Mid-term Summary

- ~ 70% AB
  - How we are going to evaluate in the final?
- Assignment
  - How weekly? – impact:~88% vs. ~34%
- Feedback
  - Assignment results, separate, acoustic, practical, materials in advance
- Logistics: lecture vs. seminar
- Cheating

Indexing

Why Indexes?

- Sometimes, we want to retrieve records by specifying the *values in one or more fields*, e.g.,
  - Find all students in the "CS" department
  - Find all students with a gpa > 3
- An *index* on a file speeds up selections on the *search key fields* for the index (accelerate search on keys)
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - *Search key* is not the same as *key* (e.g., doesn't have to be unique).

Thus, we have:

Index

Data

- ‘data entries’ == what we store at the bottom of the index pages
- what would you use as data entries?
- (3 alternatives here)

Alternatives for Data Entry **k*** in Index

1. Actual data record (with key value **k**)

123	Smith; Main str; 412-999.9999

2. <**k**, rid of matching data record>

$40	Rid-1
$40	Rid-2

…

3. <**k**, list of rids of matching data records>

$40	Rid-1	Rid-2	…

Alternatives for Data Entry **k*** in Index

1. Actual data record (with key value **k**)
2. <**k**, rid of matching data record>
3. <**k**, list of rids of matching data records>

- Choice is orthogonal to the indexing technique.
  - Examples of indexing techniques: B+ trees, hash-based structures, R trees, …
  - Typically, index contains auxiliary info that directs searches to the desired data entries
- Can have multiple (different) indexes per file.
  - E.g. file sorted on *age*, with a hash index on *name* and a B-tree index on *salary*.

Alternatives for Data Entries (Contd.)

Alternative 1:

Actual data record (with key value **k**)

- Then, this is a clustering/sparse index, and constitutes a file organization (like Heap files or sorted files).
- At most one index on a given collection of data records can use Alternative 1.
- Saves pointer lookups but can be expensive to maintain with insertions and deletions.

Alternatives for Data Entries (Contd.)

Alternative 2

<**k**, rid of matching data record>

and Alternative 3

<**k**, list of rids of matching data records>

- Easier to maintain than Alternative 1.
- If more than one index is required on a given file, at most one index can use Alternative 1; rest must use Alternatives 2 or 3.
- Alternative 3 more compact than Alternative 2, but leads to *variable sized data entries* even if search keys are of fixed length.
- Even worse, for large rid lists the data entry would have to span multiple pages!

Outline – Tree Structured Index

- **Motivation**
- **ISAM**
- **B-trees**
- **Tree vs. Hash-based index**
- **Index organization: clustered vs. nonclustered**

Motivation

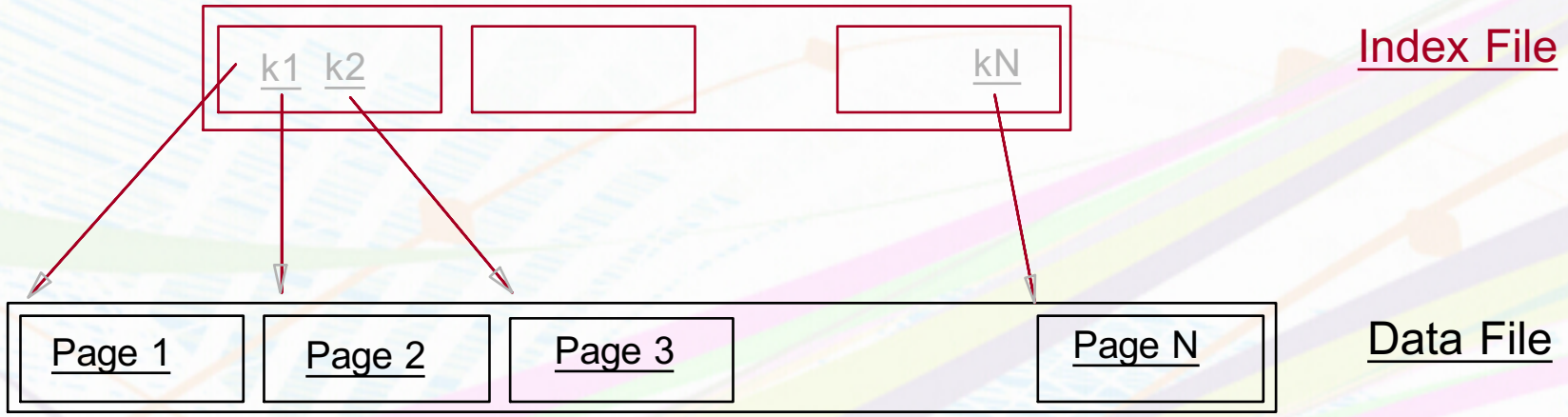
- Accelerate searches on big indexes
- How to support range searches?
- equality searches?

Range Searches

- "*Find all students with gpa > 3.0*"
- **may be slow, even on sorted file**
- **What to do?**

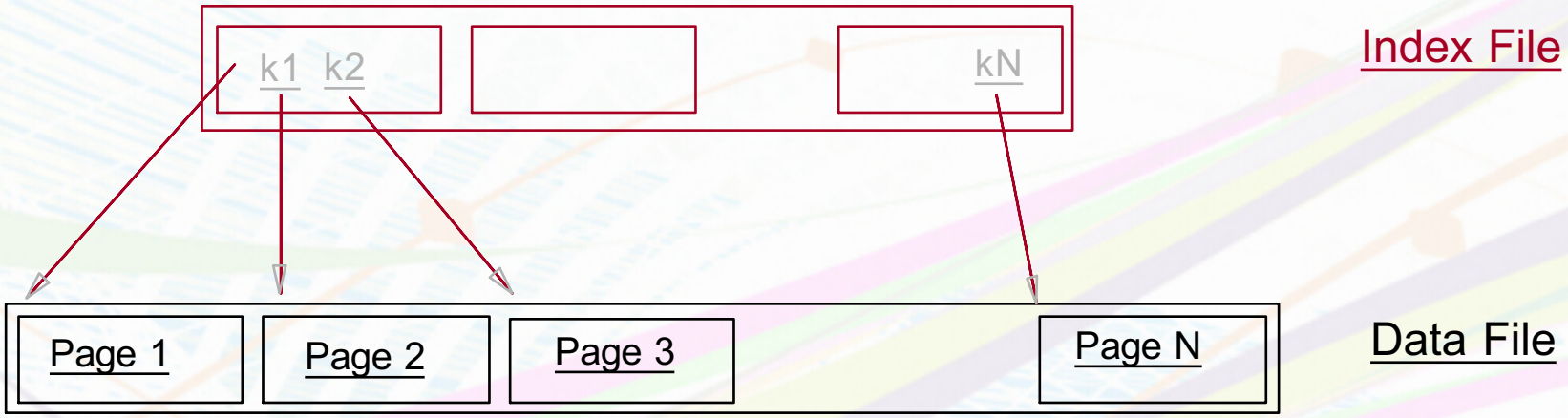
Range Searches

- "*Find all students with gpa > 3.0*"
- **may be slow, even on sorted file**
- **Solution: Create an `index' file.**



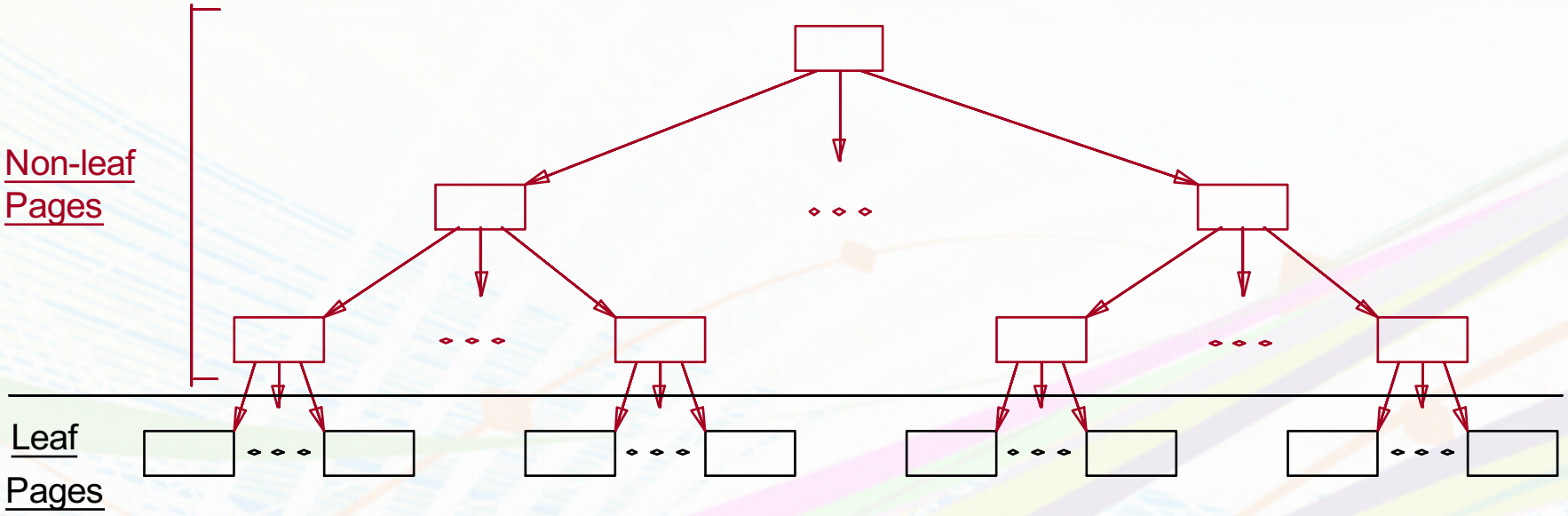
Range Searches

- **More details:**
- **if index file is small, do binary search there**
- **Otherwise??**



