structure.
  - The root of the tree must have at least 2 children unless it is the only node in the tree. This ensures the tree maintains its hierarchical structure while accommodating small datasets.
  - *Each leaf node has between **0:4B** and **B** data points, where $B \geq 3$ is a parameter.*
  - The use of B as the maximum number of points or children per node aligns with the concept of **disk paging**, where nodes are designed to fit entirely within a single page of memory to minimize disk I/O during queries.
- There is no constraint on how points should be grouped into leaf nodes, and in general, how non-leaf nodes should be grouped into nodes of higher levels. Since each point is stored only once, the entire tree consumes linear space O(N/B), where N is the cardinality of P.

R-Tree (Example)

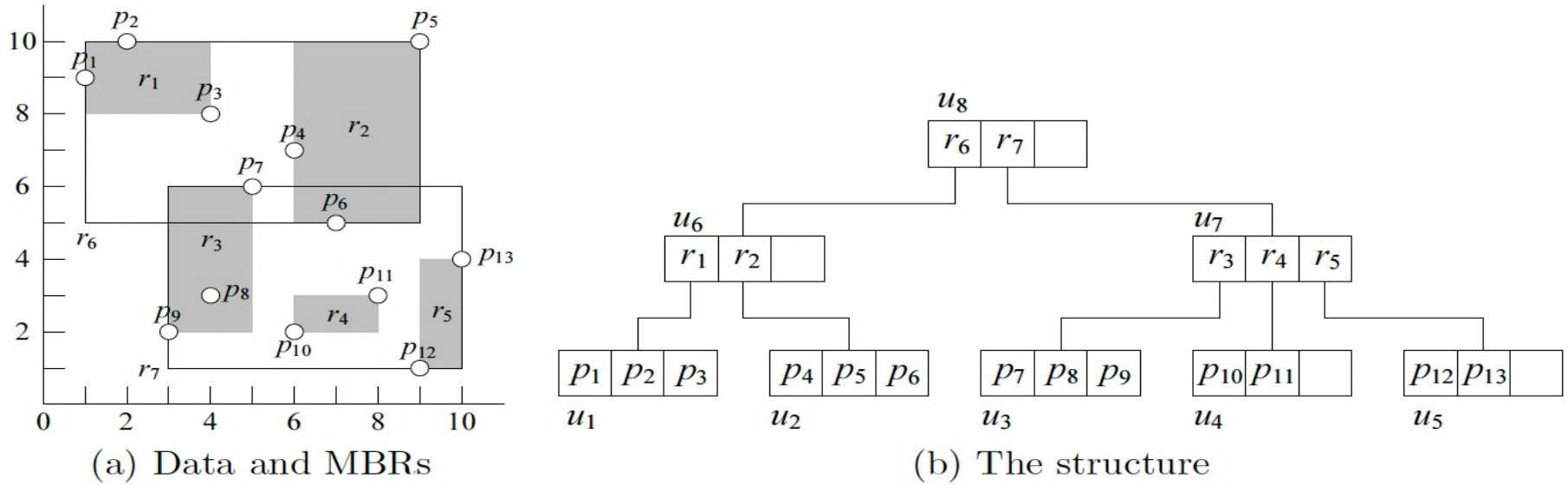
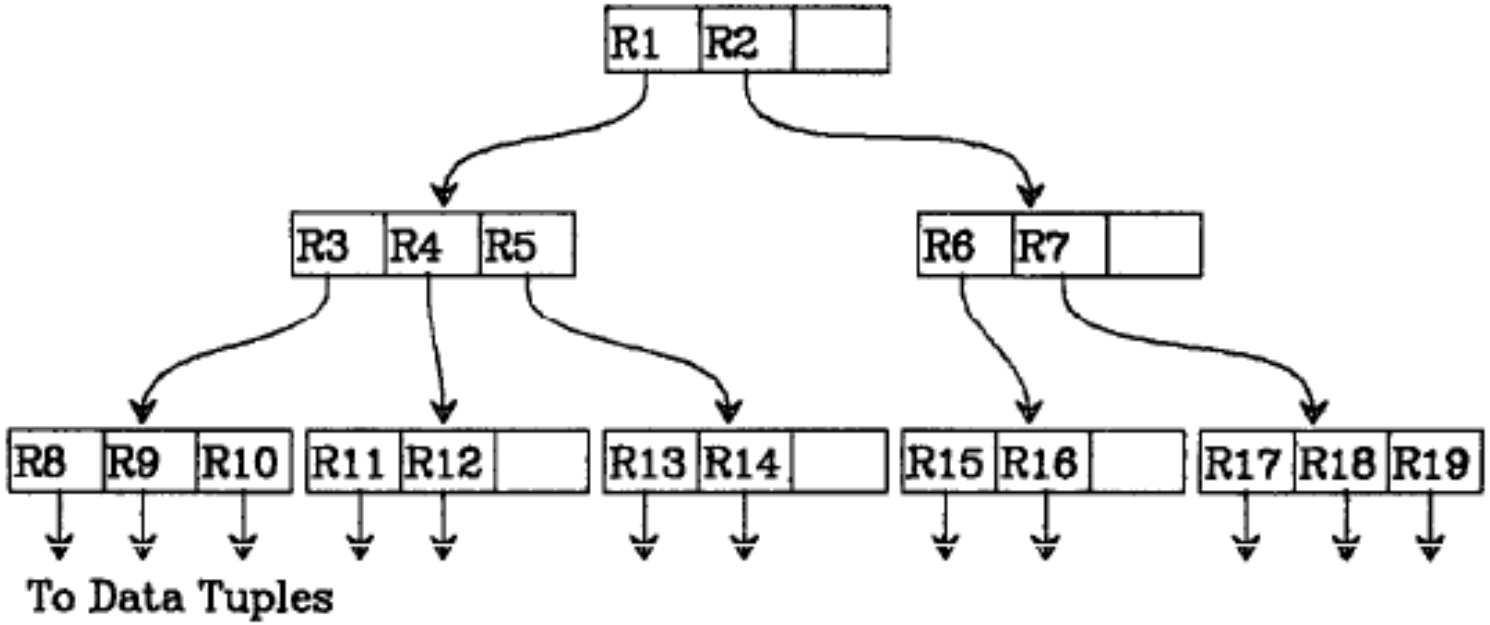


