CSE544
Data Management

Lectures 13:
Indexes and Bloom Filters

Outline

- B+ Trees
- Bloom Filters
- Next time:
  – Learned indexes (paper!), LSM trees
  – Note: Tim Kraska’s talk May 24, 9am

Terminology

A *dictionary* is a main memory data structure that supports:

- Insert(k,v) = insert a key,value pair
- Find(k) = find value of key k

- Variations: key may not be unique
- An *index* is a disk bound dictionary

A *Bloom Filter* is a main memory data structure that supports

- Insert(k) = insert k (no value)
- Member(k) = check k; false positives OK!

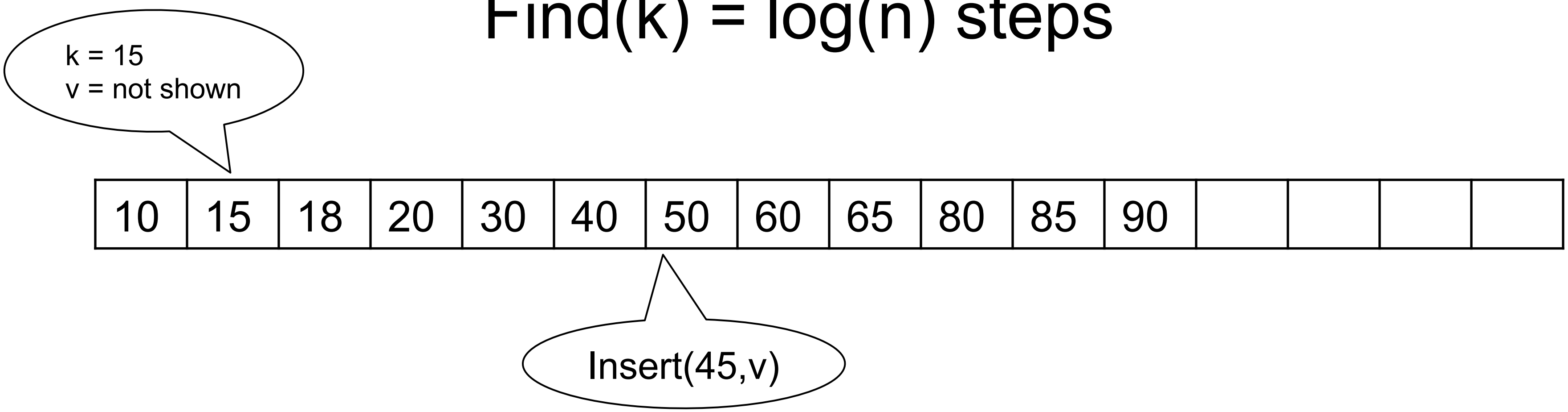
Best Dictionary for Find

Sorted array!
Find(k) = log(n) steps

Best Dictionary for Find

Sorted array!

Find(k) = log(n) steps



Very bad for insert(k,v)

Best Dictionary for Insert

A log file! (many other names)
Insert(k,v) = O(1) steps

Best Dictionary for Insert

A log file! (many other names)
Insert(k,v) = O(1) steps

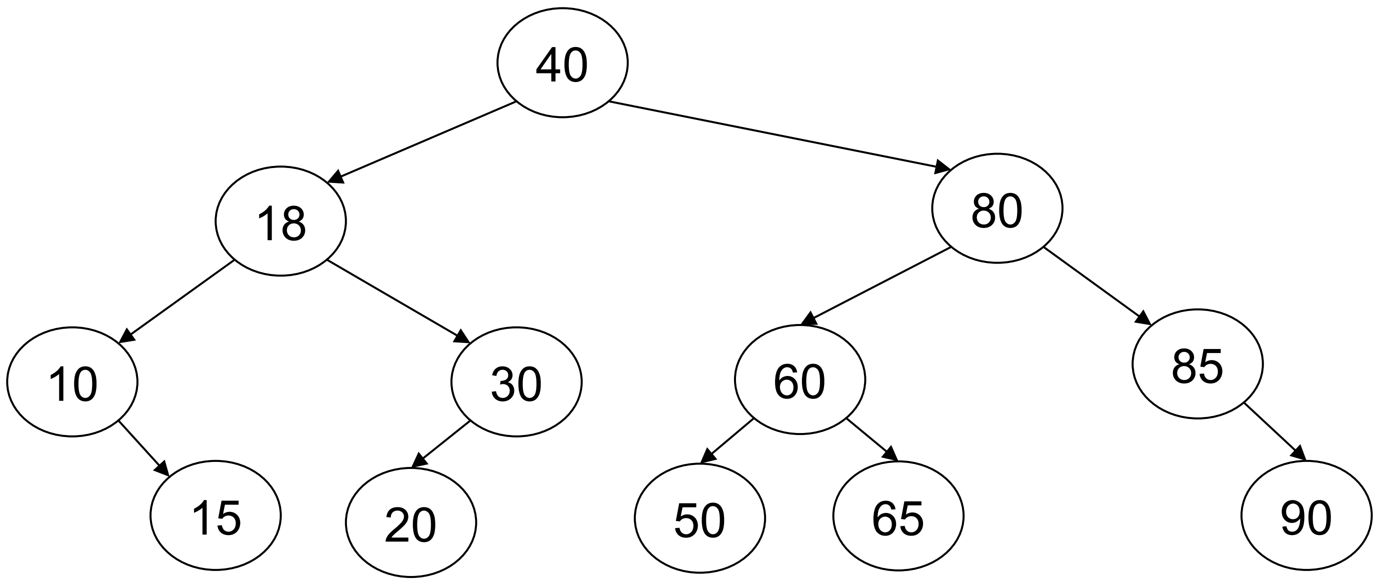
50	15	65	20	80	40	18	60	90	10	85	30				

Insert(45,v)

Very bad for find(k)

Compromise: Search Trees

Find(k) = O(log(n)) steps
Insert(k,v) = O(log(n)) steps



10	15	18	20	30	40	50	60	65	80	85	90

Compromise: Search Trees

- Main challenge: ensure height=O(log n)
- Many techniques:
  - Red/black trees
  - Splay trees
  - …
  - B-trees: special case 2-3 trees

