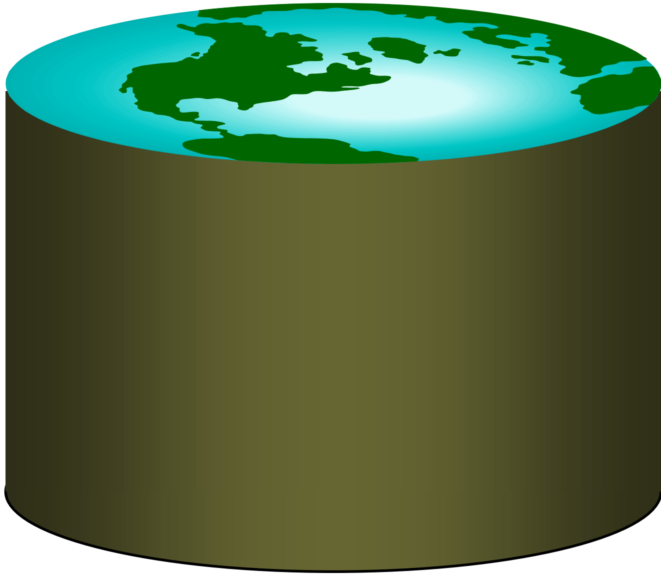
Generalized Search Trees and Spatial Indexing

(Ramakrishnan 28.1, 28.2, 28.6)



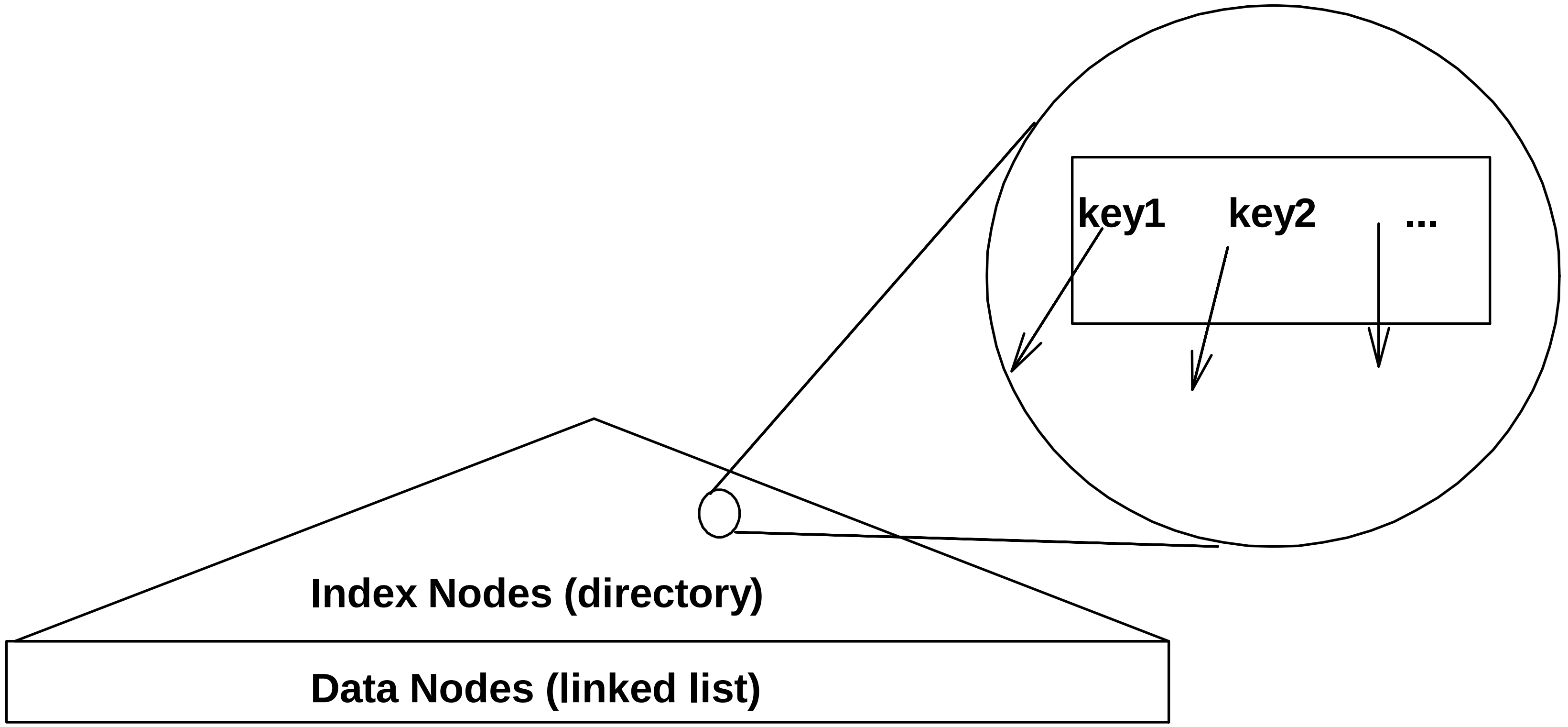
Other Search Trees

- **Question: Can B+-trees handle more complicated searches?**
  - A typical example: *"gpa > 3.7 and age < 18"*
  - Same thing: "all restaurants in downtown Berkeley"
  - Even fancier: "all pictures resembling *</tmp/sunset.gif>*"
  - (Easy: "all pictures identical to *</tmp/sunset.gif>*")
- **B+-trees exploit data order to do range search**
  - 1-d range search is not always what you want

Search Trees in General

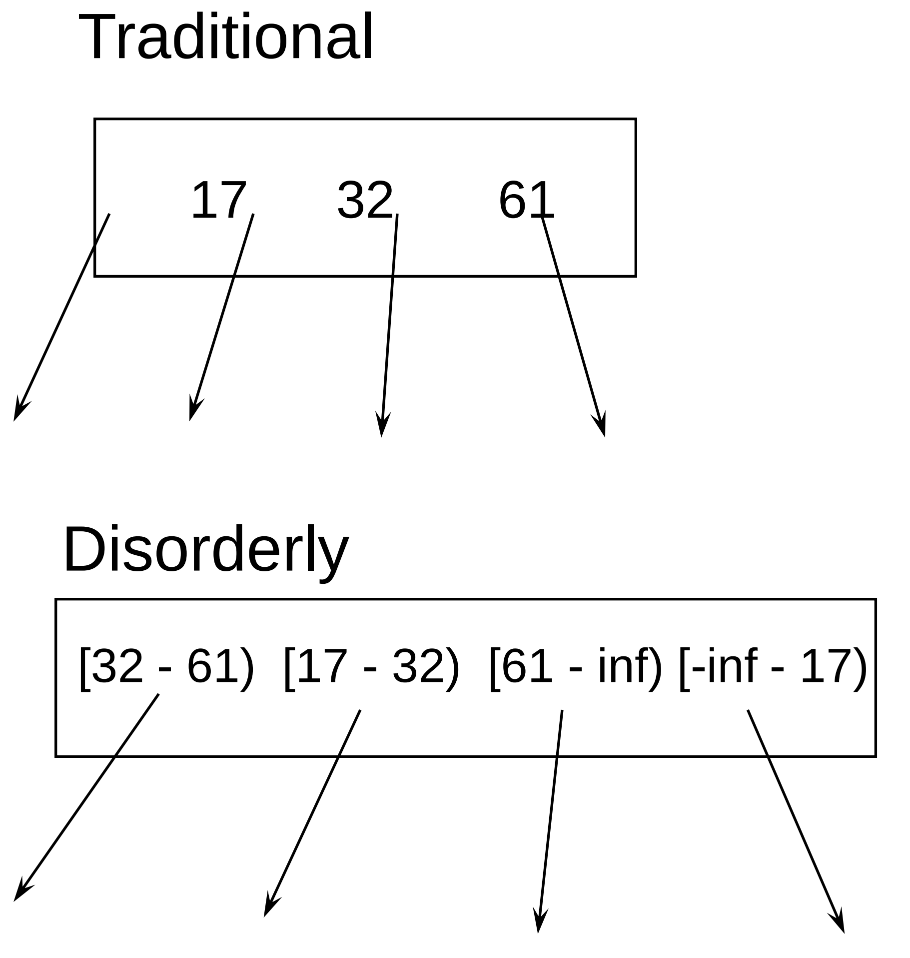
- **Lots of trees invented for multidimensional data**
  - e.g. R-trees, R*-trees, hB-trees, UB-trees, X-trees, etc. etc.
- **New tree indexes for other kinds of data**
  - Image/video search, timeseries matching, DNA sequence matching, etc.
- **Many are "unordered" B+-trees, fancy keys**
  - In some ways, B+-tree is just a "special case" of these trees.