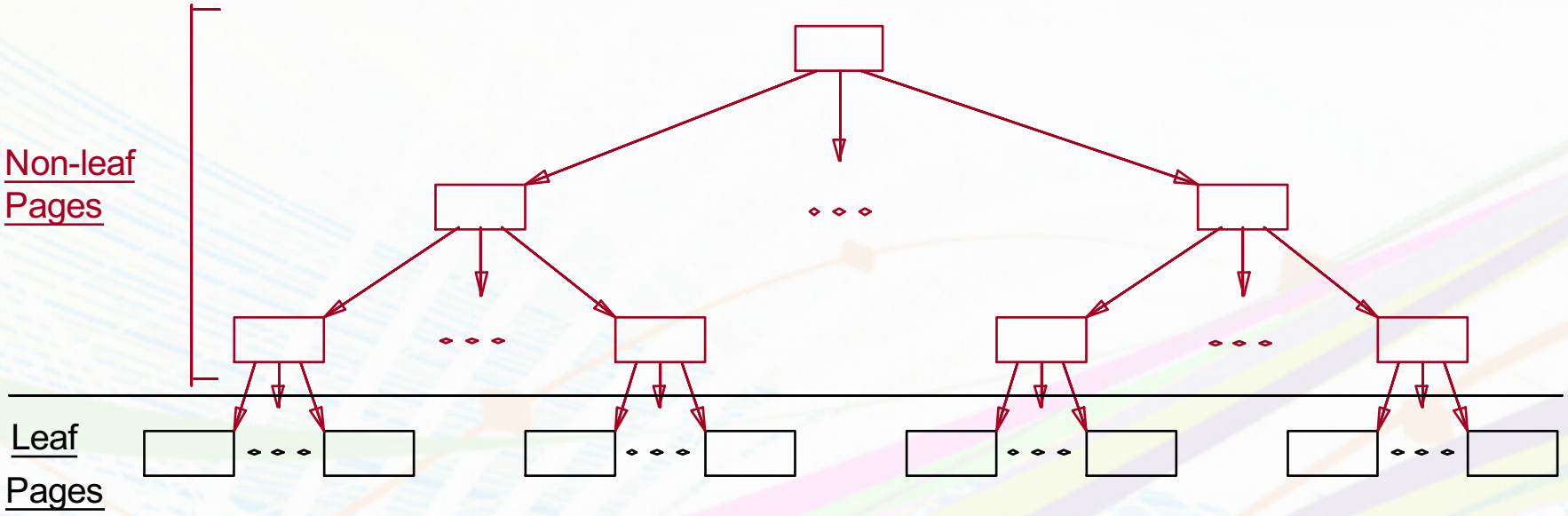
ISAM

- **Repeat recursively!**



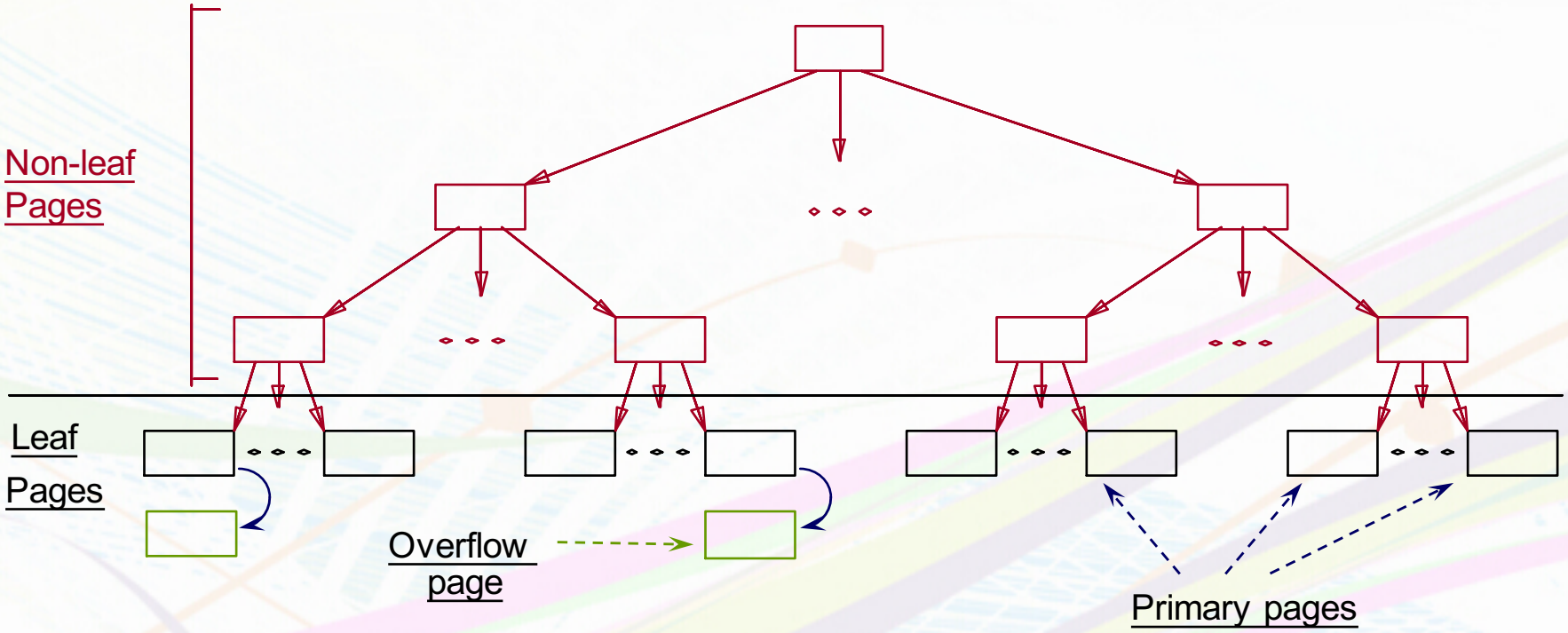
ISAM

- **OK - what if there are insertions and overflows?**



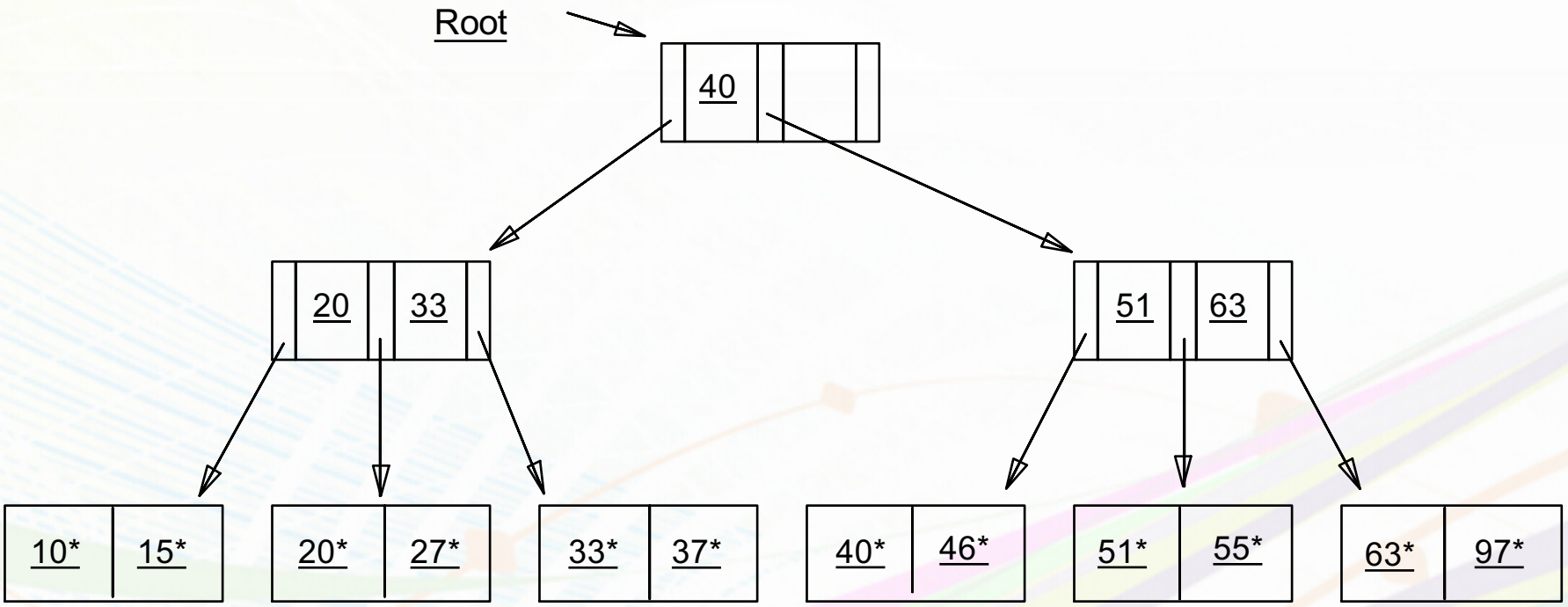
ISAM

- **Overflow pages, linked to the primary page**

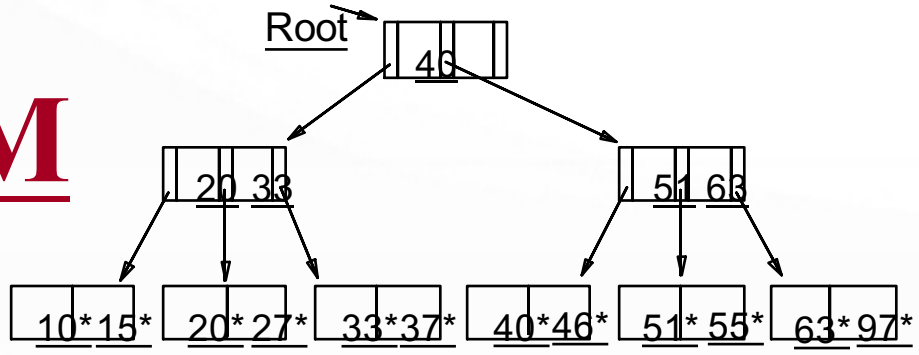


Example ISAM Tree

- 2 entries per page



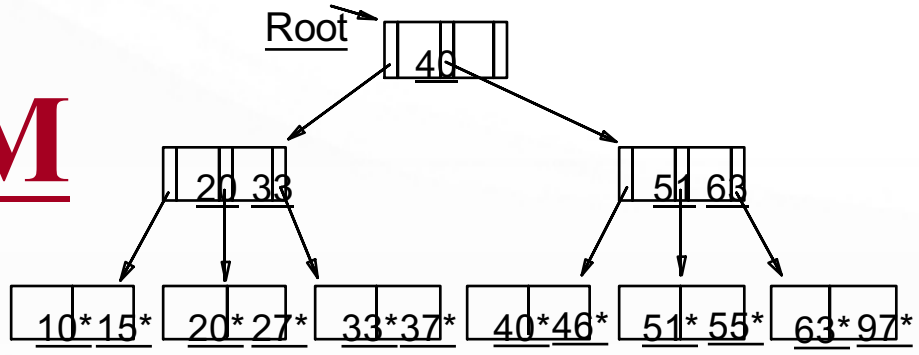
ISAM



Details

- format of an index page?
- how full should a newly created ISAM be?

ISAM

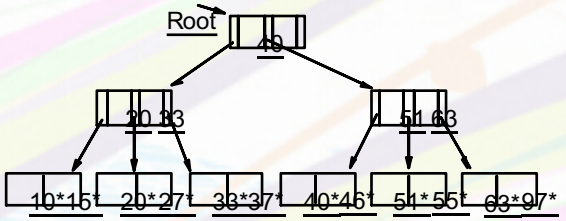


Details

- format of an index page?
- how full should a newly created ISAM be?
  - ~80-90% (not 100%)

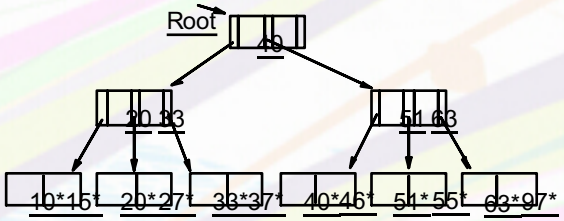
ISAM is a STATIC Structure

- that is, index pages don't change
- *File creation*: Leaf (data) pages allocated sequentially, sorted by search key; then index pages allocated, then overflow pgs.



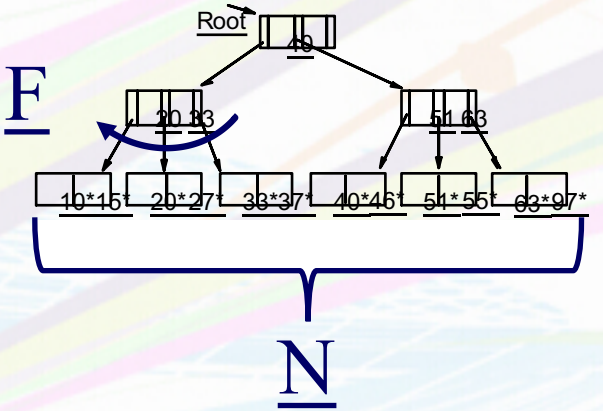
ISAM is a STATIC Structure

- ***Search*: Start at root; use key comparisons to go to leaf.**
- **Cost = ??**



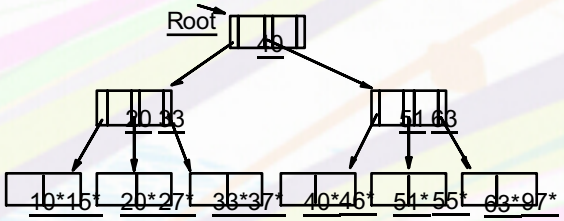
ISAM is a STATIC Structure

- ***Search*: Start at root; use key comparisons to go to leaf.**
- **Cost = ??**



ISAM is a STATIC Structure

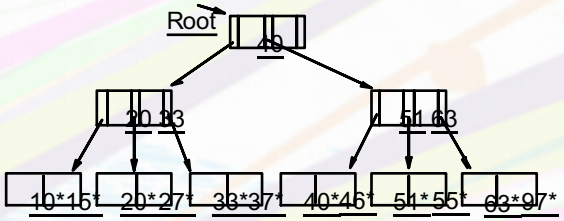
- *Search*: Start at root; use key comparisons to go to leaf.
- Cost = $\log_F N$;
- F = # entries/pg (i.e., fanout),
- N = # leaf pgs



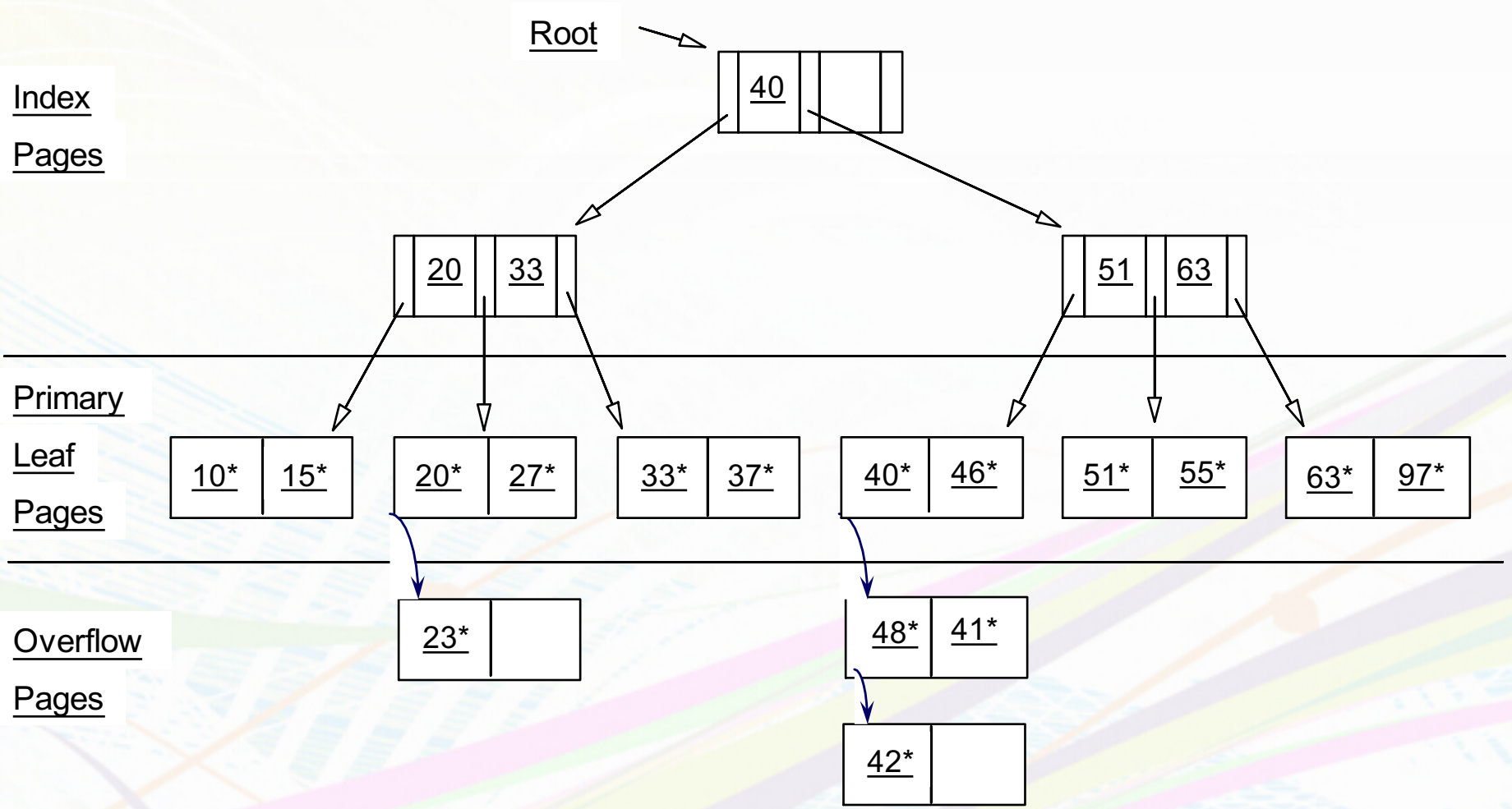
ISAM is a STATIC Structure

***Insert*: Find leaf that data entry belongs to, and put it there. Overflow page if necessary.**

***Delete*: Find and remove from leaf; if empty page, de-allocate.**

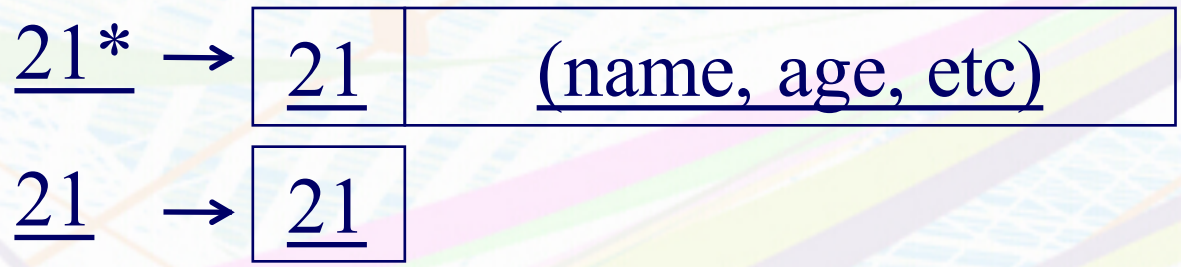


Example: Insert 48*, 23*, 41*, 42*



21* means

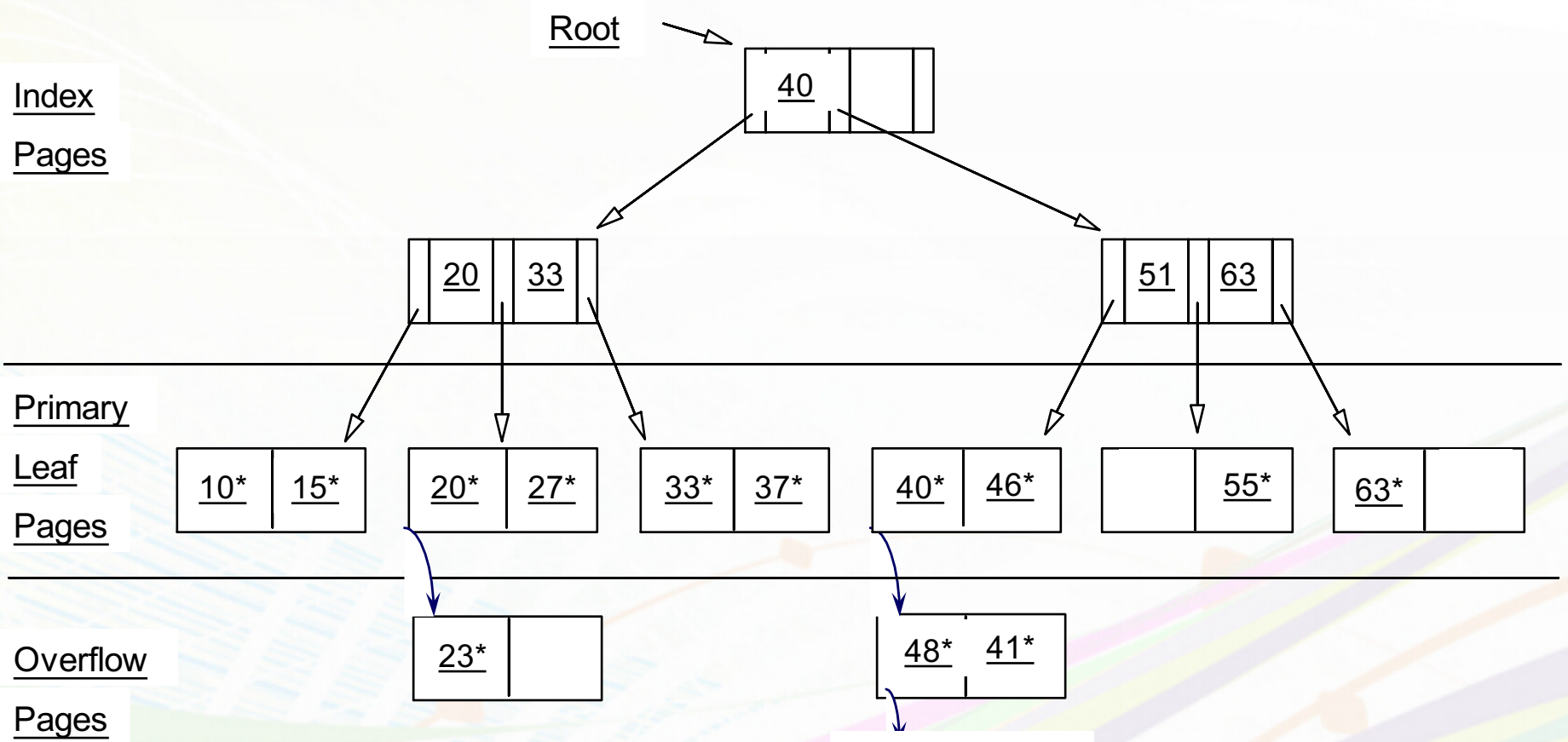
- **<21> + the rest of the record**
- **(it's a bit more complicated – but we stay with that, for the moment).**
- **'21' plain means just 4 bytes, to store integer 21**



~record

divider

... then delete 42*, 51*, 97*



☛ *Note that 51* appears in index levels, but not in leaf!*

ISAM ---- Issues?

