Chapter 6

External Memory Structures

Acknowledgements

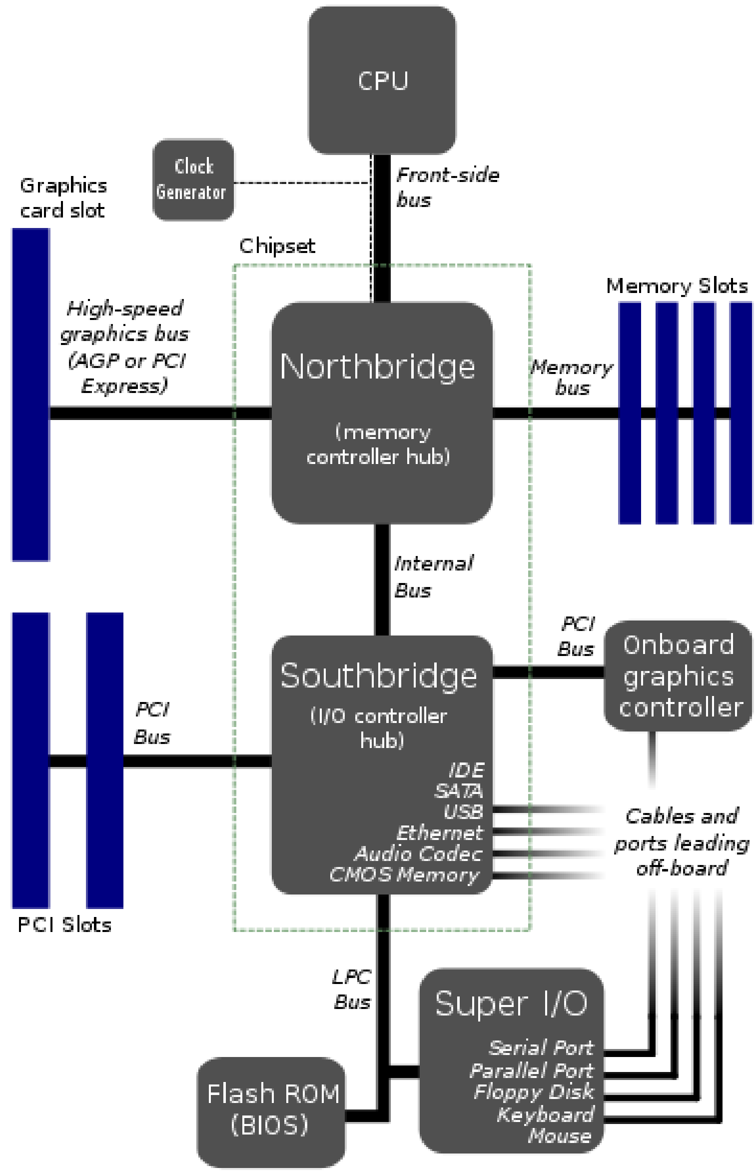
- Disk storage, Wikipedia. https://en.wikipedia.org/wiki/Disk_storage
- B-tree, Wikipedia. https://en.wikipedia.org/wiki/B-tree
- R-tree, Wikipedia. https://en.wikipedia.org/wiki/R-tree
- Multimedia Databases and Data Mining. Primary key indexing – B-trees. Christos Faloutsos – CMU
- Spatial Access Methods, Chapter 26 of book. Dr Eamonn Keogh, Computer Science & Engineering Department, University of California – Riverside, Riverside,CA 92521

Chapter Outline

- External Disk Storage
- Working with External Data
- B-tree
- B^+-tree
- R-tree

External Storage

Computer Architecture

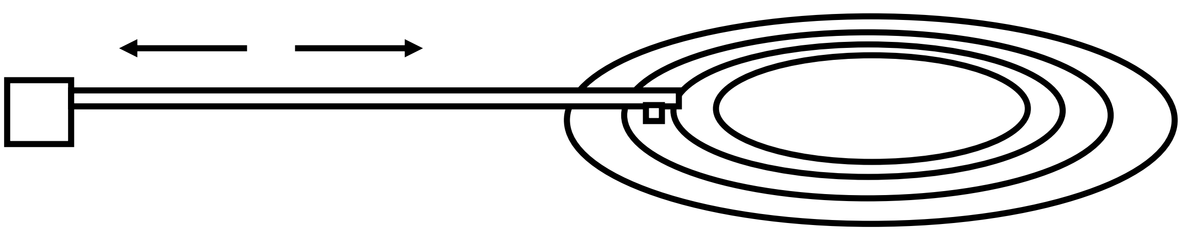


Types of External Memory

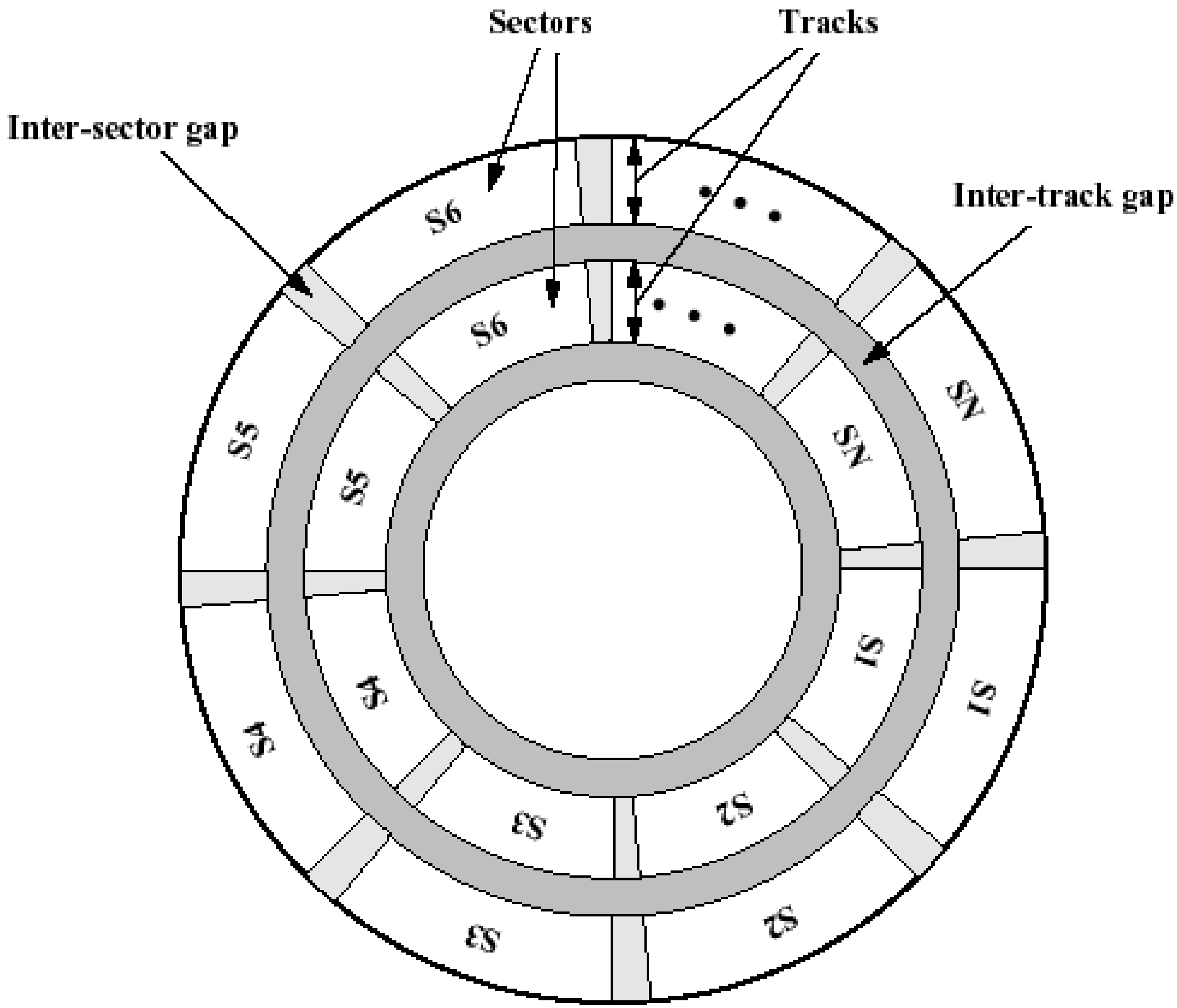
- Magnetic Tape
- Optical
  - CD (Compact Disc)
    - CD-ROM
    - CD-R
    - CD-RW
  - VCD (Video Compact Disc)
  - DVD (Digital Video/Versatile Disc)
- Magnetic Disk
  - RAID

Magnetic Disk

- Metal or plastic disk coated, on one or both sides, with magnetizable material
- Data read and written through a magnetic head (coil) by means of induction



Disk Data Layout



Data Organization and Formatting

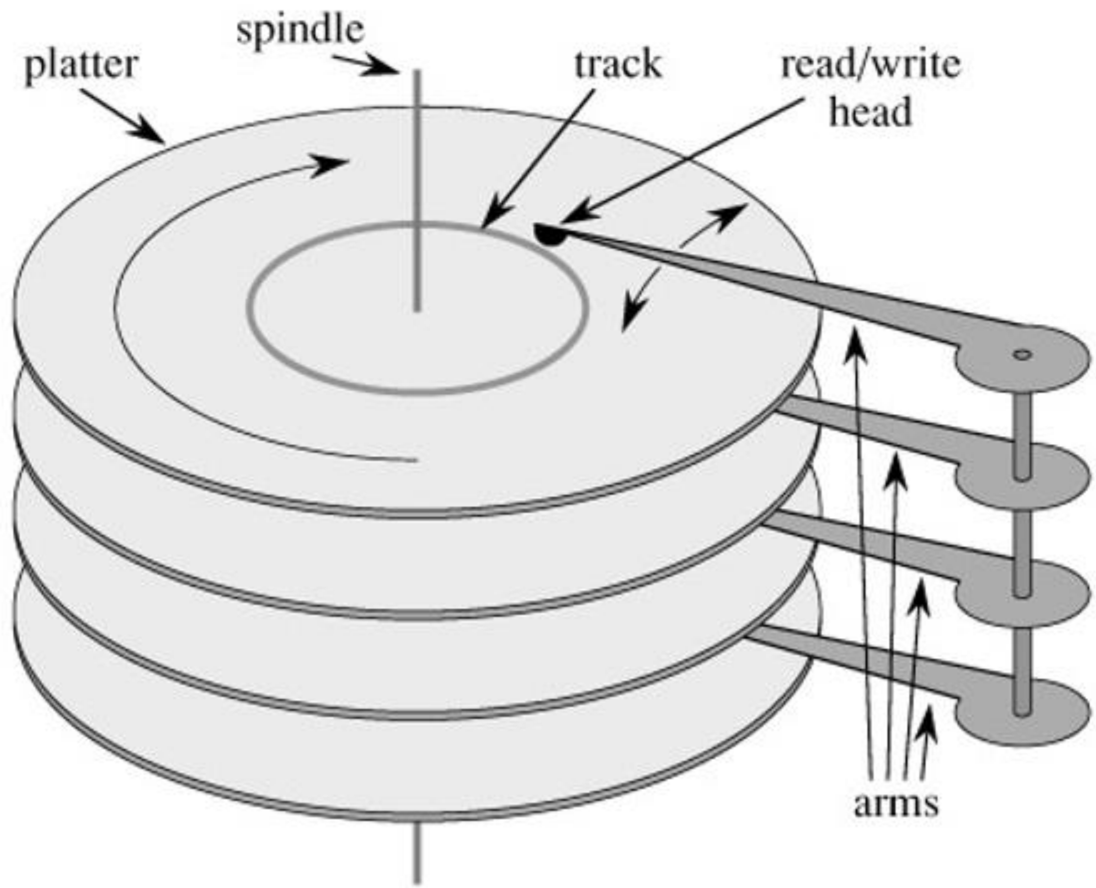
- Concentric rings or tracks
  - **Gaps** between tracks, reduce gap to increase capacity
  - Same number of **bits** per track (variable density)
  - Constant **angular velocity**
- Tracks divided into **sectors**
- Data read/written in **blocks**
  - Minimum block size is one sector
  - May have more than one sector per block

Finding Sectors

- Must be able to identify **start** of track and sector
- Format disk
  - Additional information not available to user
  - Marks tracks and sectors

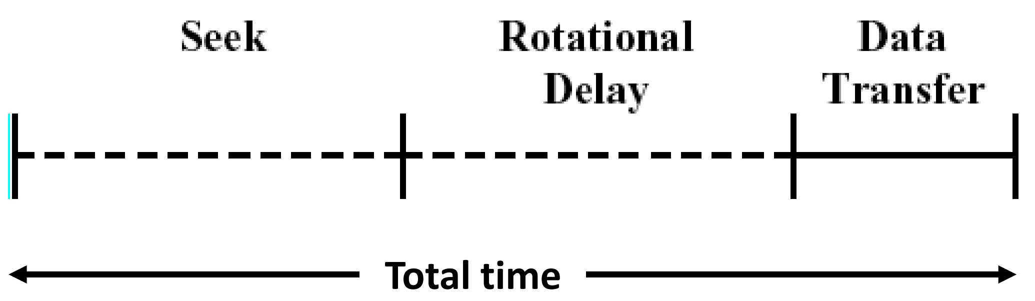
Multiple Platters

- One head per side
- Heads are joined and aligned
- Aligned tracks on each platter form cylinders
- Data is striped by cylinder
  - reduces head movement
  - increases speed (transfer rate)



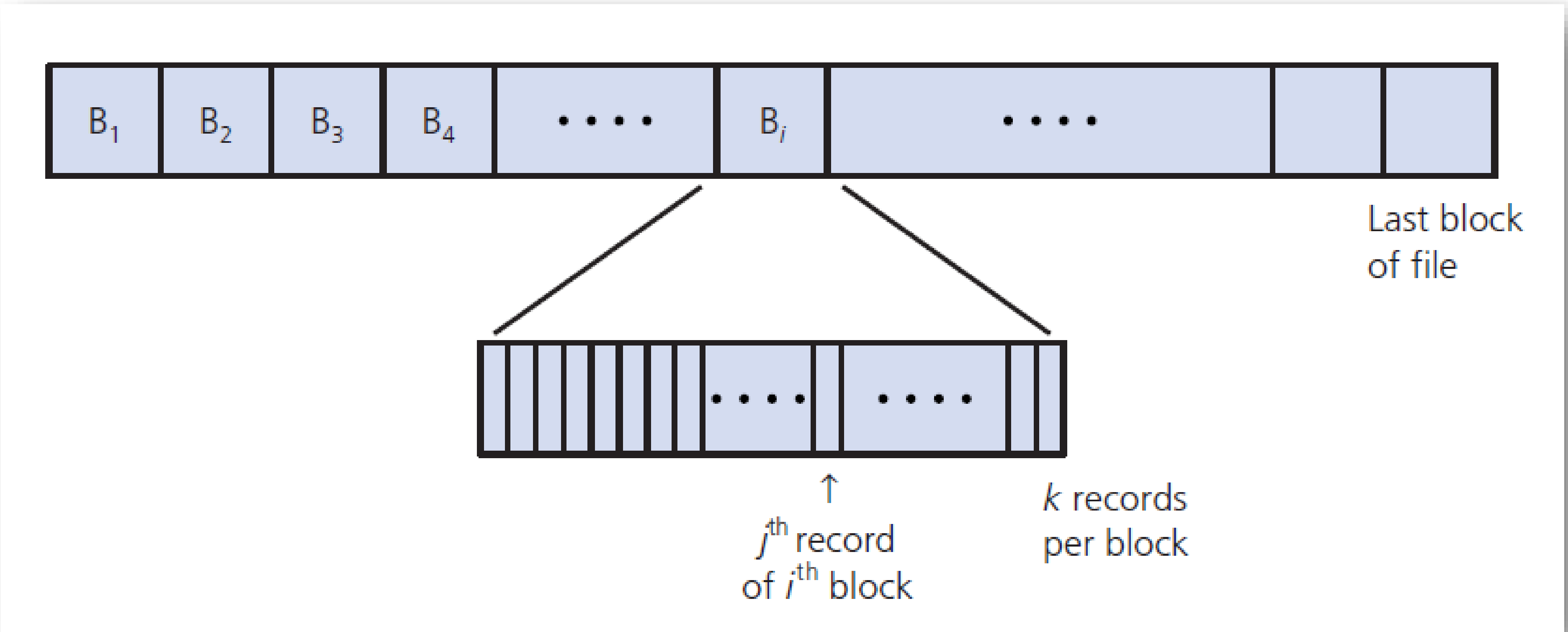
Speed

- Seek time
  - Moving head to the right track
- (Rotational) latency
  - Waiting for data to rotate under head
- Access time = Seek + Latency
- Transfer rate: speed of copying bytes from disk