Search Trees from 30,000 feet

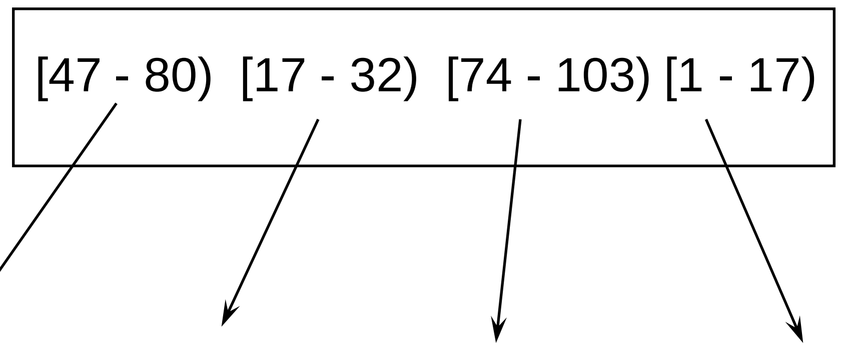


A “Disorderly” B+-tree

- **How to search a disorderly B+-tree?**
  - Equality search is identical to traditional!
  - Range search = traversing multiple paths
    - follow all pointers where key range overlaps query range.



Another Disorderly B+-Tree



- **Insert 41???**
  - keys need not cover all possible values
- **Search for 77??**
  - keys may “overlap”

Generalized Search Tree (GiST)

- **A disorderly B+-tree, with *user-defined* keys**
  - tree doesn't interpret keys.
  - "user" implements keys as an OO class, with methods that guide search, insert, delete, split, etc.
- **Structure: balanced tree of (*p*, ptr) pairs**
  - *p* is an index key or "predicate"
  - *p* holds for all data records below ptr
  - need *n* keys for *n* pointers (unlike B+-tree)

User-provided Key Methods

- **Search:**
  - **Consistent**(E,q): $E.p \wedge q$? (no/maybe)
- **Generating new keys after splits:**
  - **Union**(P): new key that holds for all tuples in P
- **Data organization:**
  - **Penalty**(E_1,E_2):
    "badness" of inserting E_2 in subtree E_1
  - **PickSplit**(P): split P into two groups of entries

Search

- **General technique:**
  - Depth-First Search where **Consistent** is TRUE
- **Incremental algorithm:**
  - Maintain a search stack of <page_id, offset> pairs from root to current data entry
  - Each "get next" call moves the offset to right
    - When nothing left on page, pop stack, move right in parent (or recurse) and go down to leaf.

Insert

- **descend tree along least increase in Penalty**
- **if there's room at leaf, insert there**
- **else split according to PickSplit**
  - do B-tree-style recursive splitting
- **propagate changes (recursively) using Union**
  - make sure all ancestor keys are consistent with inserted item

Q: what happened to B-tree "copy up" and "push up" logic?

Let's revisit with example of R-tree in a few slides

Delete

- **find the entry via Search, and delete it**
- **propagate changes (recursively) using Union**
- **on underflow:**
  - reinsert stuff on page and delete page
  - why not borrow/merge a la B+-trees?

RD-tree: GiST for Sets

{CS1, CS11, Music1, Music2, Math221, Math22, Math223}

{CS1, Bus101, Bus102, Bus103, Ec121, Ec122, Ec123}

{CS1, CS786, CS888, Math221, Music1, Music788}

{Bus101, Bus102, Bus103, CS1}
{Bus101, Ec121, Ec122, Ec123}
{CS1, Bus101, Ec121}

{CS1, CS11, Math221}
{Music1, Music2, CS1}
{CS1, Math221, Math22, Math223}

{Music1, CS1, Math221}
{Music788, CS888, CS786}
{CS1}

RD-trees

- **Logically, keys represent minimal supersets**
- **Queries: Contains, Intersects, Equals…**
- **Consistent(*E,q*):**
  - Varies slightly on query type … e.g. $E.p \cap q \mathrel{!=} \emptyset$
- **Union(*P*): set-union of keys**
- **Penalty(*E,F*): $|E.p \cup F.p| - |E.p|$**
- **PickSplit(*P*): many possible algorithms**
  - One goal: minimize sum of cardinalities of 2 pages

Used in Postgres as an alternative to inverted indexes

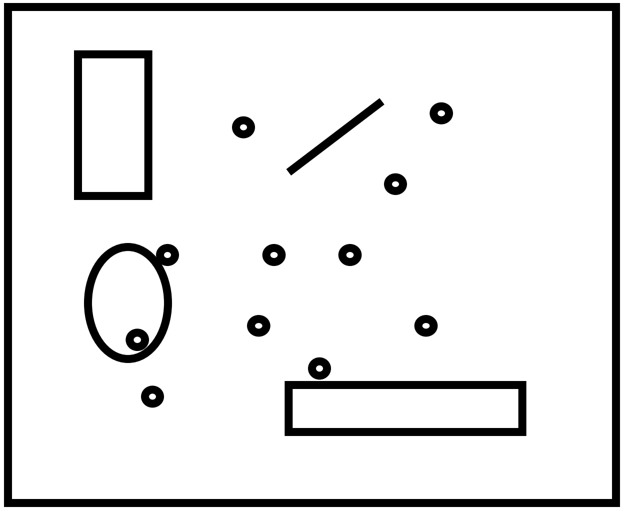
Note: need key compression here (like postings lists)

R-tree: A GiST over 2-d data

- **(Invented ~10 years before GiST or RD-Tree)**
  - Also at Berkeley ;-)
- **Logically, keys represent "bounding boxes"**
- **Queries: Contains, Overlaps, Equals … *bbox***
- **Consistent(*E,q*):**
  - Varies slightly on query type: e.g. *E.p* overlaps *q*?
- **Union(*P*): bounding box of all entries**
- **Penalty(*E,F*): size(Union({*E,F*})) - size(*E*)**
- **PickSplit(*P*):**
  - goal: minimize sum of areas of the 2 pages

R-trees, Slowly. Problem:

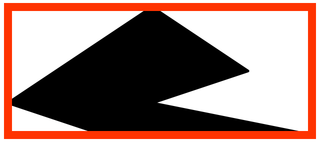
- **Given a collection of geometric objects (points, lines, polygons, ...)**
- **organize them on disk, to answer spatial queries (range, near neighbors, etc)**