- **Pros**
  - **????**
- **Cons**
  - **????**

Outline

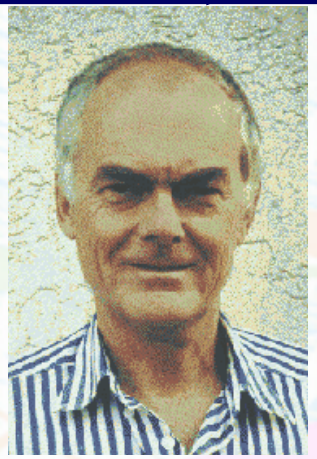
- **Motivation**
- **ISAM**
- **B-trees**
- **Tree vs. Hash-based index**
- **Index organization: clustered vs. nonclustered**

B-trees and Why?

- **the** most successful **family of index schemes (B-trees, B^{+}-trees, B^{*}-trees)**
- **Can be used for primary/secondary, clustering/non-clustering index.**
- **balanced "n-way" search trees**
- **Flexible and dynamic**

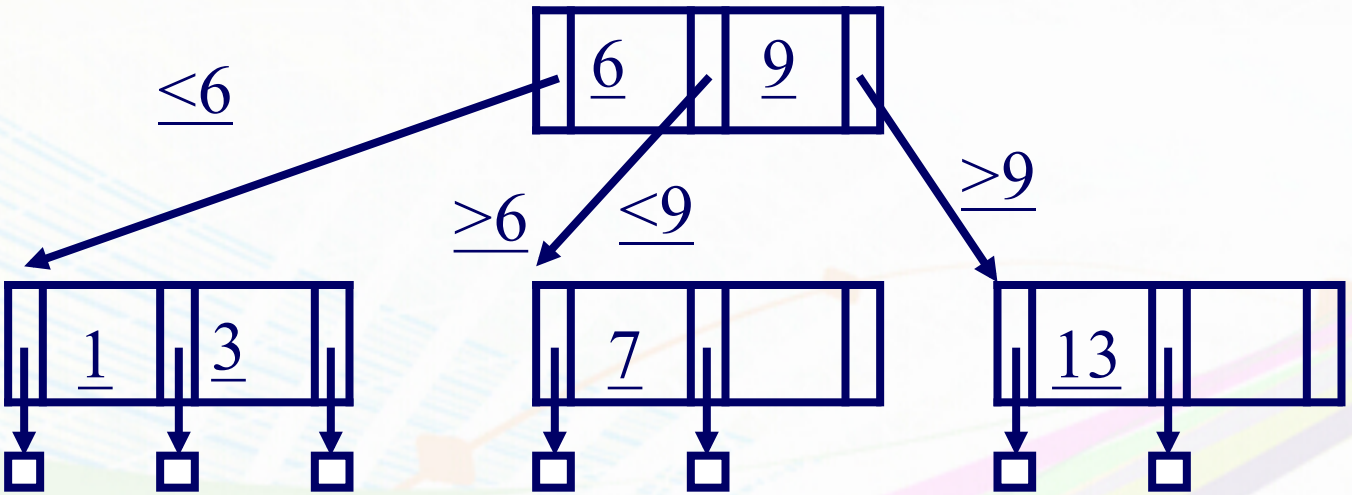
B-trees

[Rudolf Bayer and McCreight, E. M. Organization and Maintenance of Large Ordered Indexes. Acta Informatica 1, 173-189, 1972.]



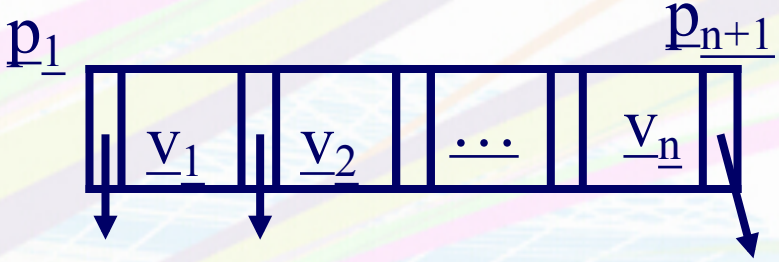
B-trees Example

B-tree of order d=1:



B - tree properties:

- **each node, in a B-tree of order *d*:**
  - **Key order**
  - **at most n=*2d keys***
  - **at least *d keys* (except root, which may have just 1 key)**
  - **all leaves at the same level**
  - **if number of pointers is *k*, then node has exactly k-1 keys**
  - **(leaves are with data entries)**

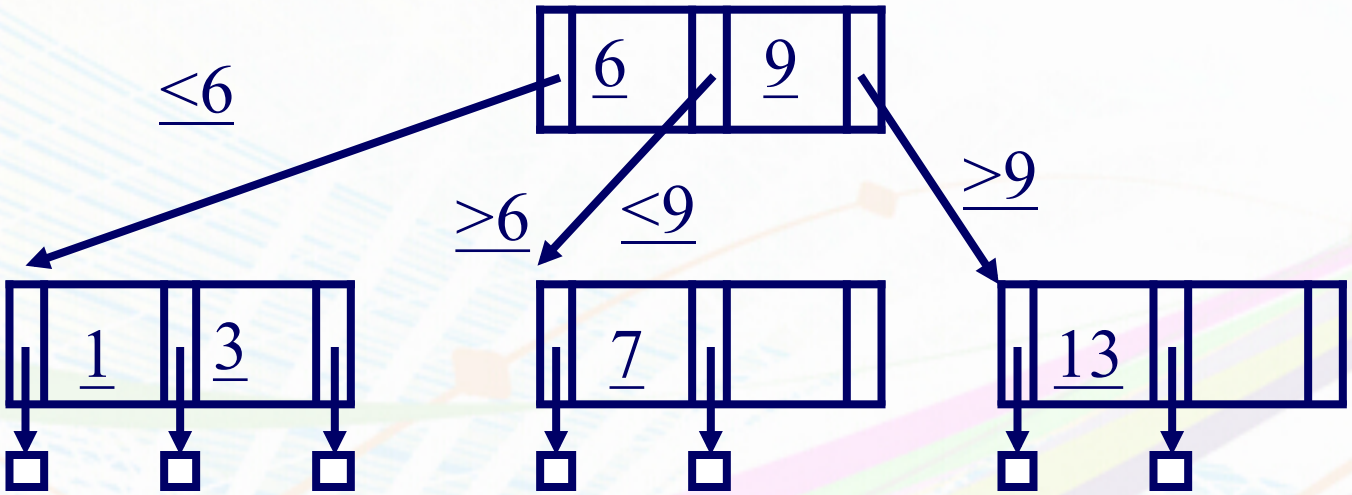


Properties

- **"block aware" nodes: each node -> disk page**
- **O(log (N)) for everything! (ins/del/search)**
- **typically, if d = 50 - 100, then 2 - 3 levels**
- **utilization >= 50%, guaranteed; on average 69%**

Queries

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



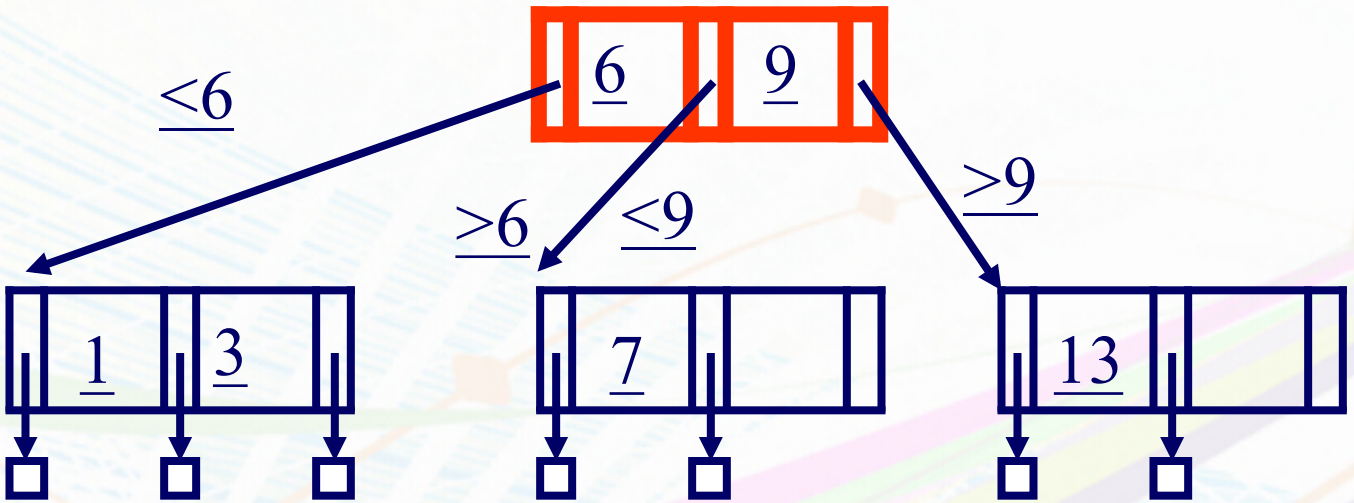
JAVA animation!

http://slady.net/java/bt/

strongly recommended! (with all usual pre-cautions – VM etc)