Fig: An Example of R-Tree

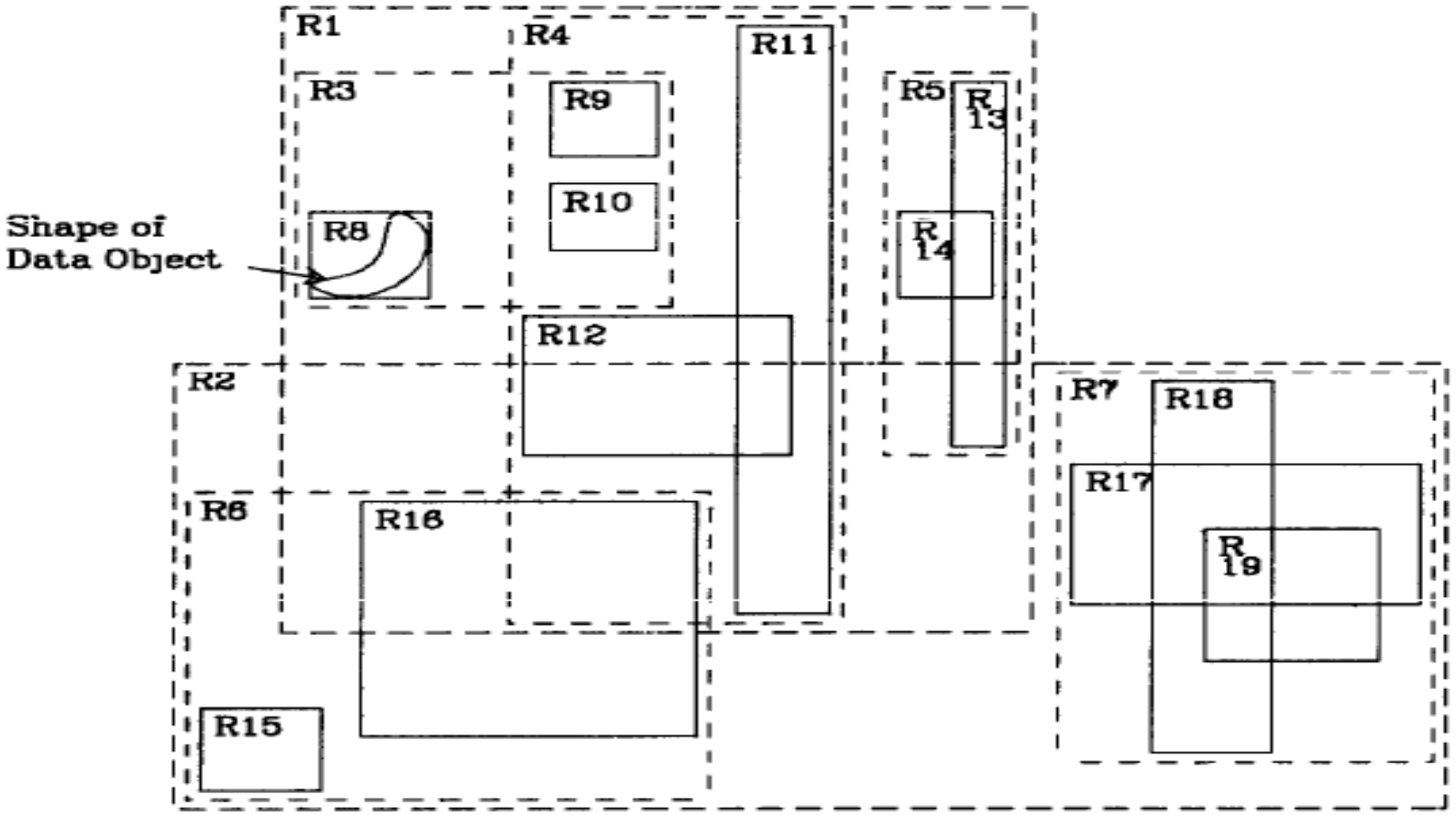
Above fig is an example of R-tree where P has 13 points p_1, p_2, …, p_{13}. Points p_1, p_2, p_3, for example, are grouped into leaf node u_1. This leaf is a child of of non-leaf node u_6, which stores an MBR r_1 for u_1. Note that r_1 tightly bounds all the points in u_1.

