

# *Spatial Data Management*

## Chapter 28



# *Types of Spatial Data*

## ❖ **Point Data**

- Points in a multidimensional space
- E.g., *Raster data* such as satellite imagery, where each pixel stores a measured value
- E.g., Feature vectors extracted from text

## ❖ **Region Data**

- Objects have spatial extent with location and boundary
- DB typically uses geometric approximations constructed using line segments, polygons, etc., called *vector data*.



# *Types of Spatial Queries*

## ❖ **Spatial Range Queries**


- *Find all cities within 50 miles of Madison*
- Query has associated region (location, boundary)
- Answer includes overlapping or contained data regions

## ❖ **Nearest-Neighbor Queries**

- *Find the 10 cities nearest to Madison*
- Results must be ordered by proximity

## ❖ **Spatial Join Queries**

- *Find all cities near a lake*
- Expensive, join condition involves regions and proximity



# *Applications of Spatial Data*

- ❖ Geographic Information Systems (GIS)
  - E.g., ESRI's ArcInfo; OpenGIS Consortium
  - Geospatial information
  - All classes of spatial queries and data are common
- ❖ Computer-Aided Design/Manufacturing
  - Store spatial objects such as surface of airplane fuselage
  - Range queries and spatial join queries are common
- ❖ Multimedia Databases
  - Images, video, text, etc. stored and retrieved by content
  - First converted to *feature vector* form; high dimensionality
  - Nearest-neighbor queries are the most common

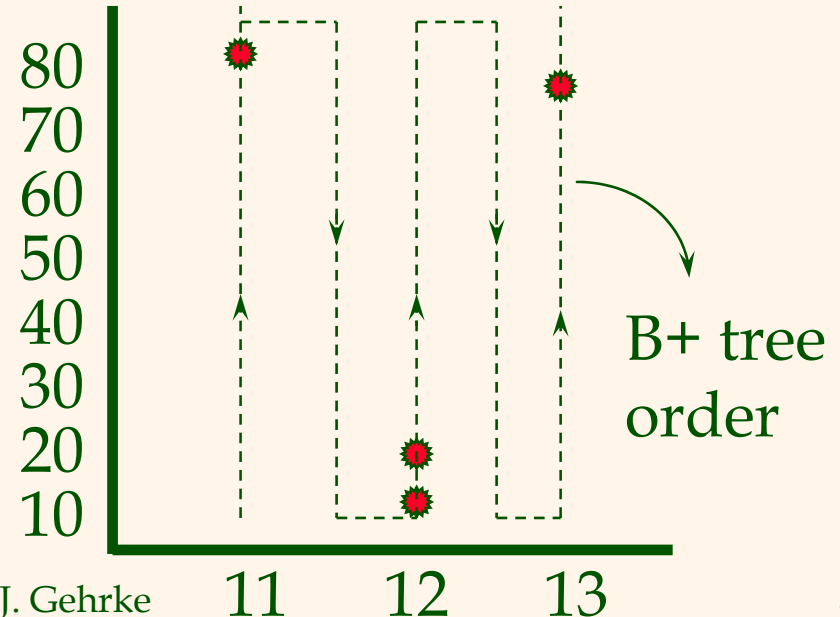
# Single-Dimensional Indexes

- ❖ B+ trees are fundamentally **single-dimensional** indexes.
- ❖ When we create a composite search key B+ tree, e.g., an index on **<age, sal>**, we effectively linearize the 2-dimensional space since we sort entries first by **age** and then by **sal**.

Consider entries:

**<11, 80>**, **<12, 10>**

**<12, 20>**, **<13, 75>**





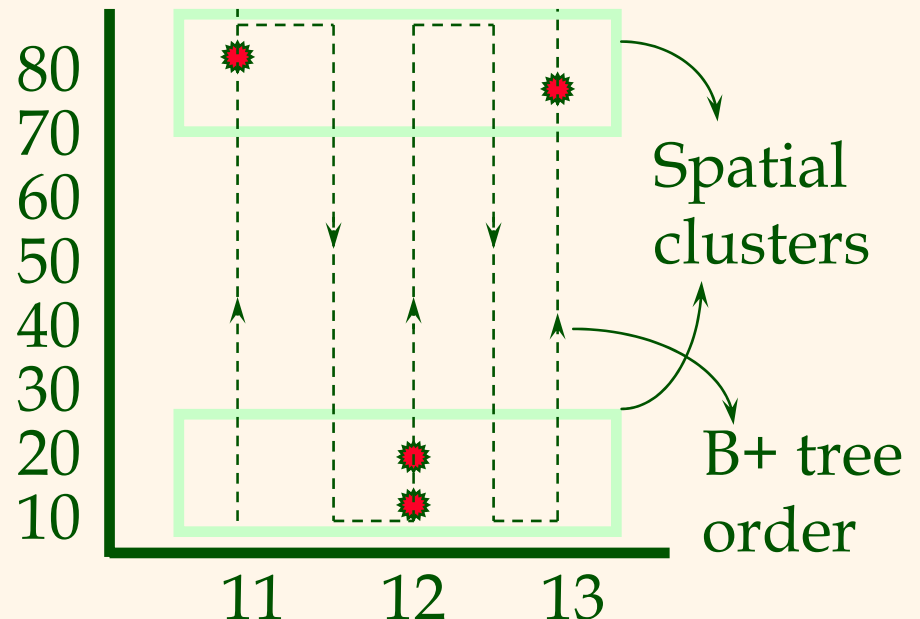
# Multidimensional Indexes

- ❖ A multidimensional index **clusters** entries so as to exploit “nearness” in multidimensional space.
- ❖ Keeping track of entries and maintaining a balanced index structure presents a challenge!

Consider entries:


$\langle 11, 80 \rangle, \langle 12, 10 \rangle$

$\langle 12, 20 \rangle, \langle 13, 75 \rangle$



# Motivation for Multidimensional Indexes

- ❖ **Spatial queries (GIS, CAD).**
  - Find all hotels within a radius of 5 miles from the conference venue.
  - Find the city with population 500,000 or more that is nearest to Kalamazoo, MI.
  - Find all cities that lie on the Nile in Egypt.
  - Find all parts that touch the fuselage (in a plane design).
- ❖ **Similarity queries (content-based retrieval).**
  - Given a face, find the five most similar faces.
- ❖ **Multidimensional range queries.**
  - $50 < \text{age} < 55$  AND  $80\text{K} < \text{sal} < 90\text{K}$
- ❖ **Partial match queries.**
  - look up all points (records) matching in one or more dimensions (attributes).



# *What's the difficulty?*

- ❖ An index based on spatial location needed.
  - One-dimensional indexes don't support multidimensional searching efficiently. (Why?)
  - Hash indexes only support point queries; want to support range queries as well.
  - Must support inserts and deletes gracefully.
- ❖ Ideally, want to support **non-point data** as well (**e.g., lines, shapes**).
- ❖ The R-tree meets these requirements, and variants are widely used today.



# *Geometric index structures*

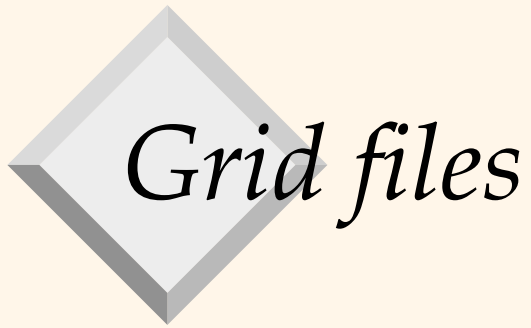
## **Hash-like indexes**

- ❖ Grid files
- ❖ Partitioned hash functions

## **Tree-based indexes**

- ❖ Multiple-key indexes
- ❖ kd-trees
- ❖ Quad trees
- ❖ R-trees

These indexes are *heuristic*, i.e., they are not (proved to be) worst case efficient.



# Grid files

- ❖ Simple idea:
  - Split each dimension into intervals.
  - Put each point into a bucket corresponding to the intervals it lies in.
- ❖ Overflow handling as in hash tables.
- ❖ Supports partial match queries, range queries, and nearest neighbor queries (how?)
- ❖ Bad case: All data along "diagonal" - need many grid lines (in internal memory) to avoid large buckets.



# *kd-trees*

- ❖ Short for “k-dimensional search tree”.
- ❖ Change from ordinary search trees:
  - Each node is associated with a dimension.
  - An internal node partitions the points of its subtree along its dimension.
  - Dimensions rotate down the tree: 1,2,..,k,1,2,..k,1,2,..
  - Supports partial match queries, range queries and nearest neighbor queries.
- ❖ Bad case: Many points with same value in some dimension makes it impossible to split points well along this dimension.