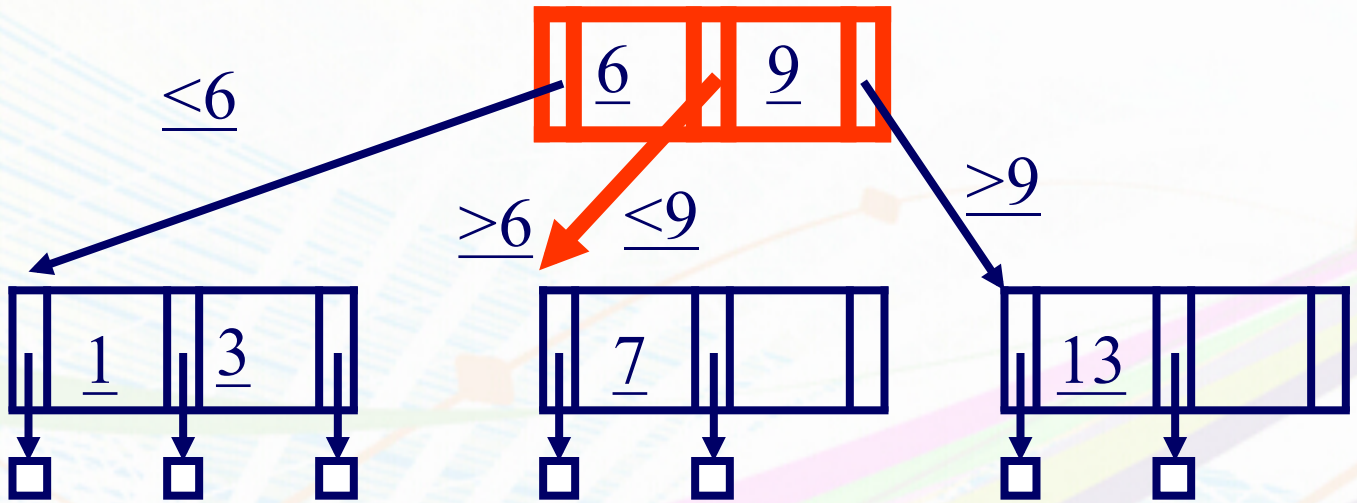
Queries

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



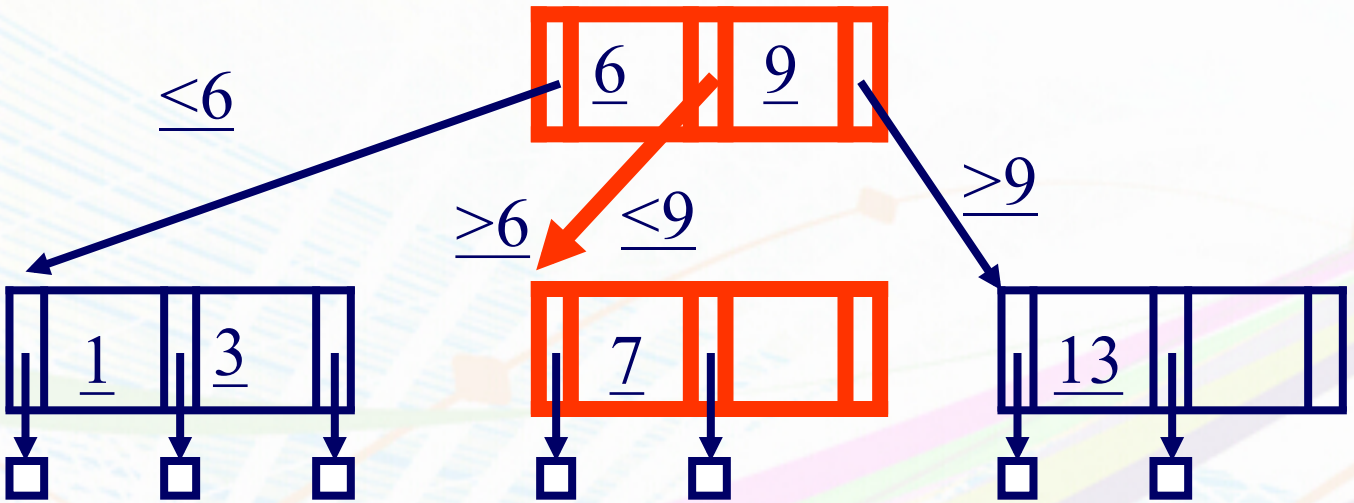
Queries

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



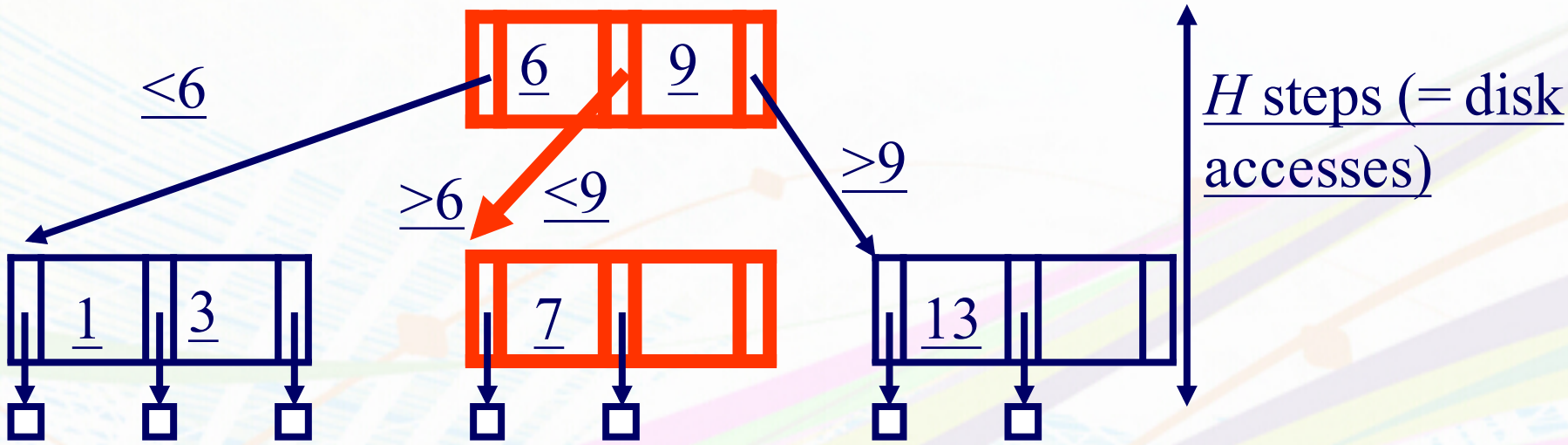
Queries

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



Queries

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

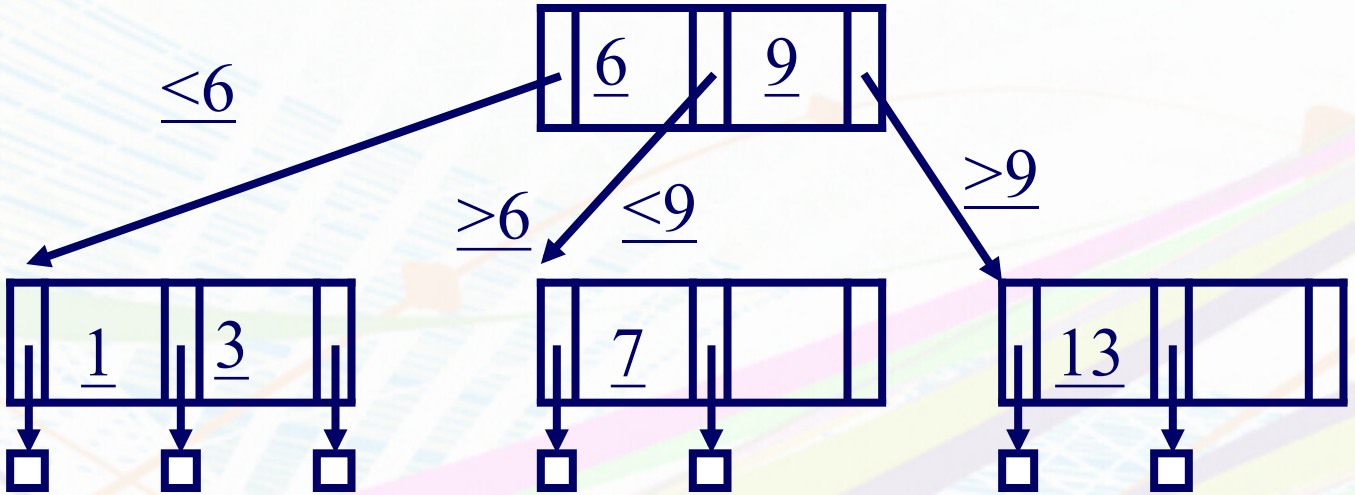


Queries

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

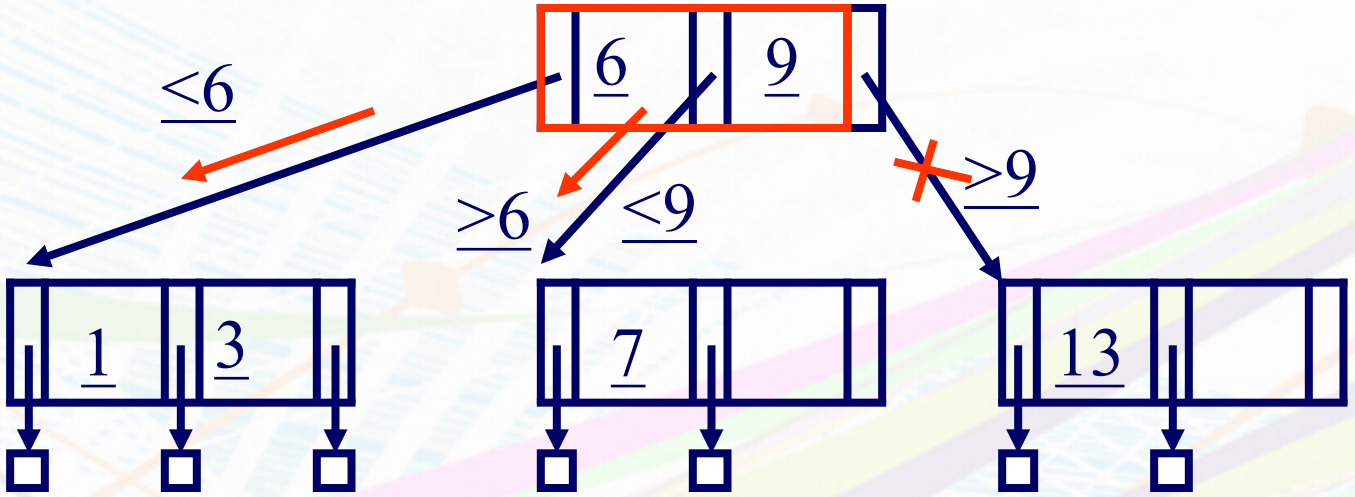
Queries

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



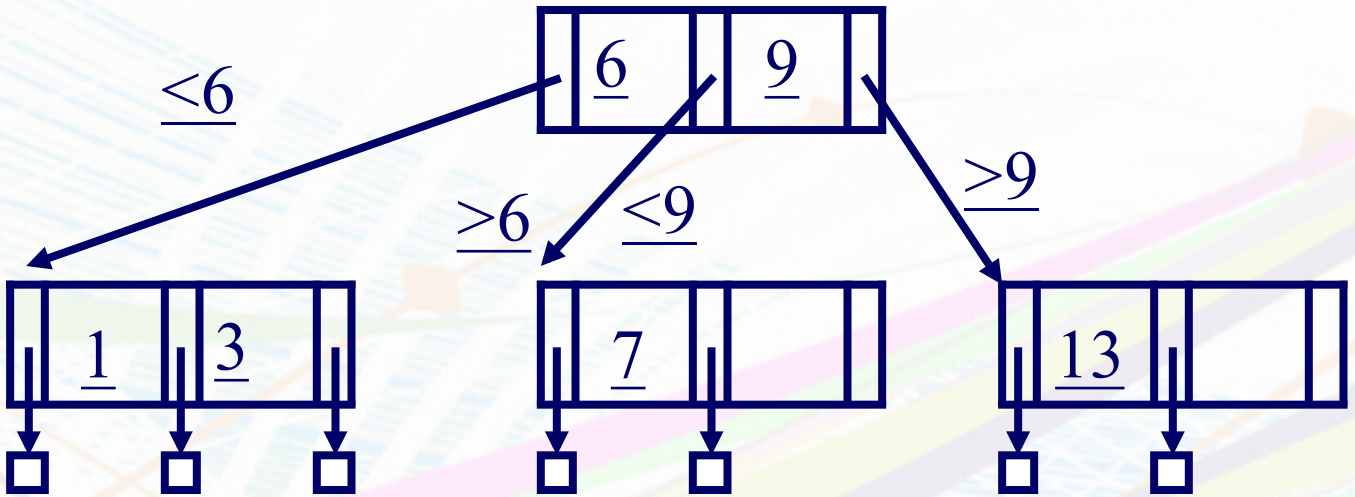
Queries

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



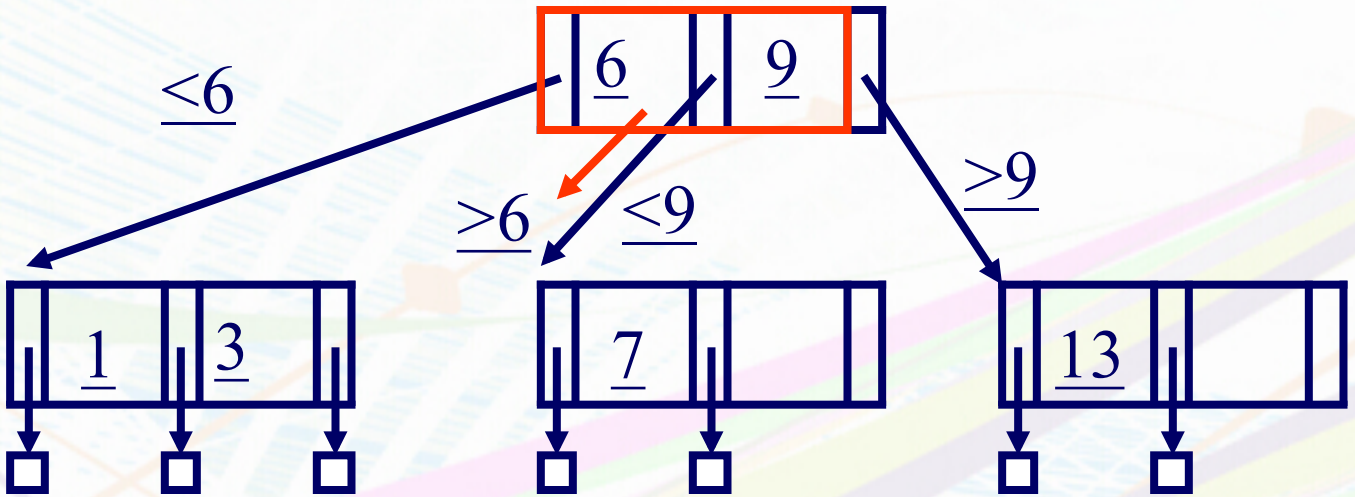
Queries

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



Queries

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

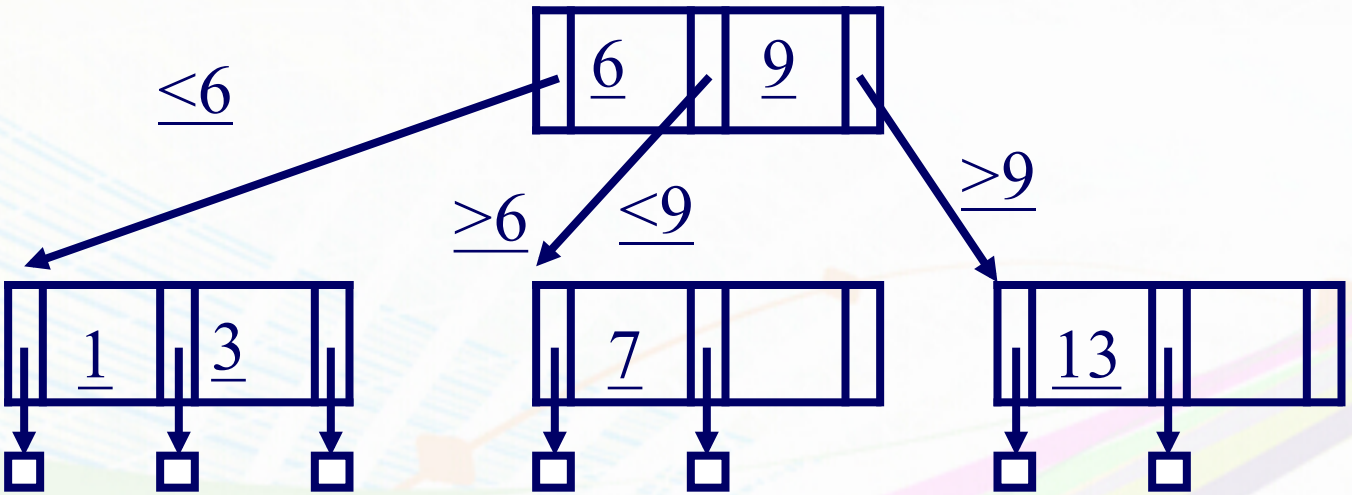


B-trees: Insertion

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

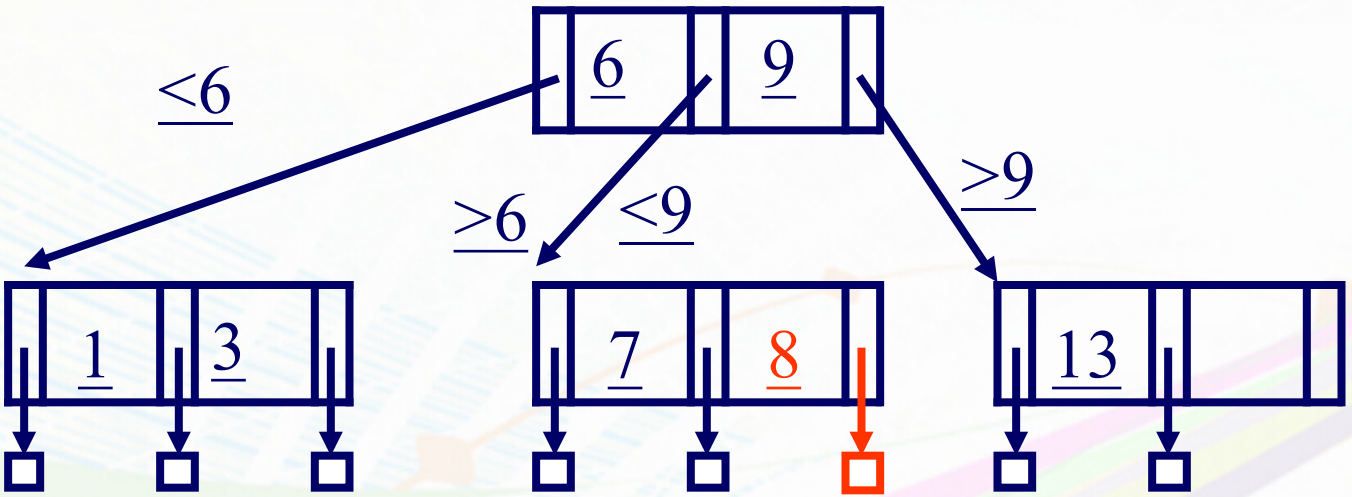
B-trees

Easy case: Tree T0; insert ‘8’



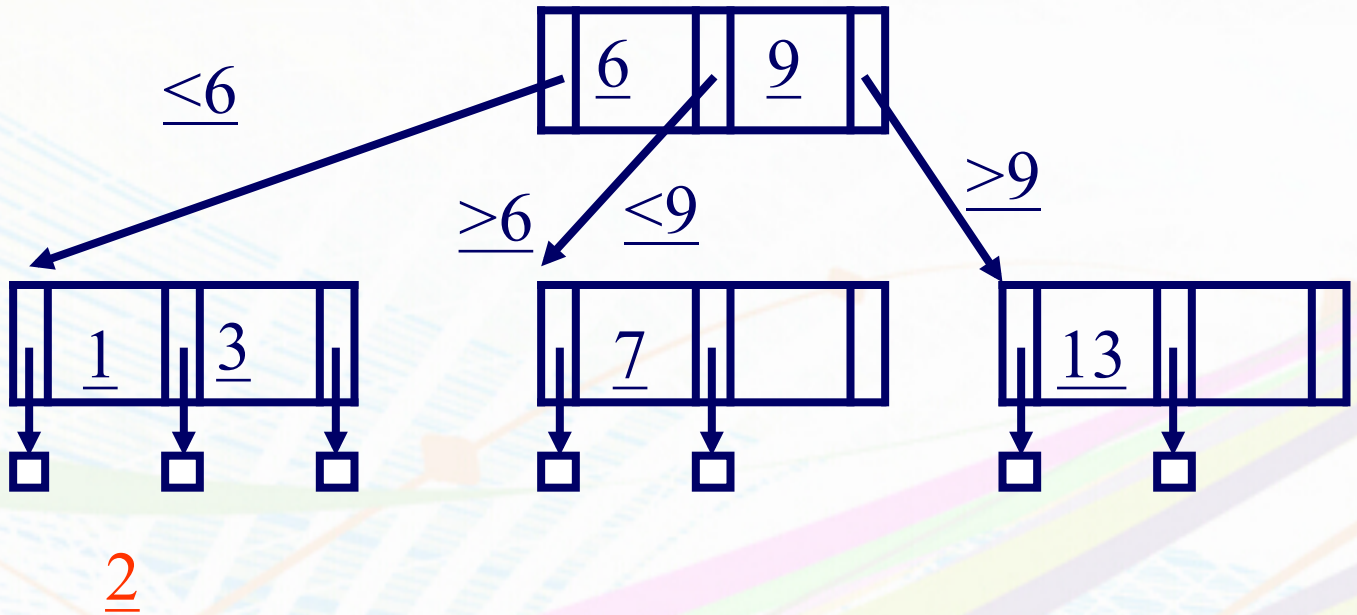
B-trees

Tree T0; insert ‘8’



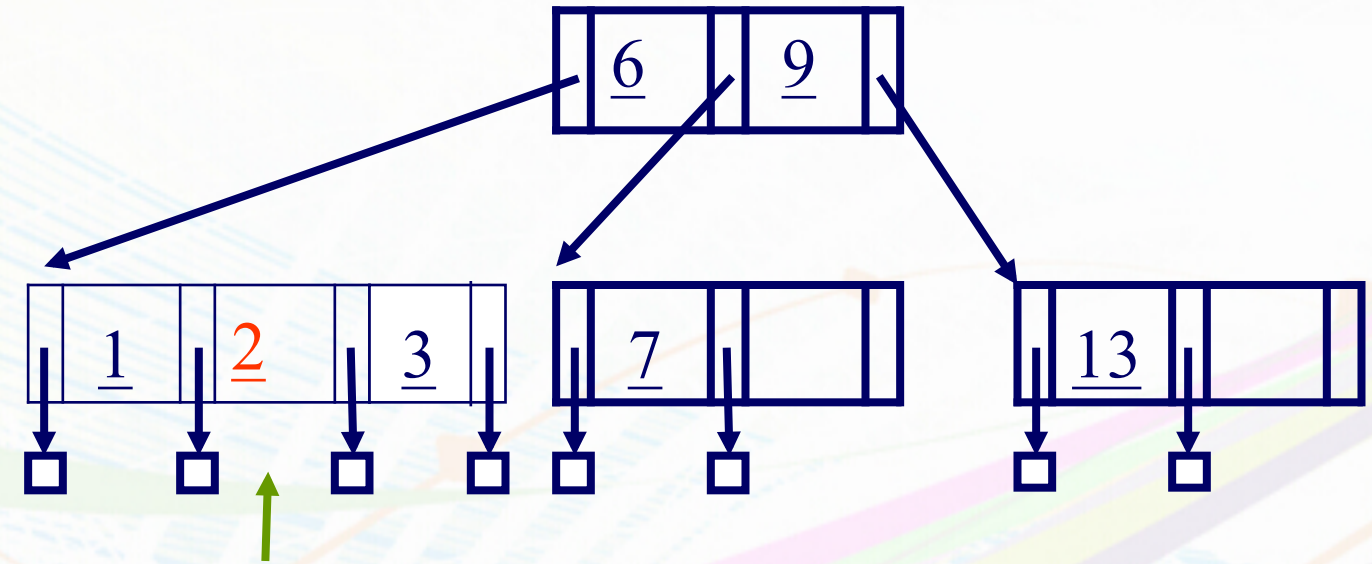
B-trees

Hardest case: Tree T0; insert ‘2’