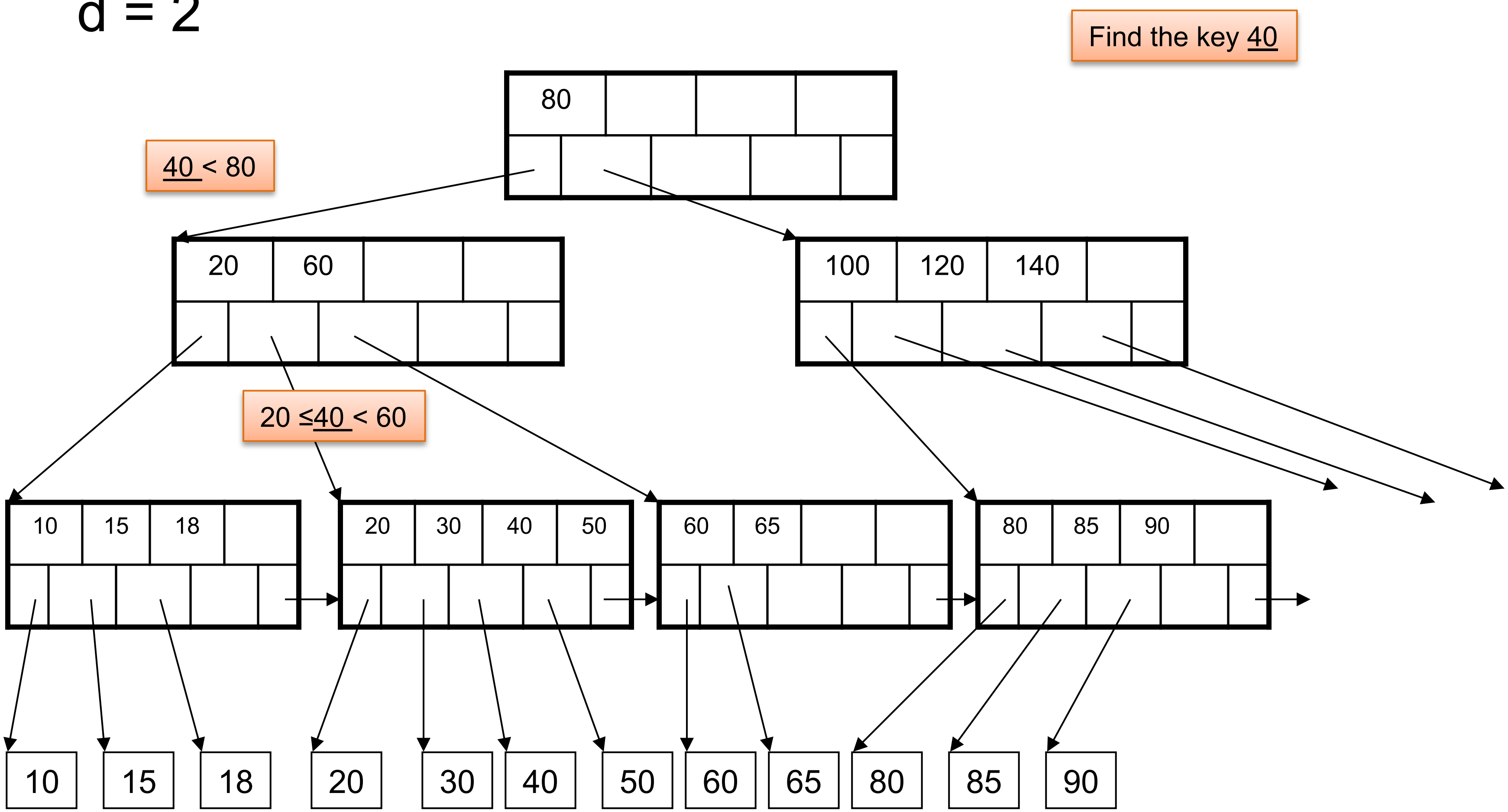
B Trees, B+ Trees

- B-tree on disk
  - Make 1 node = 1 page (= 1 block)

- B trees to B+ trees:
  - Keys are stored on the leaves (not internal nodes)
  - Leaves are linked in a list: for range queries

B+ Tree Example

d = 2



B+ Trees Properties

- For each node except the root, maintain 50% occupancy of keys
- Insert and delete must rebalance to maintain constraints

B+ Trees Details

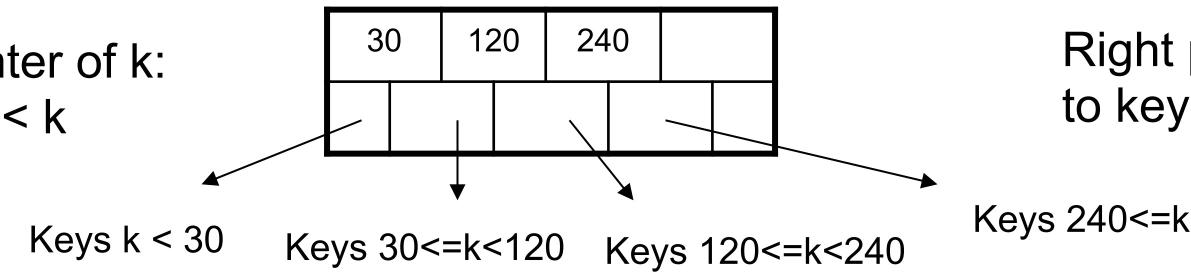
- Parameter* d = the *degree*
- Each node has **d <= m <= 2d keys** (except root)

* Textbooks define the *order* of the B tree as 2d+1

B+ Trees Details

- Parameter* d = the *degree*
- Each node has **d <= m <= 2d keys** (except root)
- Each node also has **m+1 pointers**

