R-/R*-tree

Advanced Indexing Techniques & Flexible Query Answering in DBs

Author: Do Quang Tri (09070471)
Do Quang Huy (09070443)

Outline

- R-Tree
- R* Tree

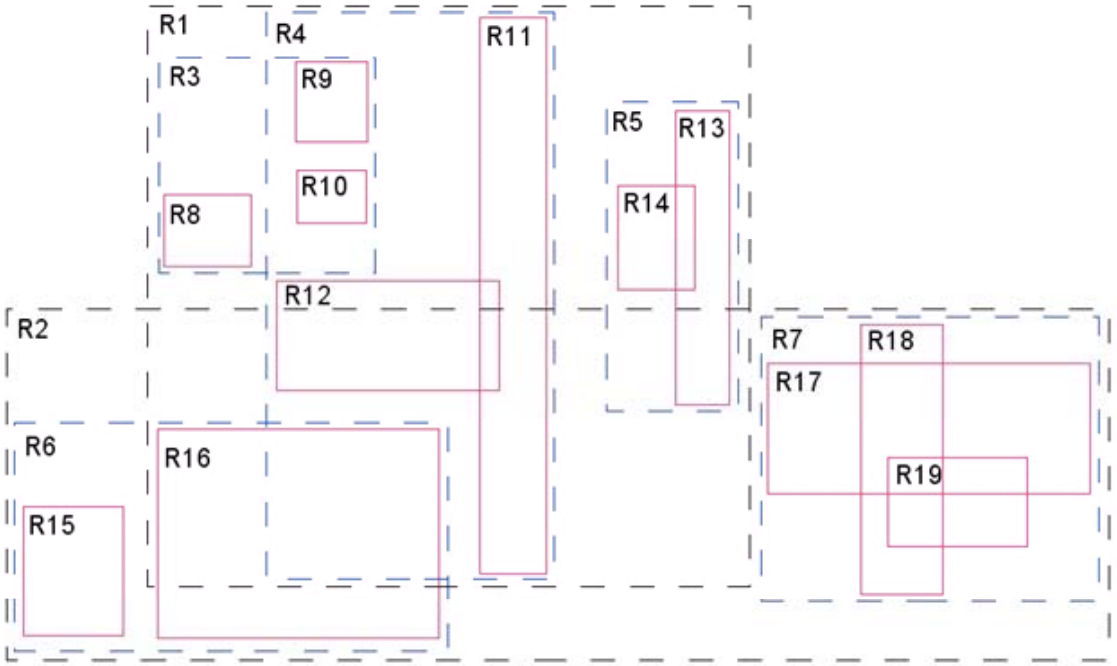
R-tree

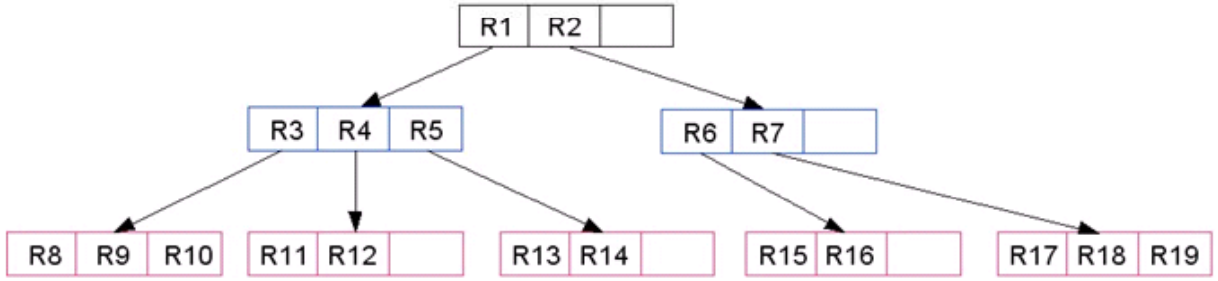
- A height-balanced tree like B-tree.
- Leaf-node entry: (I, tuple-identifier)
  - $I = (I_0, I_1, \ldots, I_{n-1})$: n-dimension rectangle bounding the spatial object
  - tuple-identifier: refers to a tuple in database
- Non-leaf node entry: (I, child-pointer)
  - child-pointer: is the address of a lower node
  - I: covers all rectangles in the lower's node entry
- Let M be the maximum number of entries in a node, and m<= M/2 be the minimum number of entries in a node

R-tree's properties

- Every node contains between *m* and *M* entries unless it is the root.
- The root has at least two children unless it is a leaf.
- All leaves appear on the same level.
- For each index record(I, tuple-identifier) in a leaf node, I is the smallest rectangle that spatially contains the n-dimensional data object
- For each entry(I, child-pointer) in a non-leaf node, I is the smallest rectangle that spatially contains the rectangle in the child nodes.