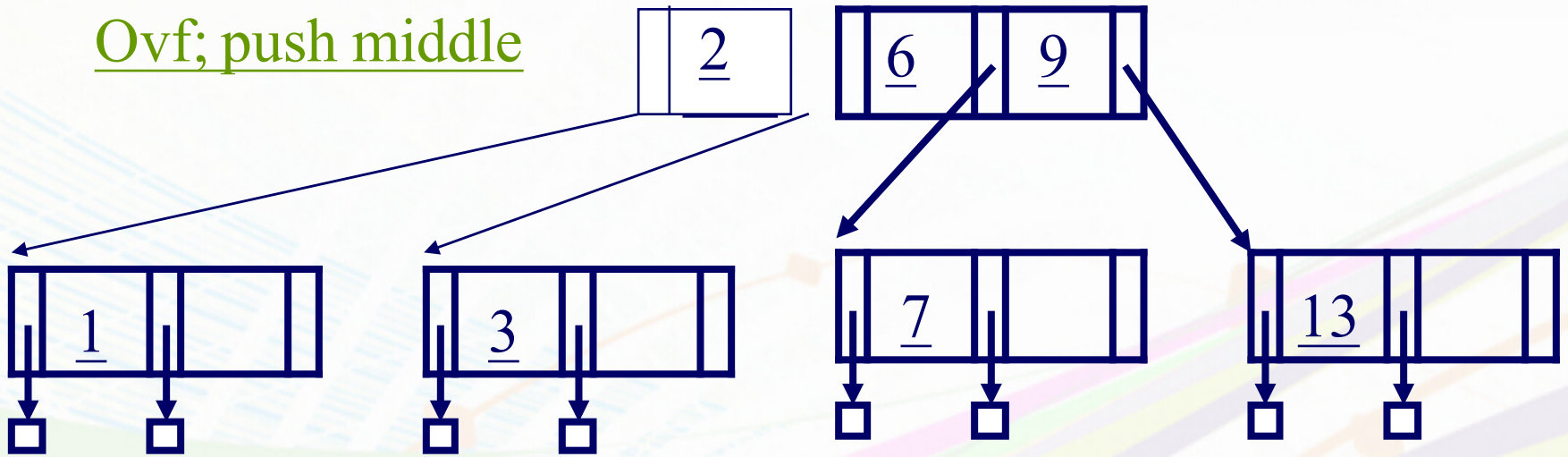
B-trees

Hardest case: Tree T0; insert ‘2’



B-trees

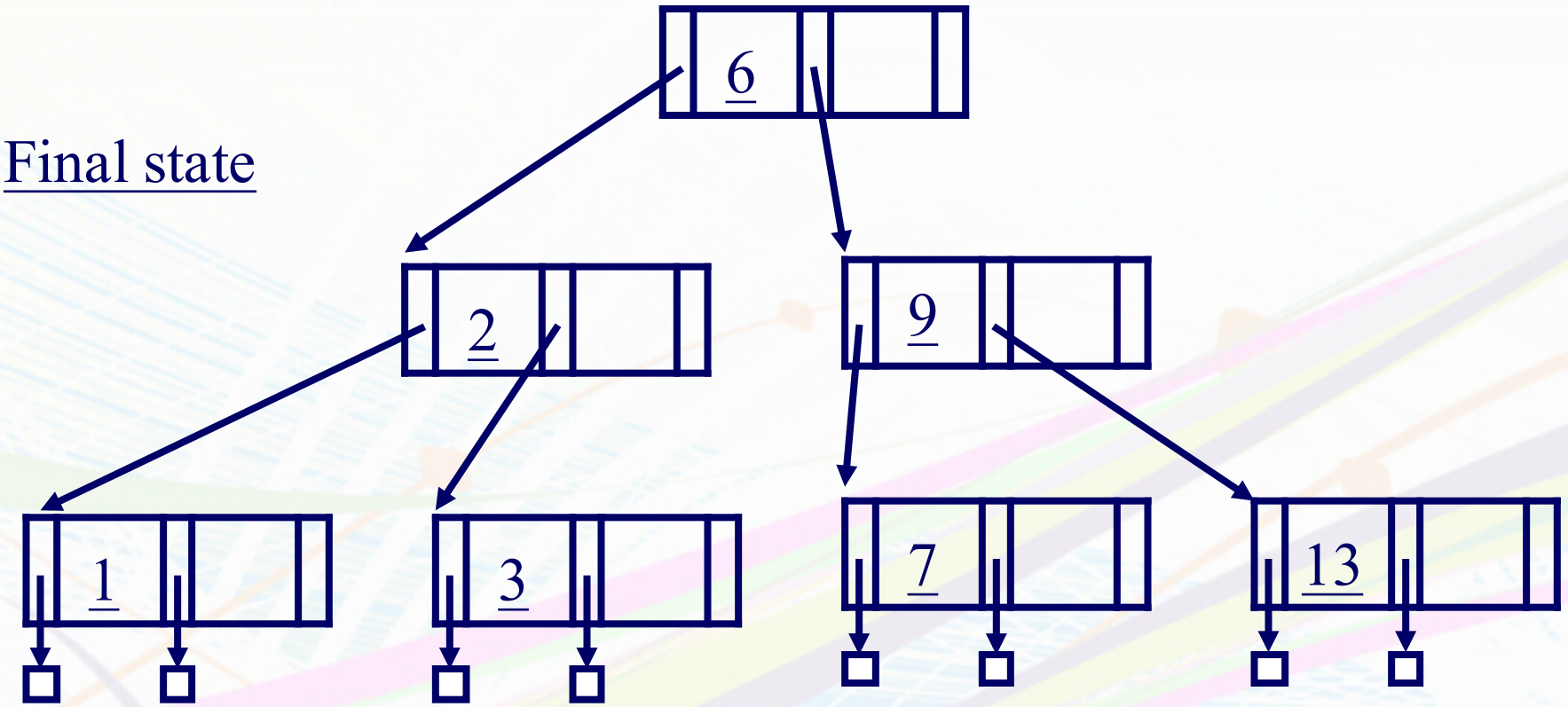
Hardest case: Tree T0; insert ‘2’



B-trees

Hardest case: Tree T0; insert ‘2’

Final state



B-trees: Insertion

- Insert in leaf; on overflow, push middle up (recursively – 'propagate split')
- split: preserves all B - tree properties (!!)
- notice how it grows: height increases when root overflows & splits
- Automatic, incremental re-organization (contrast with ISAM!)

Pseudo-code

```
INSERTION OF KEY 'K'
find the correct leaf node 'L';
if ('L' overflows ){
        split 'L', and push middle key to parent node 'P';
        if ('P' overflows){
                repeat the split recursively; }
else{
        add the key 'K' in node 'L';
        /* maintaining the key order in 'L' */ }
```

Overview

- **…**
- **B – trees**
  - **Dfn, Search, insertion,** deletion
- **…**

Deletion

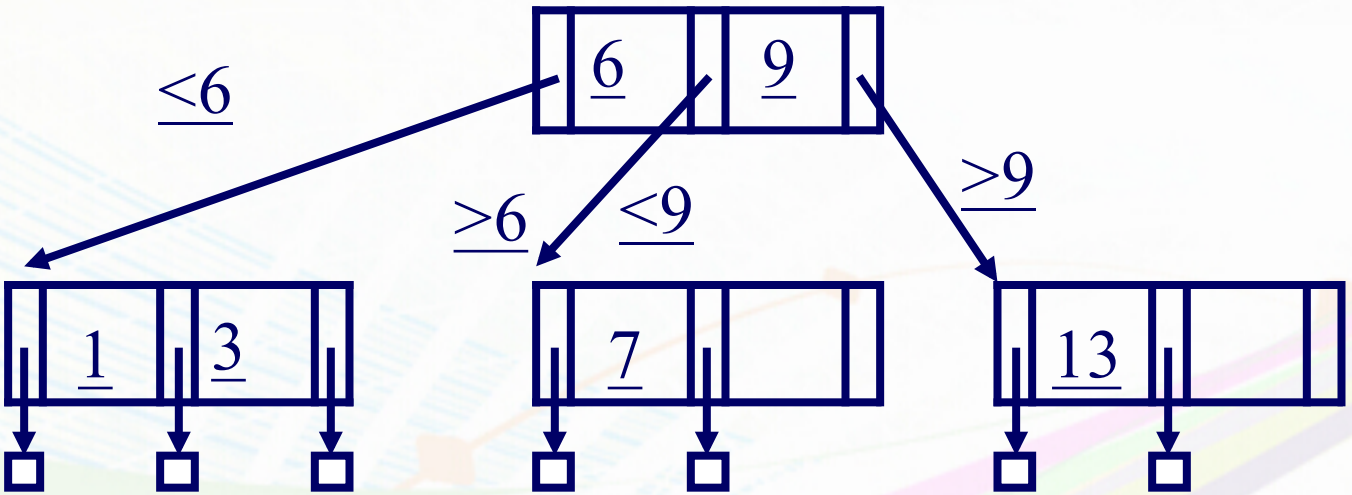
Rough outline of algo:

- **Delete key;**
- **on underflow, may need to merge**

In practice, some implementors just allow underflows to happen…

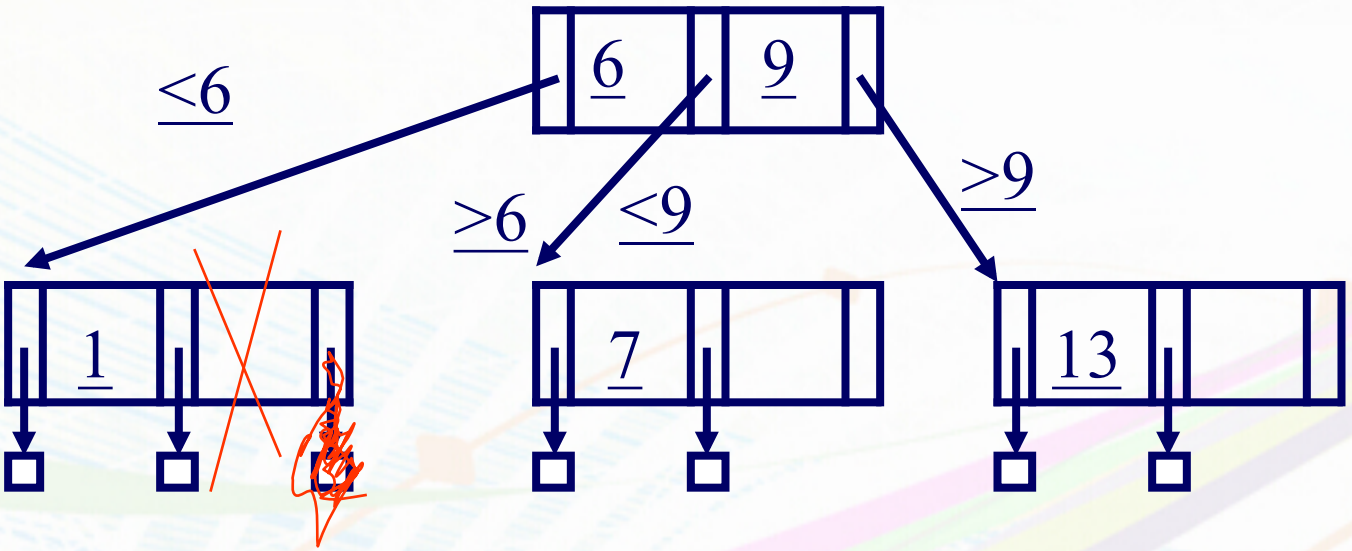
B-trees – Deletion

Easiest case: Tree T0; delete '3'



B-trees – Deletion

Easiest case: Tree T0; delete '3'

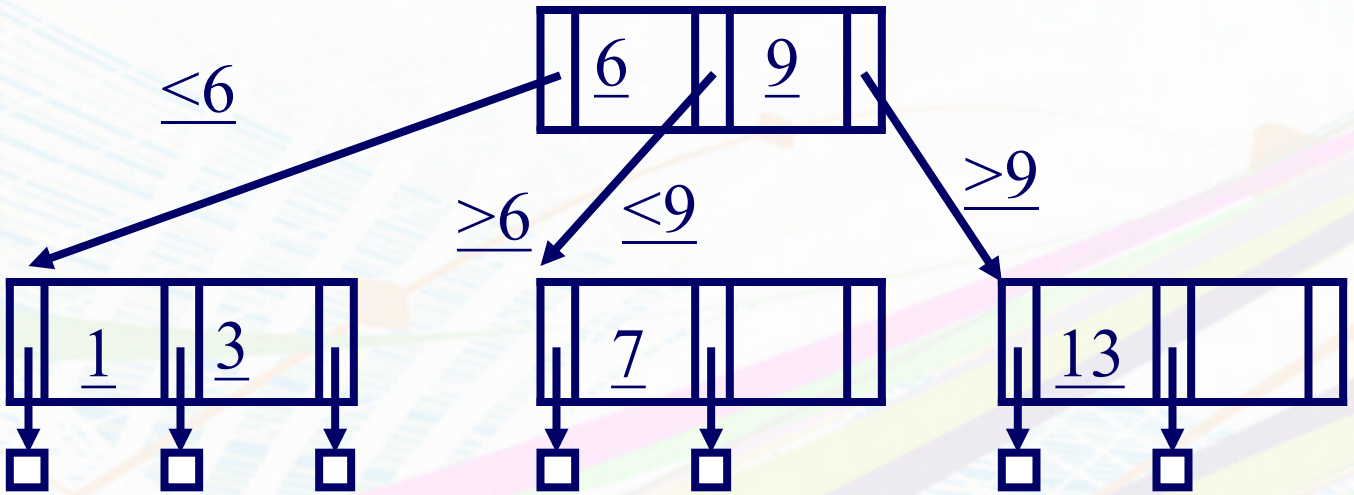


B-trees – Deletion

- 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’