R-Tree (Example)



Ref: Antonin Guttman. 1984. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data* (SIGMOD '84)

R-Tree (Example)



Ref: Antonin Guttman. 1984. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data* (SIGMOD '84)

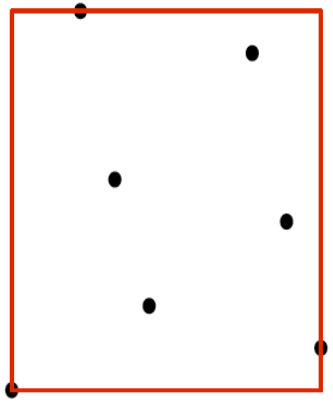
R-Tree

R-Tree

- Each leaf node has between $0.4B$ and B data points, where $B \geq 3$ is a parameter. The only exception applies when the leaf is the root, in which case it is allowed to have between 1 and B points. All the leaf nodes are at the same level.
- Each internal node has between $0.4B$ and B child nodes, except when the node is the root, in which case it needs to have at least 2 child nodes.

R-Tree

For any node u, denote by S_u the set of points in the subtree of u. Consider now u to be an internal node with child nodes $v_1, ..., v_f$ ($f \leq B$). For each v_i ($i \leq f$), u stores the minimum bounding rectangle (MBR) of S_{v_i}, denoted as $MBR(v_i)$.

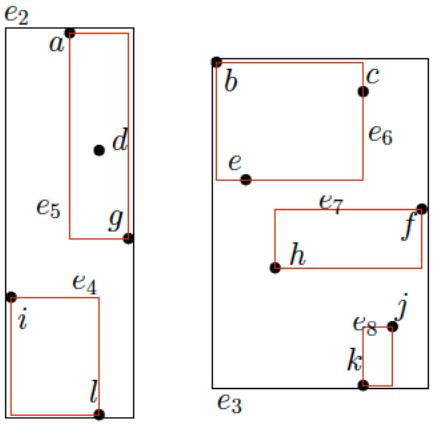


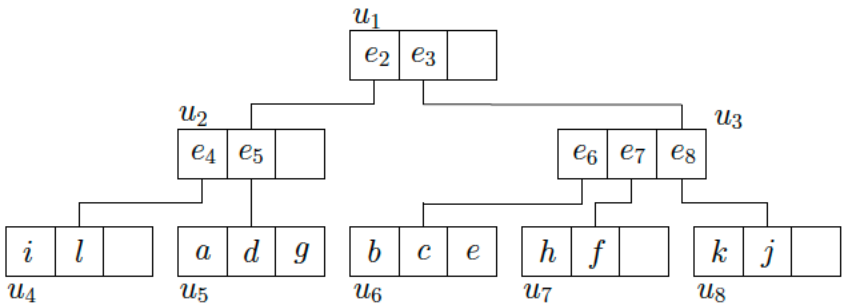
The above is an MBR on 7 points.