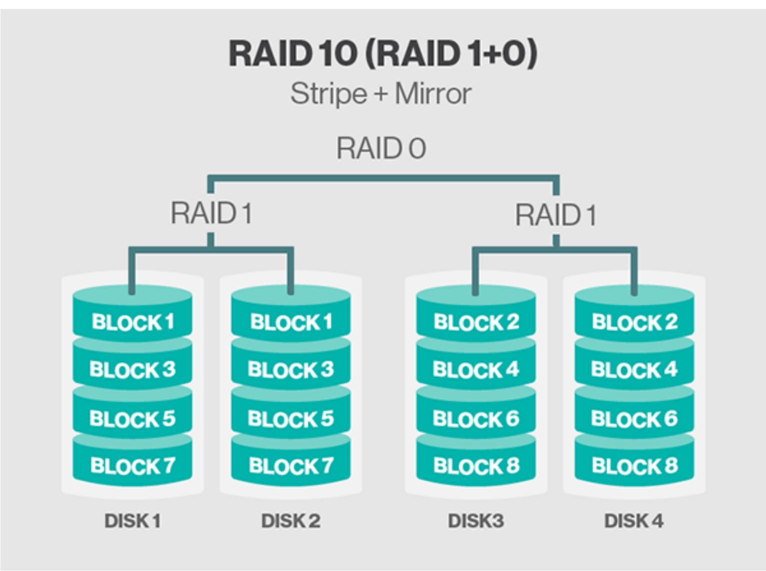
Look at External Storage

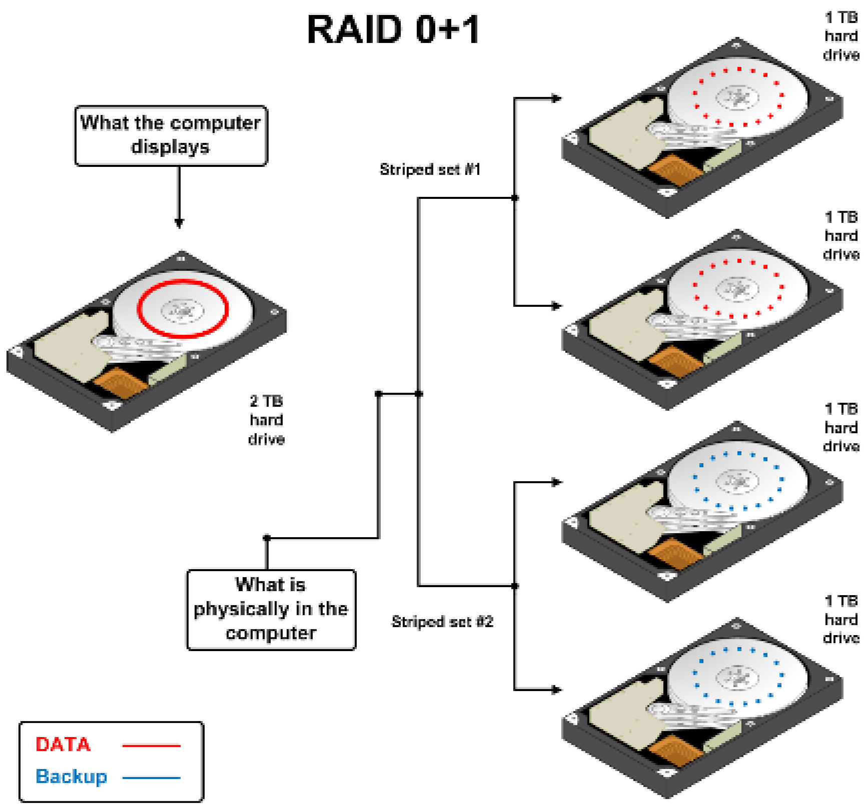
- A file partitioned into blocks of records



RAID

- Redundant Arrays of Independent Disks

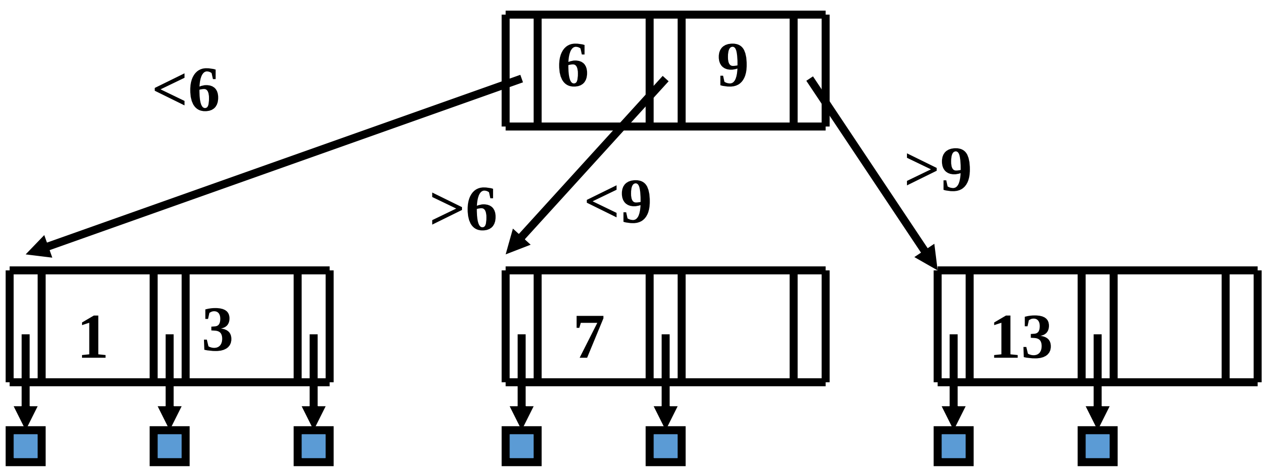




B-tree

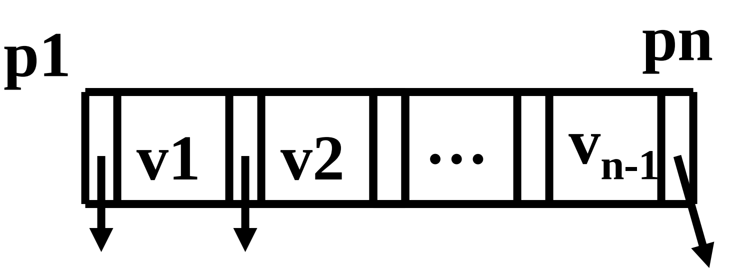
B-trees

Eg., B-tree of order 3:



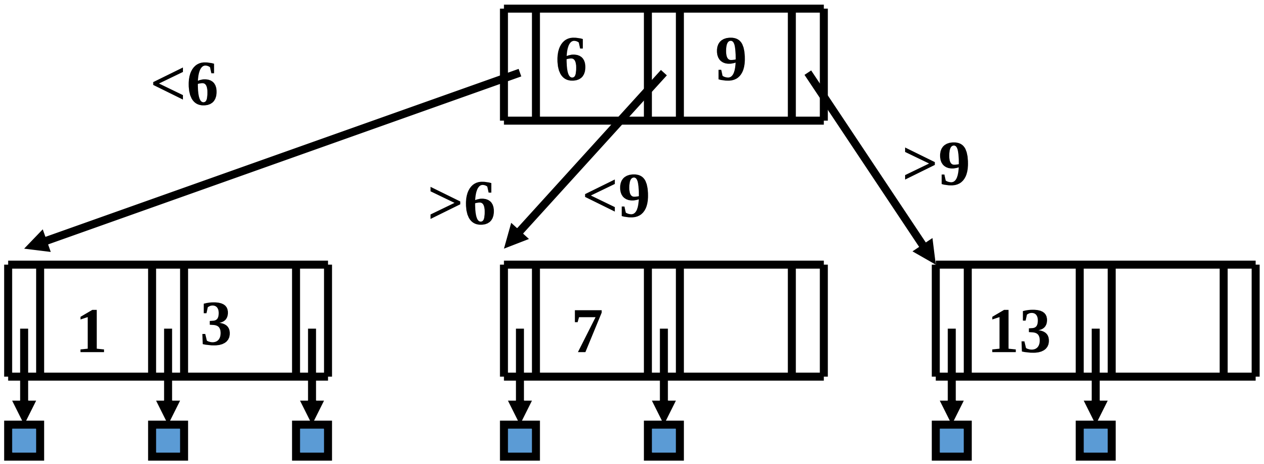
B - tree properties:

- each node, in a B-tree of order *n*:
  - Key order
  - at most *n* pointers
  - at least *n/2* pointers (except root)
  - all leaves at the same level
  - if number of pointers is *k*, then node has exactly *k-1* keys
  - (leaves are empty)



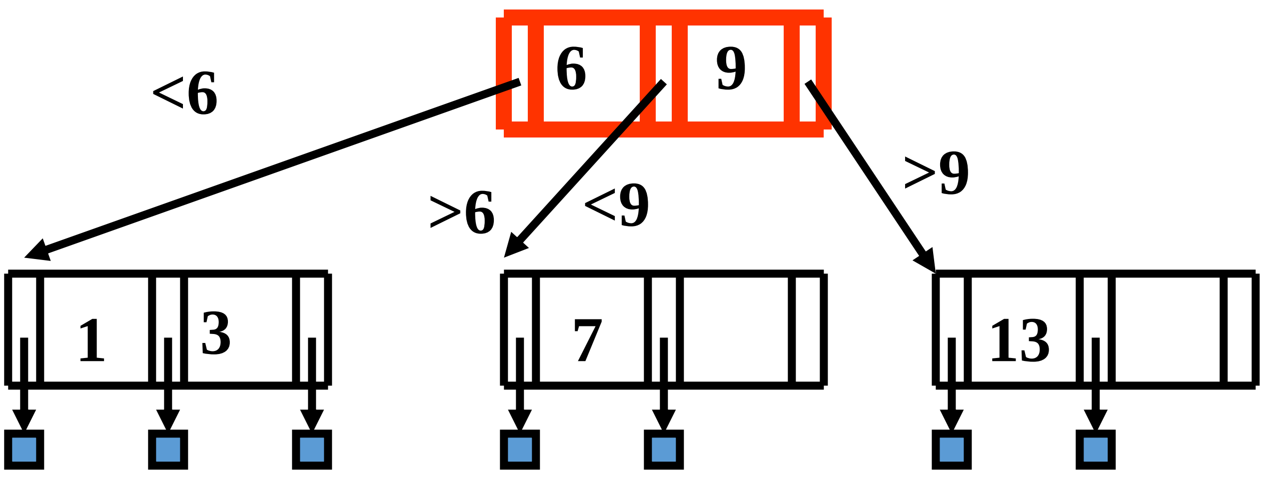
Queries

- Algo for exact match query? (eg., ssn=8?)



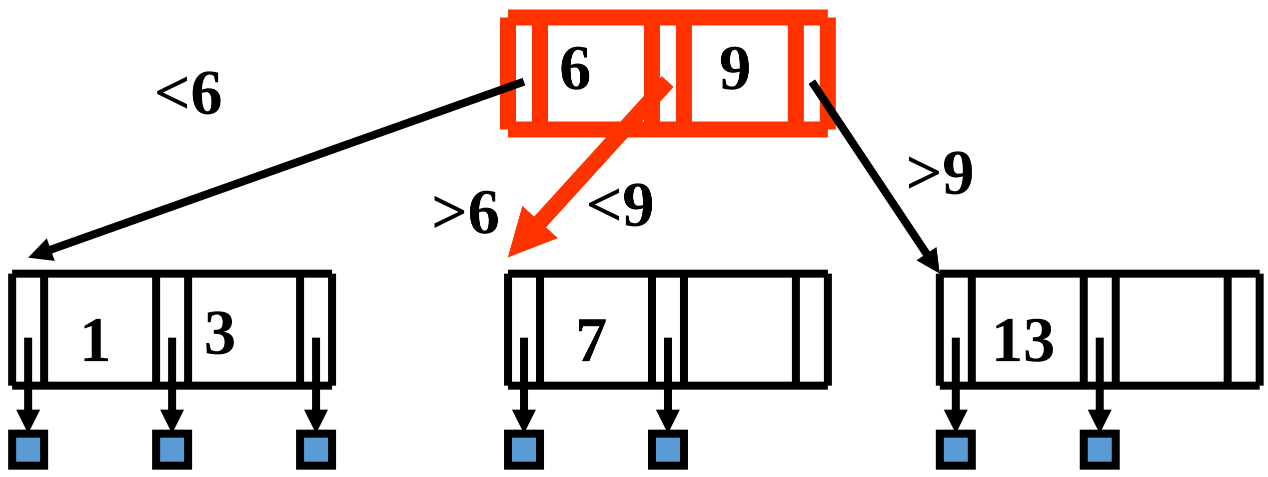
Queries

- Algo for exact match query? (eg., ssn=8?)



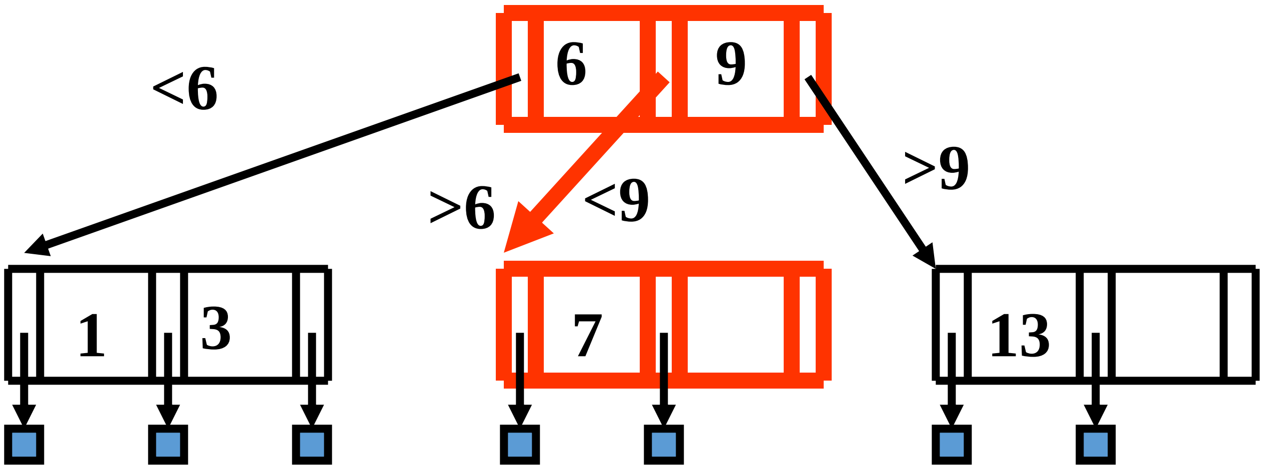
Queries

- Algo for exact match query? (eg., ssn=8?)