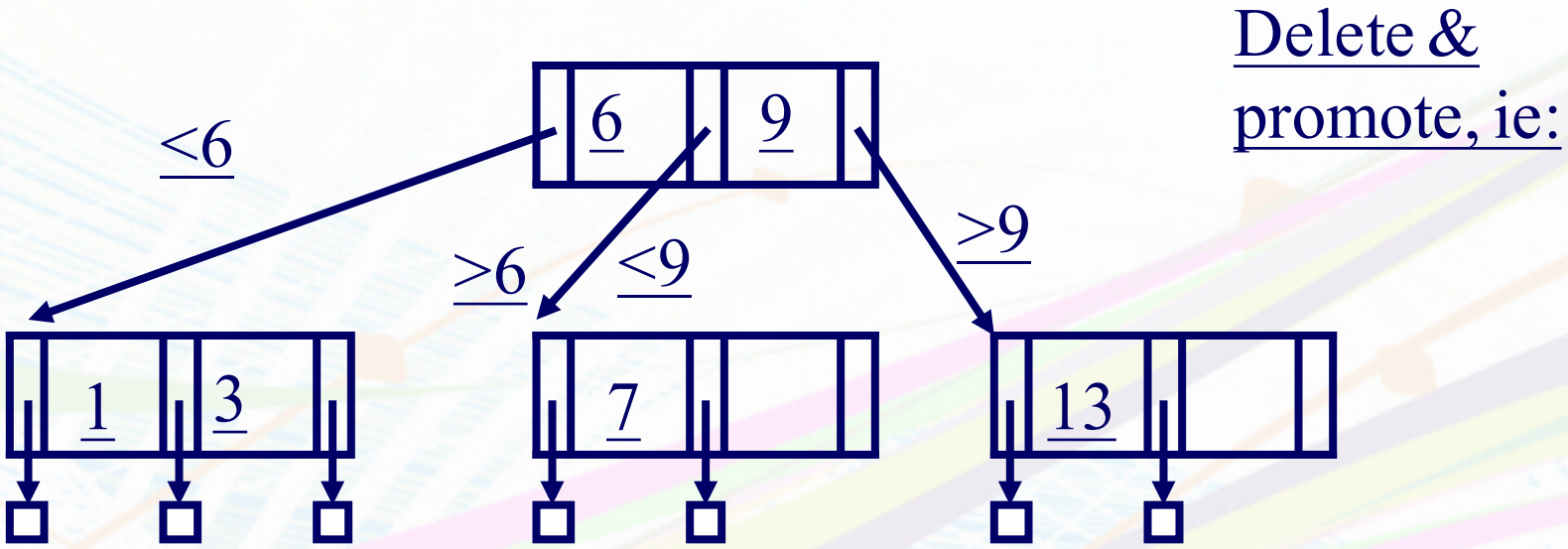
B-trees – Deletion

- **Case1: delete a key at a leaf – no underflow (delete 3 from T0)**



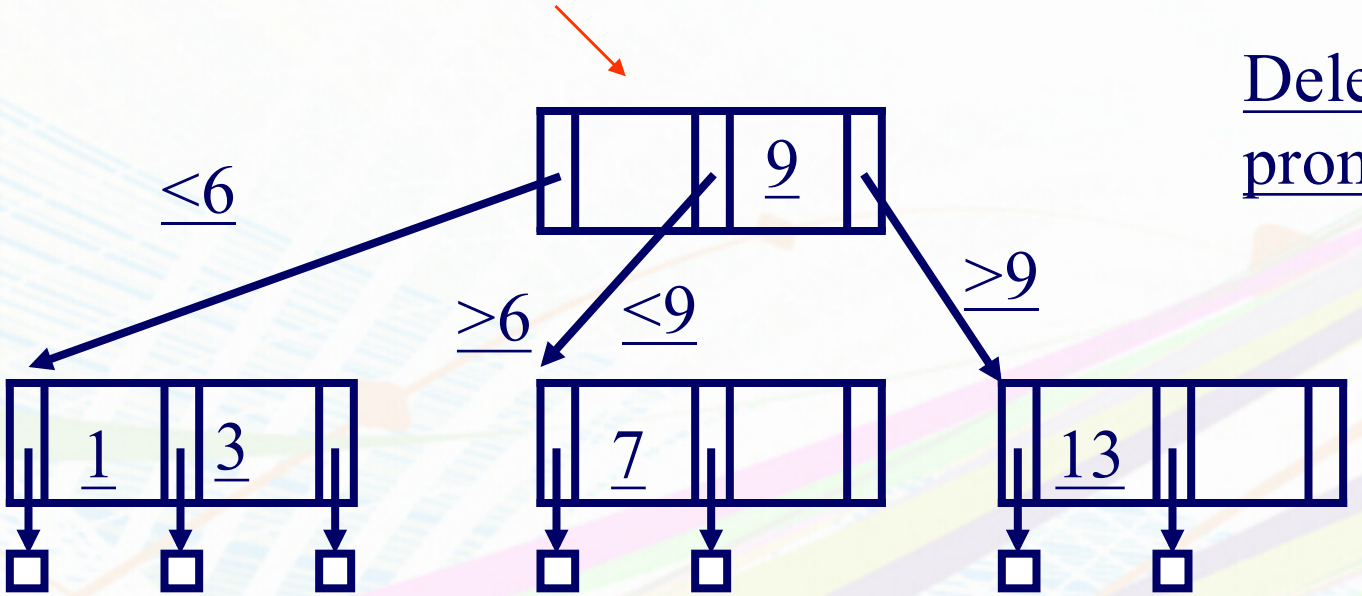
B-trees – Deletion

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



B-trees – Deletion

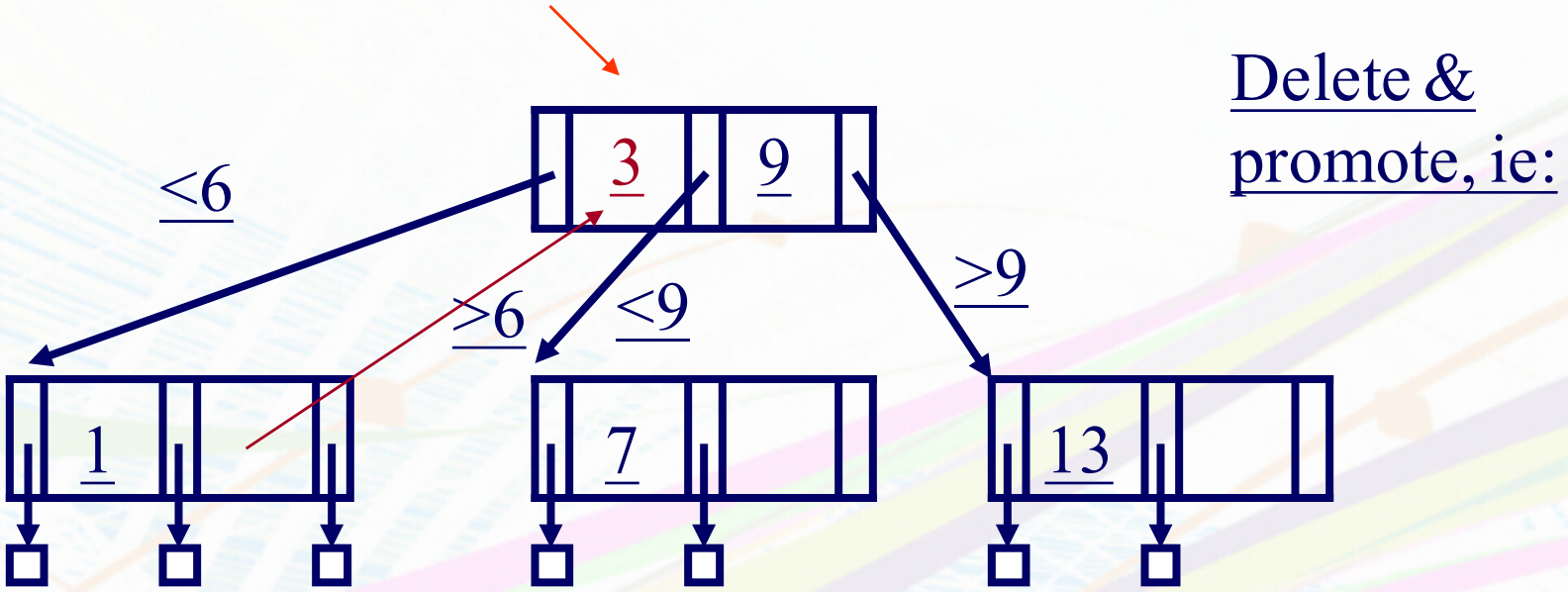
- **Case2: delete a key at a non-leaf – no underflow (eg., delete 6 from T0)**



Delete & promote, ie:

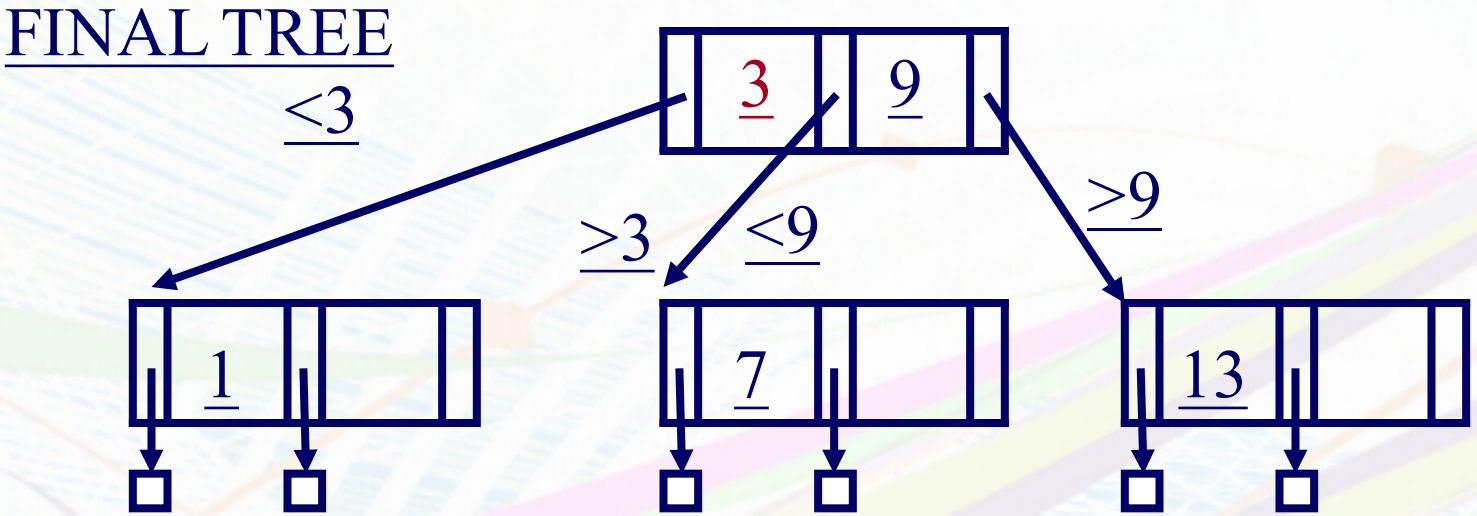
B-trees – Deletion

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



B-trees – Deletion

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



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)

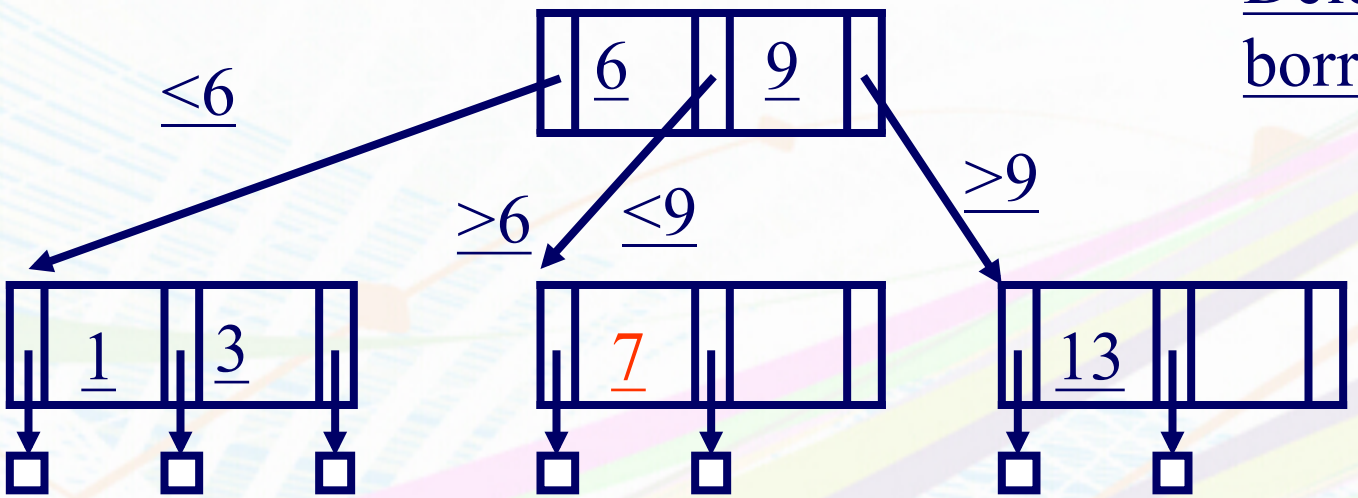
- Observation: every deletion eventually becomes a deletion of a leaf key

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’