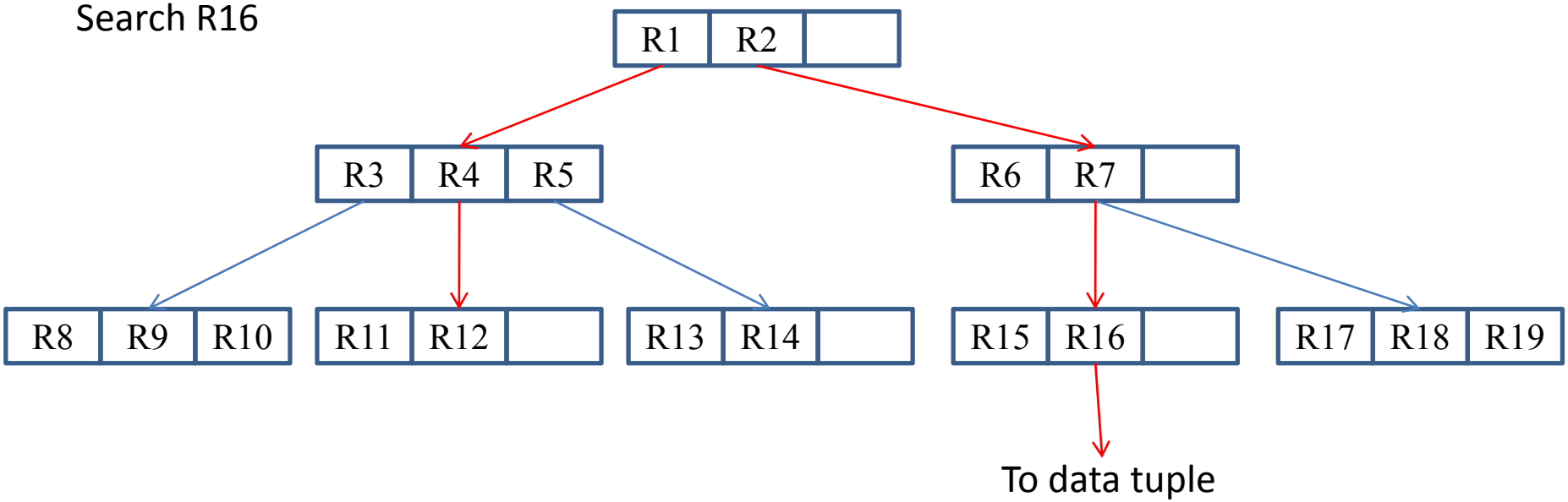
Example


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


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

Searching

Search R16

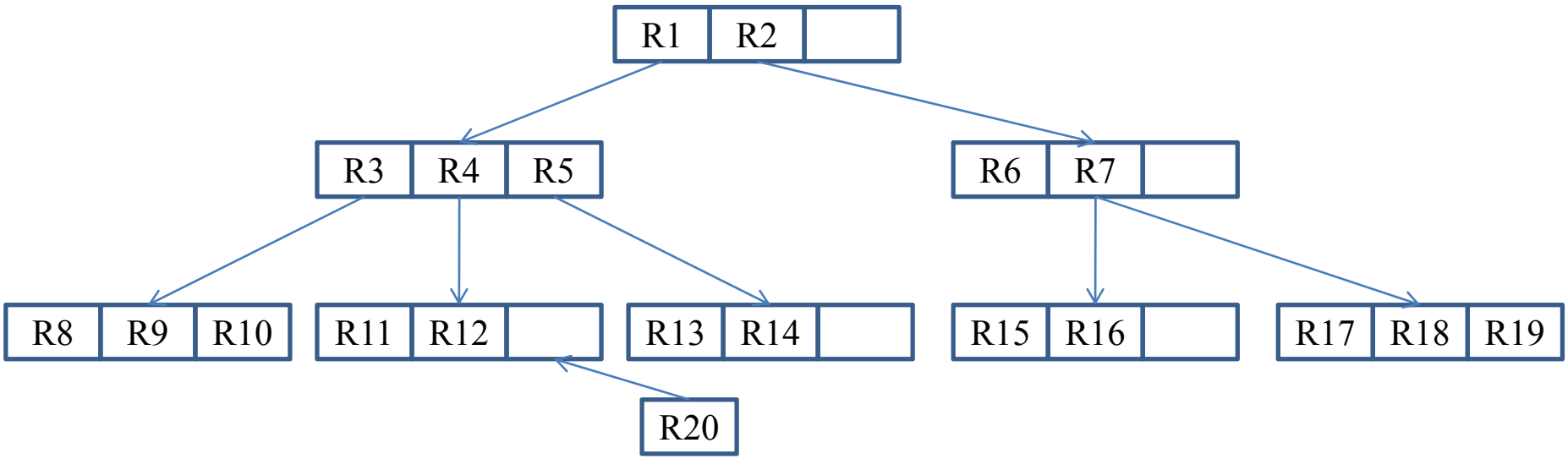


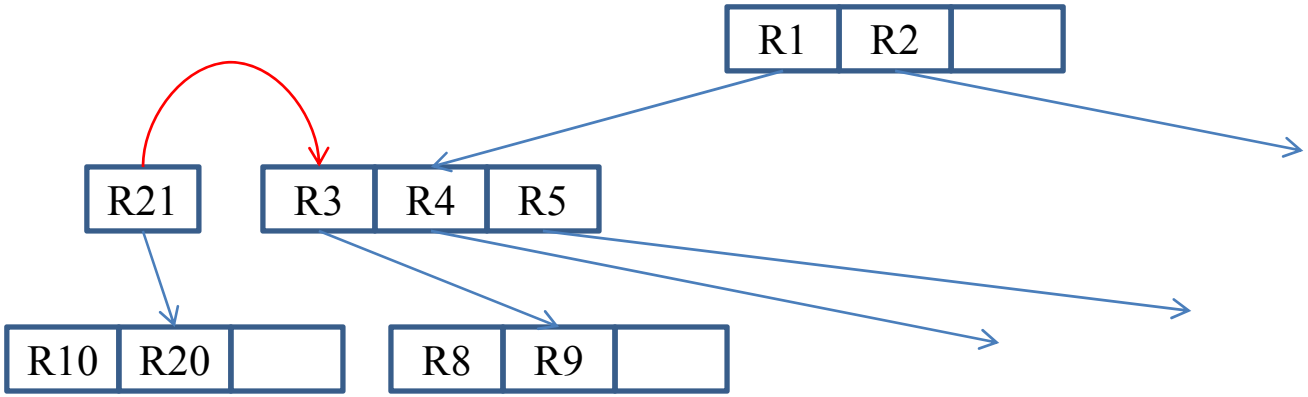
Searching

- EI: the rectangle part of an index entry
- EP: the tuple-identifier or child-pointer part
- Algorithm **Search**: Given an R-tree which T is the root and search rectangle S
  1. If T is not a leaf, check each entry E to determine whether EI overlap S. For all overlapping entries, invoke **Search** on the tree whose root node is pointed to by EP
  2. If T is a leaf, check all entries E to determine whether Ei overlap S. If so, E is qualifying record

Insertion

Insert R20





Insertion

- Algorithm **Insert**: Insert a new index entry to R-tree
