R-Trees

A Dynamic Index Structure for Spatial Searching

Based on Antonin Guttman's 1984 Paper · *February 23, 2026*

PRESENTED BY

Farhan Labib

2205071

Fatima Sad Sudipta

2205063

Bangladesh University of Engineering & Technology

Outline

- Introduction
- R-Tree Structure
- Search Simulation
- Insertion & Node Splitting
- Full Insertion Walkthrough
- Deletion
- Key Takeaways

Introduction

The Problem

Traditional Indexes

- B-trees, hash tables
- One-dimensional keys
- Exact-match queries

They struggle with. . .

- Multi-dimensional data
- Objects with *spatial extent*
- “Find all objects overlapping S”

Real-world examples

- GIS: counties, roads, buildings
- CAD: circuit layouts
- Games: collision detection
- Maps: nearest-place search

We need a better structure!

The Problem

Traditional Indexes

- B-trees, hash tables
- One-dimensional keys
- Exact-match queries

They struggle with...

- Multi-dimensional data
- Objects with *spatial extent*
- "Find all objects overlapping S"

Real-world examples

- GIS: counties, roads, buildings
- CAD: circuit layouts
- Games: collision detection
- Maps: nearest-place search

We need a better structure!

The Problem

Traditional Indexes

- B-trees, hash tables
- One-dimensional keys
- Exact-match queries

They struggle with...

- Multi-dimensional data
- Objects with *spatial extent*
- "Find all objects overlapping S"

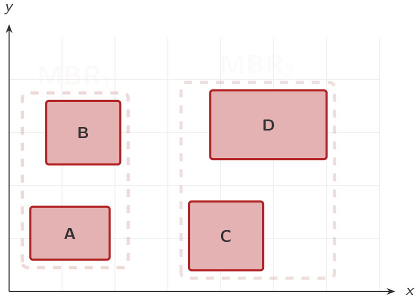
Real-world examples

- GIS: counties, roads, buildings
- CAD: circuit layouts
- Games: collision detection
- Maps: nearest-place search

We need a better structure!

R-Tree Structure

Core Idea: Minimum Bounding Rectangles

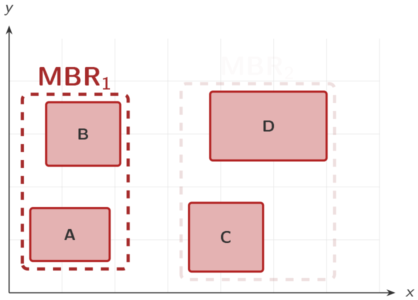


Step 1 — Objects

Four spatial objects in 2-D space.

- Every tree node $\approx$ one disk page
- $m \leq$ entries $\leq M$ per node
- All leaves at the same depth

Core Idea: Minimum Bounding Rectangles

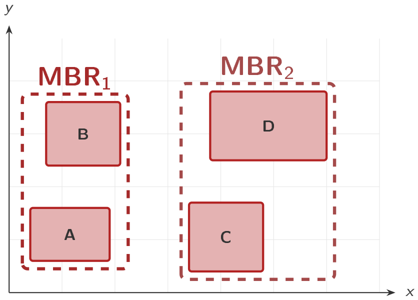


Step 2 — Group left pair

Draw the tightest rectangle around A and B: the Minimum Bounding Rectangle (MBR).

- Every tree node ≈ one disk page
- $m \leq$ entries $\leq M$ per node
- All leaves at the same depth

Core Idea: Minimum Bounding Rectangles



Step 3 — Group right pair

Each MBR becomes one entry in an R-tree internal node. MBRs *may* overlap.

- Every tree node ≈ one disk page
- $m \leq$ entries $\leq M$ per node
- All leaves at the same depth

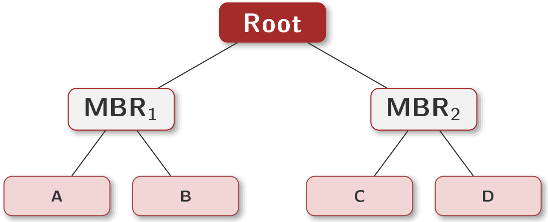
R-Tree Node Types & Tree Shape

Leaf node entry (I, *tuple-id*)

- I = MBR tightly wrapping the object
- *tuple-id* = pointer to data record

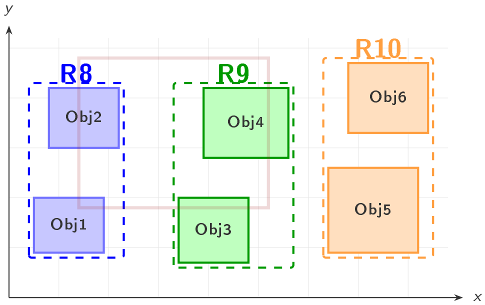
Internal node entry (I, *child-ptr*)