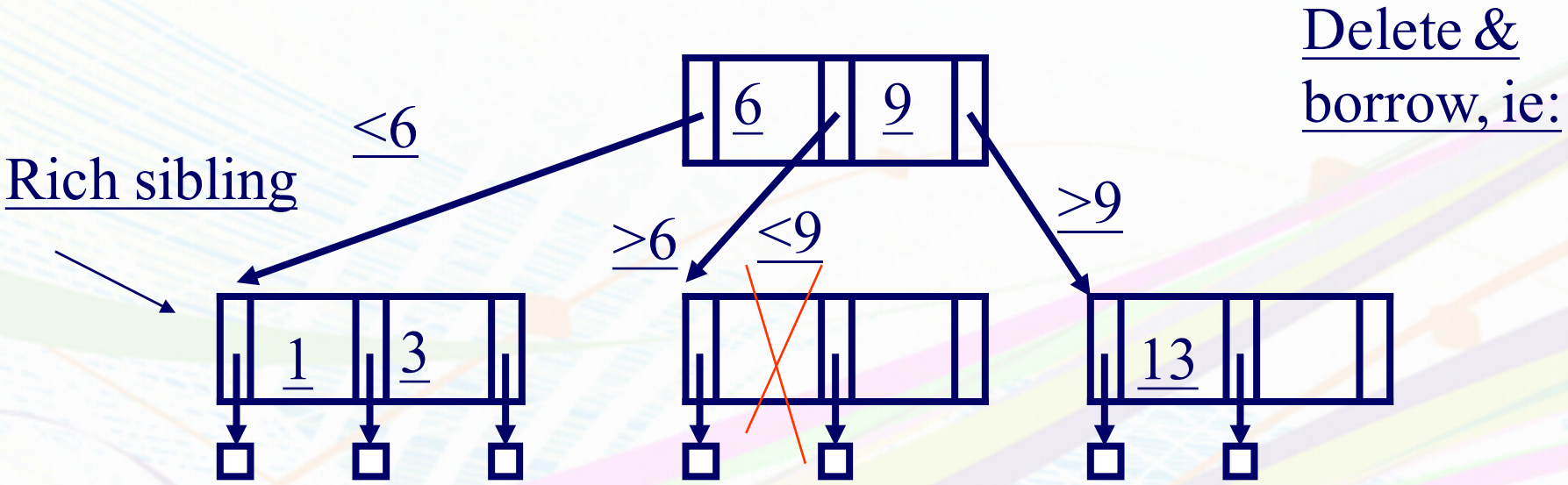
B-trees – Deletion

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



B-trees – Deletion

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



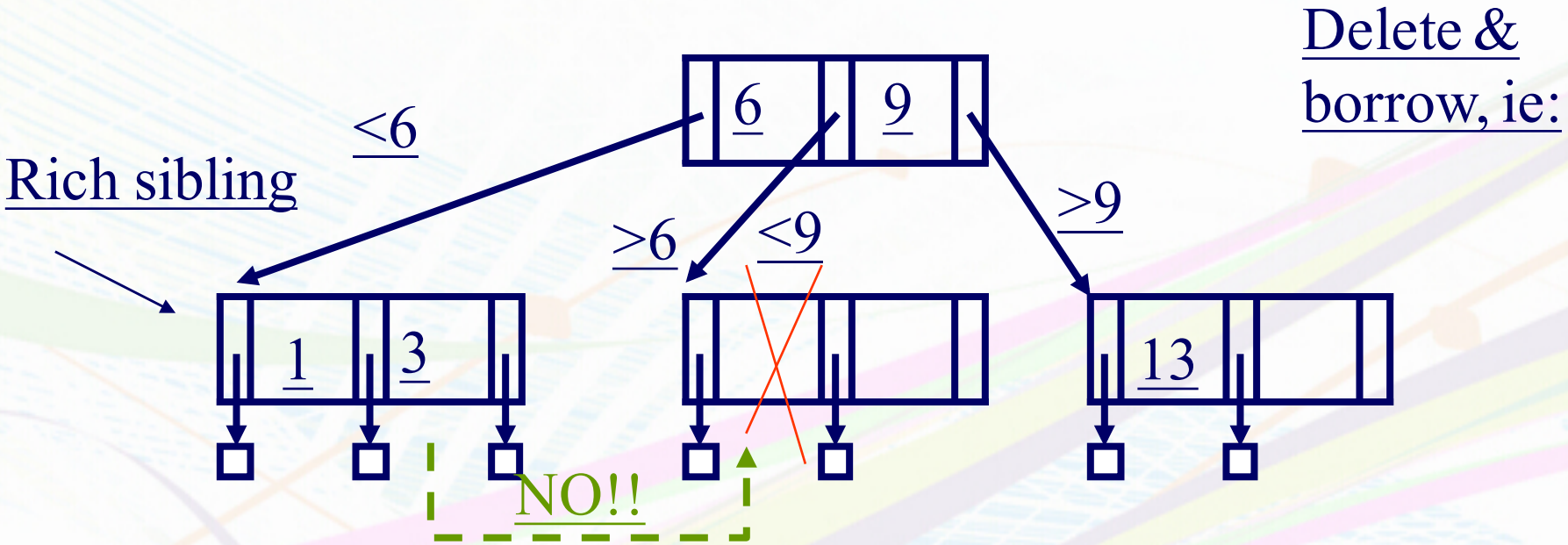
B-trees – Deletion

- **Case3: underflow & ‘rich sibling’**

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

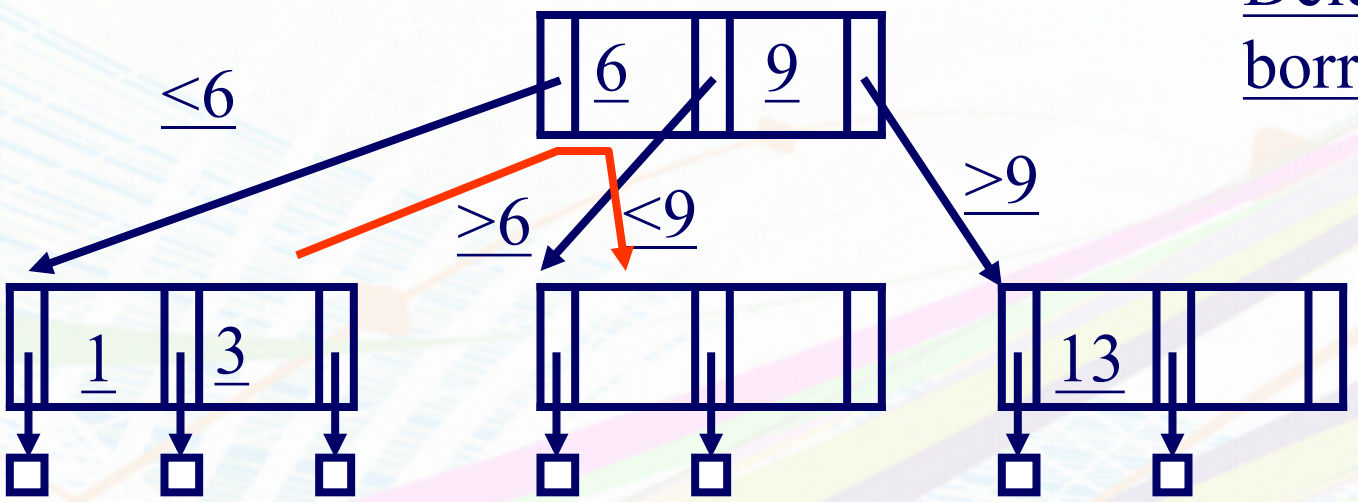
B-trees – Deletion

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



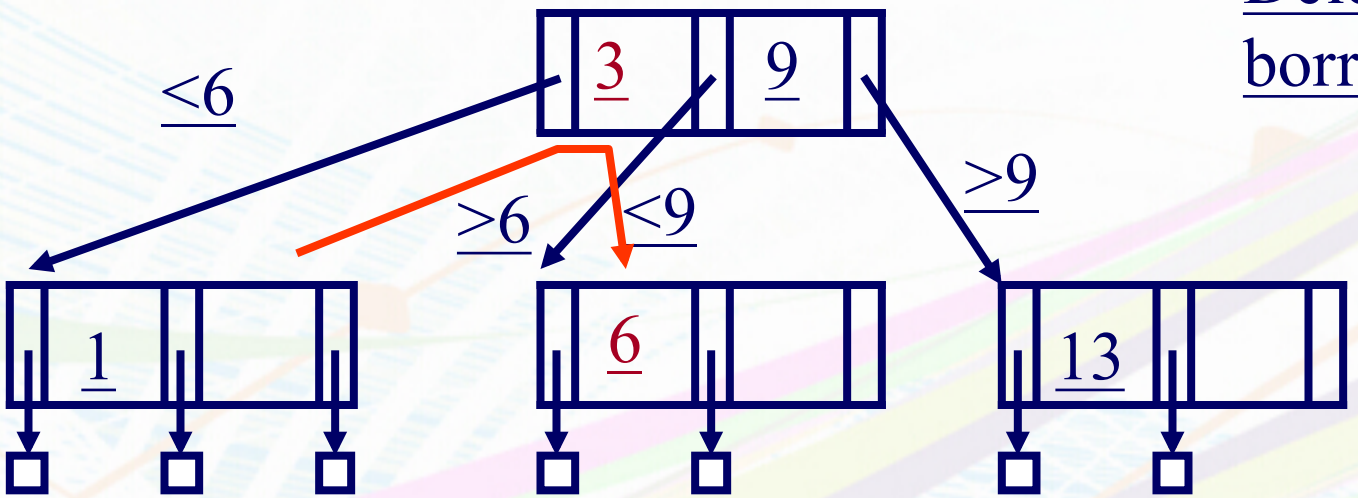
B-trees – Deletion

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



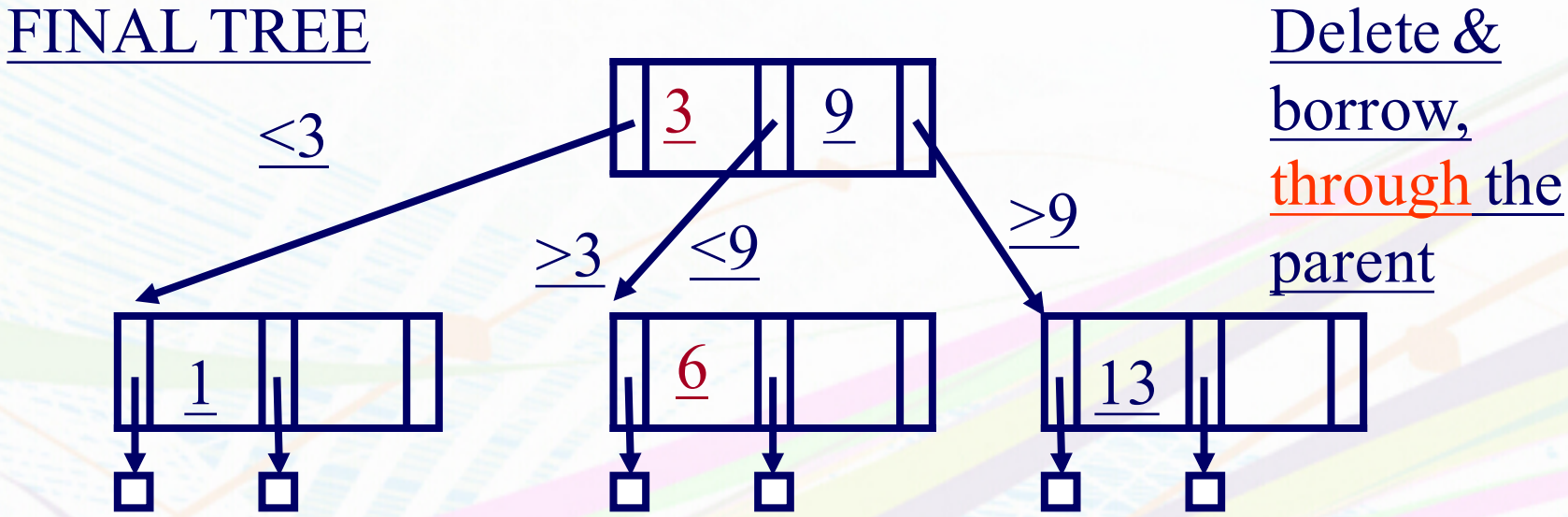
B-trees – Deletion

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



B-trees – Deletion

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



B-trees – Deletion

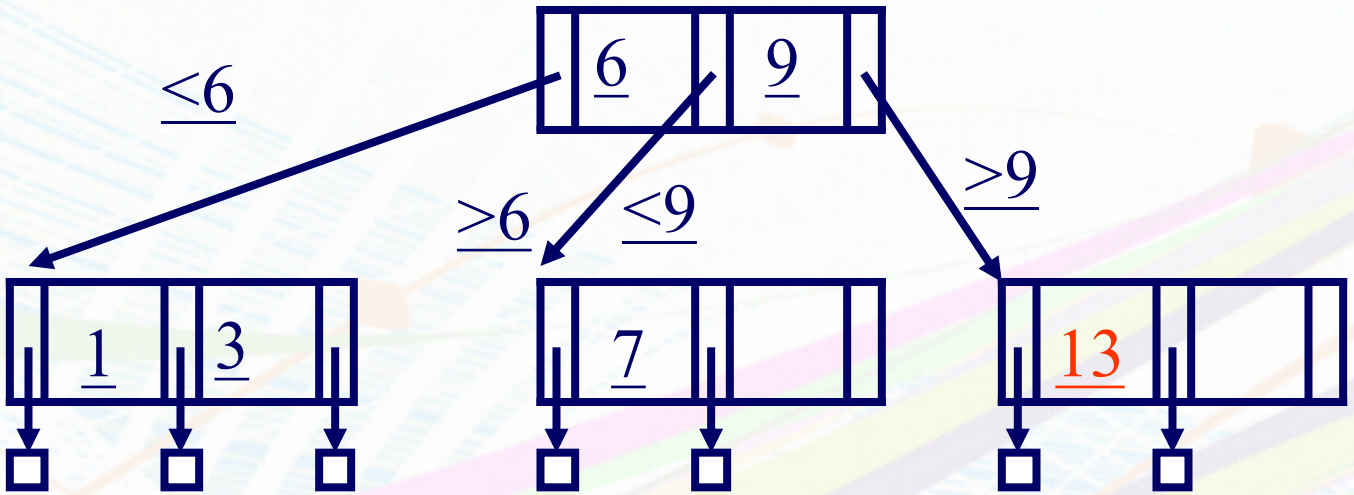
- **Case1: delete a key at a leaf – no underflow**
- **Case2: delete non-leaf key – no underflow**
- **Case3: delete leaf-key; underflow, and ‘rich sibling’**