  1. Invoke **ChooseLeaf** to select a leaf node L in which to place E.
  2. If L have room for another entry, insert E. Otherwise invoke **SplitNode** to obtain L and LL containing E and all the old entries of L.
  3. Invoke **AdjustTree** on L, also passing LL if a split was performed.
  4. If node split propagation caused the root to split, creat a new root whose children are the two resulting nodes.

Insertion

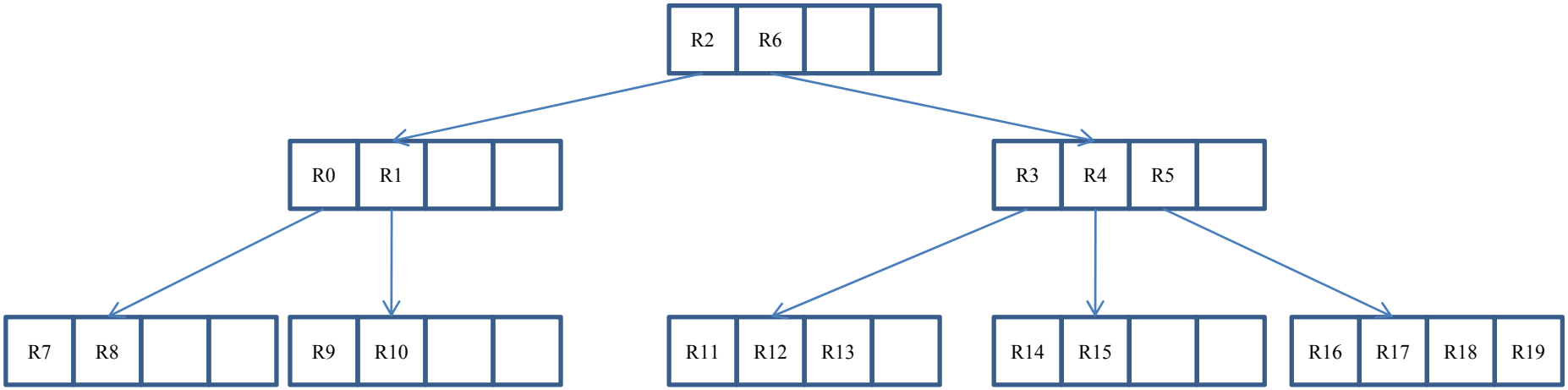
- Algorithm **ChooseLeaf** : select a leaf node in which to place a new index entry E.
  1. Set N to be the root node.
  2. If N is a leaf, return N.
  3. If N is not a leaf, let F is the entry in N whose rectangle Fi needs least enlargement to include Ei. Resolve ties by choosing the entry with the rectangle of smallest area.
  4. Set N to be the child node pointed to by Fp and repeat from 2.