R-tree slides from G. Kollios, BU

R-trees

- **Main idea: a 2-D B-tree with overlapping keys!**
  - => guaranteed 50% utilization
  - => easier insertion/split algorithms.
  - (only deal with Minimum Bounding Rectangles - **MBR**s)

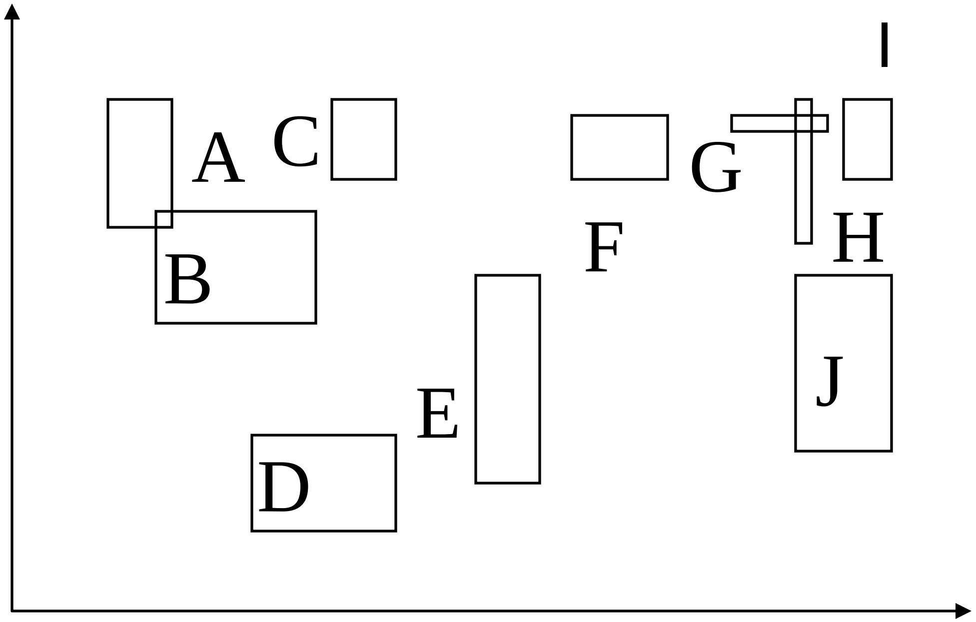


R-tree Structure

- **A multi-way tree of disk blocks**
- **Index nodes and data (leaf) nodes**
- **All leaf nodes appear on the same level**
- **Every node contains between m and M entries**
- **The root node has at least 2 entries (children)**

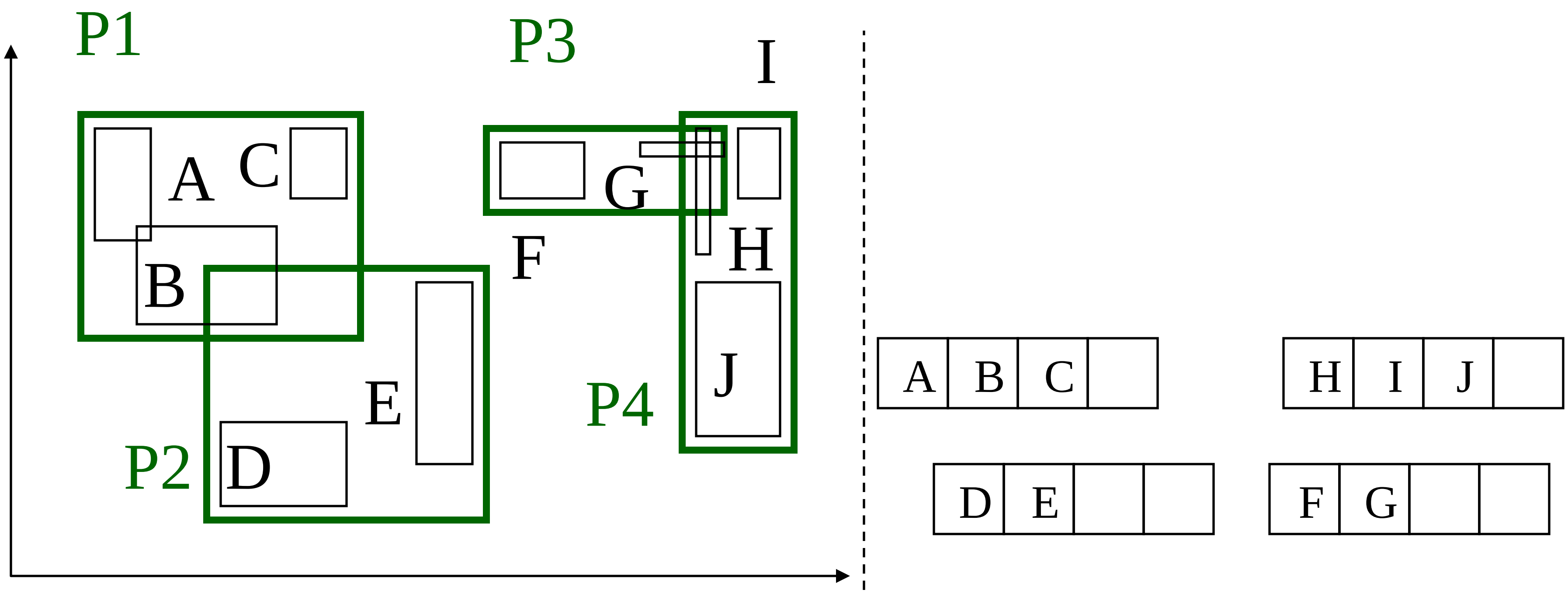
Example

- **eg., w/ fanout 4: group nearby rectangles to parent MBRs; each group -> disk page**