R-Tree

Example

Assume $B = 3$.

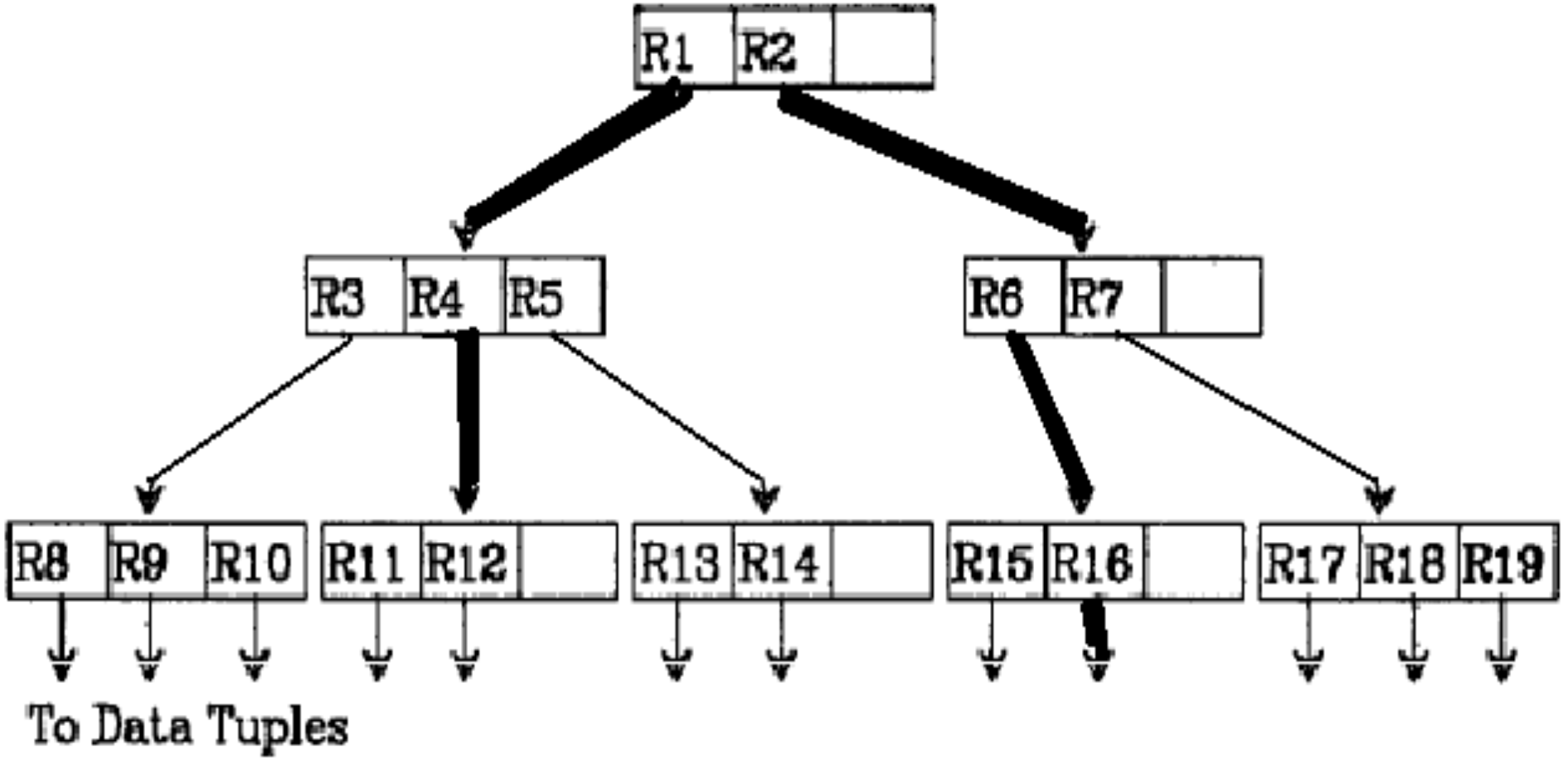




Searching

- Given a search rectangle S …
  1. Start at root and locate all child nodes whose rectangle I intersects S (via linear search).
  2. Search the subtrees of those child nodes.
  3. When you get to the leaves, return entries whose rectangles intersect S.
- Searches may require inspecting several paths.
- Worst case running time is not so good …