Insertion

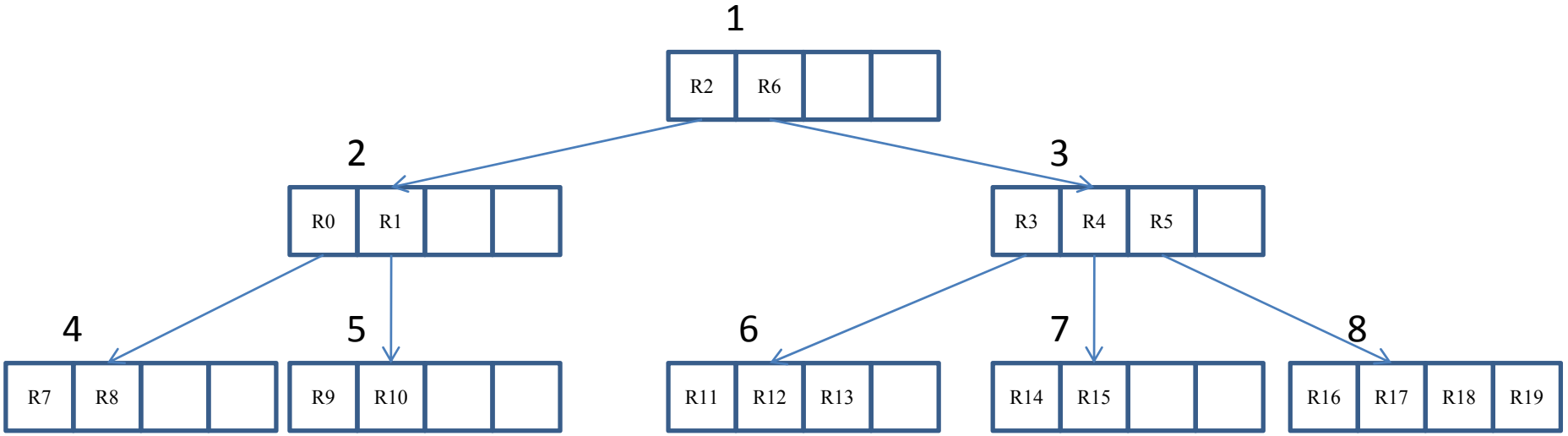
- Algorithm **AdjustTree**:
  1. Set N = L. If L was split previously, set NN to be the resulting second node.
  2. If N is the root, stop.
  3. Let P be the parent node of N, and let E_N be N's entry in P. Adjust $E_N I$ so that it tightly encloses all entry rectangle in N.
  4. If N has a partner NN resulting from an earlier split, create a new entry E_{NN} with E_{NN} P pointing to NN and $E_{NN} I$ enclosing all rectangles in NN. Add E_{NN} to P if there is a room. Otherwise invoke **SplitNode** to produce P and PP containing E_{NN} and all P's old entry
  5. Set N=P va NN=PP if a split occurred. Repeat from 2.