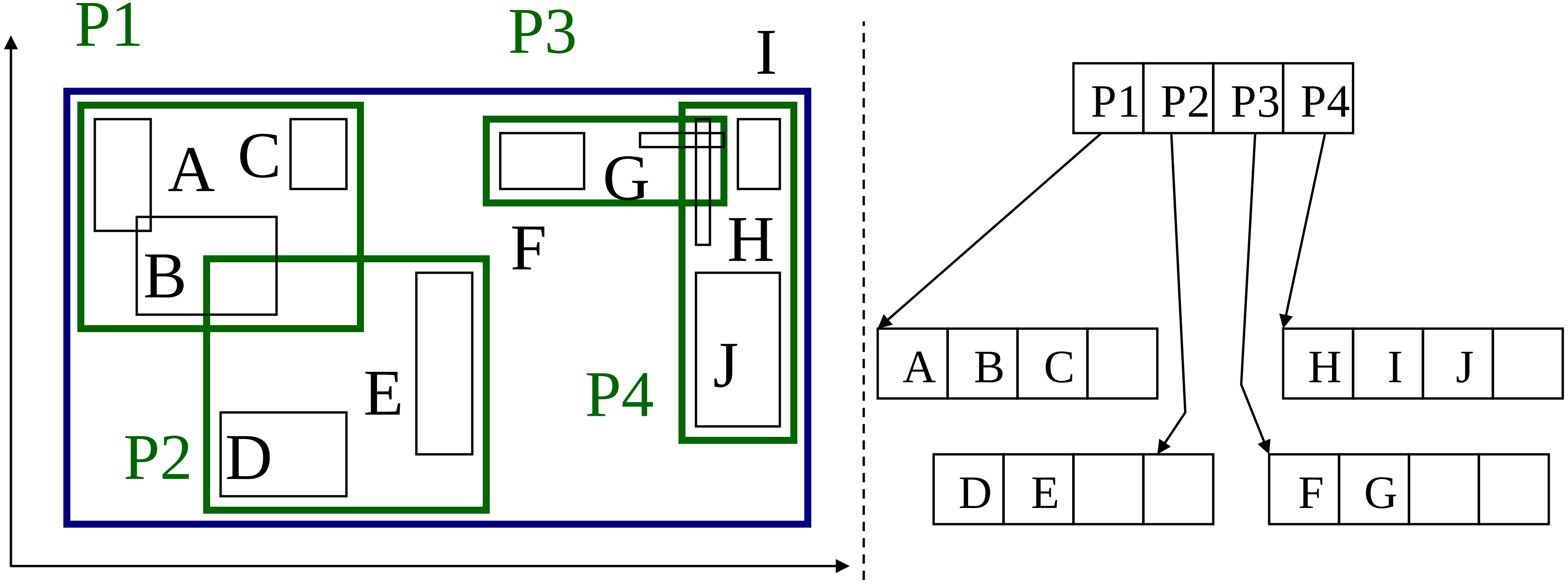
Example

- **F=4**



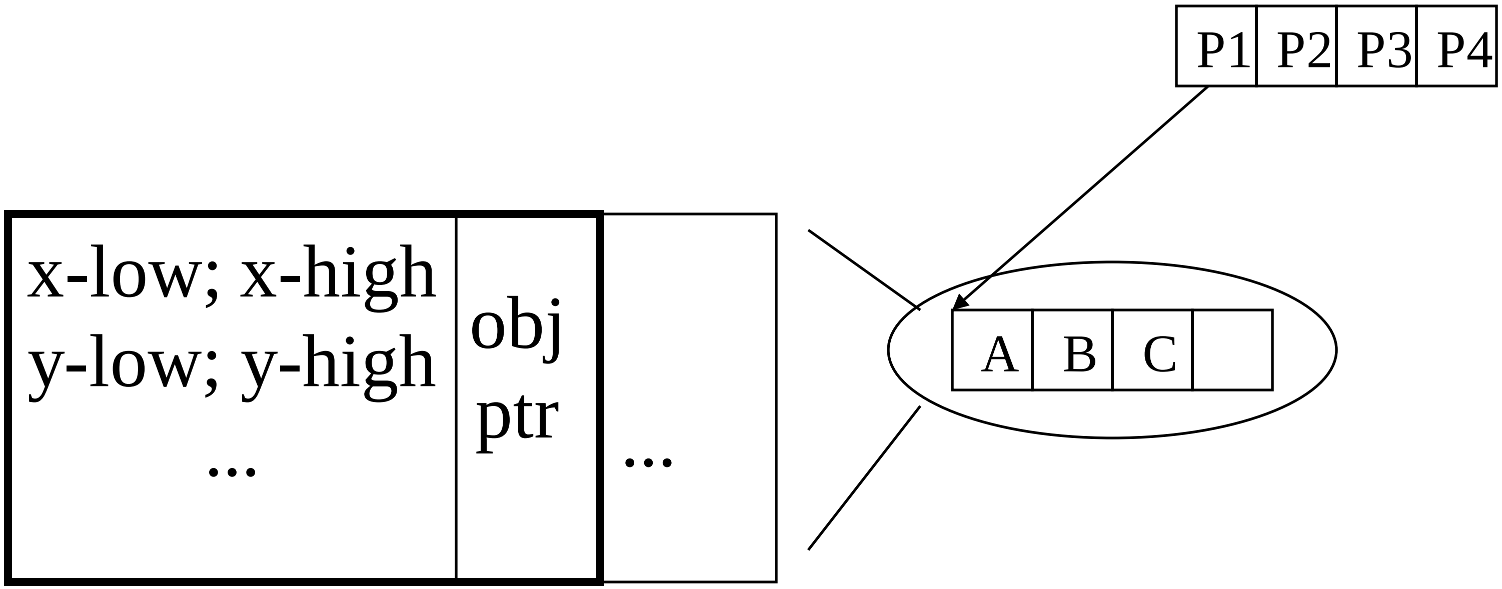
Example

- **F=4**



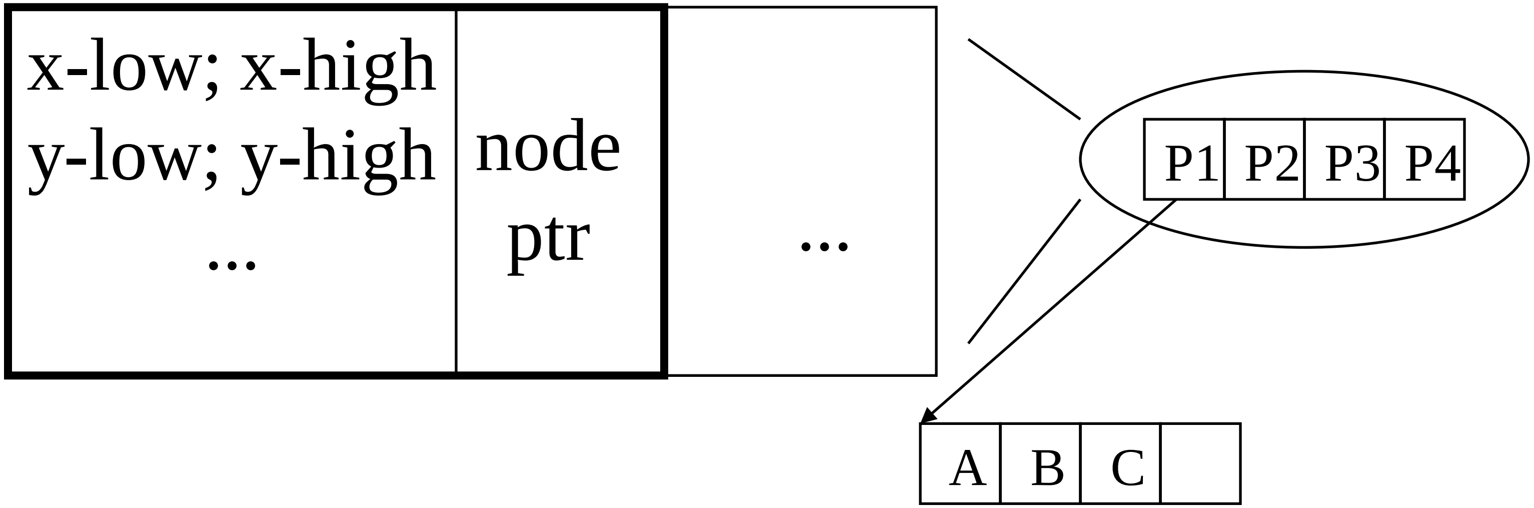
R-trees - format of nodes

- **{(MBR; obj_ptr)} for leaf nodes**

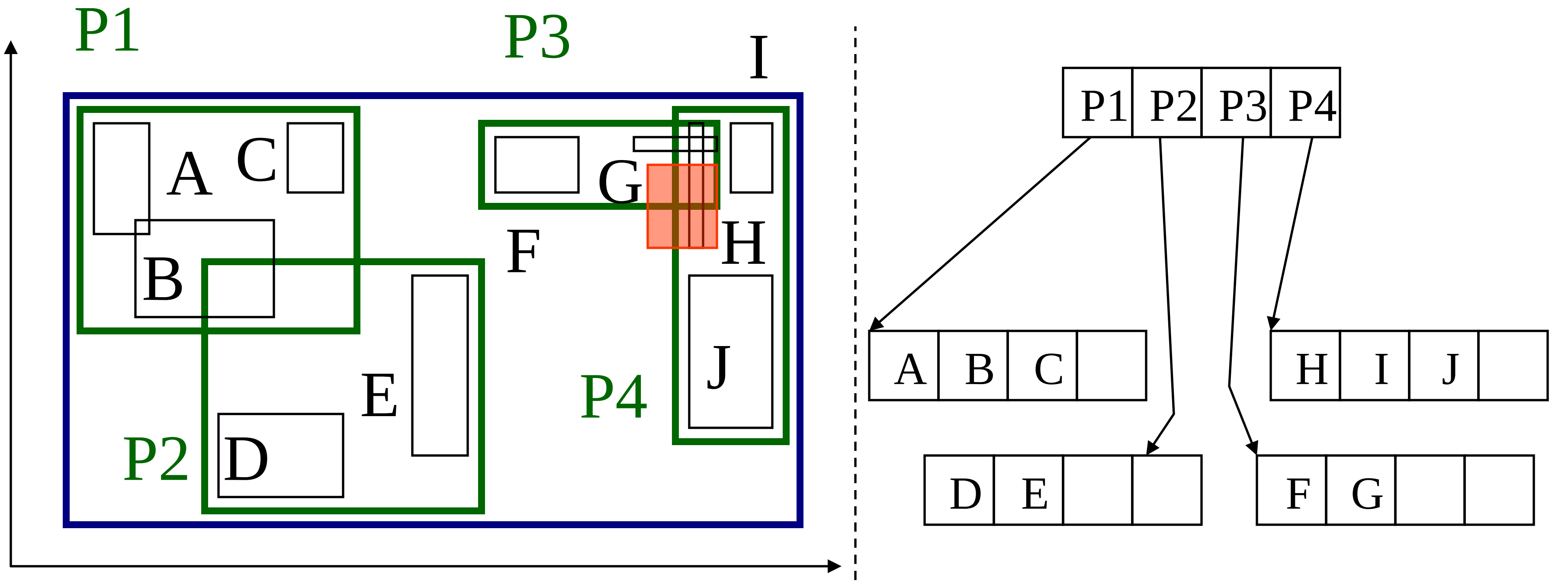


R-trees - format of nodes

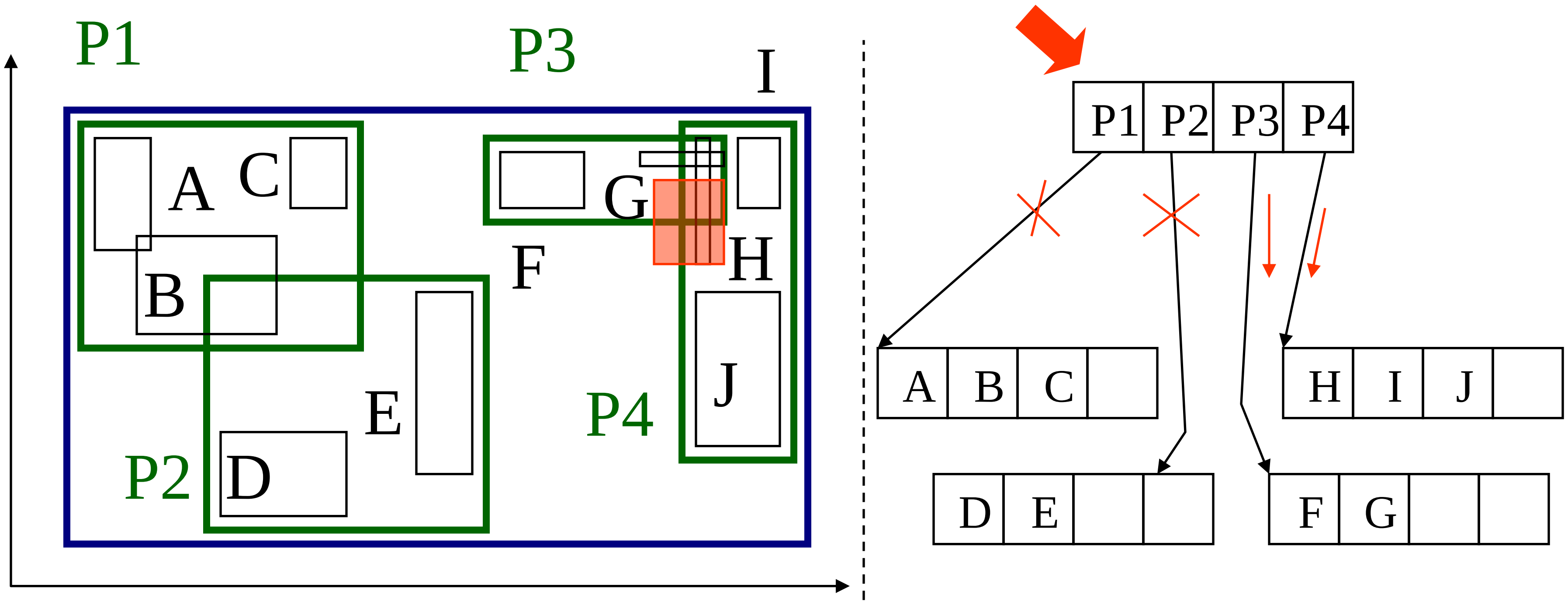
- **{(MBR; node_ptr)} for non-leaf nodes**