Left pointer of k:
to keys < k

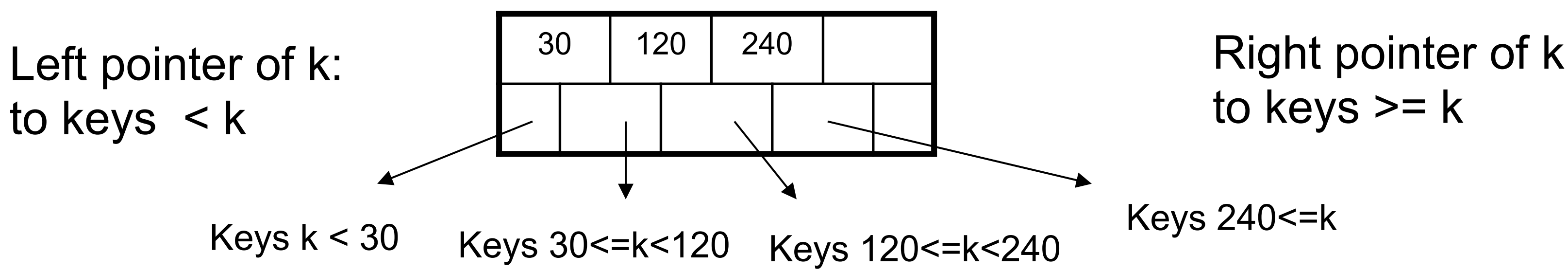


Right pointer of k:
to keys >= k

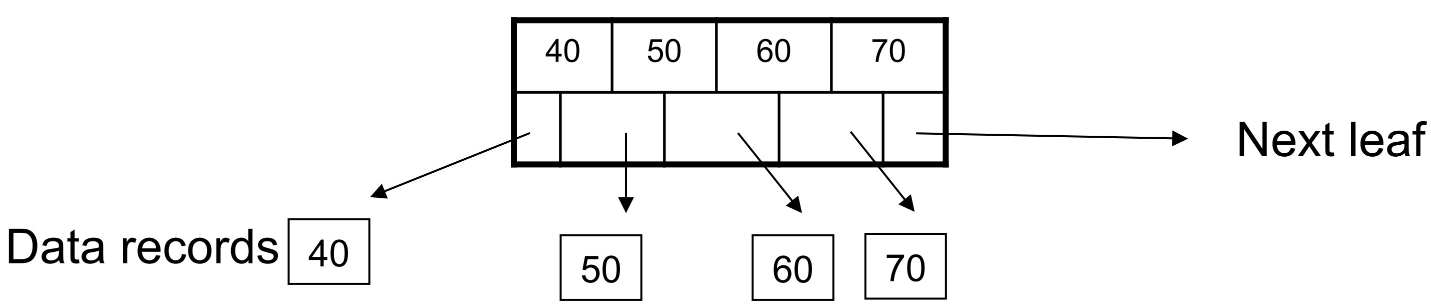
* Textbooks define the *order* of the B tree as 2d+1

B+ Trees Details

- Parameter* d = the *degree*
- Each node has **d <= m <= 2d keys** (except root)
- Each node also has **m+1 pointers**

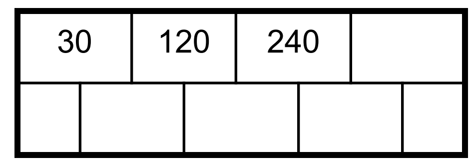


- Each leaf has **d <= m <= 2d keys**:



B+ Tree Design

- How large d ?   Make one node fit on one block
- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 bytes



(e.g. d = 2)

- 2d x 4 + (2d+1) x 8 <= 4096
- d = 170

B+ Trees in Practice

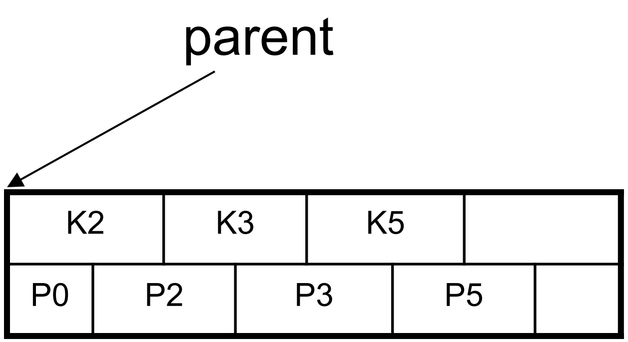
- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 Mbytes

Insertion in a B+ Tree

Insert (K, P)

- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt

Insert k1

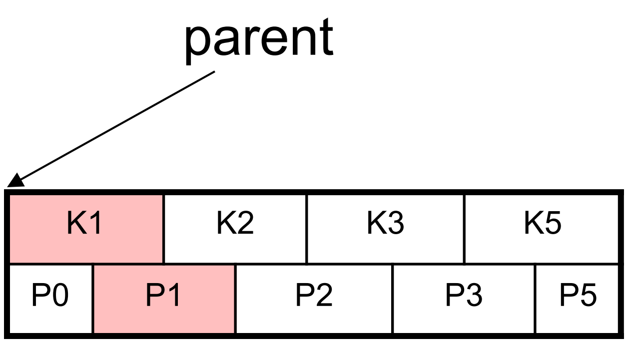


Insertion in a B+ Tree

Insert (K, P)

- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt

Insert k1

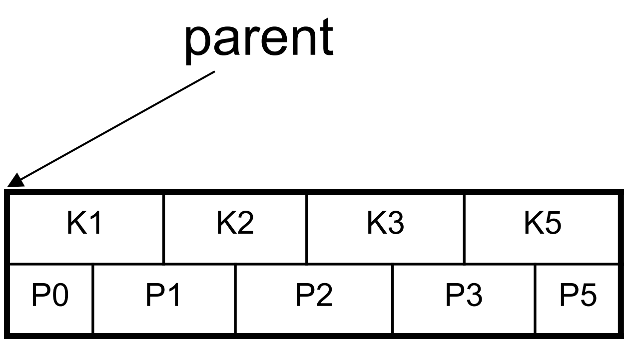


Insertion in a B+ Tree

Insert (K, P)

- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt

Insert k4

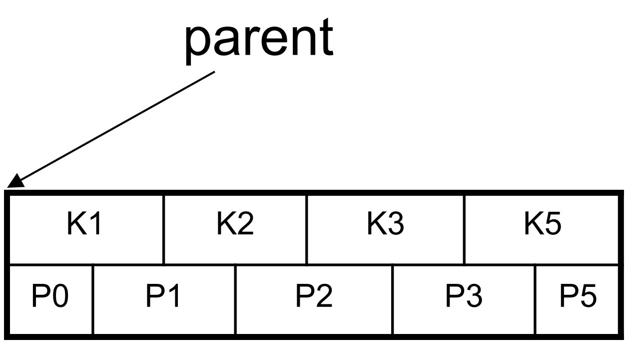


Insertion in a B+ Tree

Insert (K, P)

- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:

Insert k4

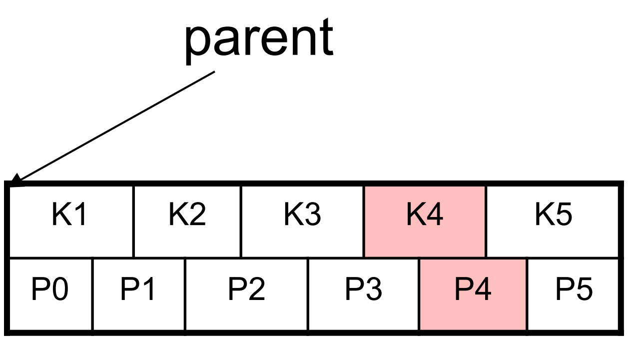


Insertion in a B+ Tree

Insert (K, P)

- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:

Insert k4

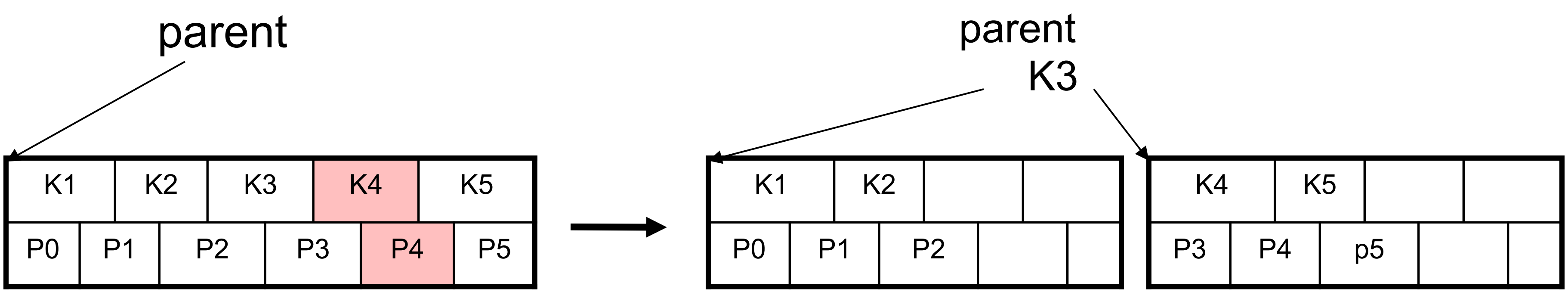


Insertion in a B+ Tree

Insert (K, P)

- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:

Insert k4

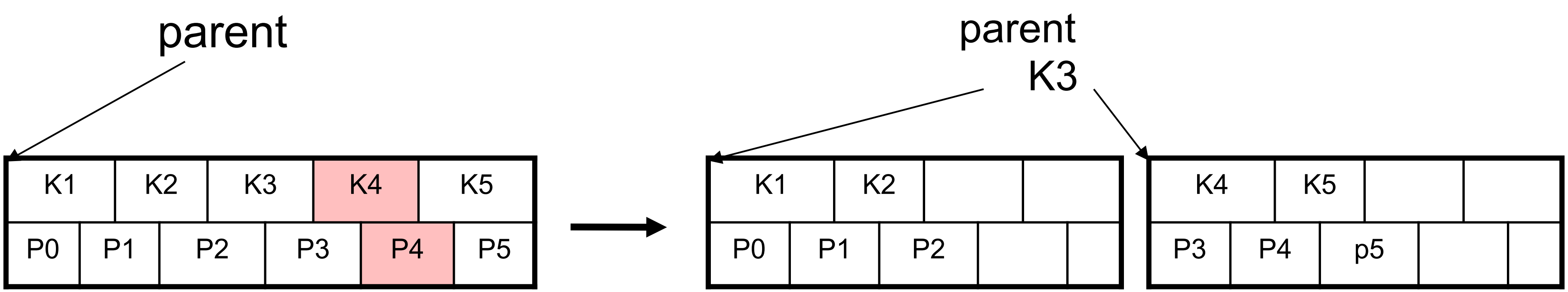


Insertion in a B+ Tree

Insert (K, P)

- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:

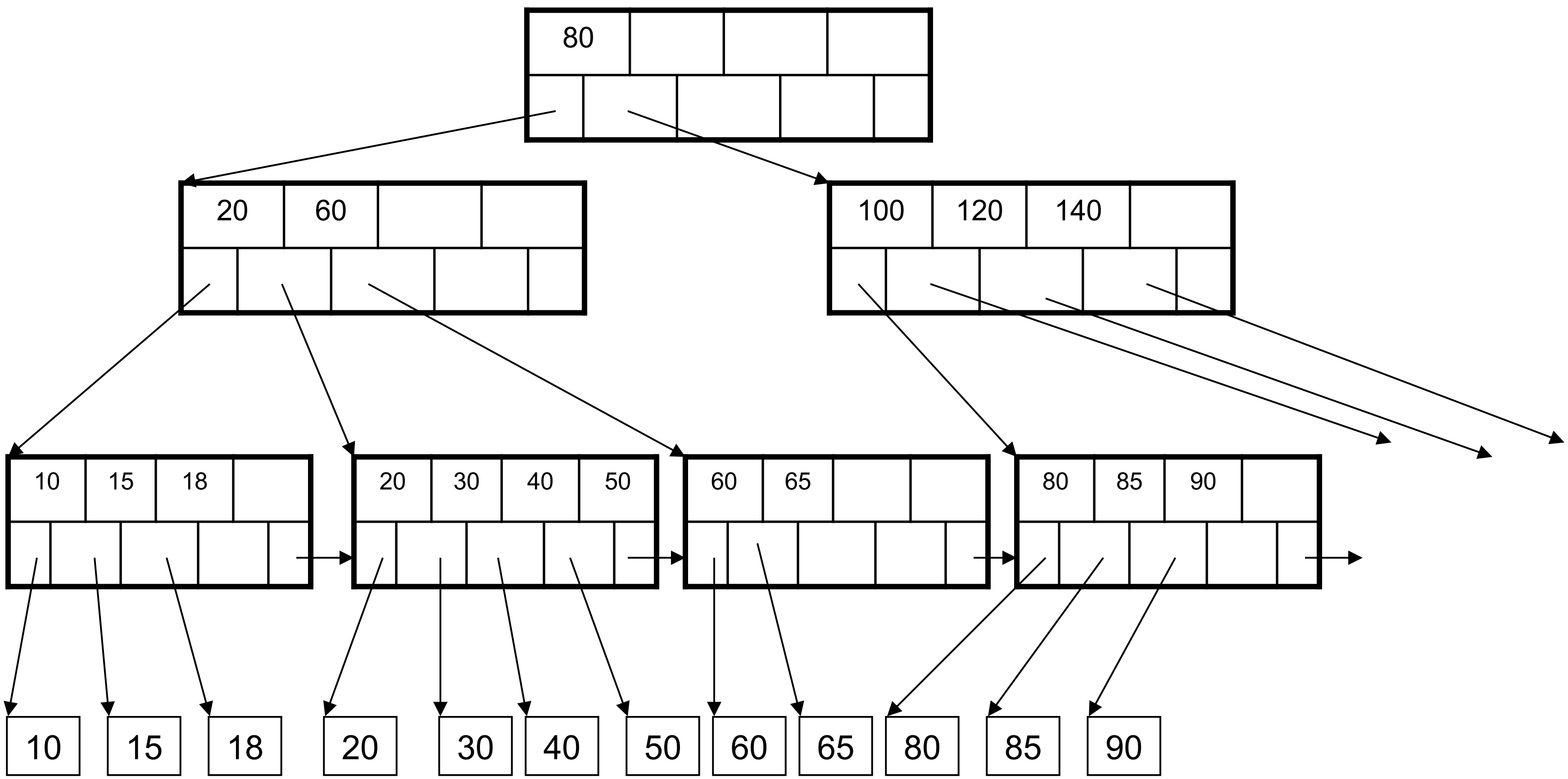
Insert k4



- If leaf, also keep K3 in right node
- When root splits, new root has 1 key only