Deletion

- R-tree: M=4, m=2
- Delete R13
- Delete R7



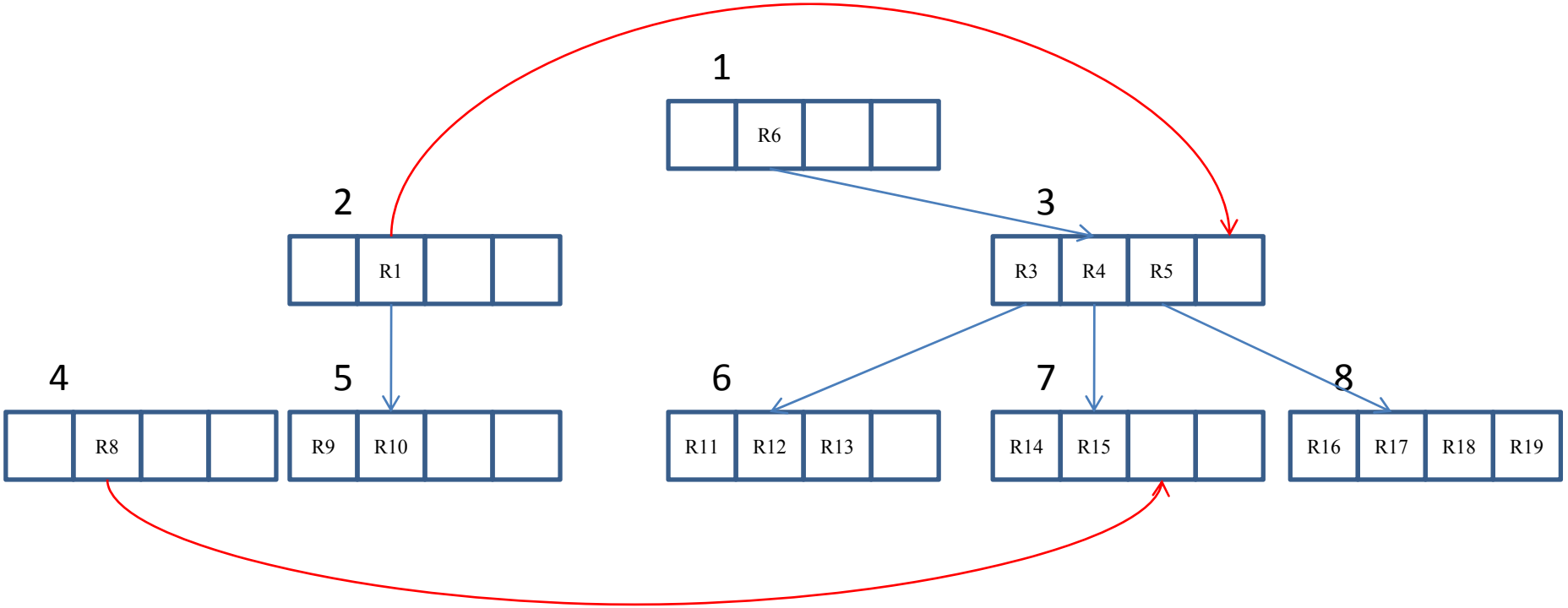
Deletion



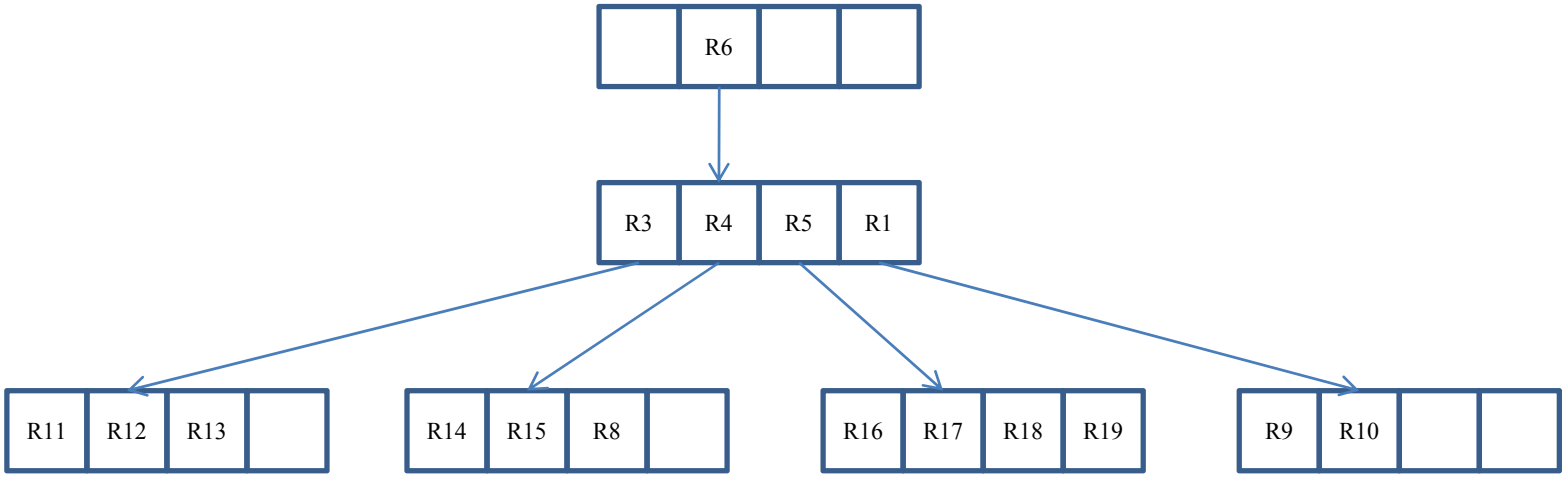
- Set Q, the eliminated node
- Delete R7, node 4 has only 1 entry. Delete R0 from node 2 and add node 4 to Q, Q = {4}.
- Delete R0, node 2 has only 1 entry. Delete R2 from node 1 and add node 2 to Q, Q = {4,2}.