R-trees:Search



R-trees:Search

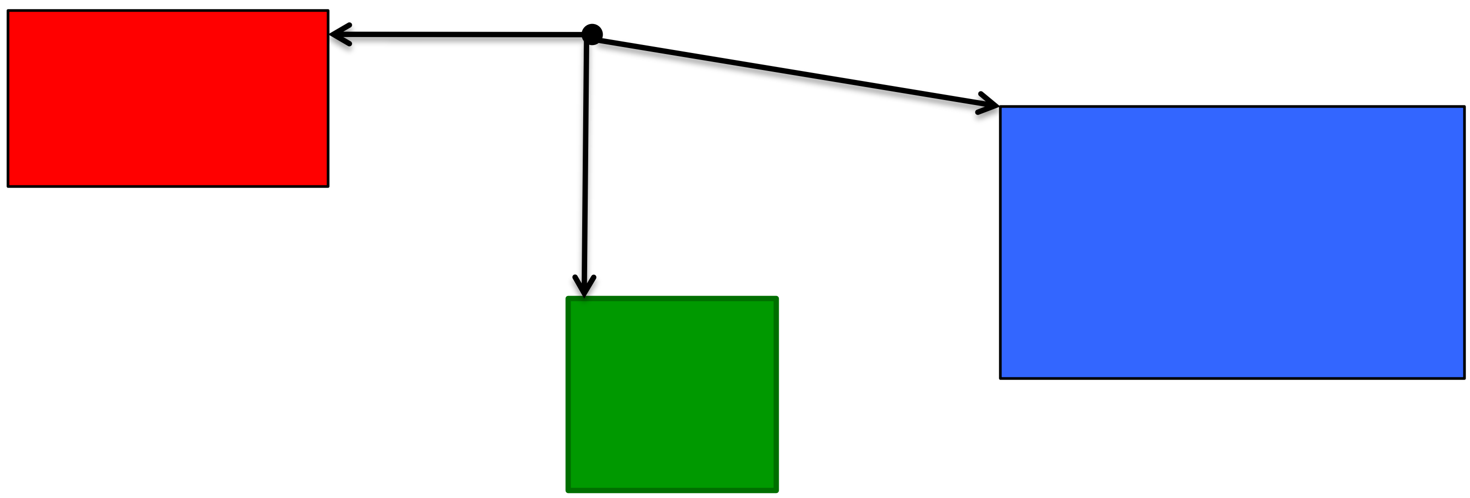


R-trees:Search

- **Main points:**
  - every parent node completely covers its 'children'
  - a child MBR may be covered by more than one parent - it is stored under ONLY ONE of them. (ie., no need for dup. elim.)
  - a point query may follow multiple branches.
  - everything works for **any(?)** dimensionality