- I = MBR covering *all* children's MBRs
- *child-ptr* = address of child page



Search Simulation

Simulation: Search — Step 1, Tree Structure



Step 1 — Tree structure

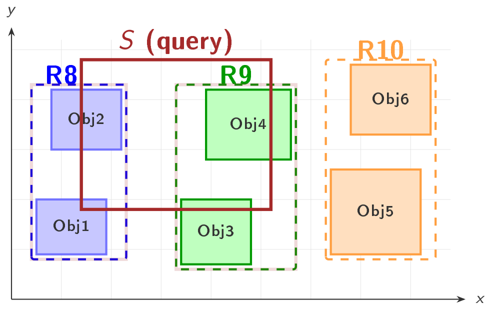
Three leaf groups: **R8**, **R9**, **R10**.

Each group's MBR tightly wraps its objects.

[

ghost]Step 2 — Coming next Issue a spatial query box S...

Simulation: Search — Step 2, Issue Query S



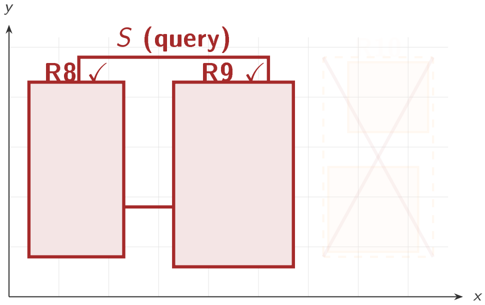
Step 2 — Issue query S

For each MBR, ask:
Does it overlap S?

Step 3 — Coming next

R8 and R9 will match. . .

Simulation: Search — Step 3, R8 & R9 Match



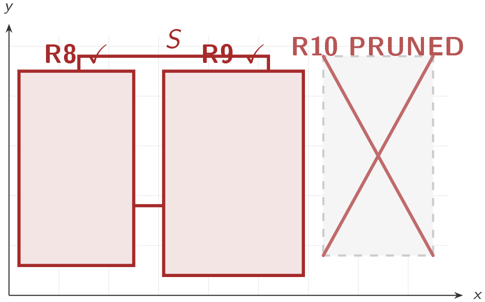
Step 3 — R8 and R9 match

Both MBRs overlap S.
Descend into both subtrees and report overlapping objects.

Step 4 — Coming next

R10 will be pruned entirely. . .

Simulation: Search — Step 4, R10 Pruned



Step 4 — R10 pruned

$R10 \cap S = \emptyset$

Skip the **entire** R10 subtree.

Zero extra disk reads!

Key lesson

Good MBRs $\Rightarrow$ more pruning

$\Rightarrow$ fewer disk reads

$\Rightarrow$ faster queries

Insertion & Node Splitting

Insertion — Overview

1. **ChooseLeaf** — pick child needing least MBR enlargement
2. **Insert** entry into chosen leaf
3. If leaf has $> M$ entries: SplitNode
4. **AdjustTree** — update ancestor MBRs upward
5. If root split: create new root one level higher

The critical step

Quality of **node splitting** drives future search performance.

Goal: small, non-overlapping MBRs.

Insertion — Overview

1. **ChooseLeaf** — pick child needing least MBR enlargement
2. **Insert** entry into chosen leaf
3. If leaf has $> M$ entries: SplitNode
4. **AdjustTree** — update ancestor MBRs upward
5. If root split: create new root one level higher

The critical step

Quality of **node splitting** drives future search performance.

Goal: small, non-overlapping MBRs.

Insertion — Overview

1. **ChooseLeaf** — pick child needing least MBR enlargement
2. **Insert** entry into chosen leaf
3. If leaf has $> M$ entries: SplitNode
4. **AdjustTree** — update ancestor MBRs upward
5. If root split: create new root one level higher

The critical step

Quality of **node splitting** drives future search performance.

Goal: small, non-overlapping MBRs.

Insertion — Overview

1. **ChooseLeaf** — pick child needing least MBR enlargement
2. **Insert** entry into chosen leaf
3. If leaf has $> M$ entries: SplitNode
4. **AdjustTree** — update ancestor MBRs upward
5. If root split: create new root one level higher

The critical step

Quality of **node splitting** drives future search performance.

Goal: small, non-overlapping MBRs.