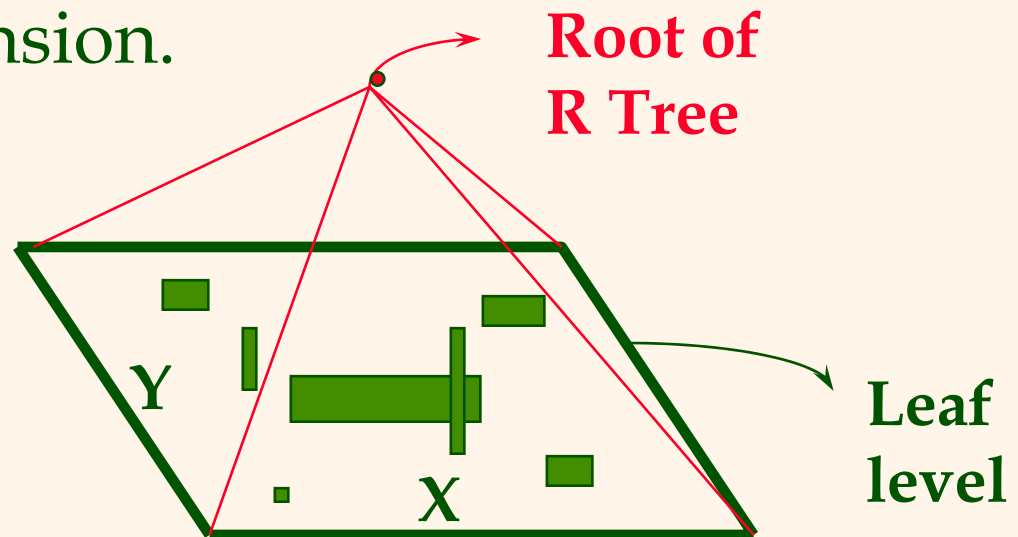


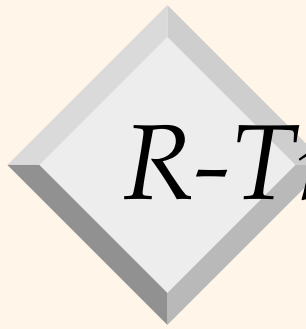
# *R Trees (Guttman 84)*

- ❖ “Region trees” – multidimensional indices
  - Any number of dimensions
  - Generally uses containment of bounding boxes, not full polygons, for efficiency
- ❖ Must deal with two issues vs. a B+ Tree:
  - Is there a complete ordering?
  - Can items overlap without being identical?
- ❖ Basic idea of the R Tree:
  - Create a tree based on containment of polygons/“rectangles” – child polygons are contained within parent polygons

# The R-Tree

- ❖ The R-tree is a tree-structured index that remains balanced on inserts and deletes.
- ❖ Each key stored in a leaf entry is intuitively a **box**, or collection of **intervals**, with one interval per dimension.
- ❖ Example in 2-D:



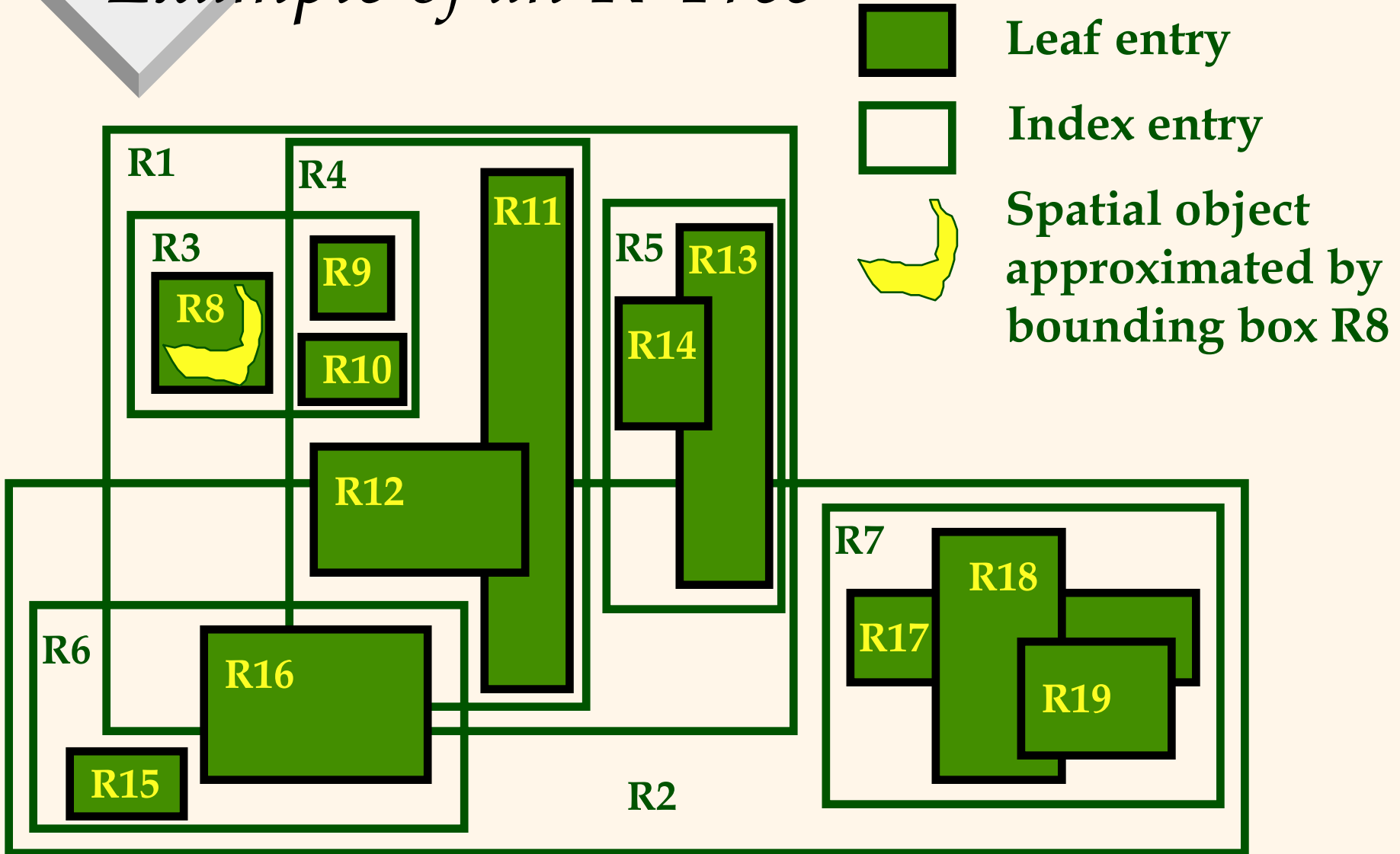


# *R-Tree Properties*

- ❖ Leaf entry =  $\langle n\text{-dimensional box, rid} \rangle$ 
  - This is Alternative (2), with *key value* being a box.
  - Box is the tightest bounding box for a data object.
- ❖ Non-leaf entry =  $\langle n\text{-dim box, ptr to child node} \rangle$ 
  - Box covers all boxes in child node (in fact, subtree).
- ❖ All leaves at same distance from root.
- ❖ Nodes can be kept 50% full (except root).
  - Can choose a parameter  $m$  that is  $\leq 50\%$ , and ensure that every node is at least  $m\%$  full.