⇨

- **Case4: delete leaf-key; underflow, and ‘poor sibling’**

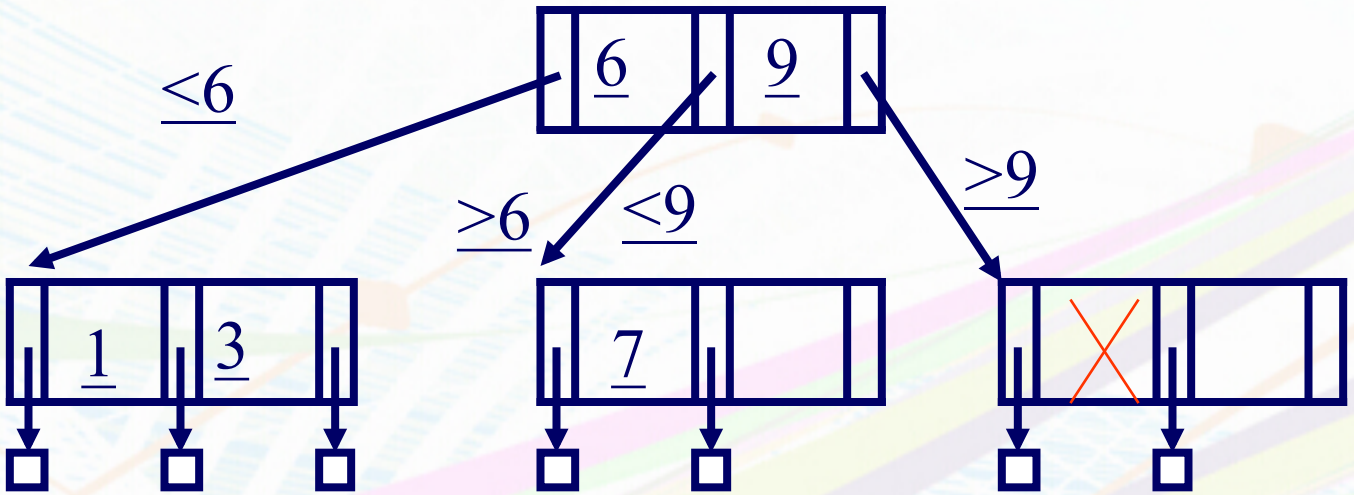
B-trees – Deletion

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



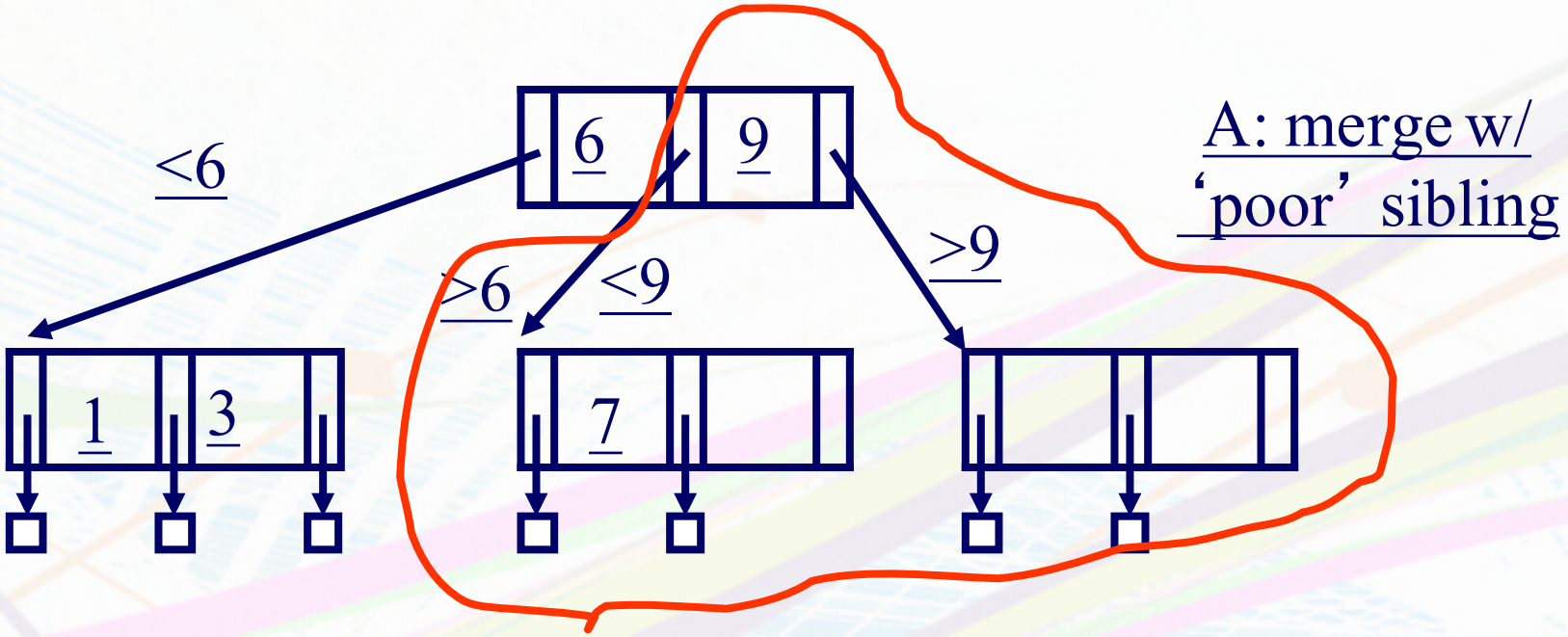
B-trees – Deletion

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



B-trees – Deletion

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



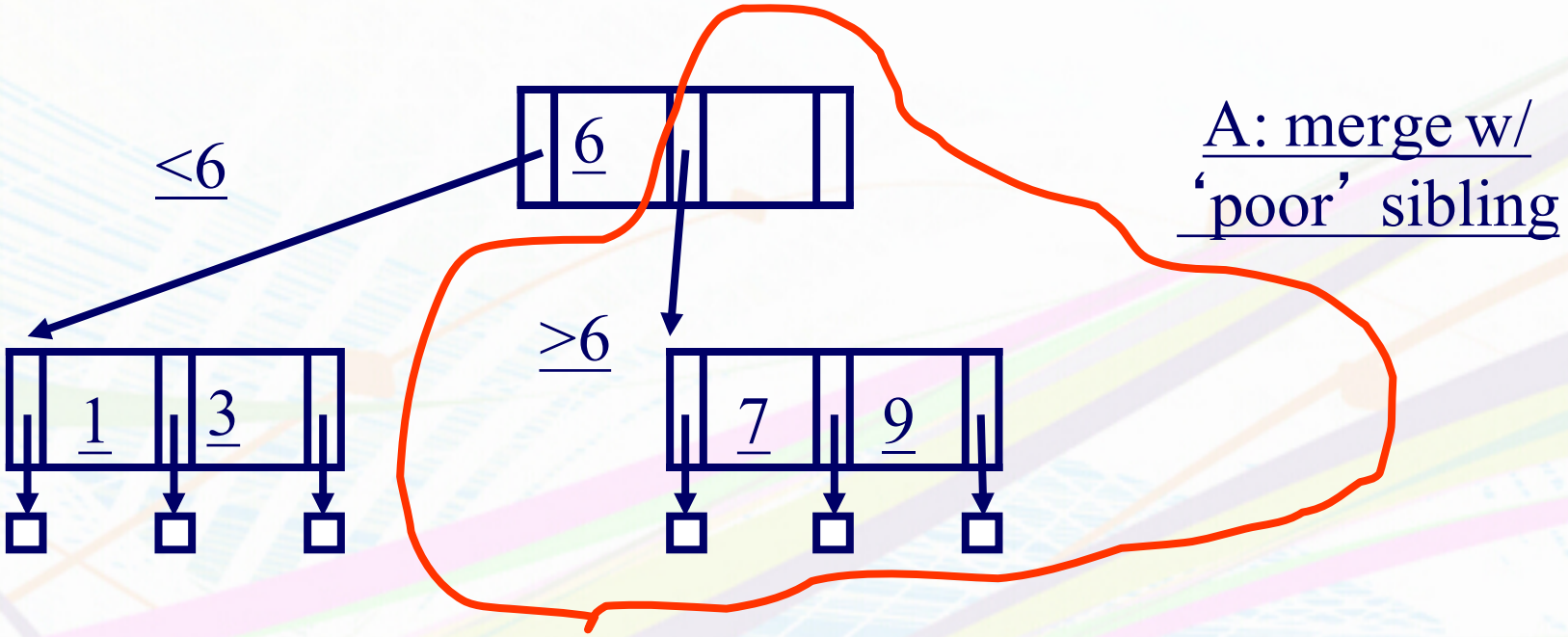
B-trees – Deletion

- **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’**
- **Ie.:**

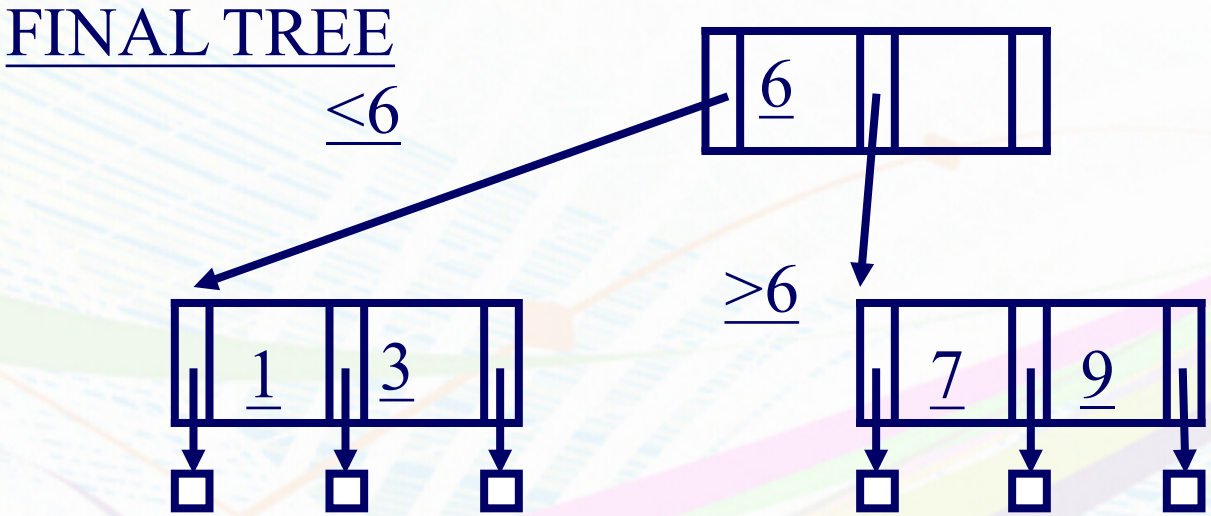
B-trees – Deletion

- **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’
- -> ‘pull key from parent, and merge’
- Q: What if the parent underflows?
- A: repeat recursively

B-tree deletion - pseudocode

```
DELETION OF KEY 'K'
  locate key 'K', in node 'N'
  if('N' is a non-leaf node) {
    delete 'K' from 'N';
    find the immediately largest key 'K1';
      /* which is guaranteed  to be on a leaf node 'L' */
    copy 'K1' in the old position of 'K';
    invoke this DELETION routine on 'K1' from the leaf
    node 'L';
  else {
/* 'N' is a leaf node */
... (next slide..)
```

B-tree deletion - pseudocode

```
/* 'N' is a leaf node */
    if( 'N' underflows ){
      let 'N1' be the sibling of 'N';
      if( 'N1' is "rich"){   /* ie., N1 can lend us a key */
        borrow a key from 'N1' THROUGH the parent node;
      }else{   /* N1 is 1 key away from underflowing */
        MERGE: pull the key from the parent 'P',
          and merge it with the keys of 'N' and 'N1' into a new
    node;
        if( 'P' underflows){ repeat recursively }
      }
  }
```

Outline

- Motivation
- ISAM
- B-trees
- Tree vs. Hash-based index
- Index organization: clustered vs. nonclustered

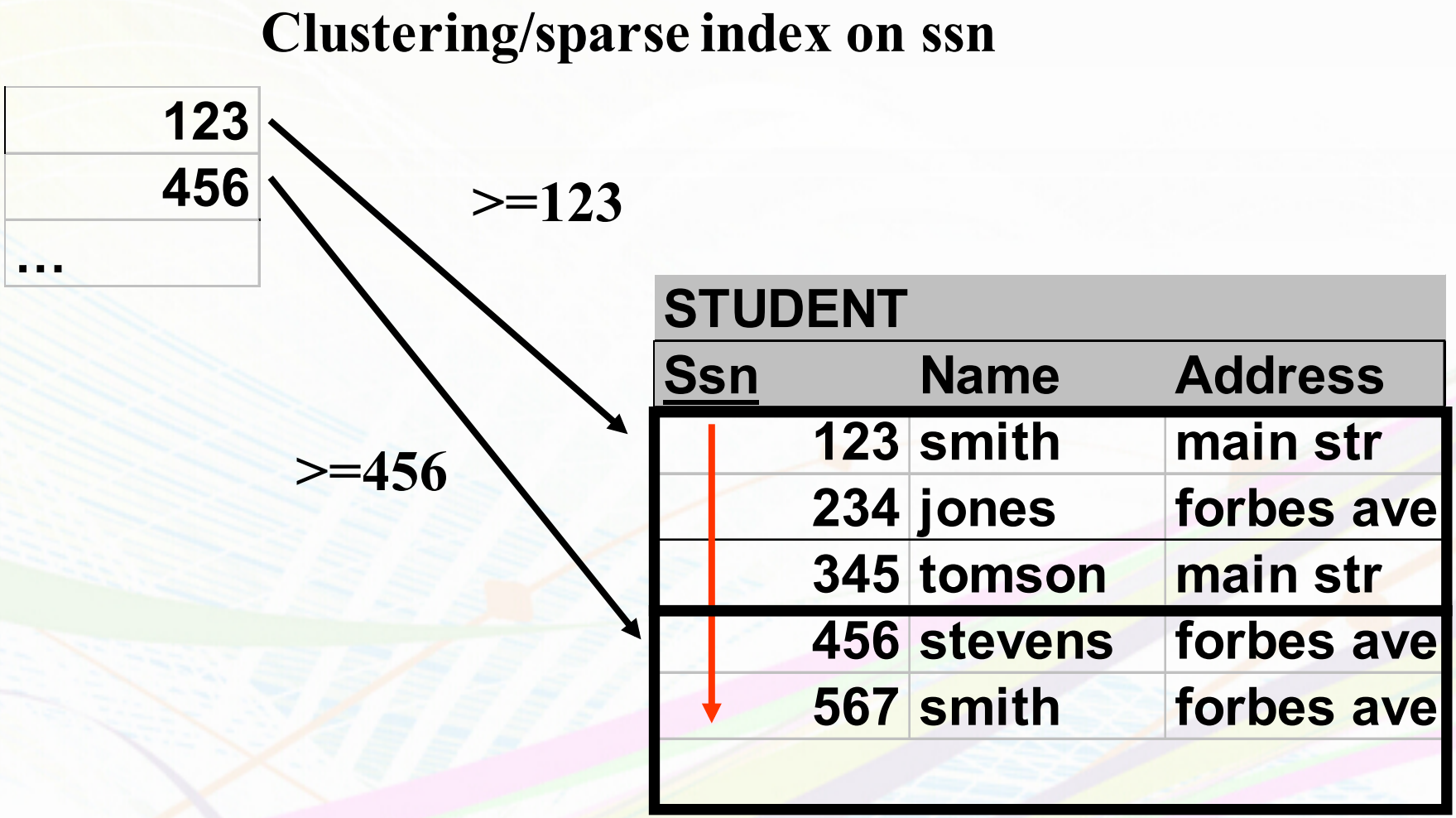
Tree vs. Hash-based index

- Hash-based index
  - Good for equality selections.
    - File = a collection of *buckets*. Bucket = *primary* page plus 0 or more *overflow* pages.
    - *Hash function* **h**: **h**(*r.search_key*) = bucket in which record *r* belongs.
- Tree-based index
  - Good for range selections, big indexes
    - Hierarchical structure (Tree) directs searches
    - Leaves contain data entries sorted by search key value
    - B+ tree: all root->leaf paths have equal length *(height)*

Outline

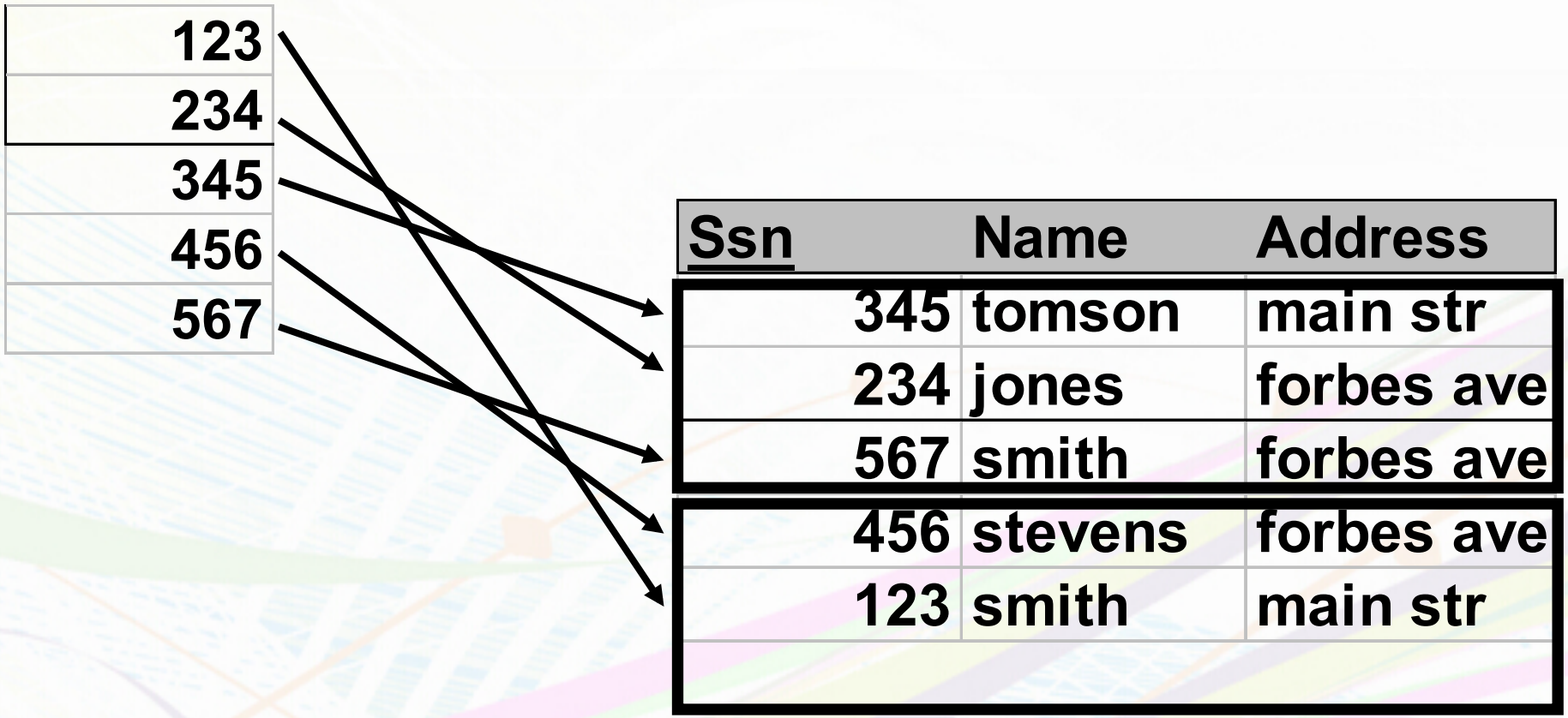
- Motivation
- ISAM
- B-trees
- Tree vs. Hash-based index
- Index organization: clustered vs. nonclustered

Indexing - clustered index example



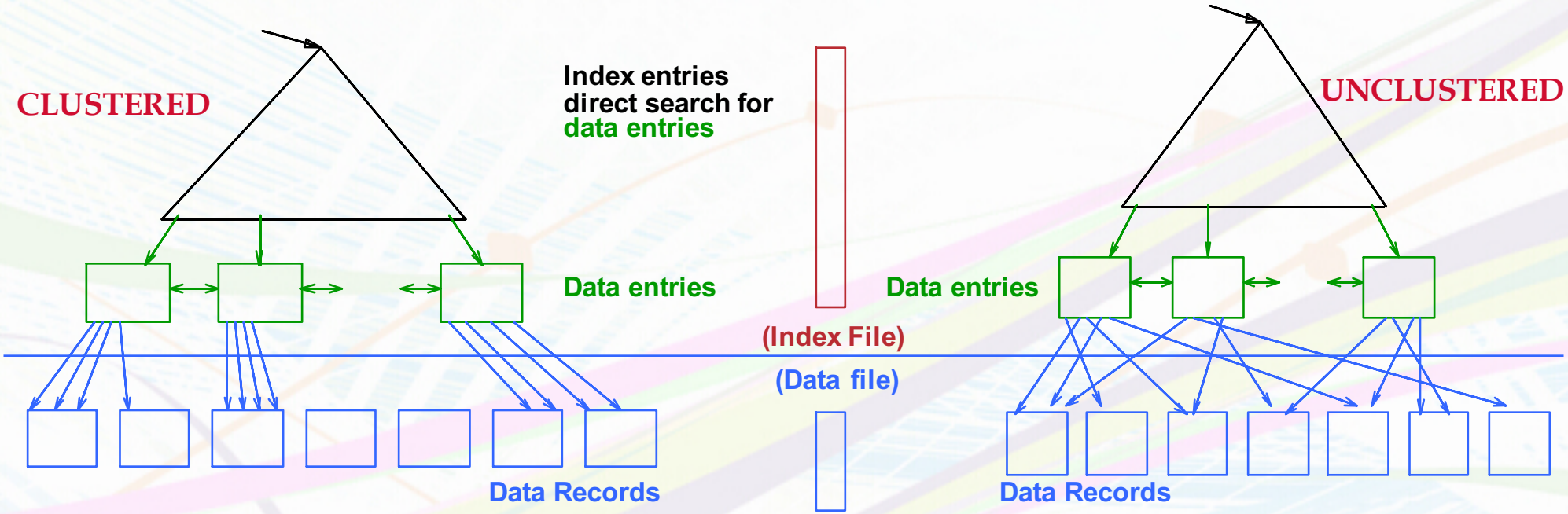
Indexing - non-clustered

Non-clustering / dense index



Index Classification - clustered

- *Clustered* vs. *unclustered*: If order of data records is the same as, or `close to', order of index data entries, then called *clustered index*.



Index Classification - clustered

- A file can have a clustered index on at most one search key.
- Cost of retrieving data records through index varies *greatly* based on whether index is clustered!
- Note: Alternative 1 implies clustered, *but not vice-versa*.

But, for simplicity, you may think of them as equivalent..

Clustered vs. Unclustered Index

- Cost of retrieving records found in range scan:
  - Clustered: cost =
  - Unclustered: cost ≈
- What are the tradeoffs????

Clustered vs. Unclustered Index

- Cost of retrieving records found in range scan:
  - Clustered: cost = # pages in file w/matching records
  - Unclustered: cost ≈ # of matching index data entries
- What are the tradeoffs????

Clustered vs. Unclustered Index

- Cost of retrieving records found in range scan:
  - Clustered: cost = # pages in file w/matching records
  - Unclustered: cost ≈ # of matching index data entries
- What are the tradeoffs????
  - Clustered Pros:
    - Efficient for range searches
    - May be able to do some types of compression
  - Clustered Cons:
    - Expensive to maintain (on the fly or sloppy with reorganization)

Chapter 8, 10

B+ tree

Exercises: 10.2-10.5, 10.8