R-trees: Near Neighbor Search

- **Rather than a stack (for depth-first), maintain a priority queue of nodes to visit**
- **Upon visiting a page, load all the children to be traversed into the priority queue**
  - Priority = MinDistance

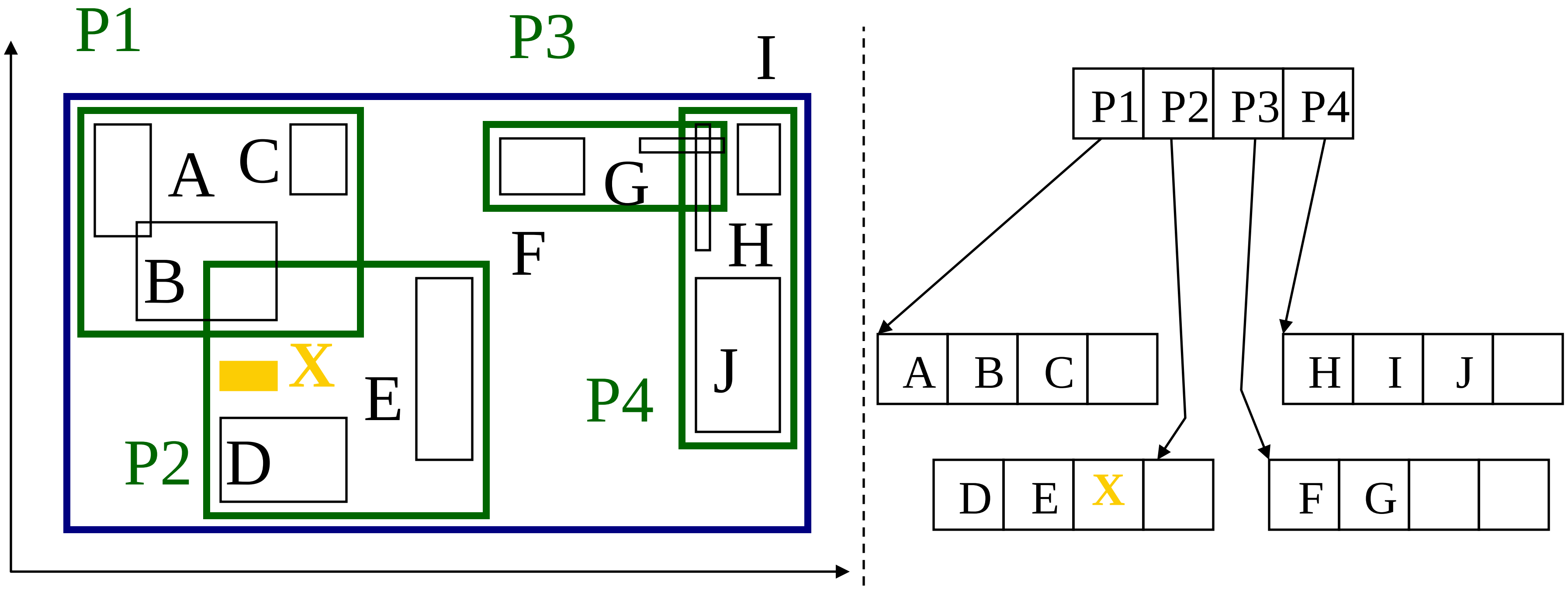


R-tree: Incremental Near Neighbors

```
PQ.insert(root)
while (!PQ.empty)
  next = PQ.deletemin
  if (next is a data entry) return (next)
  else // next is a page in the index
    for each item i on next
      PQ.insert(i)
    end for
  end if
end while
```