S = R16


R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15
R16
R17
R18
R19
To Data Tuples

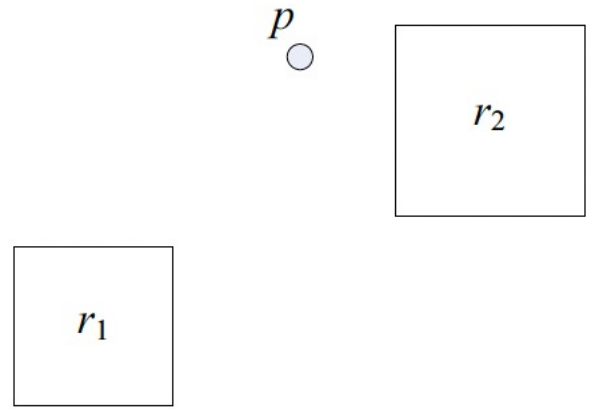
R-Tree (Insertion)

- To insert a point p, use a strategy similar to that of a B-tree. Specifically, we add p to a leaf node u by following a single root-to-leaf path. If u overflows, split it, which creates a new child of parent(u). In case parent(u) overflows, also split it, which propagates upwards in the same manner. Finally, if the root is split, then a new root is created.
- Even though all these sound familiar, however, two important differences.
  - First, although in the B-tree the insertion path is unique this is not true at all for the R-tree.
    - The new point p can be inserted into any leaf, which always results in a legal structure.
    - If a bad leaf is chosen (to contain p), its MBR may need to be enlarged substantially, thus harming the efficiency of the tree.
  - Second, the split algorithm is not as trivial as in a B-tree because now it have to tackle multiple dimensions.

R-Tree (Insertion)

Choosing a subtree to insert.

- We are essentially facing the following problem. Given a non-leaf node *u* with children $v_1, v_2, \ldots, v_f$ for some f = B, we need to pick the best child v* such that the new point *p* is best inserted into the subtree of v*.
- An approach that seems to work well in practice is a greedy one. Specifically, v* can simply be the child v_i whose MBR requires the least increase of perimeter in order to cover *p*. For example, in below figure, both MBRs r_1 and r_2 must be expanded to enclose *p*, but r_2 incurs smaller perimeter increase, and hence, is a better choice.
- It is possible that *p* falls into the overlapping region of multiple MBRs.
- All those MBRs have a tie because none of them needs any perimeter increase to cover *p*.
- In this case, the winner can be decided according to other factors such as picking the MBR having the smallest area.



Fig; MBR r_2 requires smaller perimeter increase to cover p

R-Tree (Insertion)

Node split. The node split problem can be phrased as follows. Given a set S of $B+1$ points, split it into disjoint subsets S_1 and S_2 with $S_1 \cup S_2 = S$ such that

- $|S_1| \geq \lambda B$, $|S_2| \geq \lambda B$, where constant λ is the *minimum utilization rate* of a node, and
- the sum of the perimeters of $MBR(S_1)$ and $MBR(S_2)$ is small.

- For simplicity we assume that $|S|$ is an even number, and $|S_1| = |S_2| = |S|/2$, i.e., we always aim at an even split.
- Intention is to find the optimal split that minimizes the perimeter sum of MBR(S_1) and MBR(S_2).
- Since an MBR is decided by 4 coordinates (i.e., a pair of opposite corners), it is easy to find the optimal split is significantly improved to $O(B^2)$ time from $O(B^4)$ [***Explain in other paper***]
- Unfortunately, even a quadratic split time is usually excessively long in practice. Therefore, we turn our attention to heuristics that do not guarantee optimality, but usually produce fairly good splits.
- Split algorithm that runs in **O(BlogB)** time, or **O(dBlogB)** time in general d-dimensional space.

R-Tree (Insertion)

Insertion

Let p be the point being inserted. The pseudo-code below should is invoked as `insert`$(root, p)$, where *root* is the root of the tree.