Insertion — Overview

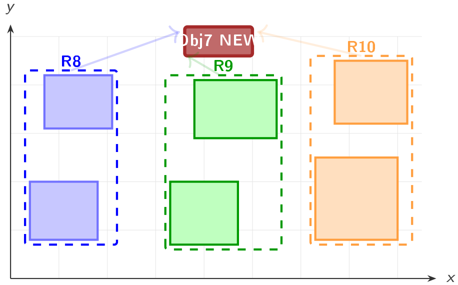
1. **ChooseLeaf** — pick child needing least MBR enlargement
2. **Insert** entry into chosen leaf
3. If leaf has $> M$ entries: SplitNode
4. **AdjustTree** — update ancestor MBRs upward
5. If root split: create new root one level higher

The critical step

Quality of **node splitting** drives future search performance.

Goal: small, non-overlapping MBRs.

Simulation: ChooseLeaf — Step 1, New Object Arrives



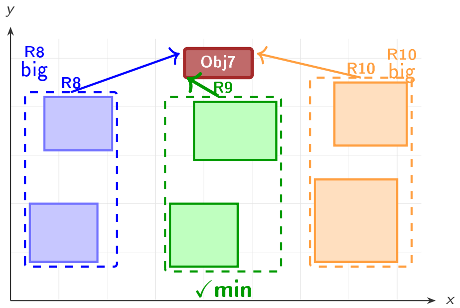
Step 1 — Obj7 arrives

Must find the best leaf node to insert into.

Step 2 — Coming next

Compute enlargements for each group. . .

Simulation: ChooseLeaf — Step 2, Compare Enlargements



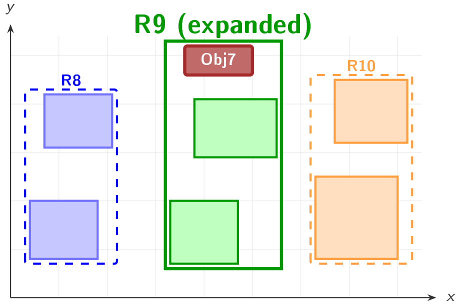
Step 2 — Enlargements

R8: large growth needed
R9: minimal growth ✓
R10: large growth needed

Step 3 — Coming next

Insert into R9 and update MBR...

Simulation: ChooseLeaf — Step 3, Insert into R9



Step 3 — Inserted!

R9 = {Obj3, Obj4, **Obj7**}
3 entries $= M$. No split.

MBR updated to wrap Obj7.

Node Splitting — Three Algorithms

Exhaustive

All 2^M groupings.
Optimal quality.
$O(2^M)$ — too slow.

Quadratic *(focus)*

Heuristic seeds.
Good quality.
$O(M^2)$ per split.

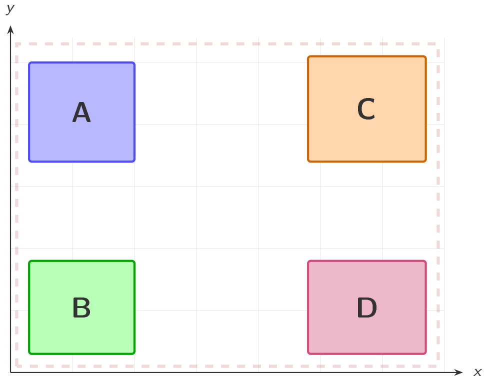
Linear

Axis extremes.
Acceptable quality.
$O(M)$ — fastest.

Goal of any split

Two nodes with small, minimally overlapping MBRs.
Better split $\Rightarrow$ better search pruning $\Rightarrow$ fewer disk reads.

Quadratic Split — Step 1: PickSeeds, Meet the Entries



Overflowing node

4 entries: A, B, C, D ($> M = 3$).

Must split into two groups.

PickSeeds formula

For every pair (E_i, E_j):

$$d = \text{area}(E_i \cup E_j) - \text{area}(E_i) - \text{area}(E_j)$$

Choose pair with largest d (most wasted space).

Quadratic Split — Step 2: PickSeeds, Worst Pair Found

MBR($A \cup D$) = max wasted area

Seeds: A and D

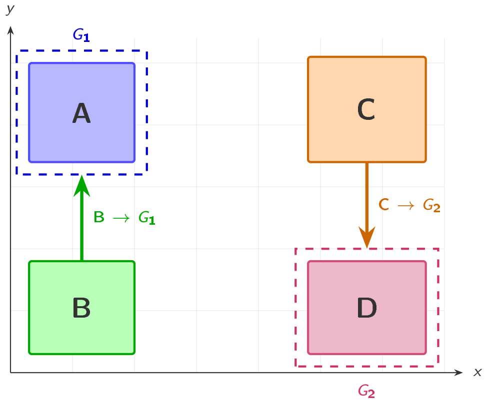
Diagonally opposite — their joint MBR wastes the most space.