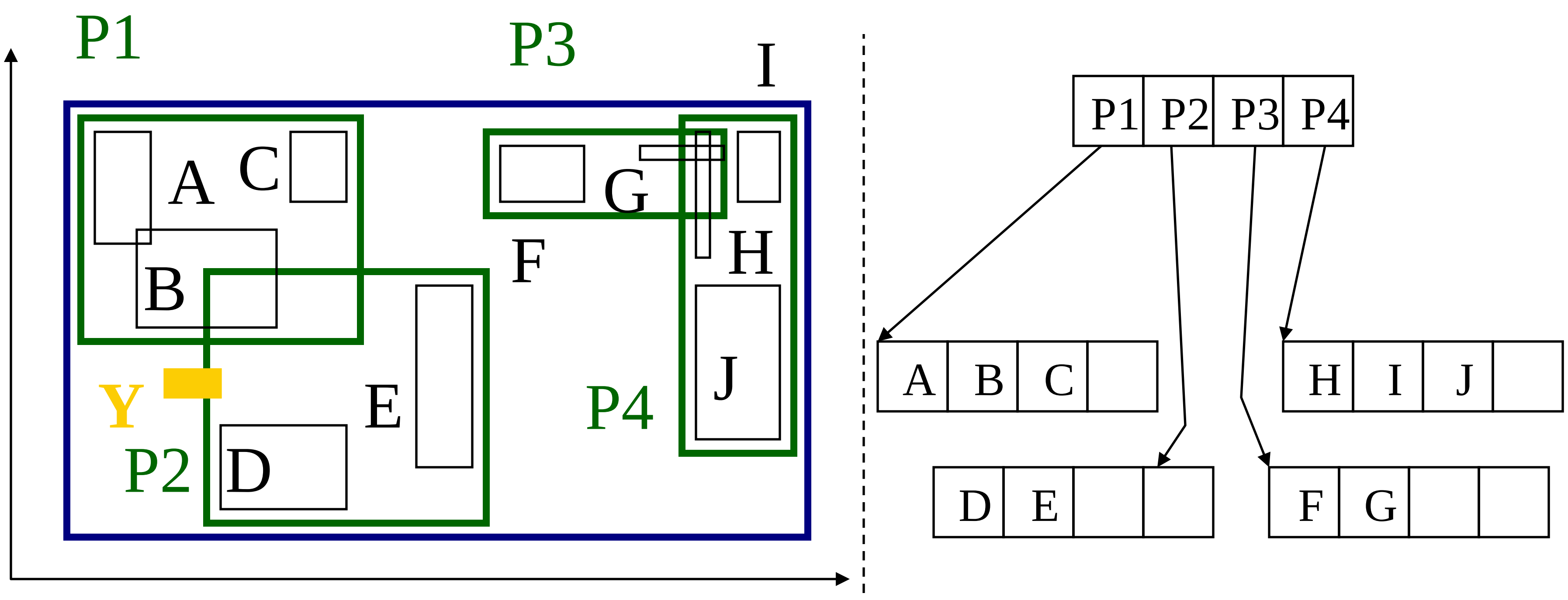
R-trees:Insertion

Insert X



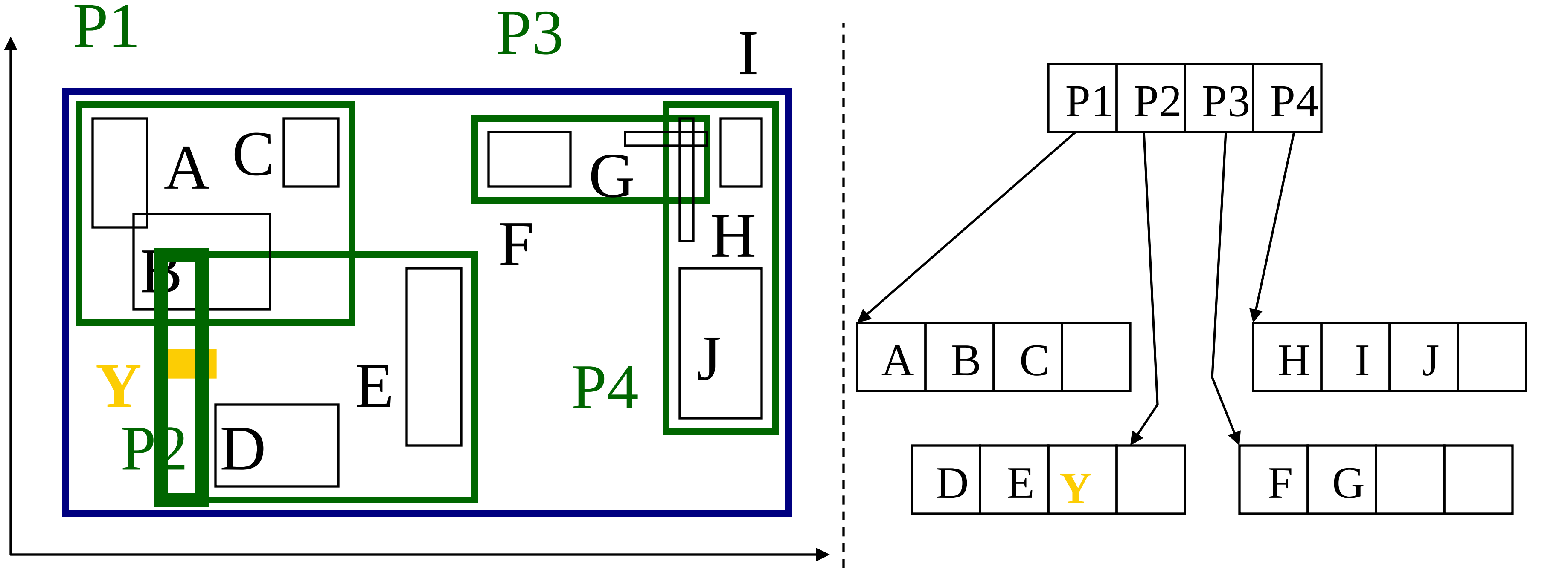
R-trees:Insertion

Insert Y



R-trees:Insertion

- **Extend the parent MBR**

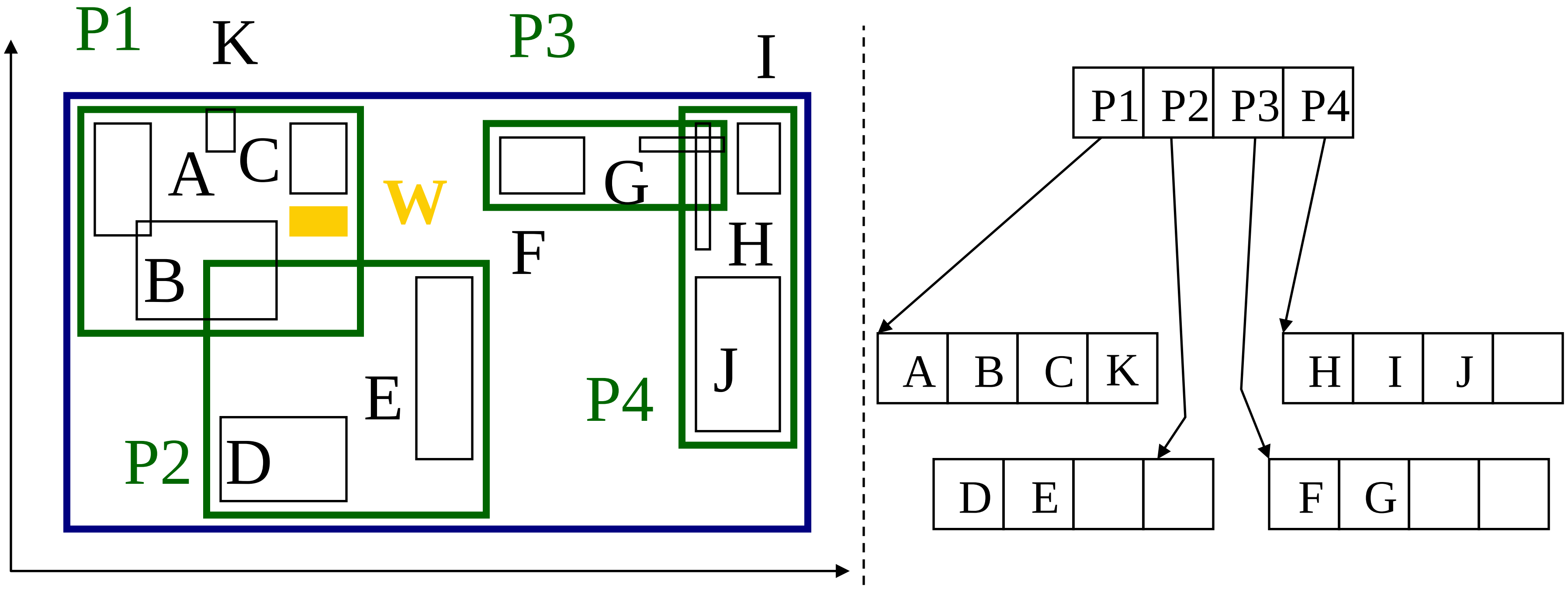


R-trees:Insertion

- **How to find the next node to insert the new object?**
  - Penalty metric: At each level, find the entry that needs the least enlargement to include Y. Resolve ties using the area (smallest)
- **Other methods have been proposed**
  - E.g. perhaps useful to minimize the perimeter of MBRs too?