Algorithm `insert`(u, p)

1. **if** u is a leaf node **then**
2. add p to u
3. **if** u overflows **then**
   /* namely, u has $B + 1$ points */
4. `handle-overflow`(u)
5. **else**
6. $v \leftarrow$ `choose-subtree`(u, p)
   /* which subtree under u should we insert p into? */
7. `insert`(v, p)

Overflow Handling

Algorithm `handle-overflow`(u)

1. `split`(u) into u and u'
2. **if** u is the root **then**
3. create a new root with u and u' as its child nodes
4. **else**
5. $w \leftarrow$ the parent of u
6. update $MBR(u)$ in w
7. add u' as a child of w
8. **if** w overflows **then**
9. `handle-overflow`(w)

R-Tree (Insertion)

Splitting a Leaf Node

Algorithm `split`(u)

1. m = the number of points in u
2. sort the points of u on x-dimension
3. **for** $i = \lceil 0.4B \rceil$ to $m - \lceil 0.4B \rceil$
4. $\quad S_1 \leftarrow$ the set of the first i points in the list
5. $\quad S_2 \leftarrow$ the set of the other i points in the list
6. $\quad$ calculate the perimeter sum of $MBR(S_1)$ and $MBR(S_2)$; record it if this is the best split so far
7. Repeat Lines 2-6 with respect to y-dimension
8. **return** the best split found

R-Tree (Insertion)

Heuristic Splitting

Splitting an Internal Node

Let S be a set of $B+1$ rectangles. Divide S into two disjoint sets S_1 and S_2 to minimize the perimeter sum of $MBR(S_1)$ and $MBR(S_2)$, subject to the condition that $|S_1| \geq 0.4B$ and $|S_2| \geq 0.4B$.

Once again, we will settle for an algorithm that is fast but does not always return an optimal split.

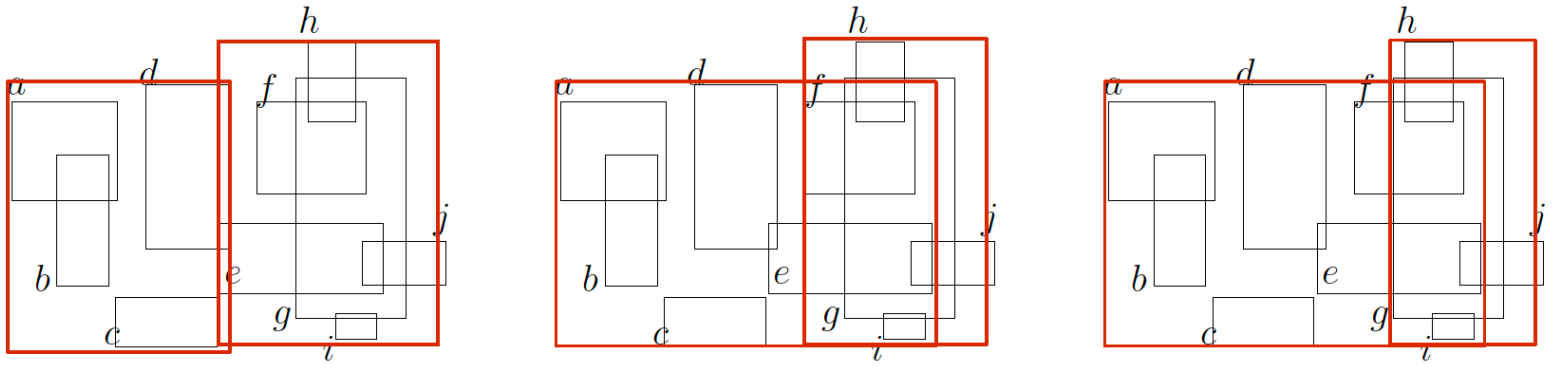
Splitting an Internal Node

Algorithm `split`(u)
/* u is an internal node */

1. m = the number of points in u
2. sort the rectangles in u by their left boundaries on the x-dimension
3. **for** $i = \lceil 0.4B \rceil$ to $m - \lceil 0.4B \rceil$
4. $S_1 \leftarrow$ the set of the first i rectangles in the list
5. $S_2 \leftarrow$ the set of the other i rectangles in the list
6. calculate the perimeter sum of $MBR(S_1)$ and $MBR(S_2)$; record it if this is the best split so far
7. Repeat Lines 2-6 with respect to the right boundaries on the x-dimension
8. Repeat Lines 2-7 w.r.t. the y-dimension
9. **return** the best split found

R-Tree (Insertion)