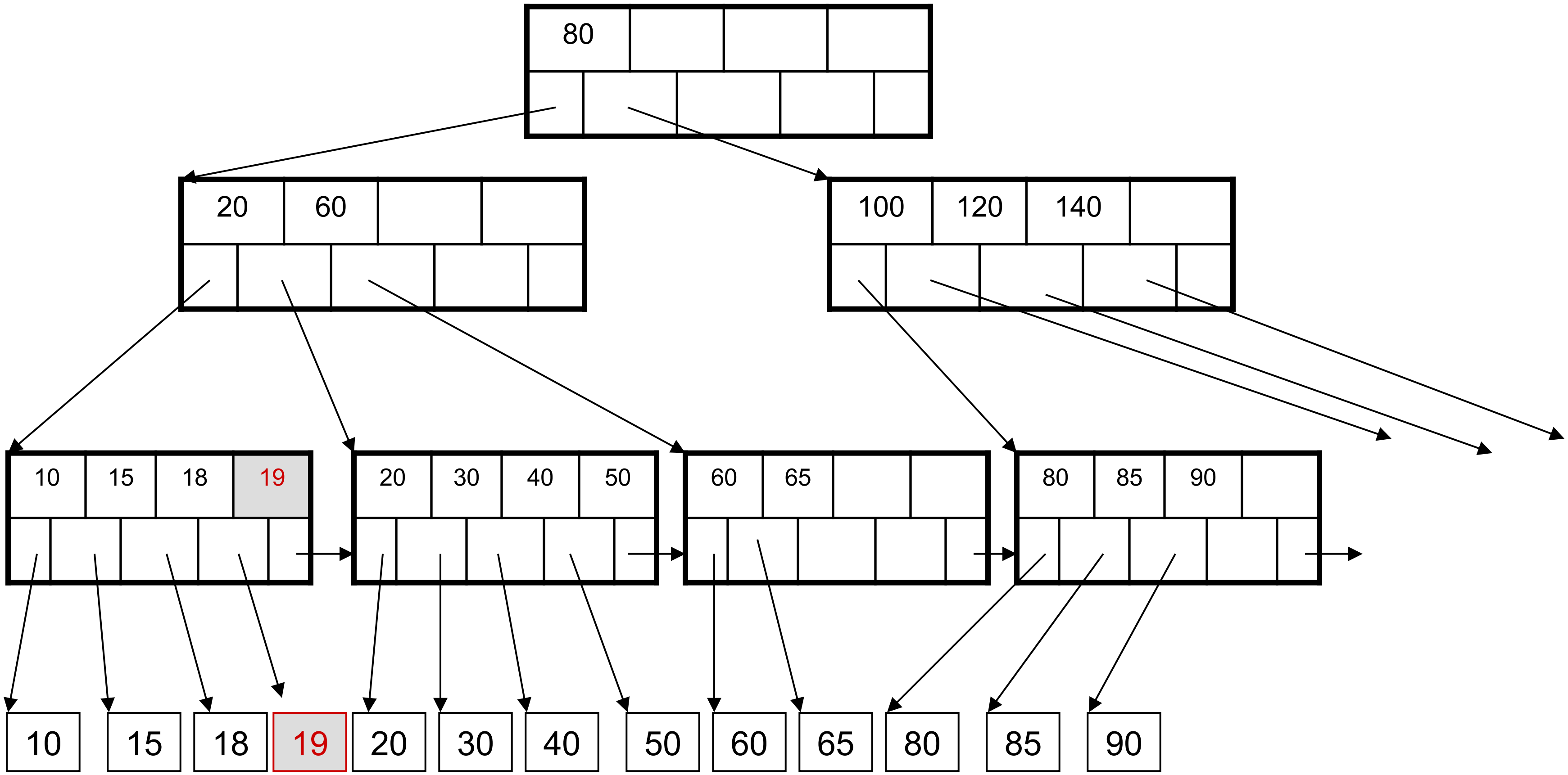
Insertion in a B+ Tree

Insert K=19



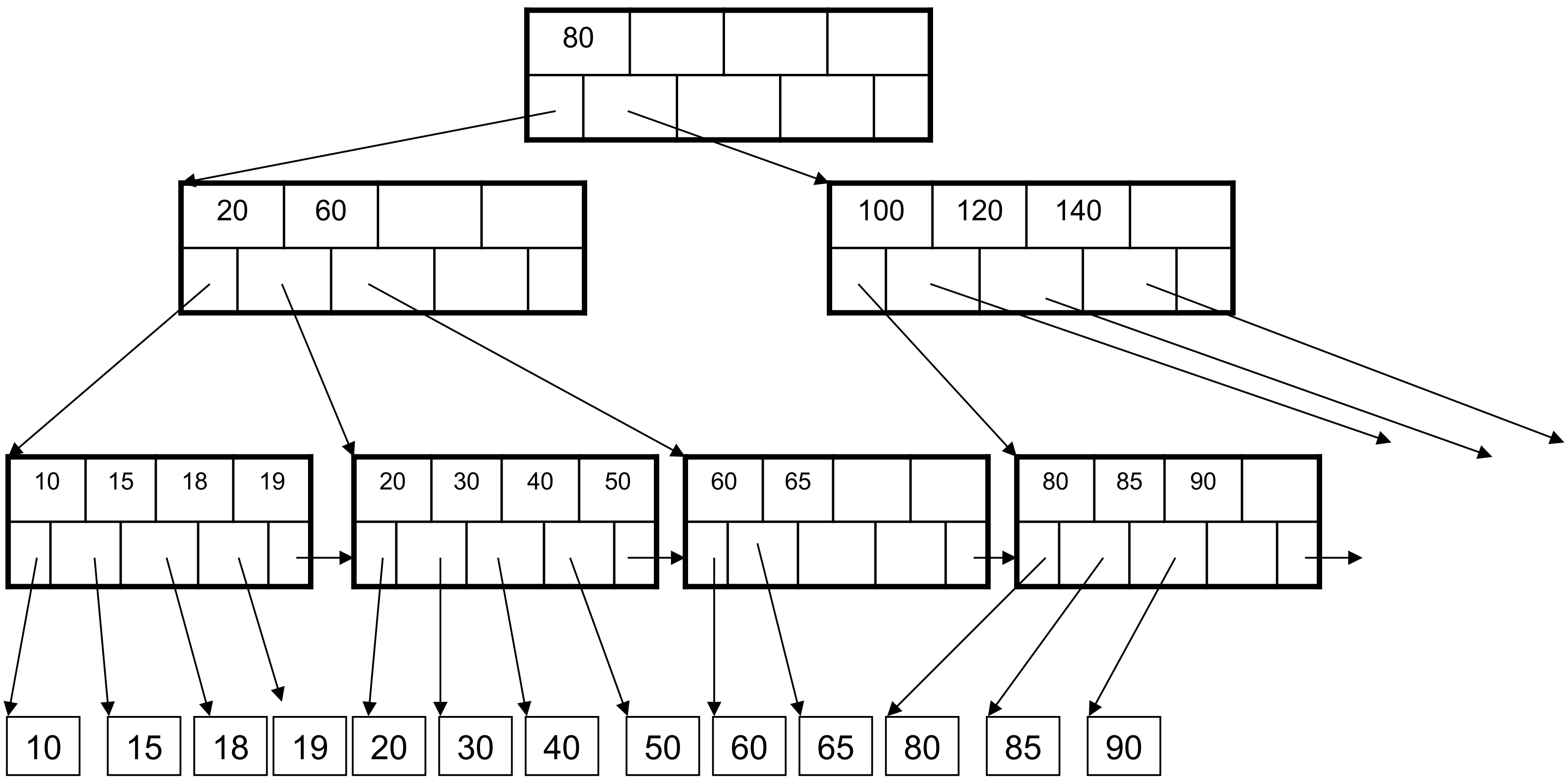
Insertion in a B+ Tree

After insertion



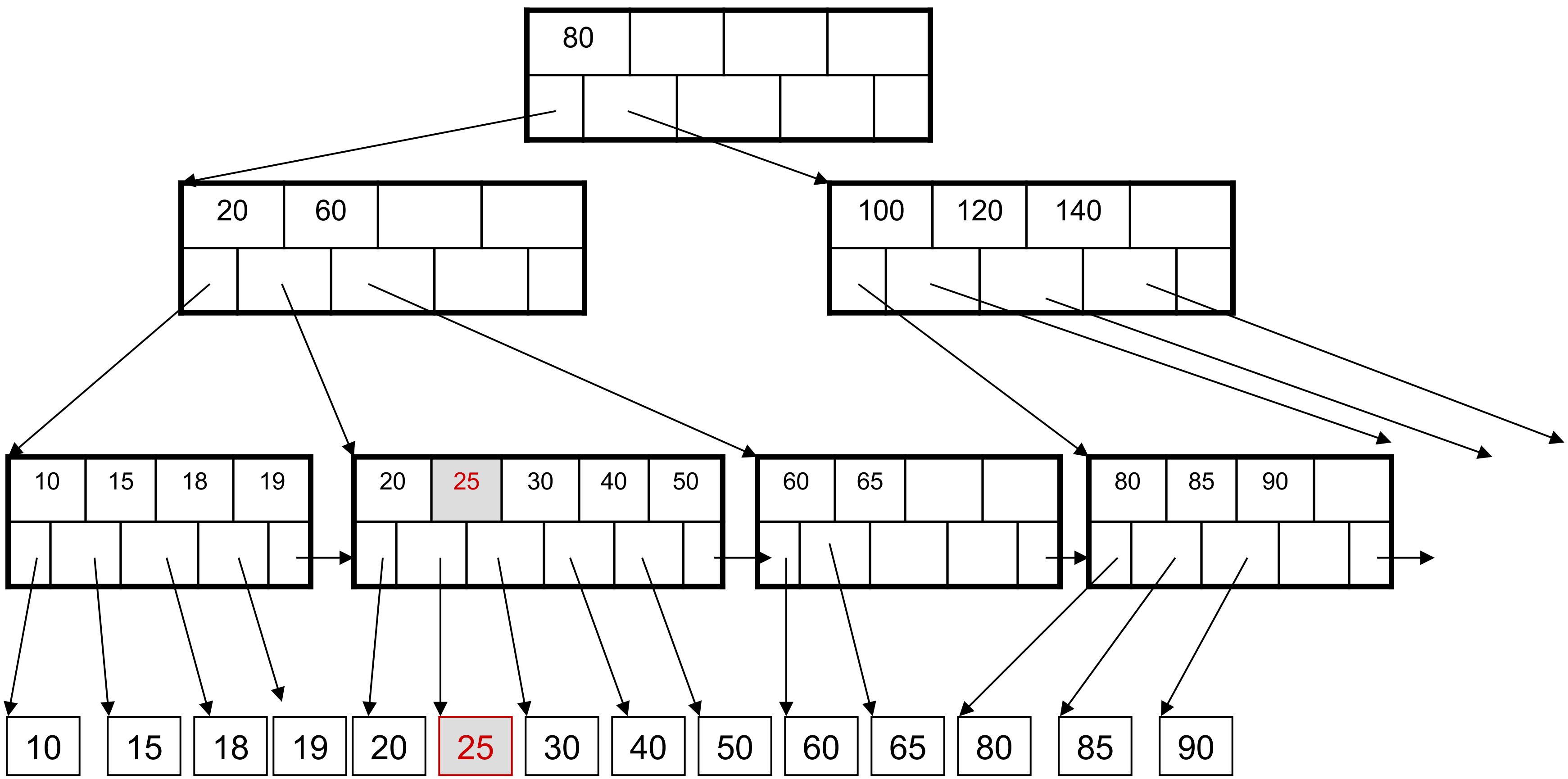
Insertion in a B+ Tree

Now insert 25