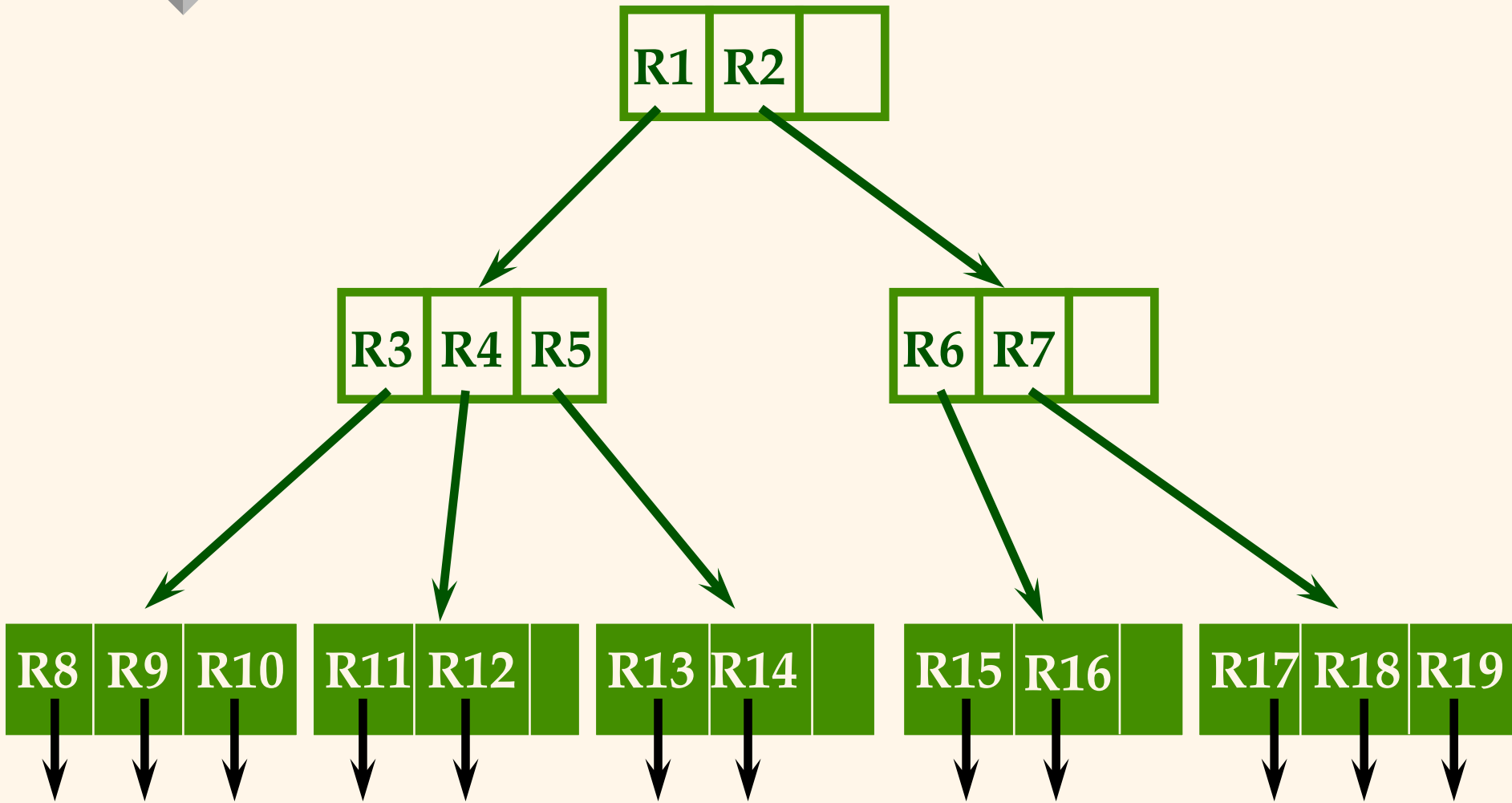


# Example of an R-Tree





# Example R-Tree (Contd.)





# Search for Objects Overlapping *Box Q*

Start at **root**.

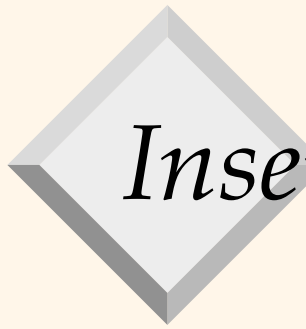
1. If current node is non-leaf, for each entry  $\langle E, ptr \rangle$ , if *box E* overlaps *Q*, search subtree identified by **ptr**.
2. If current node is leaf, for each entry  $\langle E, rid \rangle$ , if *E* overlaps *Q*, **rid** identifies an object that might overlap *Q*.

*Note: May have to search **several** subtrees at each node!  
(In contrast, a B-tree equality search goes to just one leaf.)*



# *Improving Search Using Constraints*

- ❖ It is convenient to **store boxes** in the R-tree as approximations of arbitrary regions, because boxes can be represented compactly.
- ❖ But why not use **convex polygons to approximate query regions** more accurately?
  - Will reduce overlap with nodes in tree, and reduce the number of nodes fetched by avoiding some branches altogether.
  - Cost of overlap test is higher than bounding box intersection, but it is a main-memory cost, and can actually be done quite efficiently. Generally a win.



## *Insert Entry* $\langle B, ptr \rangle$

- ❖ Start at root and go down to “best-fit” leaf L.
  - Go to **child whose box needs least enlargement** to cover B; resolve ties by going to smallest area child.
- ❖ If best-fit leaf L has space, insert entry and stop. Otherwise, split L into L1 and L2.
  - Adjust entry for L in its parent so that the box now covers (only) L1.
  - Add an entry (in the parent node of L) for L2. (This could cause the parent node to recursively split.)