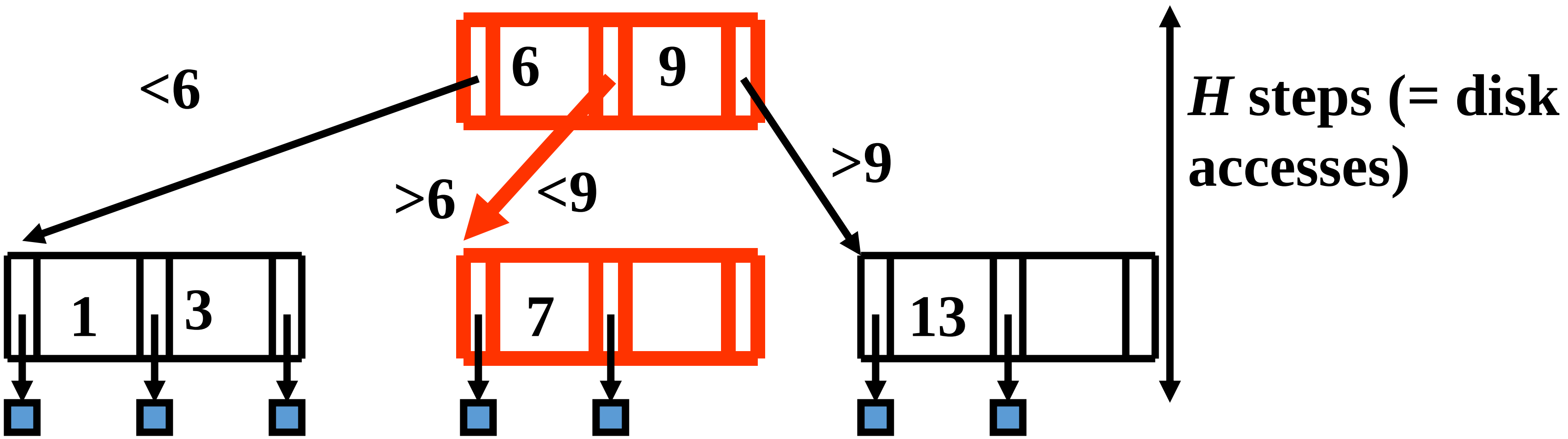
Queries

- Algo for exact match query? (eg., ssn=8?)



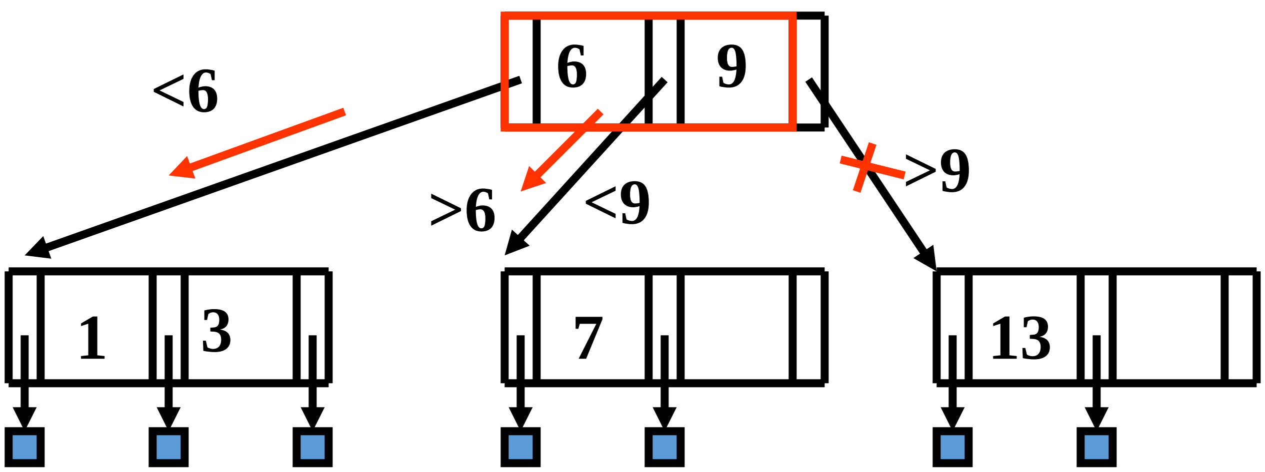
Queries

- Algo for exact match query? (eg., ssn=8?)



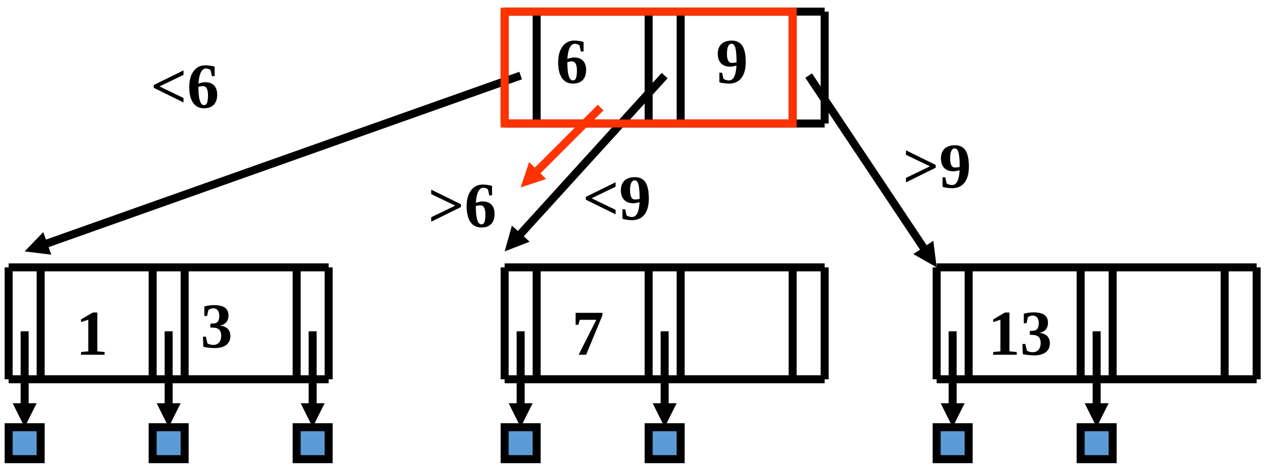
Queries

- what about range queries? (eg., ***5<salary<8***)
- Proximity/ nearest neighbor searches? (eg., *salary* ~ *8*)



Queries

- what about range queries? (eg., *5<salary<8*)
- Proximity/ nearest neighbor searches? (eg., ***salary ~ 8***)

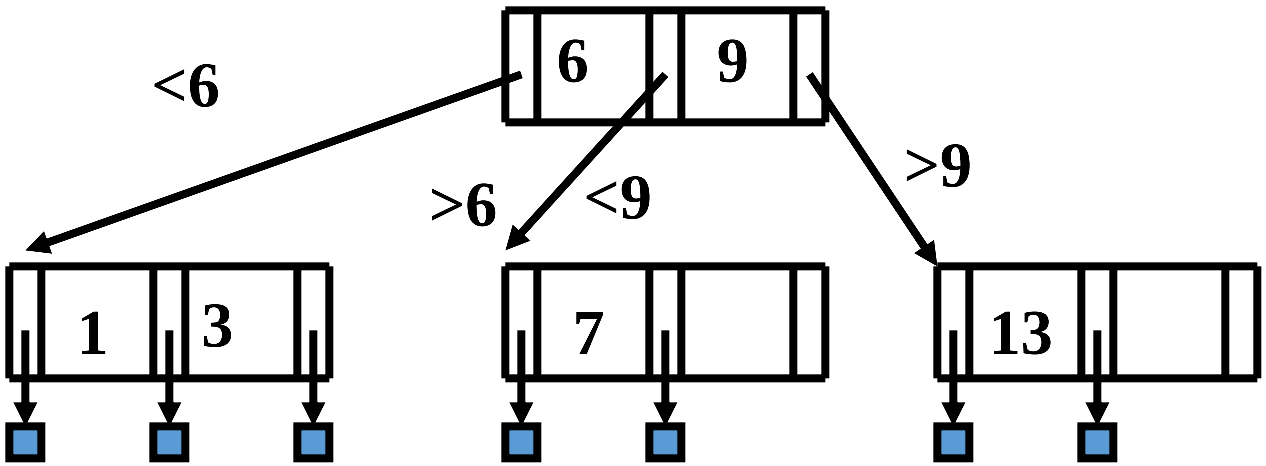


B-trees: Insertion

- Insert in leaf; on overflow, push middle up (recursively)
- split: preserves B - tree properties

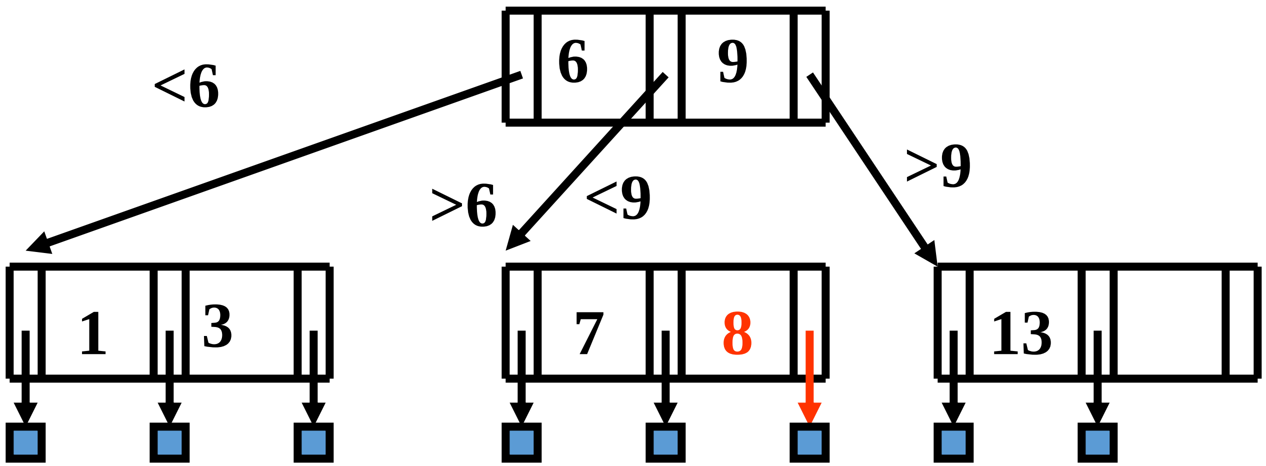
B-trees: Insertion

Easy case: Tree T0; insert '8'



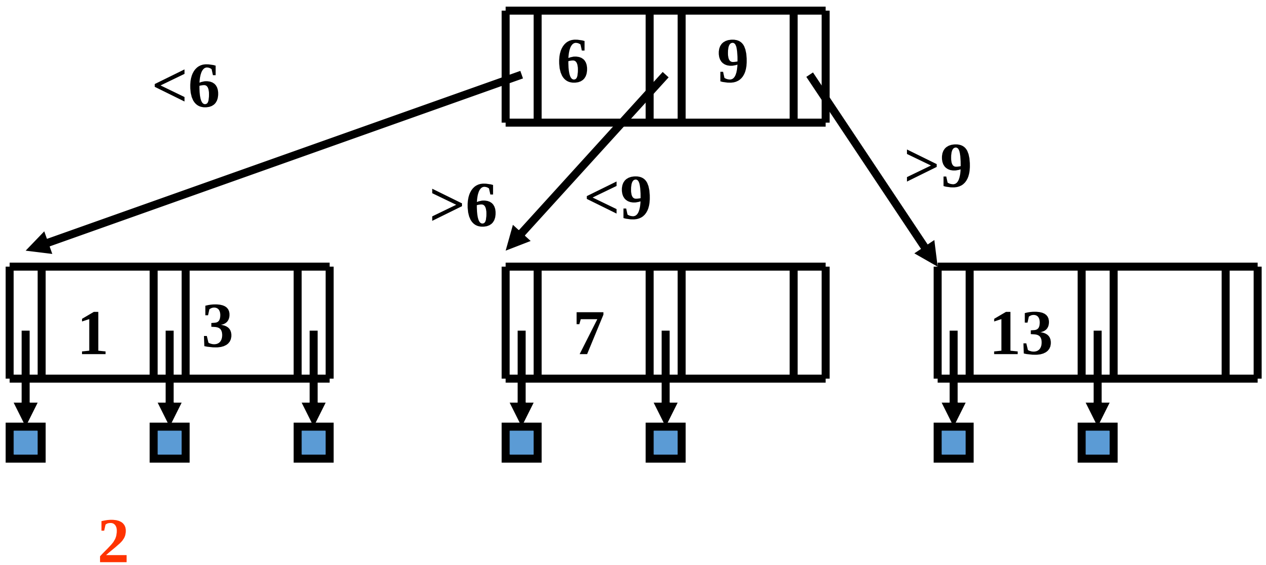
B-trees: Insertion

Tree T0; insert ‘8’



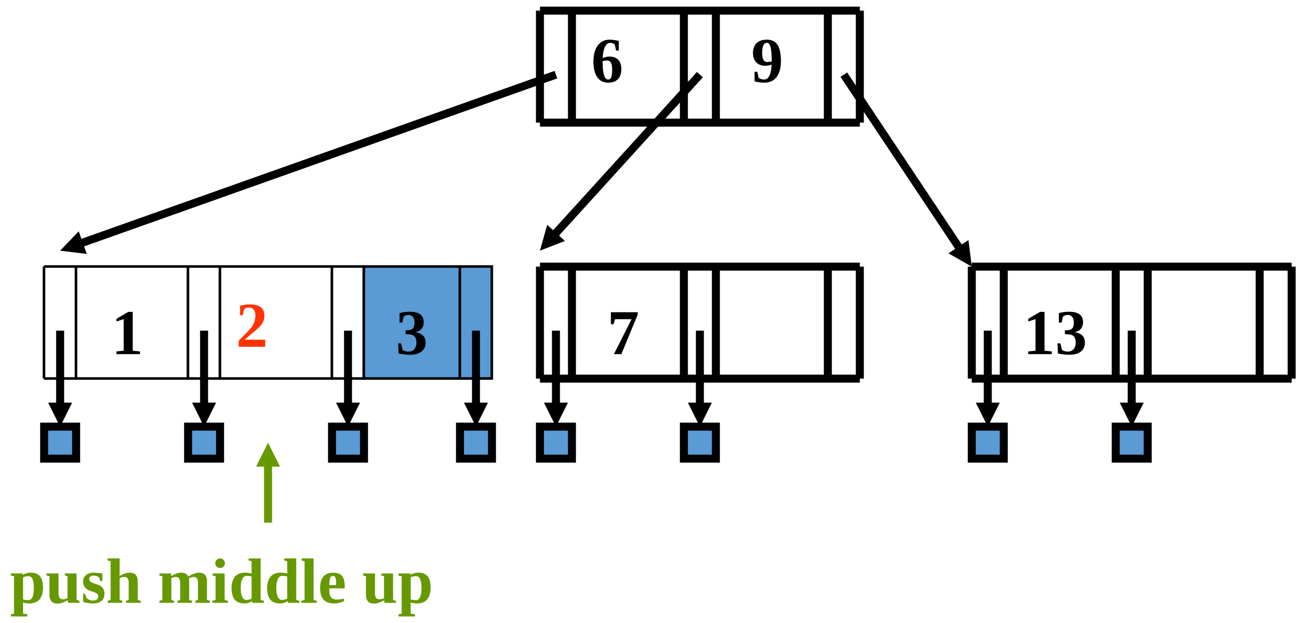
B-trees: Insertion

Hardest case: Tree T0; insert '2'