Deletion

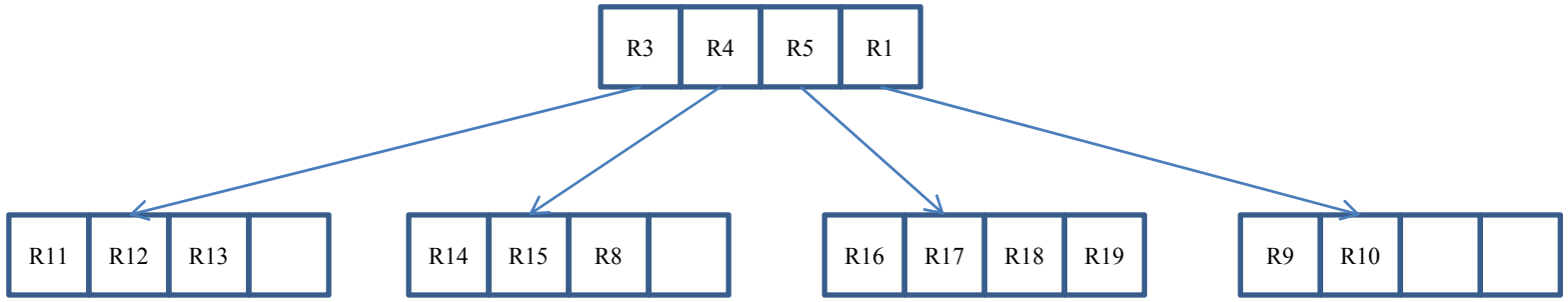
- Q = {4,2}
- **For** each node ∈ Q **Do**
  For each E ∈ node **Do**
    If Q is leaf Then Insert(E)
    Else Insert(E) as a node entry at the same node level



Deletion



- If the root node has only one child after the tree has been adjust, make the child the new root



Deletion

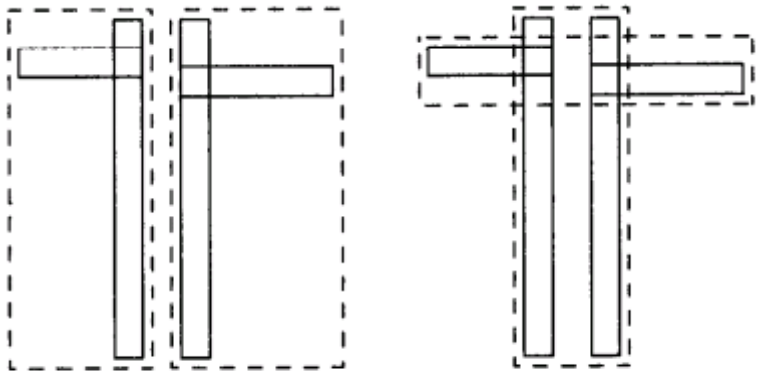
- Alogrithm **Delete**:
  1. Invoke **FindLeaf** to locate the leaf node L containing E, stop if record was not found.
  2. Remove E from L.
  3. Invoke **CondenseTree**, passing L.
  4. If the root node has only one child after the the tree has been adjusted, make the child the new root.

Deletion

- Algorithm CondenseTree:
  1. Set N=L. Set Q, the set of eliminated node, be empty.
  2. If N is the root go to 6. Otherwise let P be the parent of N, and let E_N be N's entry in P.
  3. If N has fewer than m entries, delete E_N from P and add N to set Q
  4. If N has not been eliminated, adjust $E_N I$ to tightly contain all entries in N.
  5. Set N=P and repeat from 2.
  6. Re-insert all entries of nodes in set Q. Entries from eliminated leaf node are re-inserted in tree leaves as described in Algorithm insert, but entries from higher level node must be placed higher in the tree.

Node Splitting

- Exhaustive algorithm.
- Quadratic algorithm.
- Linear time algorithm.



Bad split

Good split

Exhaustive Algorithm

- Try all possible combinations and choose the best.
- Optimal results.
- Bad running time.