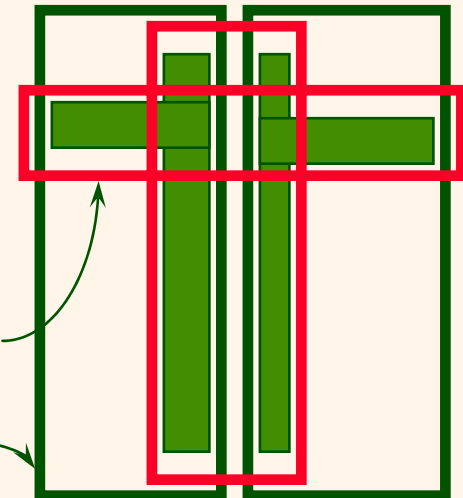
# Splitting a Node During Insertion

- ❖ The entries in node L plus the newly inserted entry must be distributed between L1 and L2.
- ❖ Goal is to reduce likelihood of both L1 and L2 being searched on subsequent queries.
- ❖ **Idea:** Redistribute so as to **minimize area** of L1 plus area of L2.

Exhaustive algorithm is too slow;  
quadratic and linear heuristics are  
described in the paper.

**GOOD SPLIT!**

**BAD!**





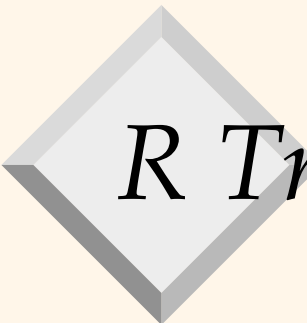
# *R-Tree Variants*

- ❖ The **R\* tree** uses the concept of **forced reinserts** to reduce overlap in tree nodes. When a node overflows, instead of splitting:
  - Remove some (say, 30% of the) entries and reinsert them into the tree.
  - Could result in all reinserted entries fitting on some existing pages, avoiding a split.
- ❖ R\* trees also use a different heuristic, minimizing **box perimeters** rather than **box areas** during insertion.
- ❖ Another variant, the **R+ tree**, avoids overlap by inserting an object into multiple leaves if necessary.
  - Searches now take a single path to a leaf, at cost of redundancy.




## *Comments on R-Trees*

- ❖ Deletion consists of searching for the entry to be deleted, removing it, and if the node becomes under-full, deleting the node and then re-inserting the remaining entries.
- ❖ Overall, works quite well for 2 and 3 D datasets. Several variants (notably, R<sup>+</sup> and R<sup>\*</sup> trees) have been proposed; widely used.
- ❖ Can improve search performance by using a convex polygon to approximate query shape (instead of a bounding box) and testing for polygon-box intersection.



# *R Trees in Practice*

- ❖ One of many (dozens of!) multidimensional index structures
  - Perhaps the most popular, though not as ubiquitous as the B+ Tree
  - Implemented in a few DBMSs, such as Oracle, Illustra (Stonebraker startup, acquired by Informix, then IBM) and MySQL



# *Generalizing B+ Trees and R Trees: GiST*

- ❖ The Generalized Search Tree (GiST) abstracts the “tree” nature of a class of indexes including B+ trees and R-tree variants.
  - Striking similarities in insert/delete/search and even concurrency control algorithms make it possible to provide “templates” for these algorithms that can be customized to obtain the many different tree index structures.
  - B+ trees are so important (and simple enough to allow further specialization) that they are implemented specially in all DBMSs.
  - GiST provides an alternative for implementing other tree indexes in an ORDBS.



# *Indexing High-Dimensional Data*

- ❖ Typically, high-dimensional datasets are collections of points, not regions.
  - E.g., Feature vectors in multimedia applications.
  - Very sparse
- ❖ Nearest neighbor queries are common.
  - R-tree becomes worse than sequential scan for most datasets with more than a dozen dimensions.
- ❖ As dimensionality increases **contrast** (ratio of distances between nearest and farthest points) usually decreases; “nearest neighbor” is not meaningful.
  - In any given data set, advisable to empirically test contrast.



# Summary

- ❖ Spatial data management has many applications, including GIS, CAD/CAM, multimedia indexing.
  - Point and region data
  - Overlap/containment and nearest-neighbor queries
- ❖ Many approaches to indexing spatial data
  - R-tree approach is widely used in GIS systems
  - Other approaches include Grid Files, Quad trees, and techniques based on “space-filling” curves.
  - For high-dimensional datasets, unless data has good “contrast”, nearest-neighbor may not be well-separated