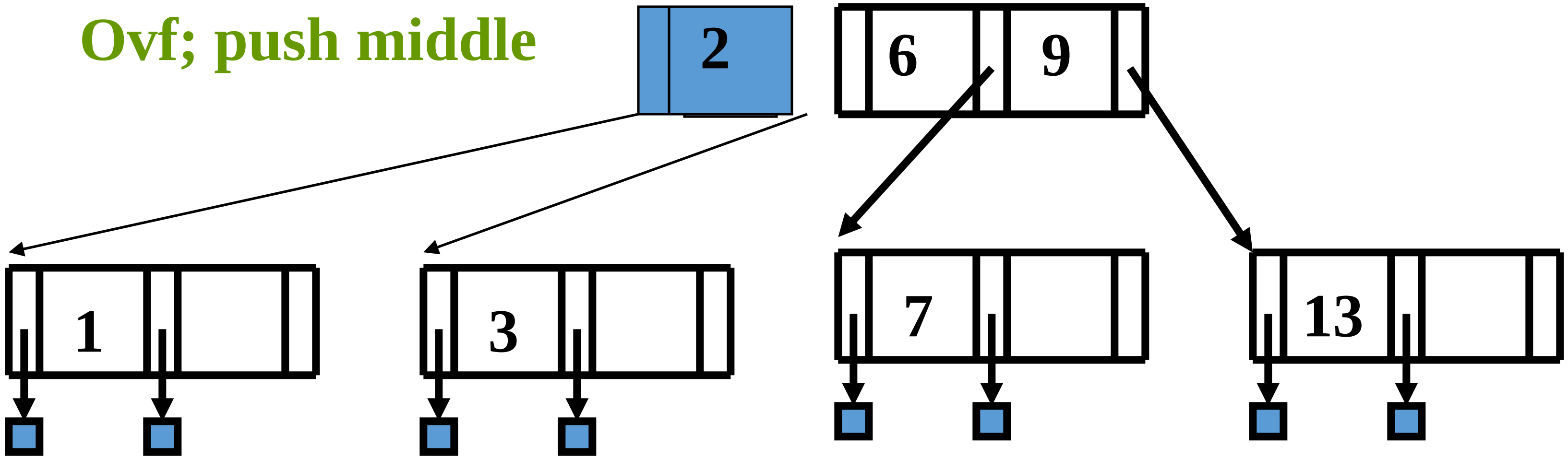
B-trees: Insertion

Hardest case: Tree T0; insert '2'



B-trees: Insertion

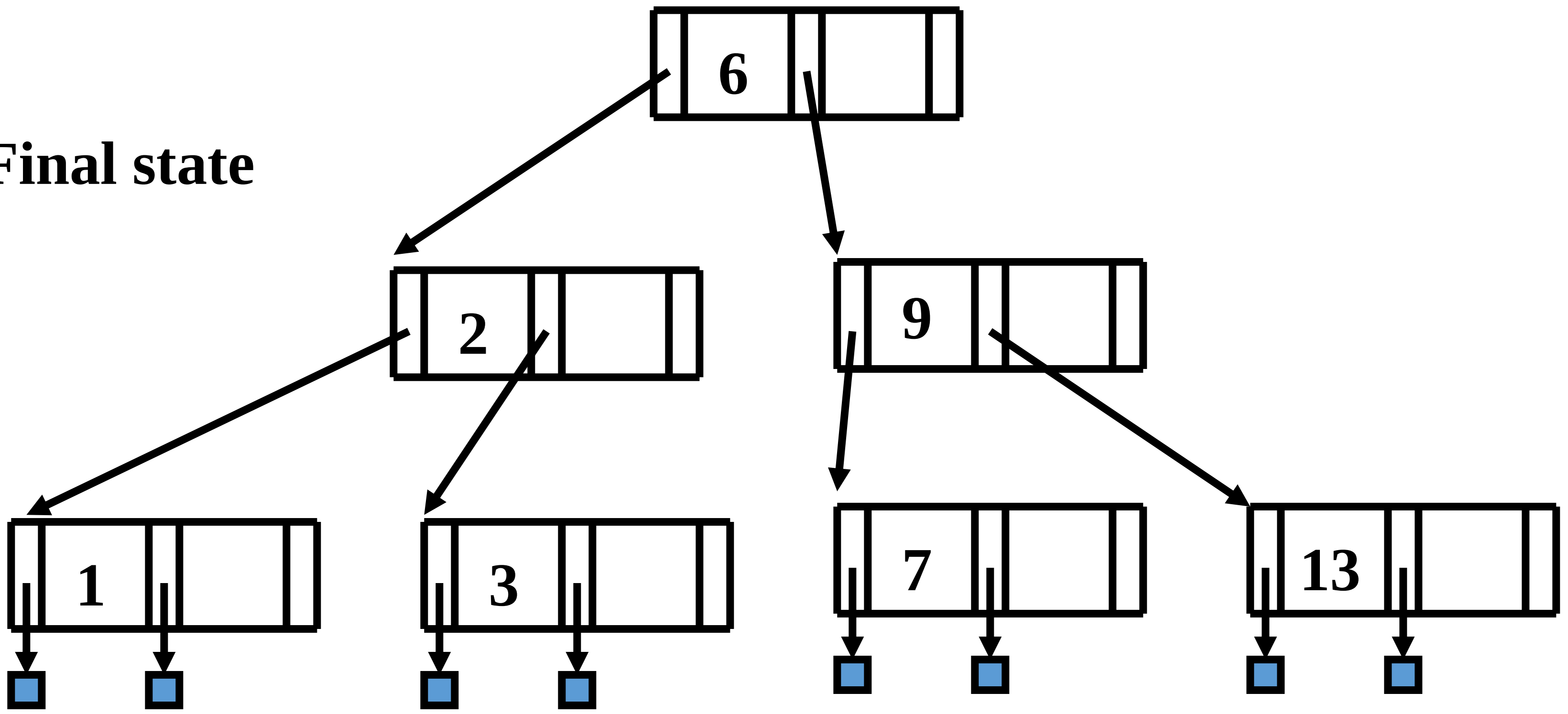
Hardest case: Tree T0; insert ‘2’



B-trees: Insertion

Hardest case: Tree T0; insert ‘2’

Final state

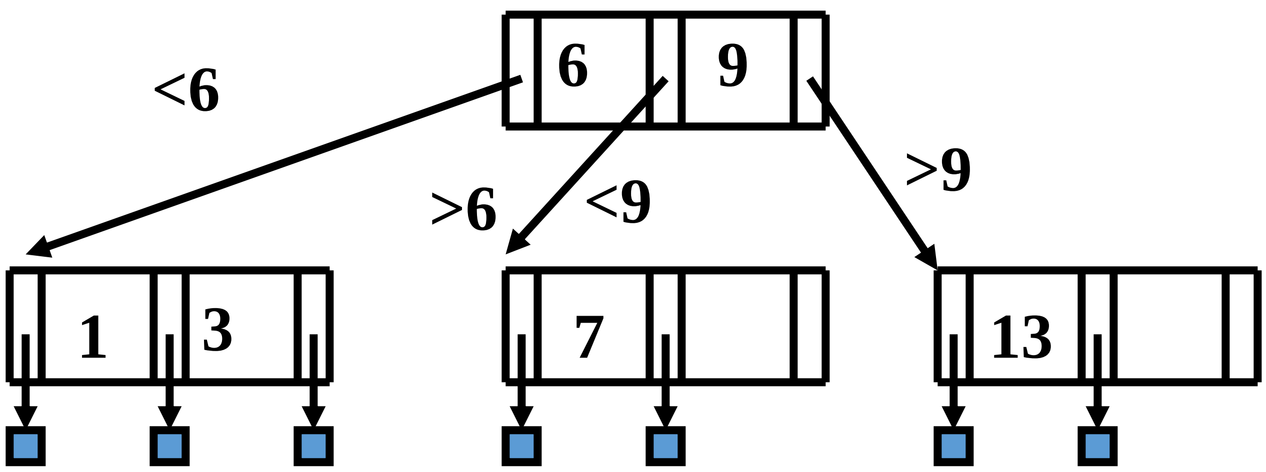


B-trees: Deletion

- Case1: delete a key at a leaf – no underflow
- Case2: delete non-leaf key – no underflow
- Case3: delete leaf-key; underflow, and ‘rich sibling’
- Case4: delete leaf-key; underflow, and ‘poor sibling’

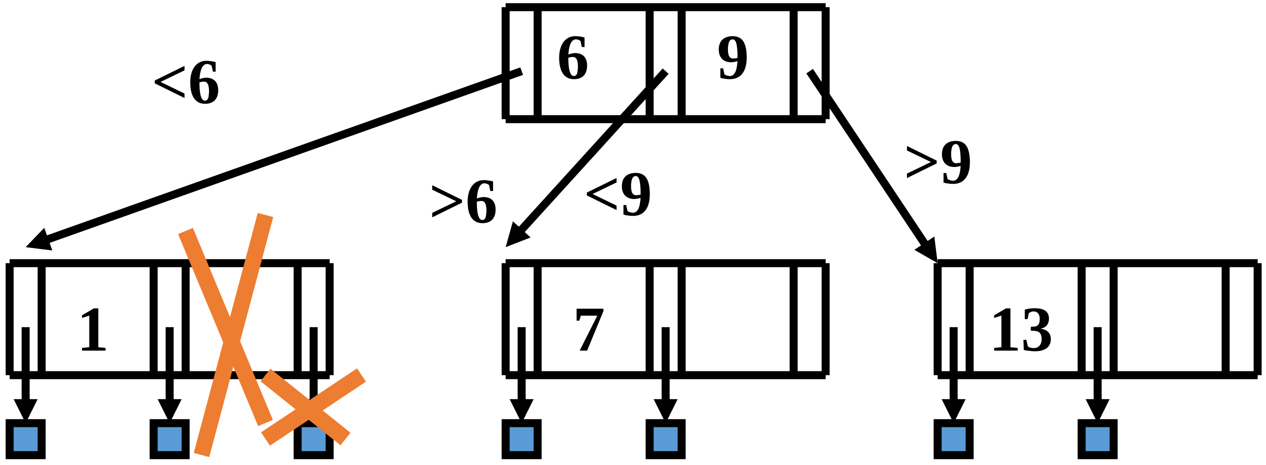
B-trees: Deletion

Easiest case: Tree T0; delete '3'



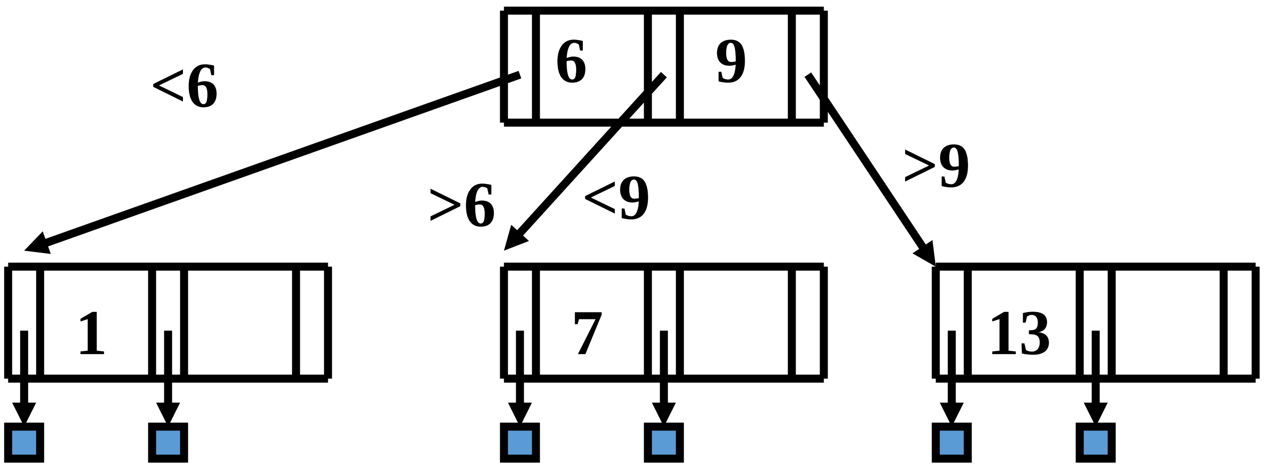
B-trees: Deletion

Easiest case: Tree T0; delete '3'



B-trees: Deletion

Easiest case: Tree T0; delete '3'

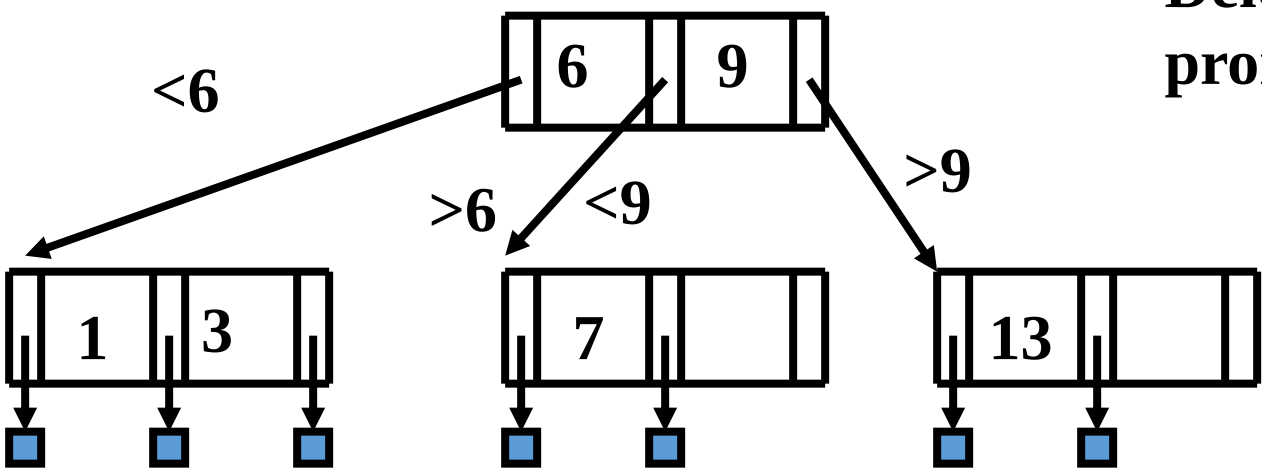


B-trees: Deletion

- Case1: delete a key at a leaf – no underflow
- Case2: delete non-leaf key – no underflow
- Case3: delete leaf-key; underflow, and ‘rich sibling’
- Case4: delete leaf-key; underflow, and ‘poor sibling’