Quadratic-cost algorithm

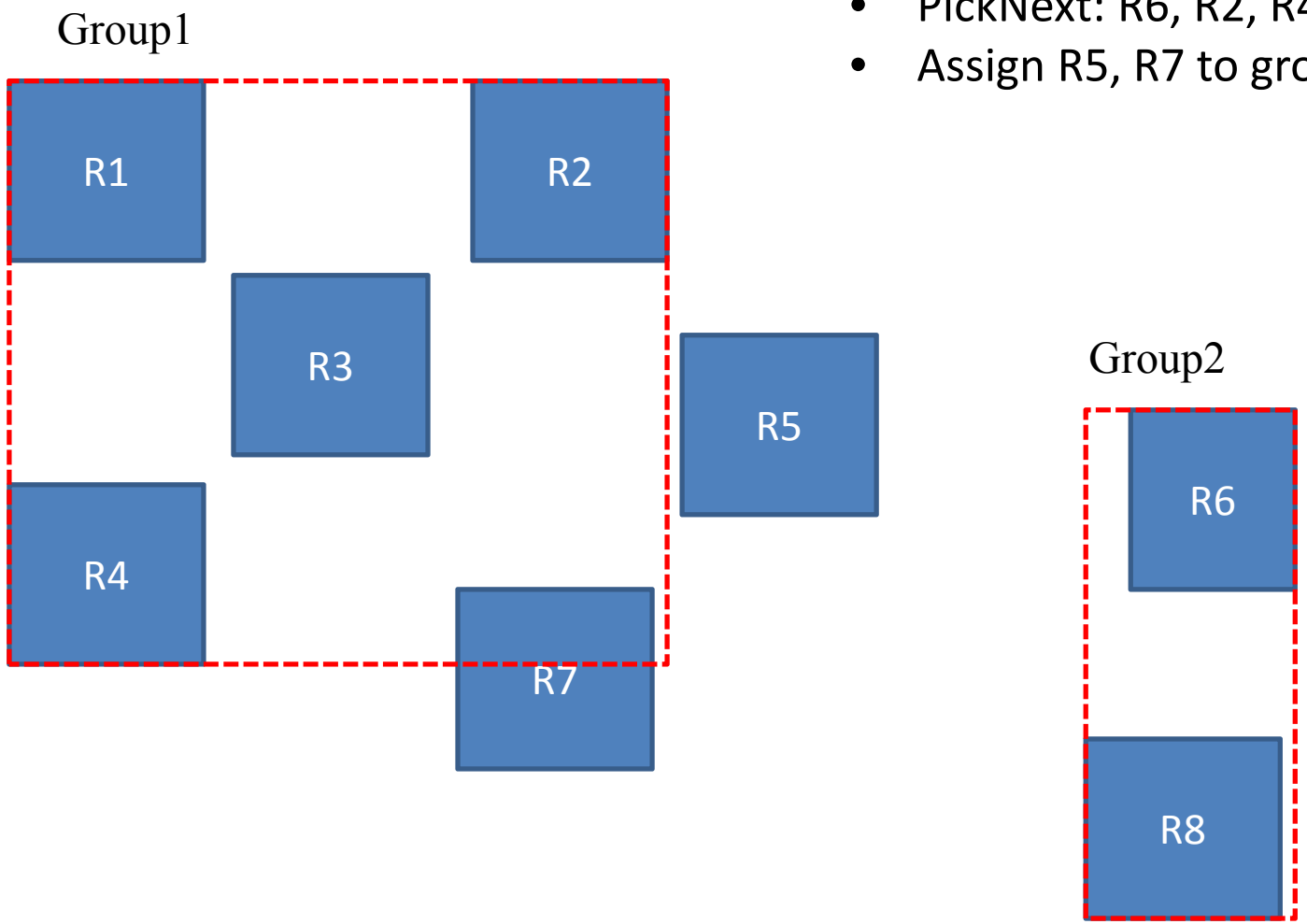
- Algorithm Quaratic split
  1. Apply alogrithm **PickSeeds** to choose two entries to be the first elements of the groups. Assign each to a group.
  2. If all entries have been assigned, stop. If one group have so few entries that all the rest must be assigned to it in order for it to have the minimum number m, assign them and stop.
  3. Invoke algorithm **PickNext** to choose the next entry to assign. Add it to the group whose covering rectangle will have to be enlarged least to accomadate it. Resolves ties by adding the entry to the group with smaller area, then to the one with fewer entries, then to either. Repeat from 2.

Quaratic-cost algorithm

- Algorithm **PickSeeds**:
  1. For each pair of entries E_1 and E_2, compose a rectangle J including E_1I and E_2I. Calculate d = area(J) – area(E_1I) – area (E_2I).
  2. Choose the pair with the largest d.
- Algorithm **PickNext**:
  1. For each entry E not yet in group, calculate d1 = the area increase required in the covering of Group1 to include EI. Calculate d2 similarly for Group2
  2. Choose any entry with the maximum difference between d1 and d2.