Example

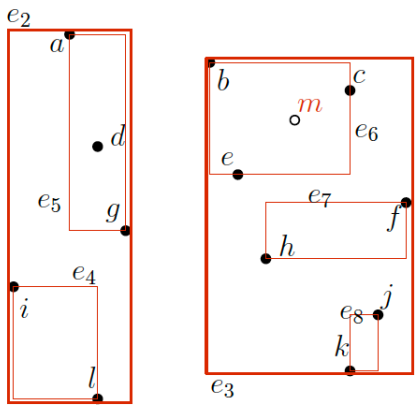


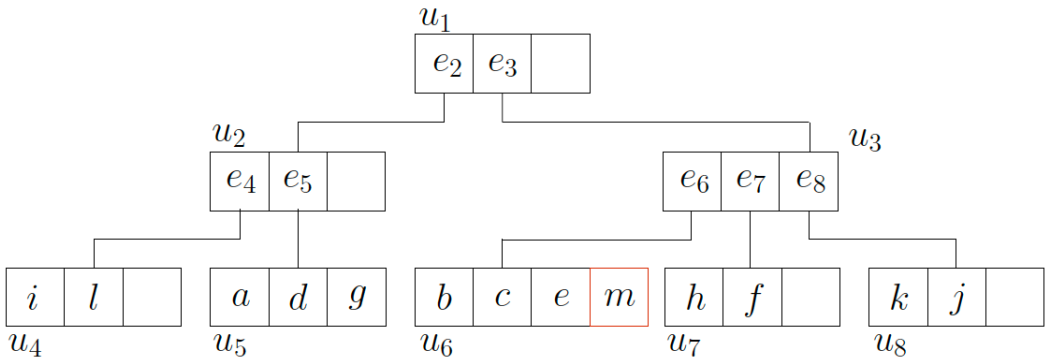
There are 3 possible splits w.r.t. the left boundaries on the x-dimension. Remember that each node must have at least $0.4B = 4$ points (here $B = 10$).

R-Tree (Insertion)

Insertion Example

Assume that we want to insert the white point m. By applying `choose-subtree` twice, we reach the leaf node u_6 that should accommodate m. The node overflows after incorporating m (recall $B = 3$).

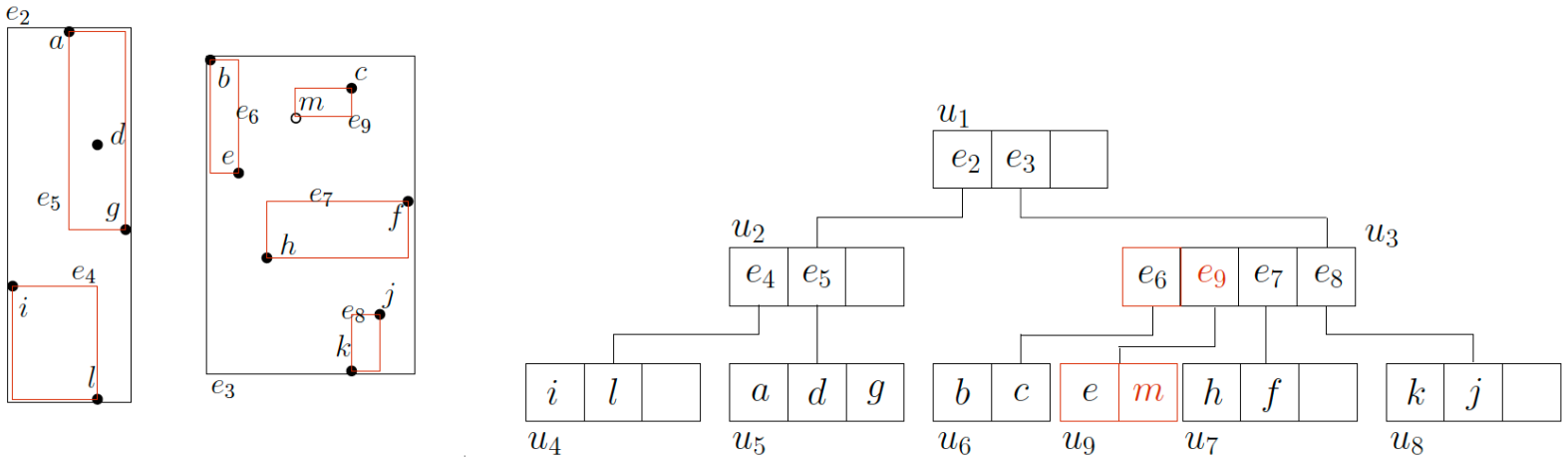




R-Tree (Insertion)