B-trees: Deletion

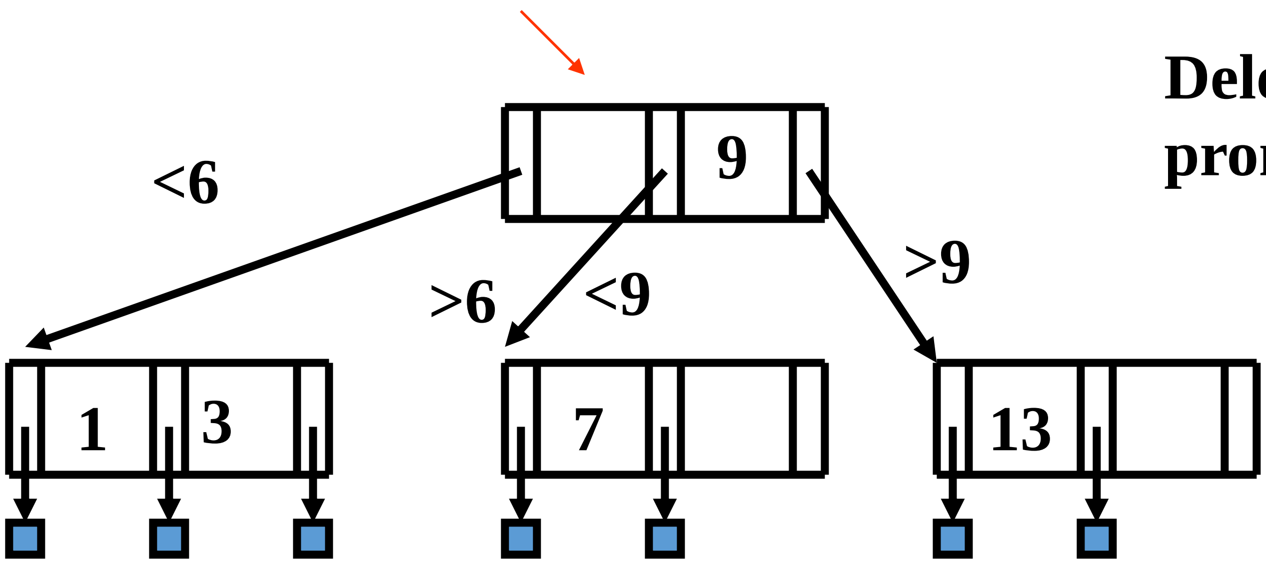
- Case2: delete a key at a non-leaf – no underflow (eg., delete 6 from T0)



Delete & promote, ie:

B-trees: Deletion

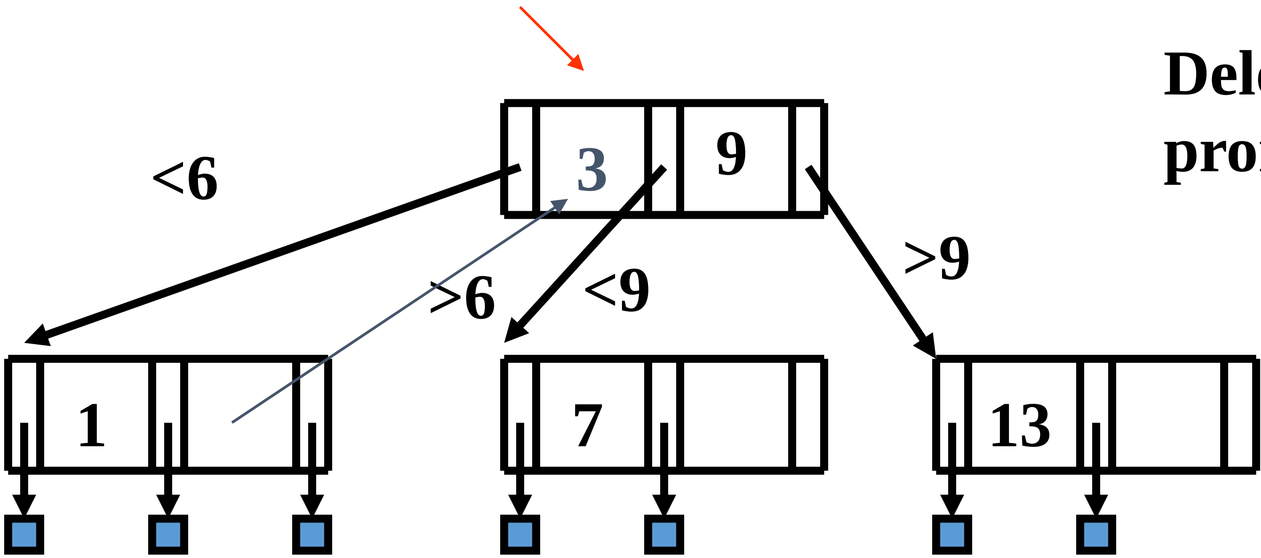
- Case2: delete a key at a non-leaf – no underflow (eg., delete 6 from T0)



Delete & promote, ie:

B-trees: Deletion

- Case2: delete a key at a non-leaf – no underflow (eg., delete 6 from T0)

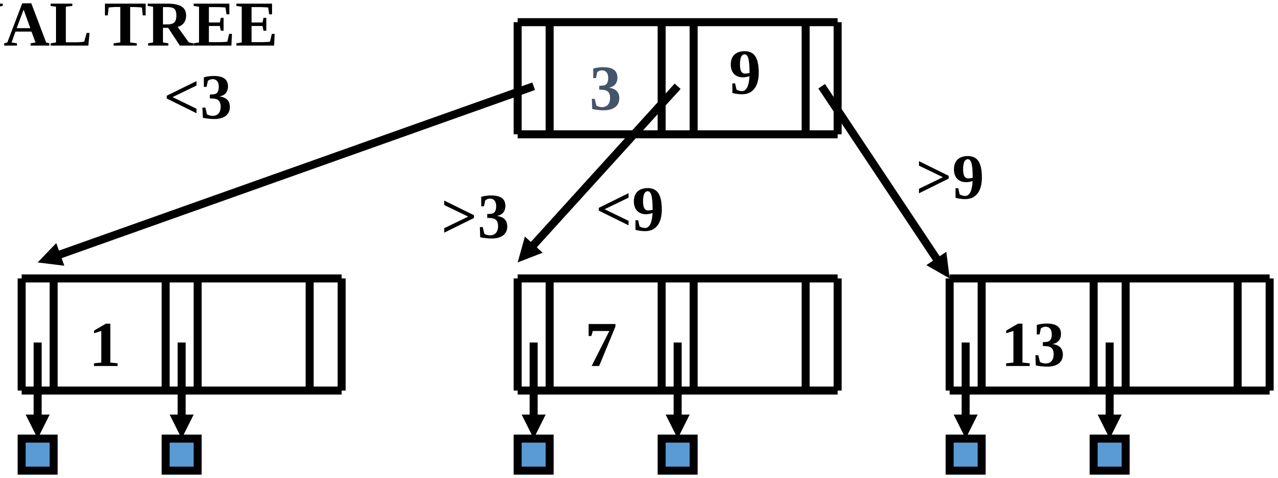


Delete & promote, ie:

B-trees: Deletion

- Case2: delete a key at a non-leaf – no underflow (eg., delete 6 from T0)

FINAL TREE



B-trees: Deletion

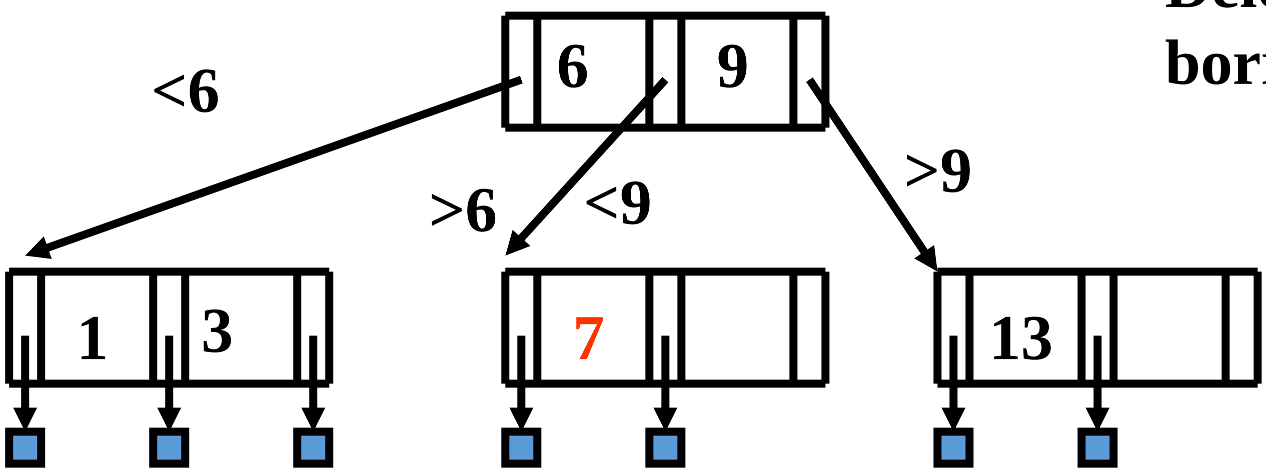
- Case2: delete a key at a non-leaf – no underflow (eg., delete 6 from T0)
- Q: How to promote?
- A: pick the largest key from the left sub-tree (or the smallest from the right sub-tree)
- Observation: every deletion eventually becomes a deletion of a leaf key

B-trees: Deletion

- Case1: delete a key at a leaf – no underflow
- Case2: delete non-leaf key – no underflow
- Case3: delete leaf-key; underflow, and ‘rich sibling’
- Case4: delete leaf-key; underflow, and ‘poor sibling’

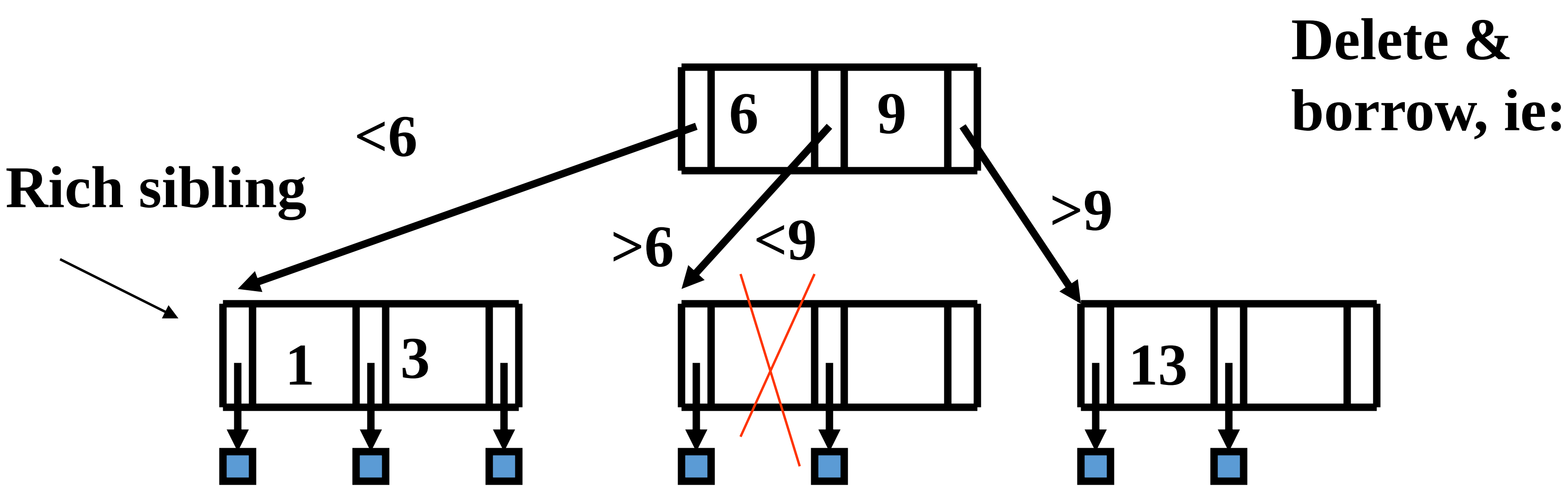
B-trees: Deletion

- Case3: underflow & 'rich sibling' (eg., delete 7 from T0)



B-trees: Deletion

- Case3: underflow & 'rich sibling' (eg., delete 7 from T0)



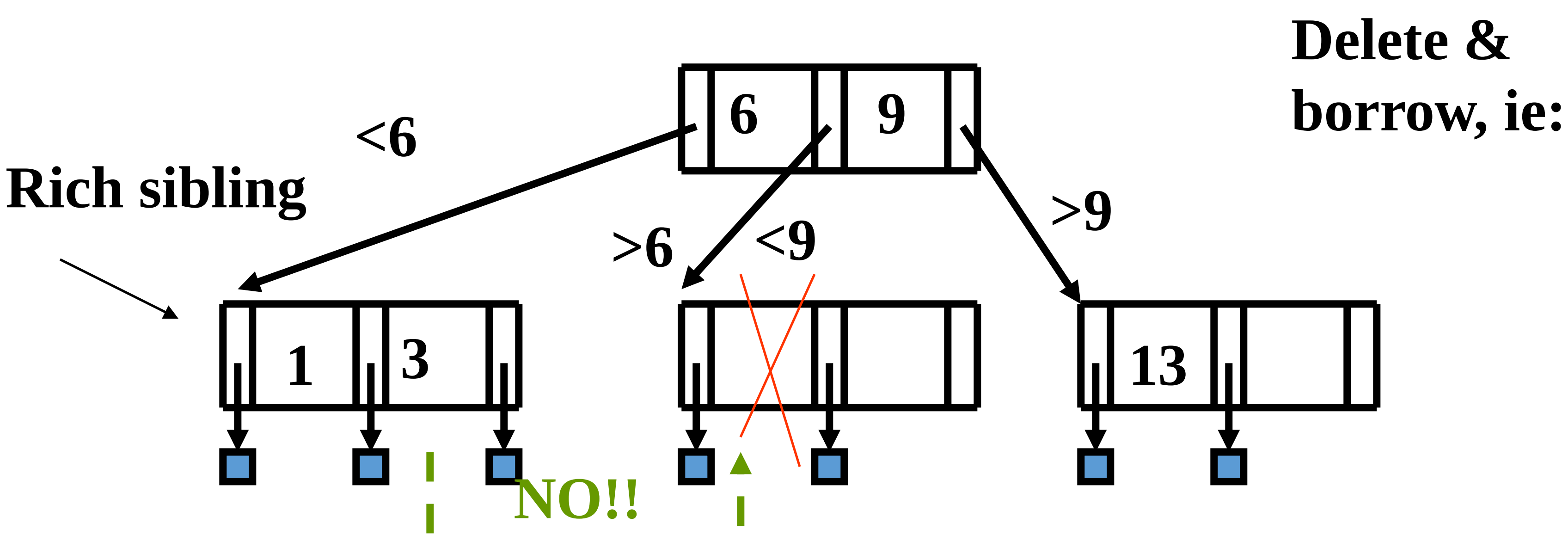
B-trees: Deletion

- Case3: underflow & 'rich sibling'

- 'rich' = can give a key, without underflowing
- 'borrowing' a key: THROUGH the PARENT!