Quaratic-cost algorithm

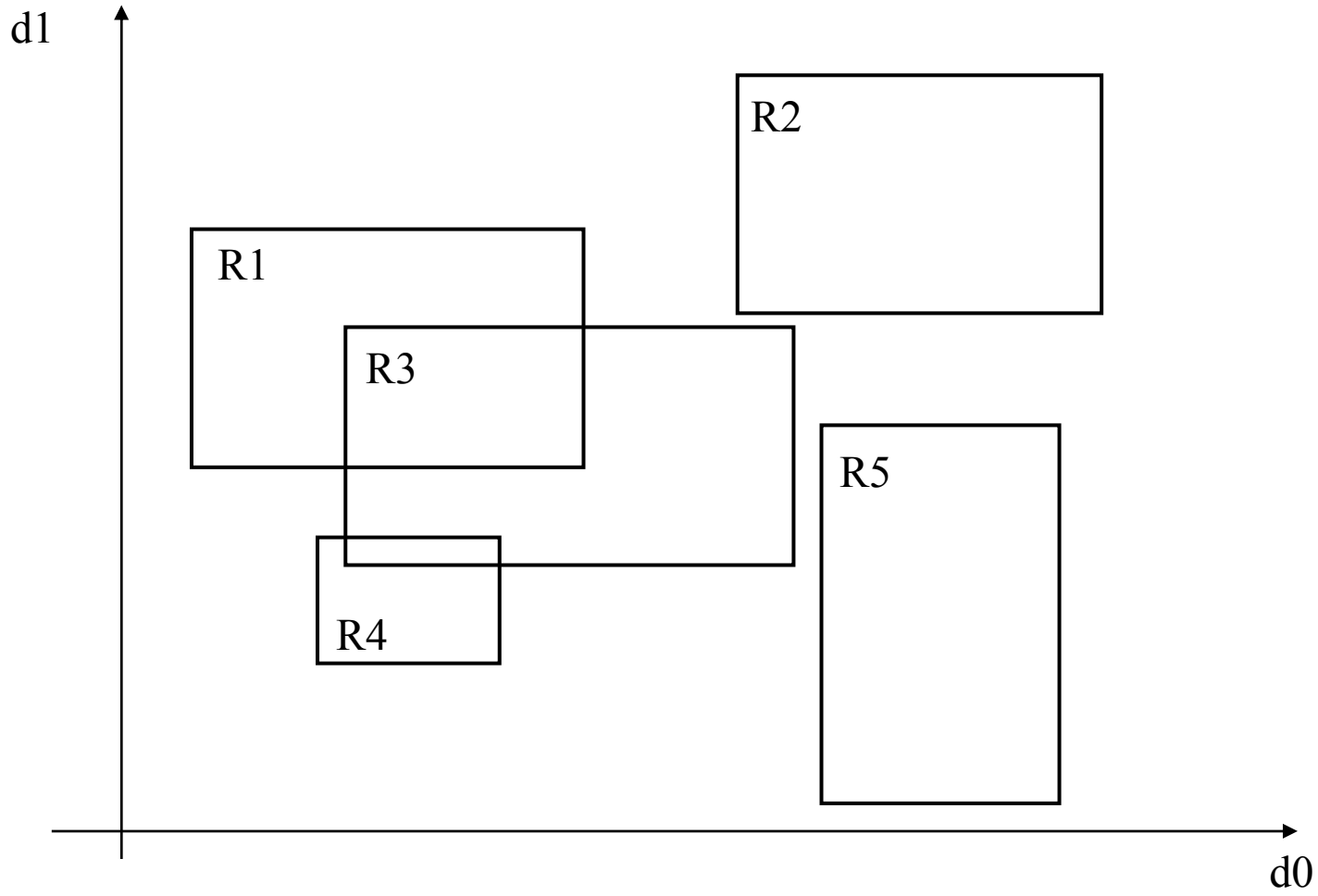
- Two seeds: R1, R8.
- PickNext: R6, R2, R4, R3
- Assign R5, R7 to group 2.



Linear-cost Algorithm

- Algorithm LinearPickSeed:
  1. Along each dimension, find the entry whose rectangle has the highest low side, and the one with the lowest high side. Record the separation.
  2. Normalize the separations by dividing the width of the entire set along the corresponding dimension.
  3. Choose the pair with the greatest normalized separation along any dimension.

Linear-cost Algorithm



R*-Tree

- Optimization on R-tree
- R-Tree consider area parameter
- More parameters are considered: area, overlap, margin in different combinations

Insertion

- Insertion
  - At levels above leaf-1, as before R-Tree
  - At leaf-1 level, choose sub tree with minimum overlap (resolve ties by choosing the entry whose rectangle needs least area enlargement)
  - overlap(E, node) = sum of area(E ∩ entry) for all entry in node
  - only marginally better

Split strategy

- For each dimension, sort M+1 values by the lower value (use upper value to break ties)
- Consider groups containing the first m-1+k and the remaining M+2-m-k entries with k in [1,M-2m+2]
- Evaluate the area-value, margin-value, and overlap-value for each split point

Split strategy

- Area-value(split) = area(first group) + area(second group)
  - Smaller area reduces access probability of access
- Margin-value(split) = margin(first group) + margin(second group)
  - Small margin produces better packing and less overlaps
- Overlap-value(split) = common area of two groups
  - Minimize common search area
- Choose split axis as the one containing the smallest sum of all margin-values of the different distributions.
- Along the split axis, choose the splitting point to be the one that gives the minimum overlap-value. Use area-value to resolve ties

Forced reinserts

- If a split occurs at level k(not level root) and this is the first time overflow occurs in the given level:
  - sort the entries in overflowing node in a descending order based on the distance of their centroid from the node centroid
  - Remove the first p entries and adjust the bounding rectangle of the overflowing node
  - Reinsert the p removed entries (data or index)
  - Value for p = 30%
- If not, split as before.
- This reduces overlap and leads to a better structure

Benefits

- Forced reinsert changes entries between neighboring nodes and thus decreases the overlap
- As a side effect, storage utilization is improved
- Due to more restructuring, less split occur
- Since the outer rectangles of a node are reinserted, the shape of the directory rectangles will be more quadratic

References

- Antonin Guttman, “R-Trees: A Dynamic Index Structure for Spatial Searching”, Proc. 1984 ACM SIGMOD International Conference on Management of Data, pp. 47-57.
- Norbert Beckmann,H.-P. Kriegel, R. Schneider, B. Seeger, “The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles”, SIGMOD Conference 1990: 322-331
- Volker Gaede, Oliver Gunther, “Multidimension access method”, ACM computing Surveys, Vol. 30, No.2, June 1998.