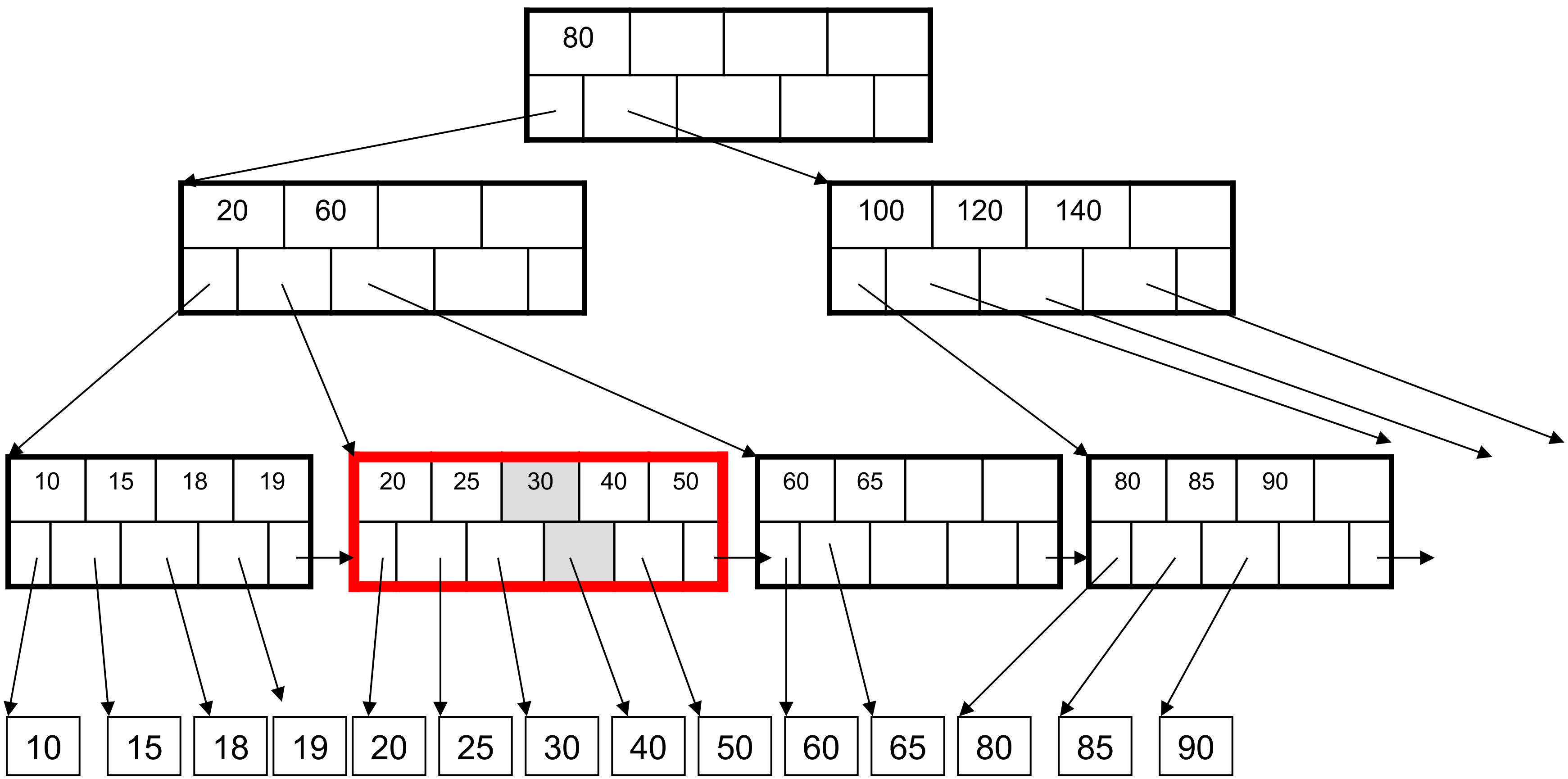
Insertion in a B+ Tree

After insertion



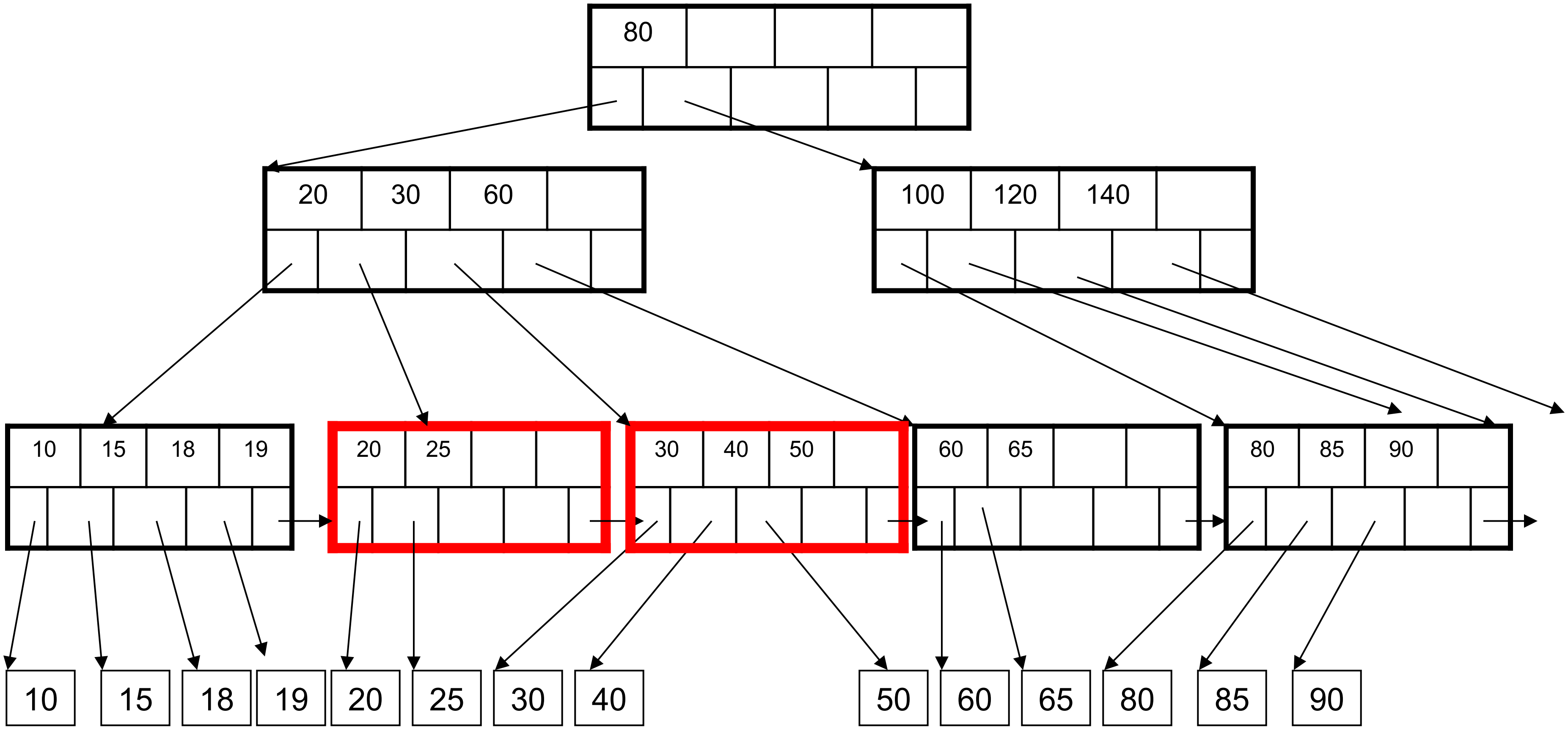
Insertion in a B+ Tree

But now have to split !



Insertion in a B+ Tree

After the split



Insert: Summary

- Find the leaf, insert it there
- If current node p too big:
  – Split it into two nodes: p, p’
  – Insert(k,p’) where k = some separator
  – Recurse on the parent (may split again)
- If root node p splits:
  – New root: key k, children p, p’