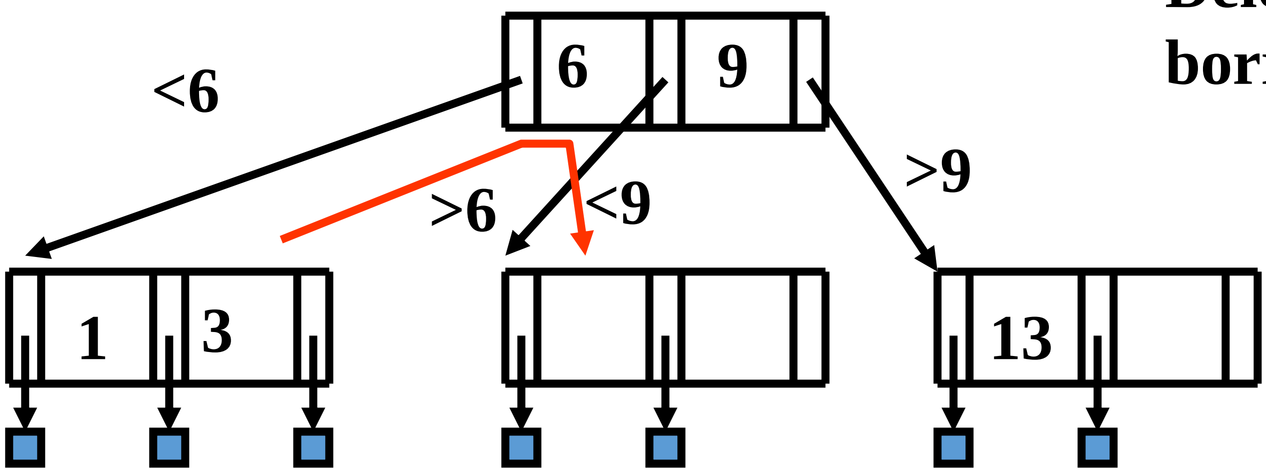
B-trees: Deletion

- Case3: underflow & 'rich sibling' (eg., delete 7 from T0)



B-trees: Deletion

- Case3: underflow & 'rich sibling' (eg., delete 7 from T0)

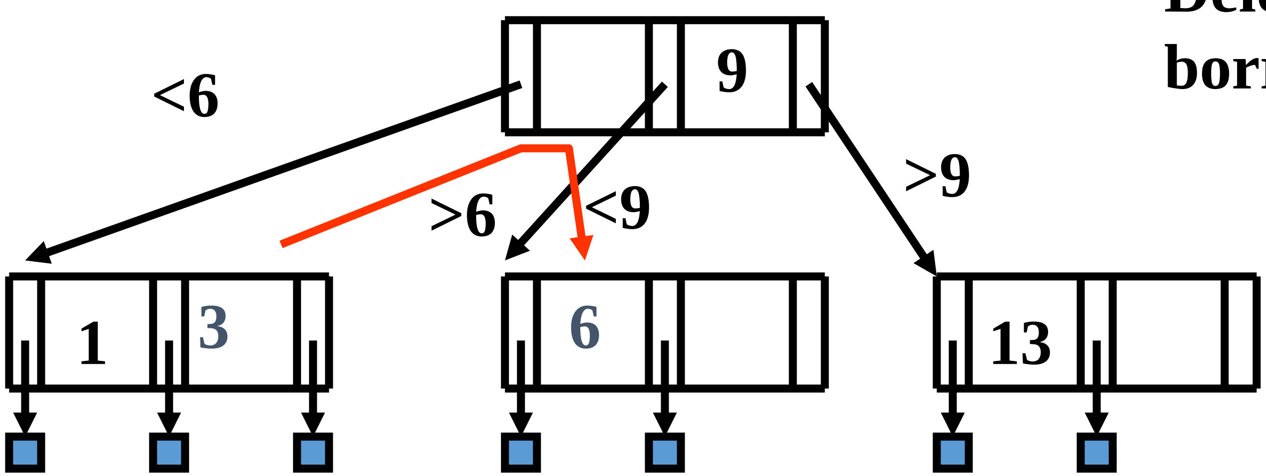


Delete & borrow, ie:

B-trees: Deletion

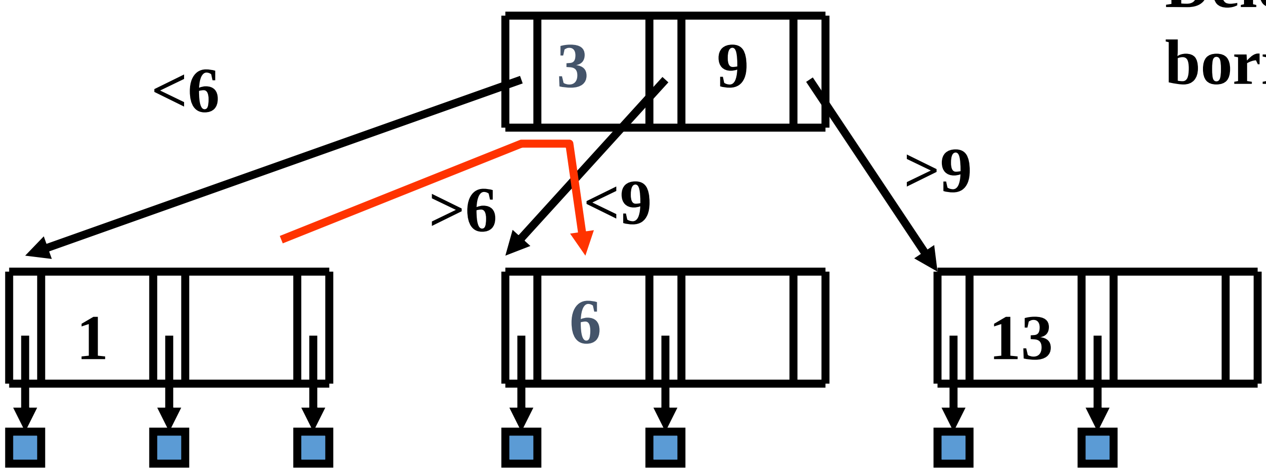
- Case3: underflow & 'rich sibling' (eg., delete 7 from T0)

Delete & borrow, ie:



B-trees: Deletion

- Case3: underflow & 'rich sibling' (eg., delete 7 from T0)

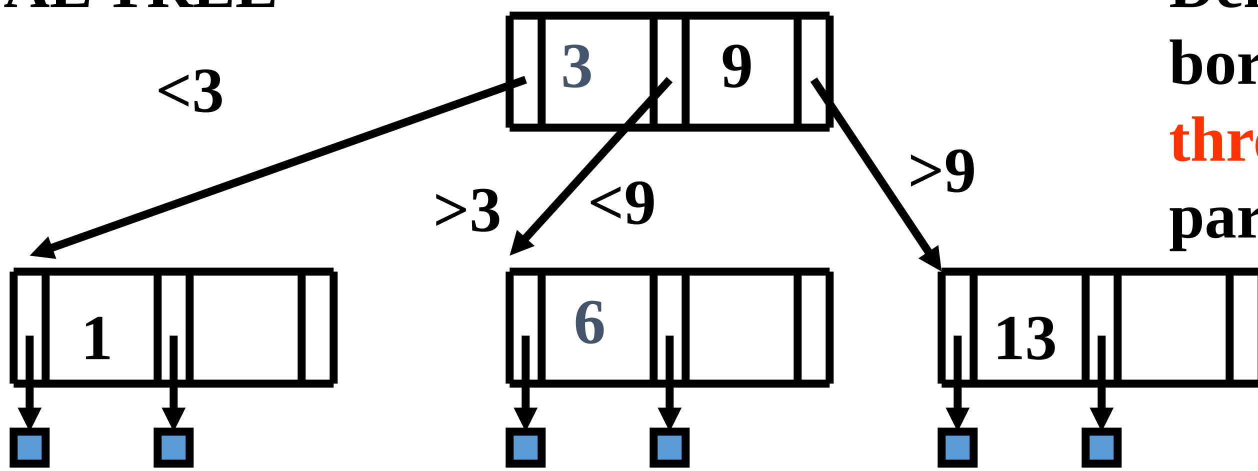


Delete & borrow, ie:

B-trees: Deletion

- Case3: underflow & 'rich sibling' (eg., delete 7 from T0)

FINAL TREE



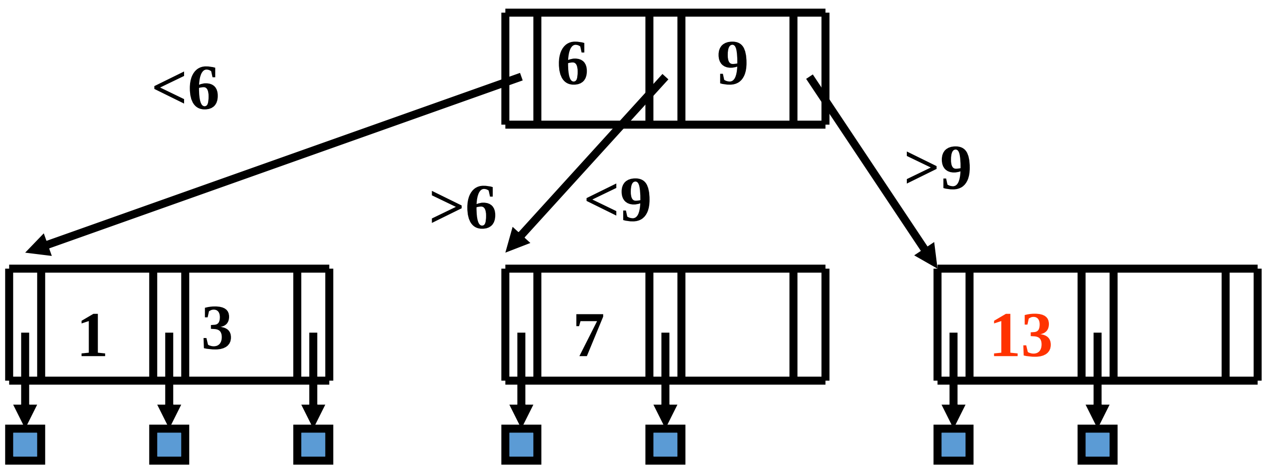
Delete & borrow, through the parent

B-trees: Deletion

- Case1: delete a key at a leaf – no underflow
- Case2: delete non-leaf key – no underflow
- Case3: delete leaf-key; underflow, and ‘rich sibling’
- Case4: delete leaf-key; underflow, and ‘poor sibling’

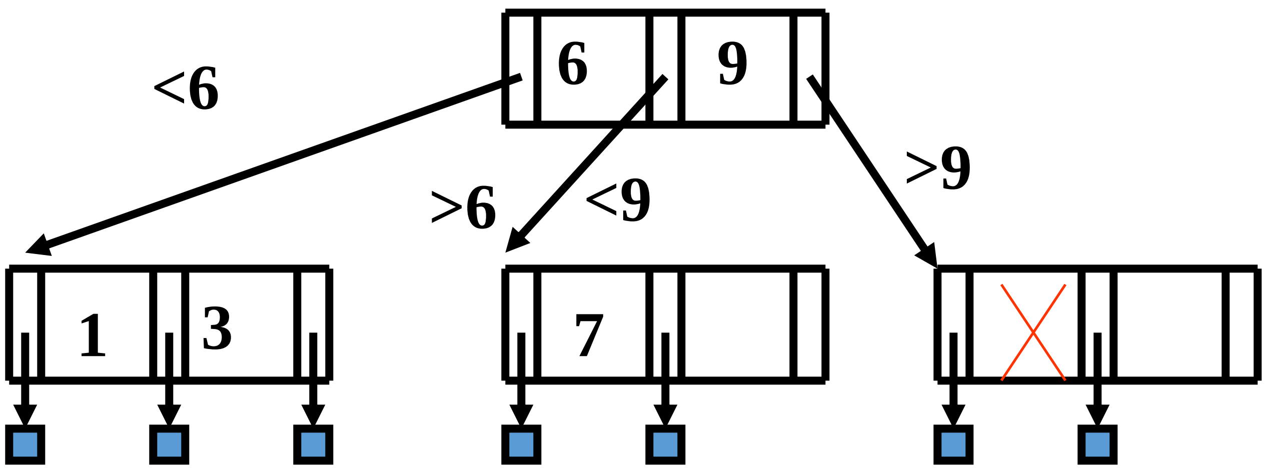
B-trees: Deletion

- Case4: underflow & 'poor sibling' (eg., delete 13 from T0)



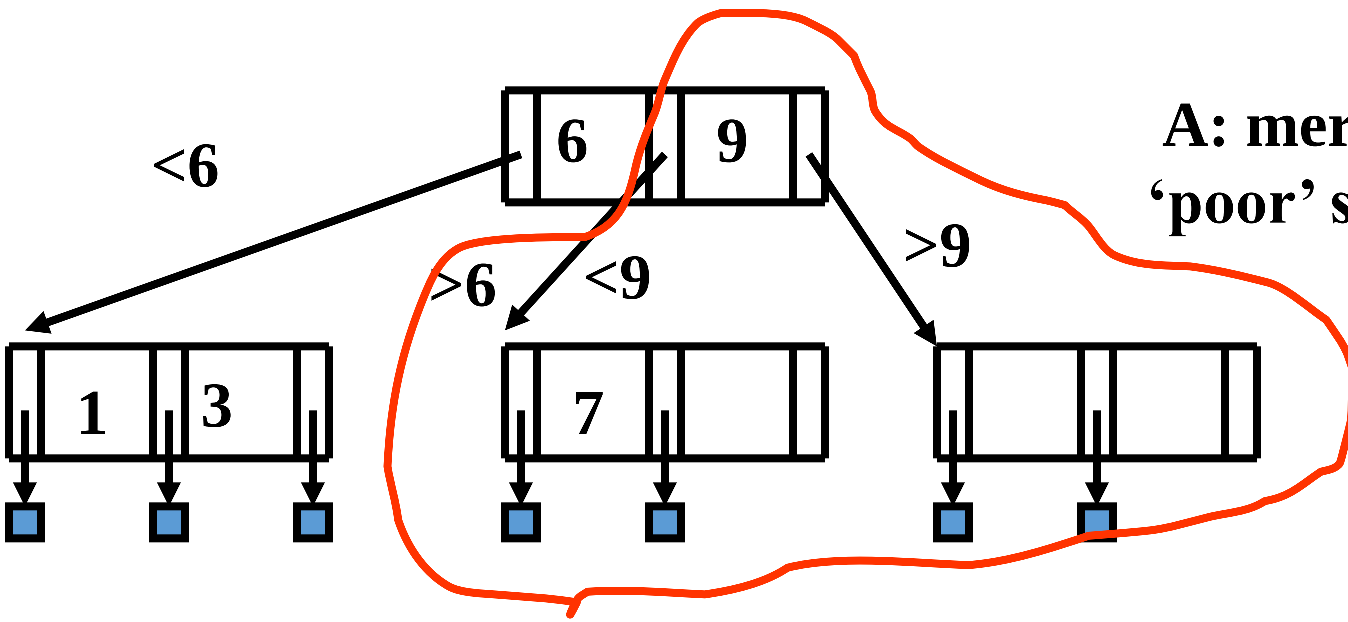
B-trees: Deletion

- Case4: underflow & 'poor sibling' (eg., delete 13 from T0)



B-trees: Deletion

- Case4: underflow & ‘poor sibling’ (eg., delete 13 from T0)