$s_1 = A \to G_1$
$s_2 = D \to G_2$

Next — PickNext

Assign B and C to their preferred group. . .

Quadratic Split — Step 3: PickNext, Assign Remaining



Rule

$\delta_i = \Delta\text{area}(G_1) - \Delta\text{area}(G_2)$

Assign entry with largest $|\delta_i|$ first.

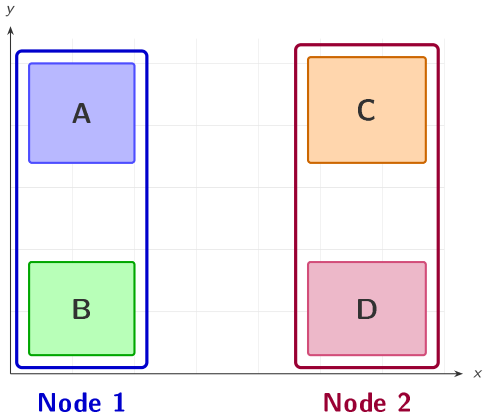
Round 1 result

B strongly prefers G_1 (near A).

C strongly prefers G_2 (near D).

Both assigned. **Split complete!**

Quadratic Split — Step 4: Final Groups



Result

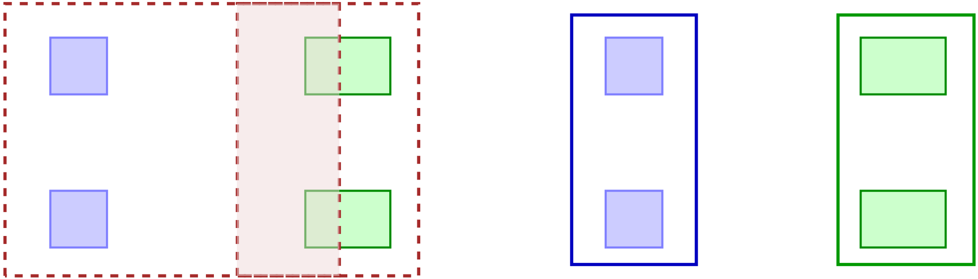
Node 1: {A, B} — left cluster
Node 2: {C, D} — right cluster

Compact, non-overlapping MBRs.

AdjustTree

Parent gets two new entries pointing to Node 1 and Node 2.
Ancestor MBRs updated upward.

Good Split vs Bad Split



Bad Split

Large overlap

$\Rightarrow$ query visits BOTH branches

Good Split

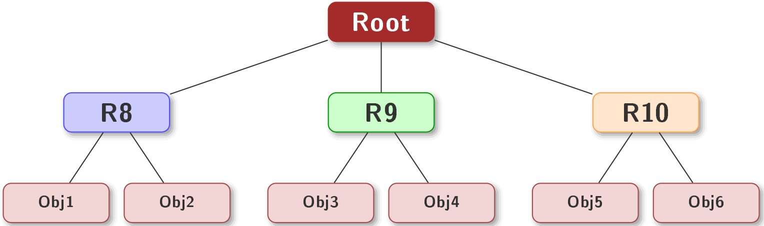
No overlap

$\Rightarrow$ query prunes one branch entirely

Full Insertion Walkthrough

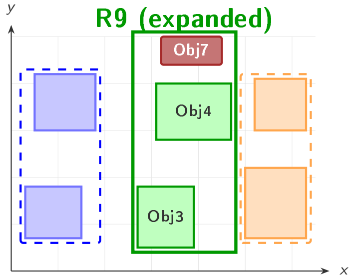
Walkthrough — Setup

R-tree with $M = 3$, starting state:



Plan: insert **Obj7** (near R9) — no overflow.
Then insert **Obj8** (near R9) — triggers overflow + split.

Walkthrough — Insert Obj7, R9 Updated

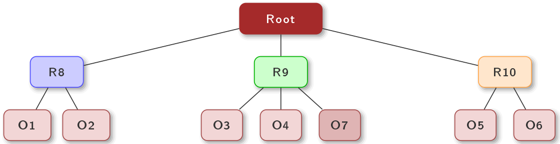


ChooseLeaf $\rightarrow$ R9

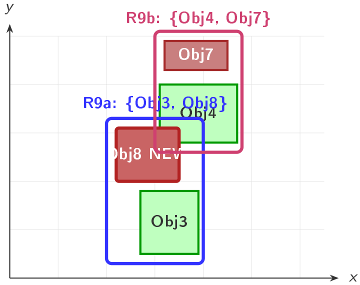
R9 needs minimum enlargement.