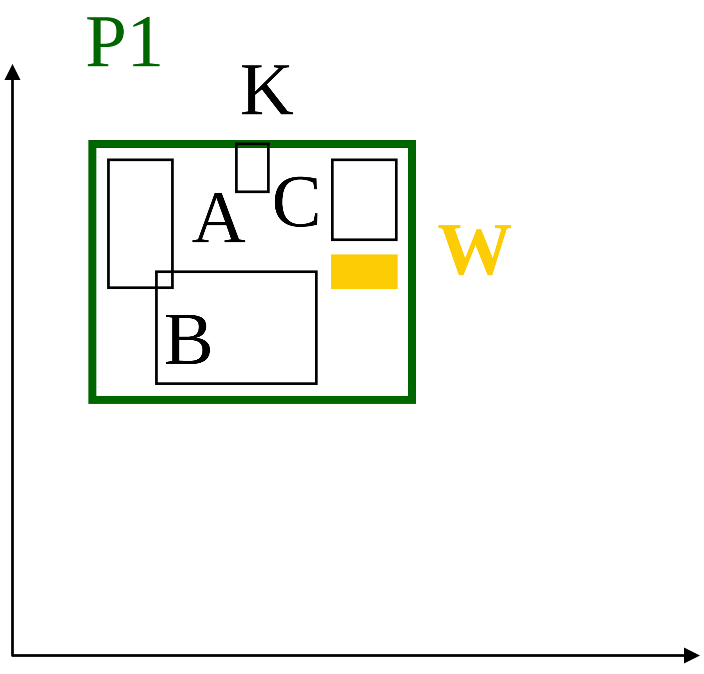
R-trees:Insertion

- **If node is full then <u>Split :</u> ex. Insert w**



R-trees:PickSplit

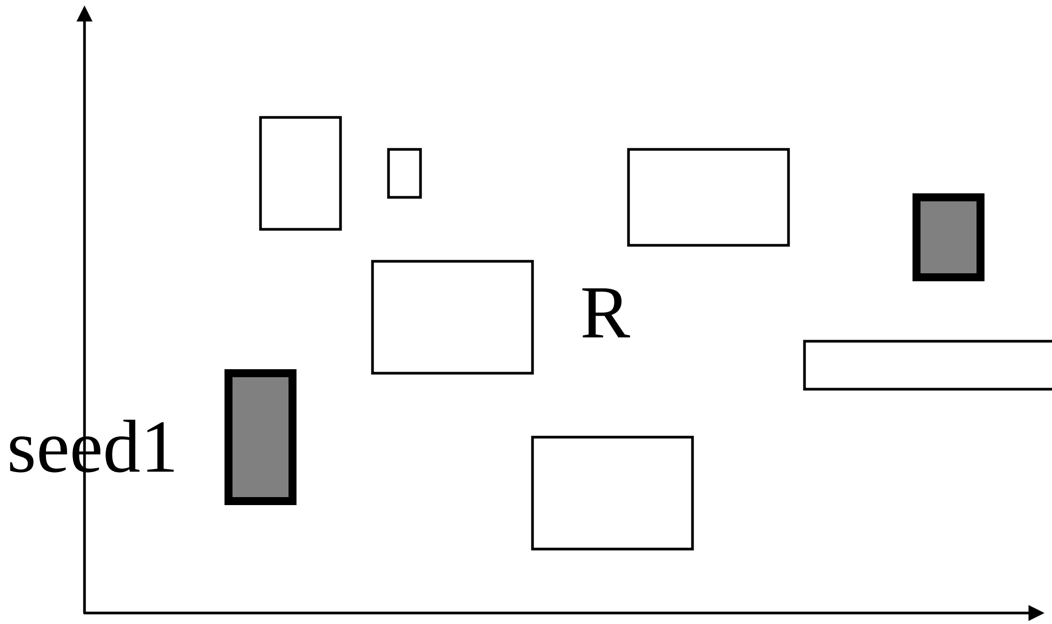
- **Split node P1: partition the MBRs into two groups.**



- Naïve: exhaustive
  - cost?
- quadratic split
- linear split

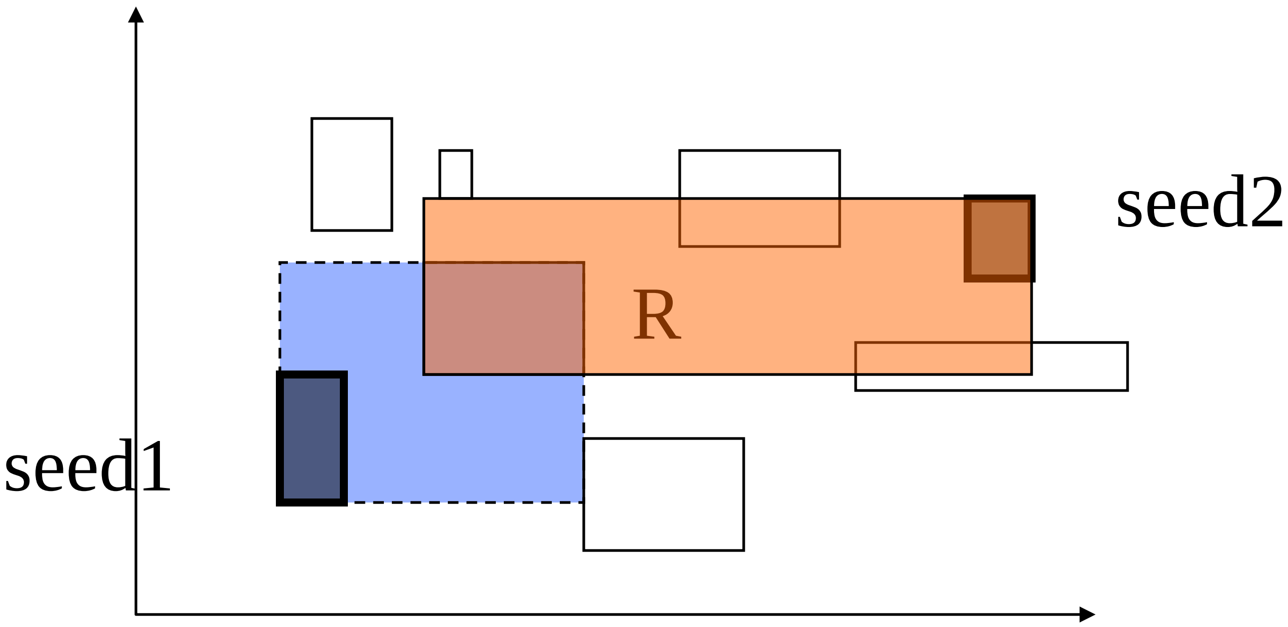
R-tree PickSplit: idea

- pick two rectangles as 'seeds';
- assign each rectangle 'R' to the 'closest' 'seed'



R-trees:Split

- **pick two rectangles as 'seeds';**
- **assign each rectangle 'R' to the 'closest' 'seed':**
- **'closest': the smallest increase in area**



R-trees:Split

- **How to pick Seeds:**
  - Linear: for each dimension
    - Find the highest low point, lowest high point; difference is separation
    - Normalize: divide separation by extent of that dimension
    - Across all dimensions, choose pair with biggest normalized separation
  - Quadratic: For each pair E1 and E2, calculate:
    - the rectangle J=MBR(E1, E2)
    - d = area(J) – (area(E1) + area(E2)) (inefficiency)
    - Choose the pair with the largest d

R-trees: Variations

- **There are many variations on R-trees**
  - Some change the key type (e.g. bounding spheres, “holey” bounding boxes)
  - Some fiddle with picksplit
  - Some complicate the structure (i.e. not exactly GiST)
  - And more…
- **Why so many?**
  - What’s wrong with R-trees
  - How good are R-trees anyway?

GiST Performance

- **B+-trees have O(log *n*) performance**
- **R-trees, RD-trees have no such guarantee**
  - search may have to traverse multiple paths
  - worst-case O(2*n*) to traverse entire tree
  - aggravated by random I/O
- **SO: when does it pay to build/use/invent an index? "Indexability"**
- **Basic questions:**
  1. Can data that's co-retrieved be put together in leaf pages?
  2. Can an efficient "directory" be built on top?
- **Often, if (1) is possible, (2) rests on the key "shape" being accurate**

The Gist of the GiST

- **Boil search trees down to their essence**
  - this is the “right” way to think about tree indexes
  - details of B+-trees, etc. are important, but not fundamental
  - the main idea is a hierarchy of clusters & labels, which grows by splitting bottom-up.
- **Unify B+-tree, R-tree, etc. in one ADT.**
  - code reuse!
- **Extensible in terms of data *and queries*.**
- **Raises nice theoretical questions of indexability.**

More on GiST

- **Implemented in PostgreSQL**
  - Including high-concurrency and recovery
  - PostgreSQL include GiST extensions for R-trees over boxes/polygons/circles
    - Basis of the PostGIS Geographic Info System
  - Also includes RD-trees for text search
    - Recommended for indexing transactional text data
    - Use inverted (“GIN”) indexes for sloppy text data
- **Was implemented in Informix**
  - Purchased by IBM
- **More? http://gist.cs.berkeley.edu**