A: merge w/ ‘poor’ sibling

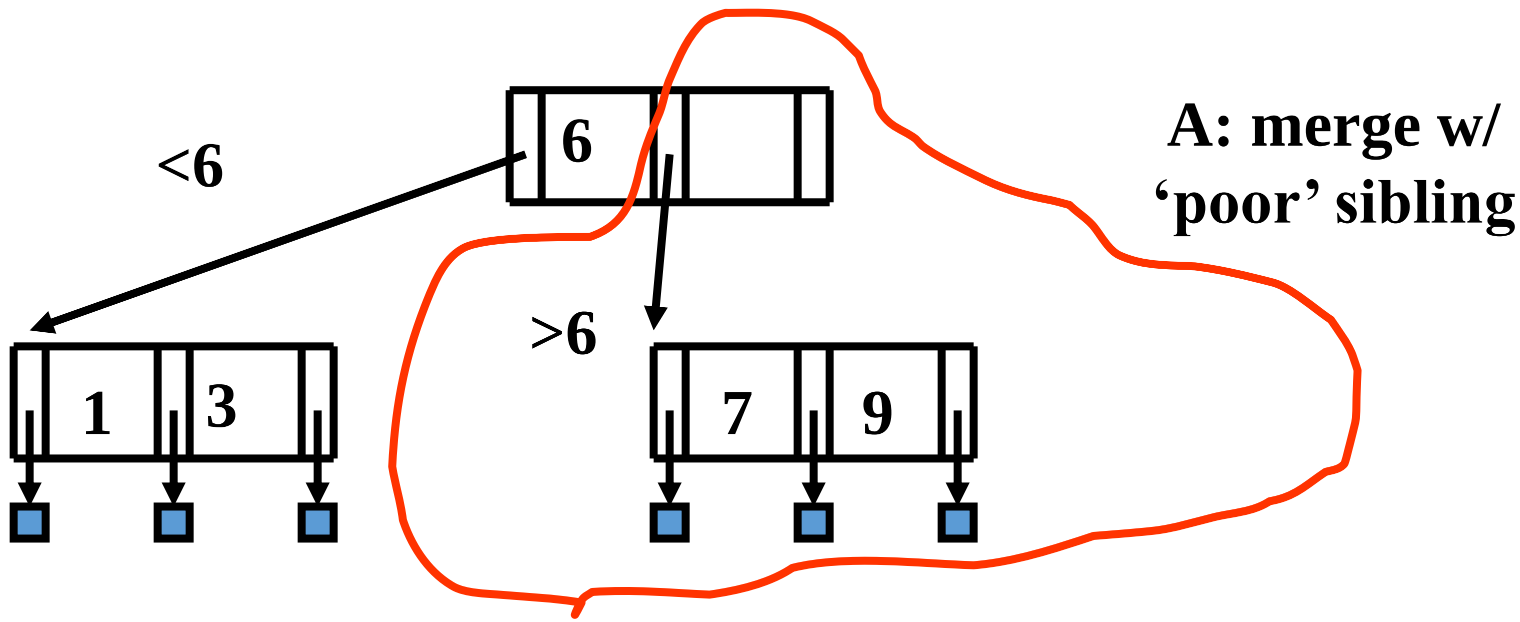
B-trees: Deletion

- Case4: underflow & ‘poor sibling’ (eg., delete 13 from T0)

- Merge, by pulling a key from the **parent**
- exact reversal from insertion: ‘split and push up’, vs. ‘merge and pull down’

B-trees: Deletion

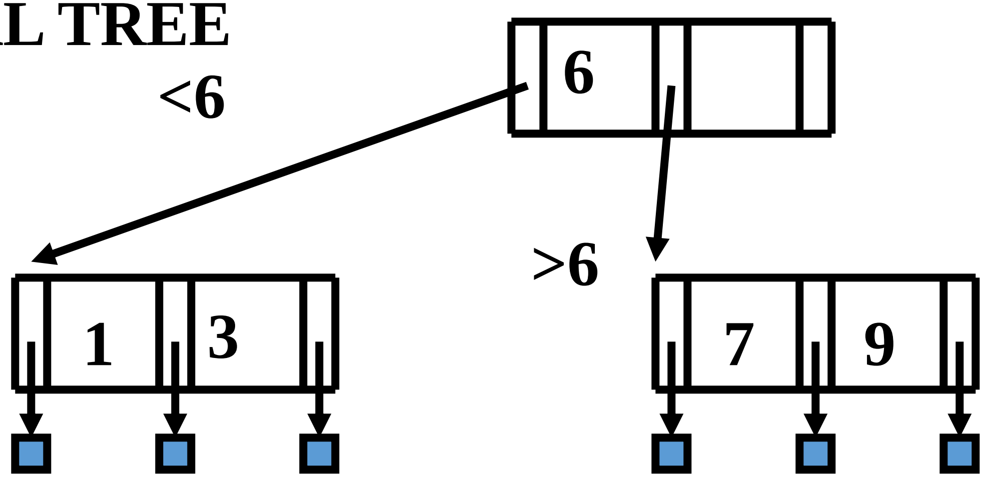
- Case4: underflow & ‘poor sibling’ (eg., delete 13 from T0)



B-trees: Deletion

- Case4: underflow & 'poor sibling' (eg., delete 13 from T0)

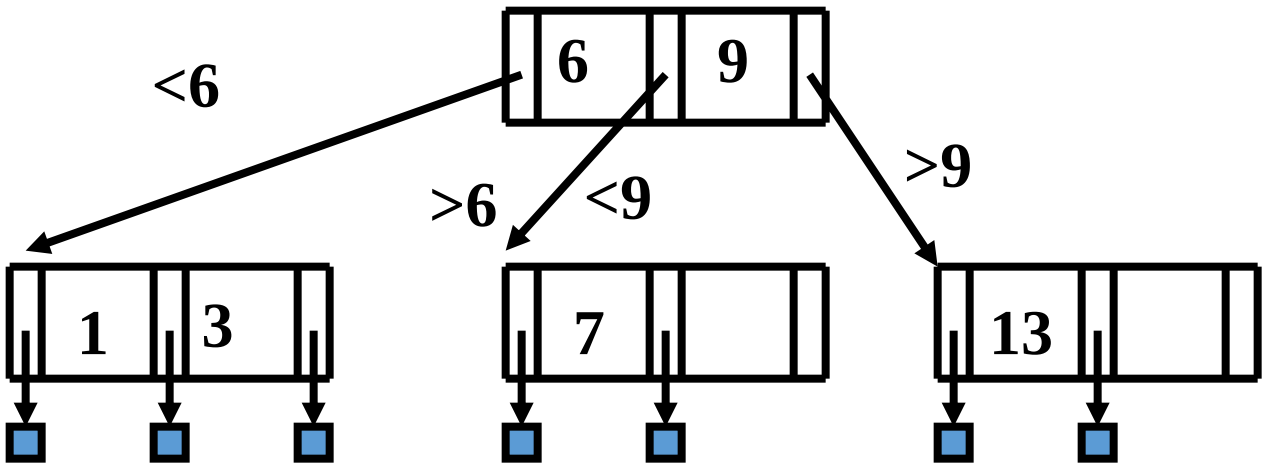
FINAL TREE



B^+-Tree

B+-trees: Motivation

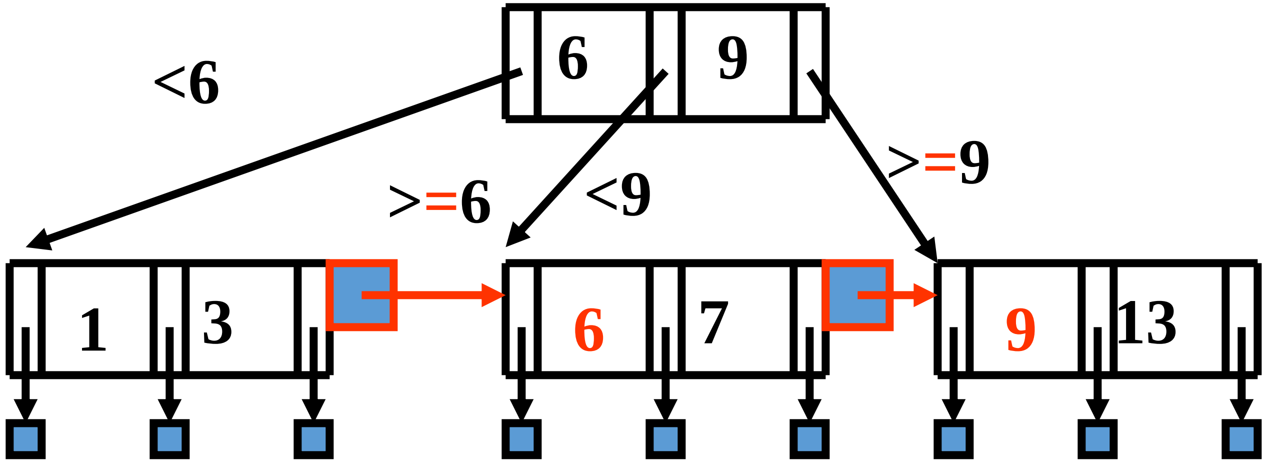
if we want to store the whole record with the key –> problems (what?)



Solution: B^+-trees

- They string all leaf nodes together
- AND
- replicate keys from non-leaf nodes, to make sure every key appears at the leaf level

B+ trees



R-Tree

Spatial Data

Name	ID	Type	Phone	Location	Grade
Marios Pizza	1	ITA	888-1212	244,365	D
Joes Bugers	2	US	848-1298	34,764	A
Tinas Mexican	3	MEX	878-1333	123,32	A
Sues Pasta	4	ITA	878-1342	876,65	B

- Given such a database we can easily answer queries by using SQL, such as
  - *List all Mexican restaurants.*
  - *List all Grade **A** restaurants.*
- However, classic databases do not allow queries such as
  - *List all Mexican restaurants within five miles of UCR*
  - *List the pizza restaurant nearest to 91 and 60.*
- These kinds of queries are called *spatial queries*
  - Nearest neighbor queries
  - Range queries
  - Spatial joins

Indexing Spatial Data

- So, we call always index 1-dimensional data (*if you can sort it, you can index it*), such that we can answer 1-nearest neighbor queries by accessing just O(log(*n*)) of the database. (*n* is the number of items in the database). (i.e. the B-tree)
- But we cannot sort 2 dimensional data…
- Solution: **R-Tree**
  - introduced by Guttman in the 1984 SIGMOD conference.

R-Trees

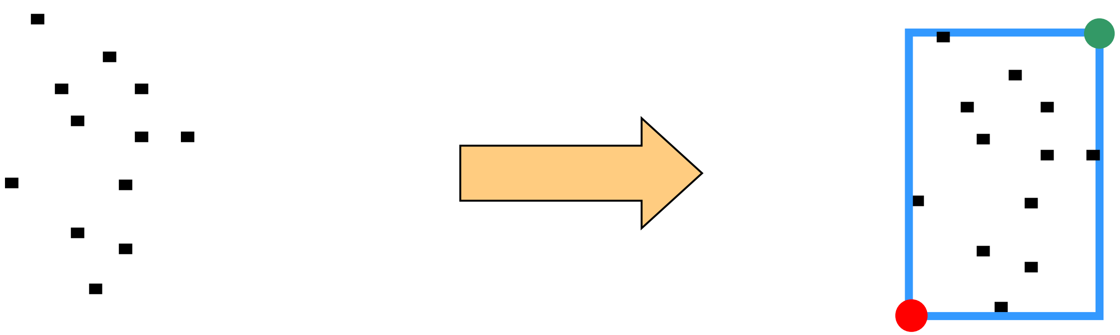
- **R-trees** are a N-dimensional extension of B^+-trees, useful for indexing sets of rectangles and other polygons.
- Supported in many modern database systems, along with variants like R^+ -trees and R*-trees.
- Basic idea: generalize the notion of a one-dimensional interval associated with each B+ -tree node to an
  N-dimensional interval, that is, an N-dimensional rectangle.
- Will consider only the two-dimensional case ($N = 2$)
  - generalization for $N > 2$ is straightforward, although R-trees work well only for relatively small N

R-Trees

- A rectangular **bounding box** is associated with each tree node.
  - Bounding box of a leaf node is a minimum sized rectangle that contains all the rectangles/polygons associated with the leaf node.
  - The bounding box associated with a non-leaf node contains the bounding box associated with all its children.
  - Bounding box of a node serves as its key in its parent node (if any)
  - *Bounding boxes of children of a node are allowed to overlap*
- A polygon is stored only in one node, and the bounding box of the node must contain the polygon
  - The storage efficiency or R-trees is better than that of k-d trees or quadtrees since a polygon is stored only once

MBR

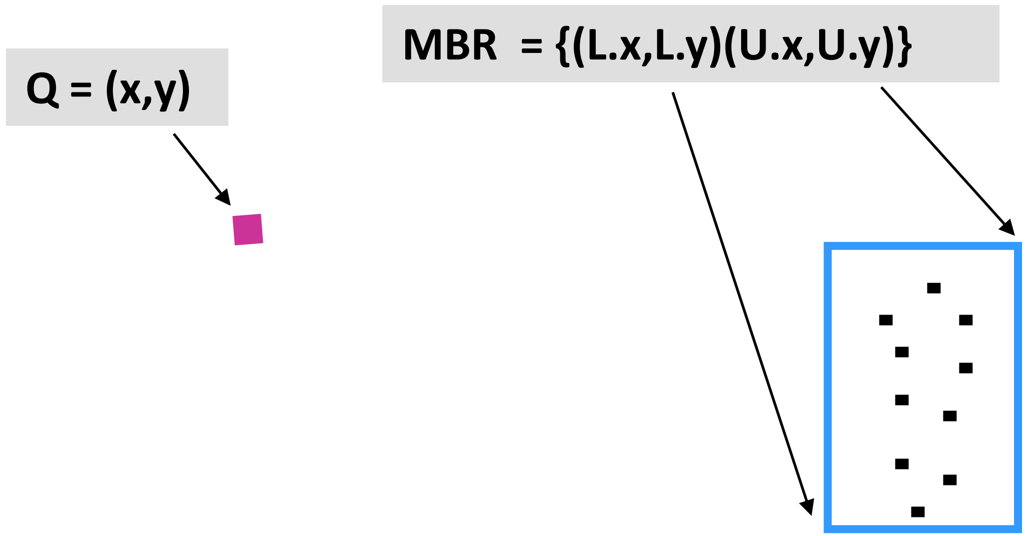
- Suppose we have a cluster of points in 2-D space...
- We can build a "box" around points. The smallest box (which is axis parallel) that contains all the points is called a **Minimum Bounding Rectangle** (MBR)



MBR = {(L.x,L.y)(U.x,U.y)}

MINDIST

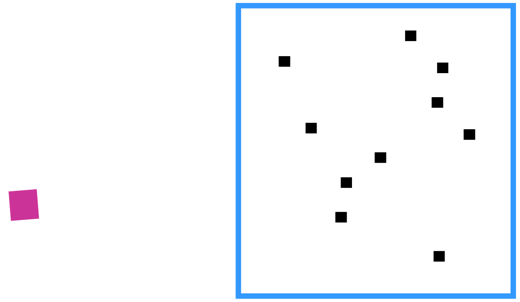
- The formula for the distance between a point and the closest possible point within an MBR



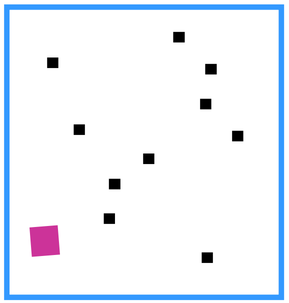
MINDIST(Q,MBR)

if L.x < x < U.x **and** L.y < y < U.y **then** 0
elseif L.x < x < U.x **then** min($(L.y - y)^2$, $(U.y - y)^2$)
elseif ….