Result — No Split

R9 = {Obj3, Obj4, Obj7}
3 entries $= M$. Tree unchanged.



Walkthrough — Insert Obj8, Overflow!



R9 overflows! (4 entries)

Overflow

R9 would have 4 entries $> M = 3$.
SplitNode triggered.

Quadratic split result

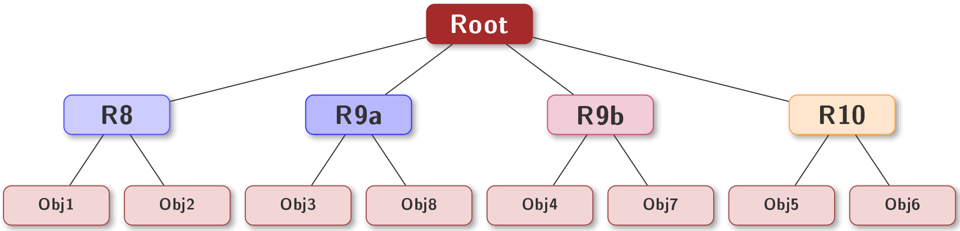
Seeds: Obj3 & Obj7 (max wasted area).

R9a = {Obj3, Obj8}
R9b = {Obj4, Obj7}

AdjustTree

Root gains R9a and R9b.
4 children total — still $\leq M$.

Walkthrough — Final Tree After Split



Root now has 4 children. Tree height unchanged. All MBRs compact thanks to the quadratic split.

Deletion

Deletion Algorithm

1. **FindLeaf** — locate the leaf containing E
2. **Remove** E from that leaf
3. **CondenseTree** — if node has $< m$ entries:
   delete it; re-insert orphaned entries
4. Propagate MBR shrinkage upward
5. If root has one child: shrink tree height

Why re-insert, not merge?

- Reuses Insert logic
- Entries find natural home
- Gradually improves structure
- Simpler than spatial merge

Deletion Algorithm

1. **FindLeaf** — locate the leaf containing E
2. **Remove** E from that leaf
3. **CondenseTree** — if node has $< m$ entries:
   delete it; re-insert orphaned entries
4. Propagate MBR shrinkage upward
5. If root has one child: shrink tree height

Why re-insert, not merge?

- Reuses Insert logic
- Entries find natural home
- Gradually improves structure
- Simpler than spatial merge

Deletion Algorithm

1. **FindLeaf** — locate the leaf containing E
2. **Remove** E from that leaf
3. **CondenseTree** — if node has $< m$ entries:
   delete it; re-insert orphaned entries
4. Propagate MBR shrinkage upward
5. If root has one child: shrink tree height

Why re-insert, not merge?

- Reuses Insert logic
- Entries find natural home
- Gradually improves structure
- Simpler than spatial merge

Deletion Algorithm

1. **FindLeaf** — locate the leaf containing E
2. **Remove** E from that leaf
3. **CondenseTree** — if node has $< m$ entries:
   delete it; re-insert orphaned entries
4. Propagate MBR shrinkage upward
5. If root has one child: shrink tree height

Why re-insert, not merge?

- Reuses Insert logic
- Entries find natural home
- Gradually improves structure
- Simpler than spatial merge

Deletion Algorithm

1. **FindLeaf** — locate the leaf containing E
2. **Remove** E from that leaf
3. **CondenseTree** — if node has $< m$ entries:
   delete it; re-insert orphaned entries
4. Propagate MBR shrinkage upward
5. If root has one child: shrink tree height

Why re-insert, not merge?

- Reuses Insert logic
- Entries find natural home
- Gradually improves structure
- Simpler than spatial merge

Key Takeaways

Summary

What we covered

1. MBRs group nearby spatial objects
2. Search prunes non-overlapping subtrees
3. ChooseLeaf: least-enlargement heuristic
4. Quadratic split: PickSeeds + PickNext
5. Split quality drives search efficiency
6. Deletion re-inserts orphaned entries

The golden rule

Minimise **overlap** and **area** of MBRs at every split.

Used in practice

PostgreSQL/PostGIS, QGIS,
ArcGIS, game engines,
GEOS, Shapely, SQLite R*

References

Antonin Guttman
R-Trees: A Dynamic Index Structure for Spatial Searching
ACM SIGMOD, 1984

Wikipedia contributors
R-tree — `en.wikipedia.org/wiki/R-tree`

Thank You

Questions & Discussion