Insertion Example

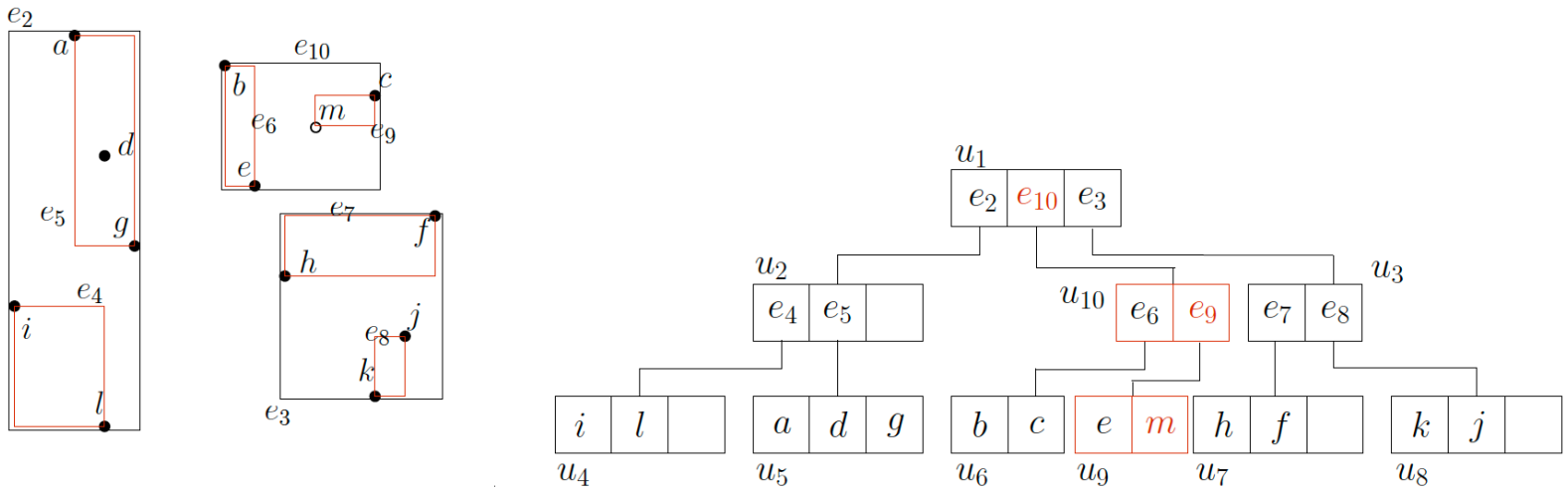
Node u_6 splits, generating u_9. Adding u_9 as a child of u_3 causes u_3 to overflow.



R-Tree (Insertion)

Insertion Example

Node u_3 splits, generating u_{10}. The insertion finishes after adding u_{10} as a child of the root.



R-Tree (Deletion)

- Node underflows are handled in a way that differs considerably from the conventional merging approach as in a B-tree.
- let p be the point to be deleted. First, we need to find the leaf node u where p is stored. This can be achieved with a special range query using p itself as the search region. Then, p is removed from u. The deletion finishes if u still has λB items, where λ denotes the minimum node utilization.
- Otherwise, u underflows, which is handled by first removing u from its parent, and then re-inserting all the remaining points in u (using exactly the insertion algorithm).
- Note that removing u from parent(u) may cause parent(u) to underflow too.
- In general, the underflow of a non-leaf node u' is also handled by re-insertions, with the only difference that the items re-inserted are MBRs, and each MBR is re-inserted to the same level of u'.
- Note that re-insertion actually gives better search performance than merging-based algorithms to handle node underflows.
- This is because the structure of an R-tree is sensitive to the insertion order of the data points. Re-insertion gives the the early inserted points to be inserted in other (better) branches, thus improving the overall structure.

Reference

- Quad Tree and kd trees:
  - https://www.cs.umd.edu/class/spring2022/cmsc420-0101/Lects/lect13-kd-dict.pdf
  - https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/PRquadtree.html
  - https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/KDtree.html
- K-nearest Search :
  - http://45.33.90.109/courses/cs201/w22/notes/trees-kd.html#sec-2
- R-Tree :
  - Antonin Guttman. 1984. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data* (SIGMOD '84)
  - https://www.cse.cuhk.edu.hk/~taoyf/course/infs4205/lec/rtree.pdf