MINDIST Example



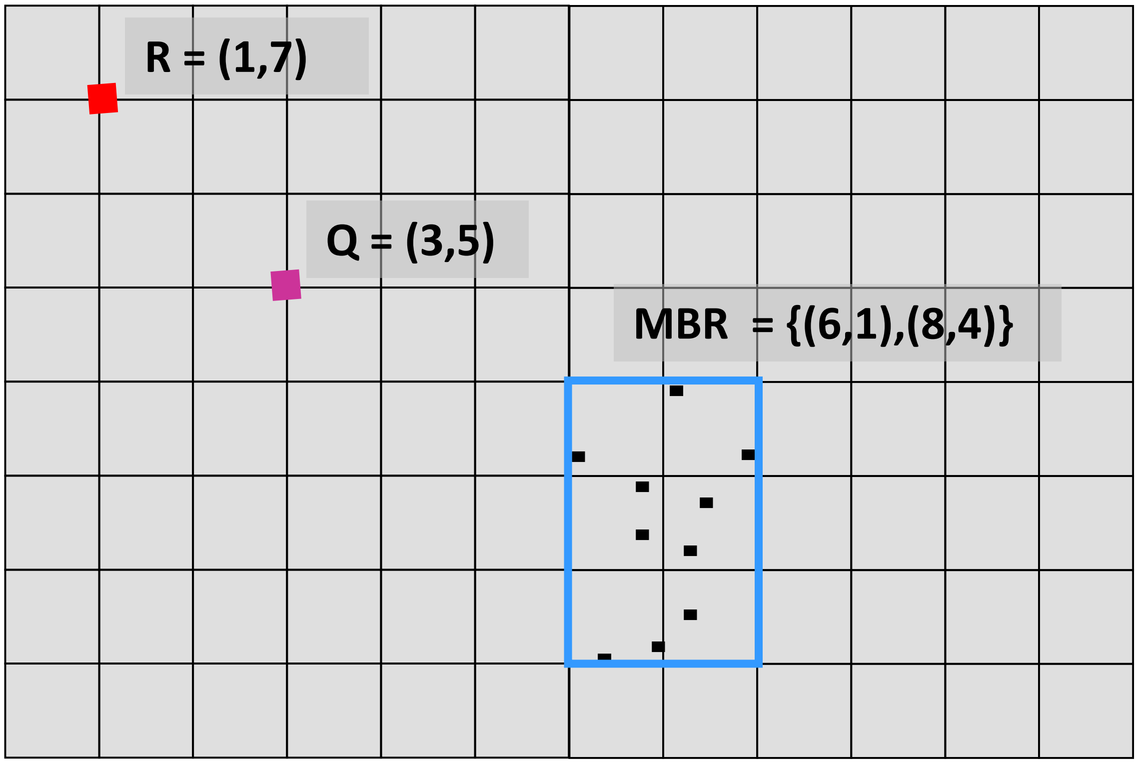
MINDIST(*point*, *MBR*) = 5



MINDIST(*point*, *MBR*) = 0

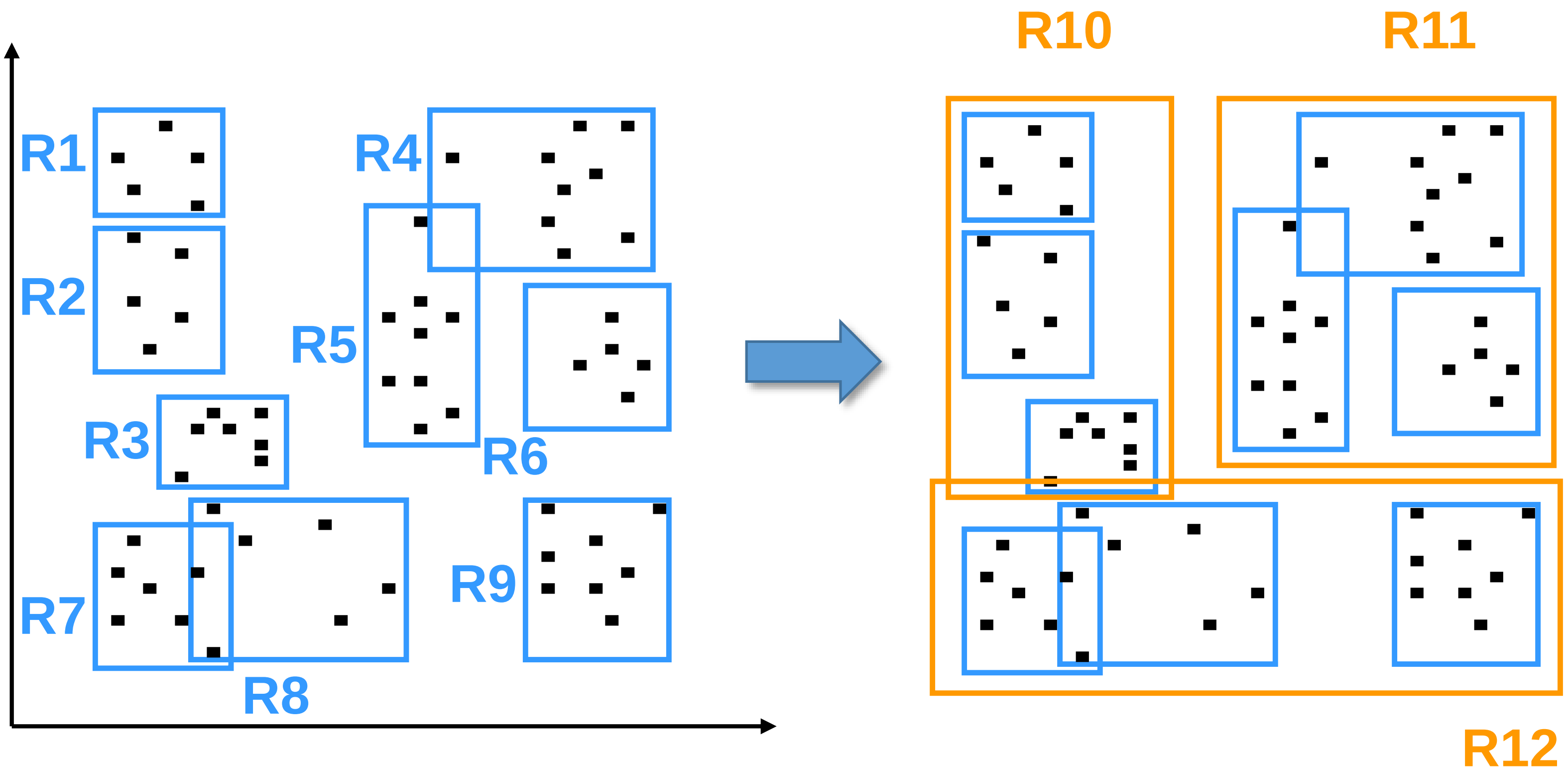
MINDIST Example

- Suppose we have a query point *Q* and one known point *R*. Could any of the points in the MBR be closer to *Q* than *R* is?



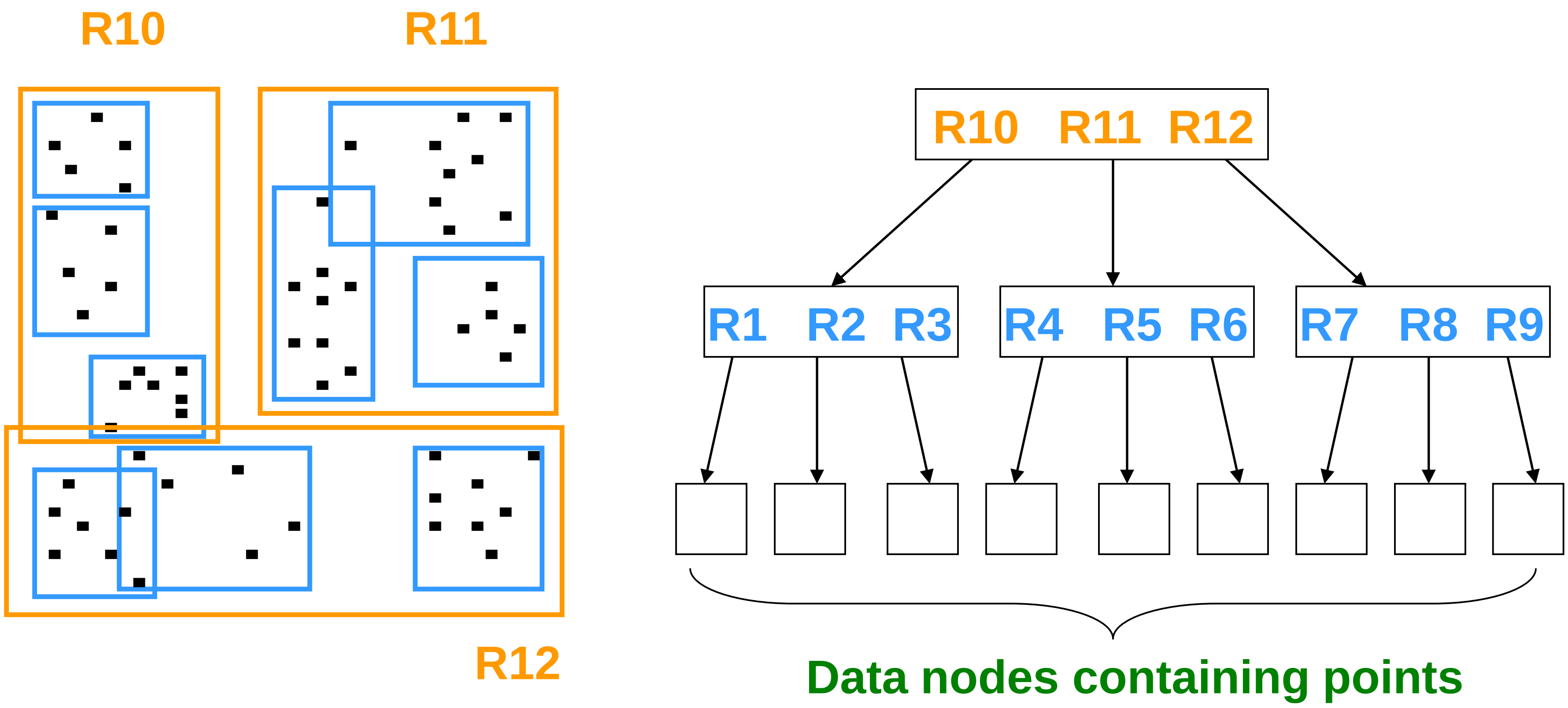
Constructing MBR

- Each MBR can be represented with just two points. The lower left corner, and the upper right corner.
- We can further recursively group MBRs into larger MBRs….



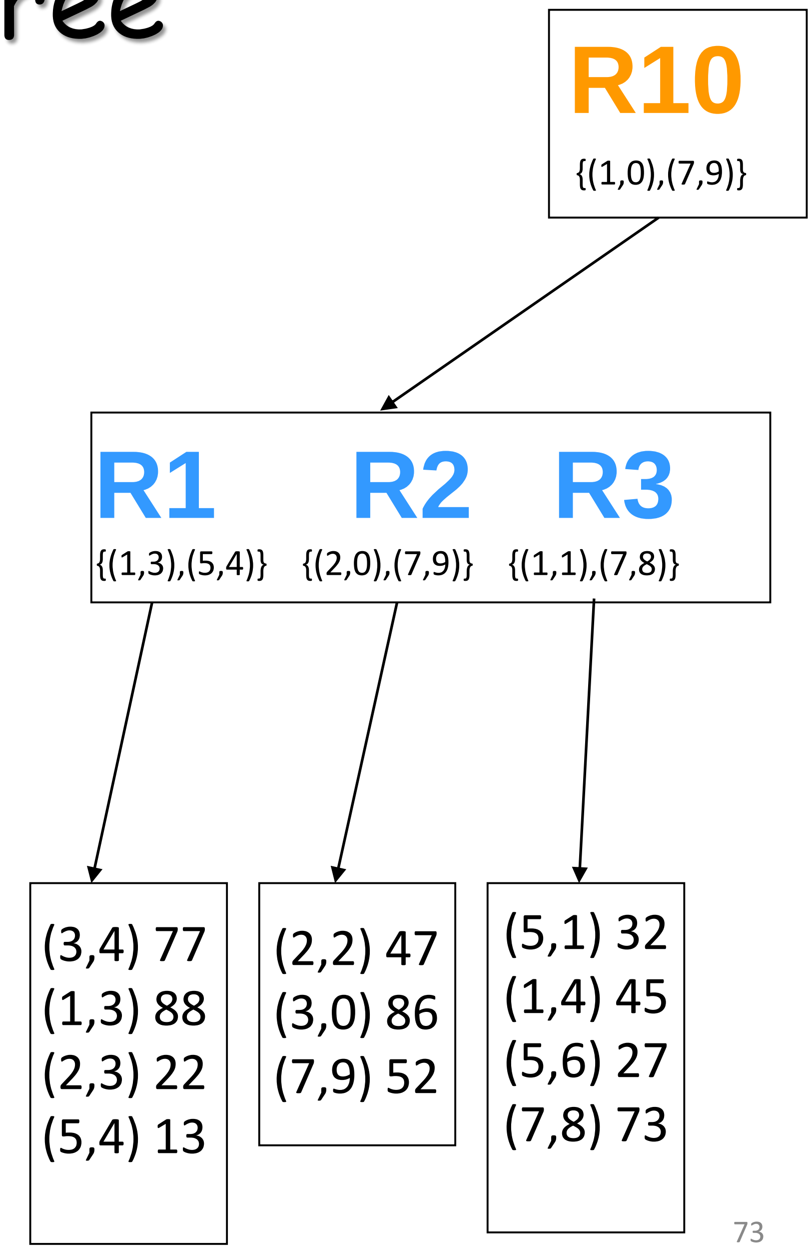
Constructing R-Tree

- ...these nested MBRs are organized as a tree (called a spatial access tree or a multidimensional tree).



Constructing R-Tree

- At the leave nodes we have the location, and a pointer to the record in question
- At the internal nodes, we just have MBR information



Search in R-Tree

- To find data items (rectangles/polygons) intersecting (overlaps) a given query point/region, do the following, starting from the root node:
  - If the node is a leaf node, output the data items whose keys intersect the given query point/region.
  - Else, for each child of the current node whose bounding box overlaps the query point/region, recursively search the child
- Can be very inefficient in worst case since multiple paths may need to be searched
  - but works acceptably in practice.
- Simple extensions of search procedure to handle predicates *contained-in* and *contains*

Insertion in R-Tree

- To insert a data item:
  - Find a leaf to store it, and add it to the leaf
    - To find leaf, follow a child (if any) whose bounding box contains bounding box of data item, else child whose overlap with data item bounding box is maximum
  - Handle overflows by splits (as in B+ -trees)
    - Split procedure is different though (see below)
  - Adjust bounding boxes starting from the leaf upwards
- Split procedure:
  - Goal: divide entries of an overfull node into two sets such that the bounding boxes have minimum total area
    - This is a heuristic. Alternatives like minimum overlap are possible
  - Finding the “best” split is expensive, use heuristics instead
    - See next slide

Splitting an R-Tree Node

- **Quadratic split** divides the entries in a node into two new nodes as follows
  1. Find pair of entries with "maximum separation"
     - that is, the pair such that the bounding box of the two would has the maximum wasted space (area of bounding box – sum of areas of two entries)
  2. Place these entries in two new nodes
  3. Repeatedly find the entry with "maximum preference" for one of the two new nodes, and assign the entry to that node
     - Preference of an entry to a node is the increase in area of bounding box if the entry is added to the *other* node
  4. Stop when half the entries have been added to one node
     - Then assign remaining entries to the other node
- Cheaper **linear split** heuristic works in time linear in number of entries,
  - Cheaper but generates slightly worse splits.

Deleting in R-Trees

- Deletion of an entry in an R-tree done much like a B^+-tree deletion.
  - In case of underfull node, borrow entries from a sibling if possible, else merging sibling nodes
  - Alternative approach removes all entries from the underfull node, deletes the node, then reinserts all entries
  - As always, deletion tends to be rarer than insertion for many real world databases.

End of Chapter 6