All leaf nodes remain at the same depth

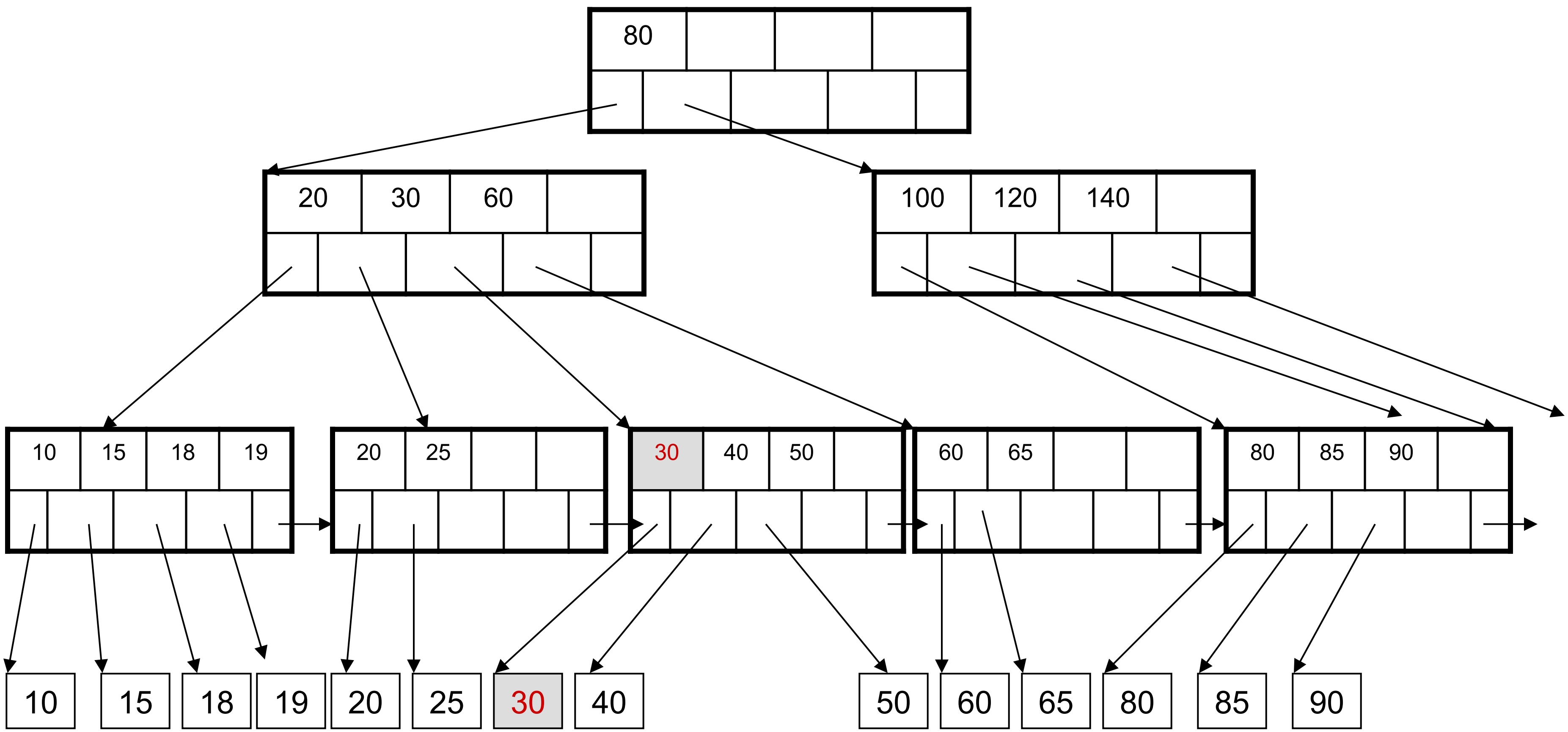
Deletion in a B+ Tree

Delete (K, P)

- Find leaf node where K belongs, delete
- Check for capacity; if above min capacity: **Stop**
- If node below capacity, try to rotate from sibling then **Stop**
- If adjacent nodes are at minimum capacity, then merge: This removes a key/child from parent; **Recurse** on parent

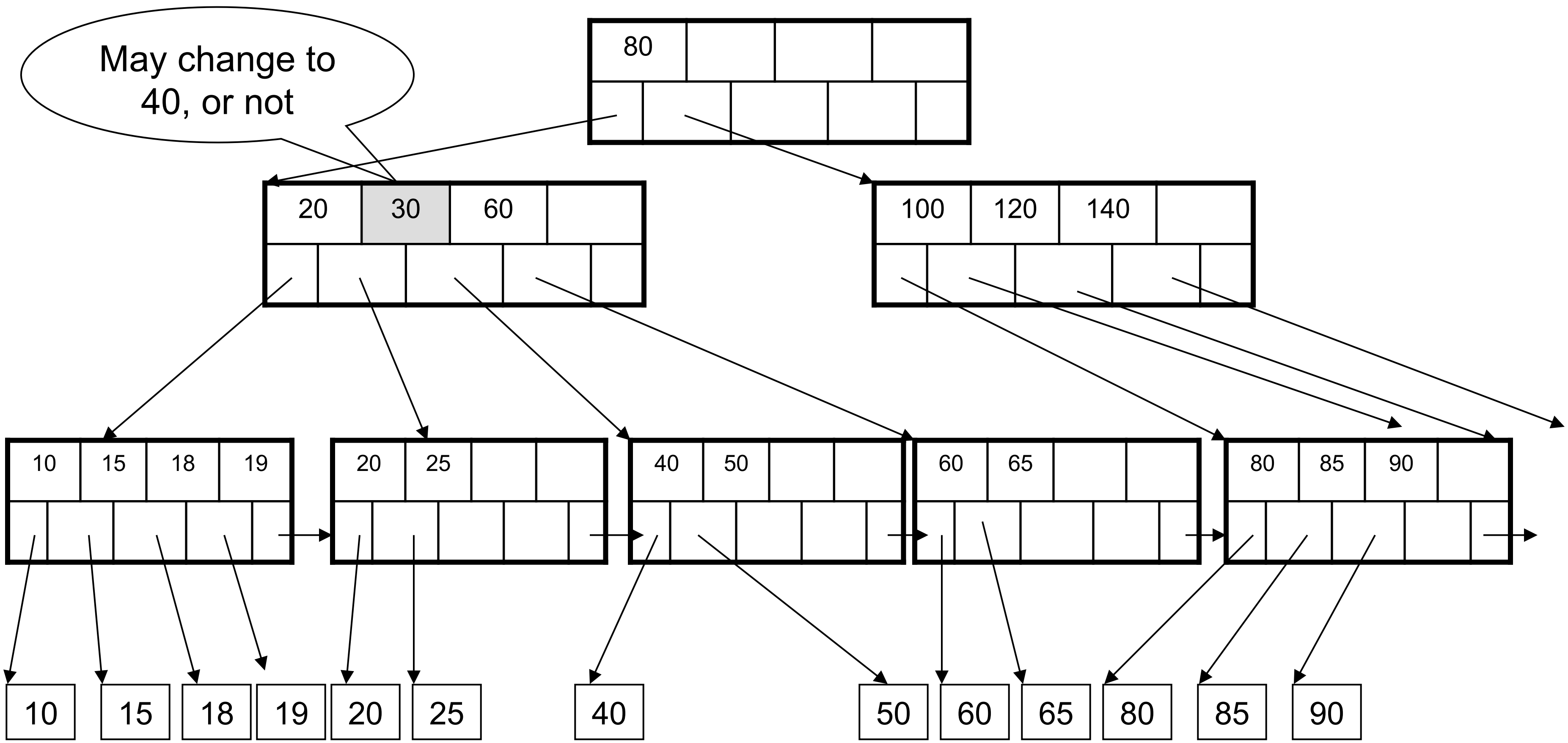
Deletion from a B+ Tree

Delete 30



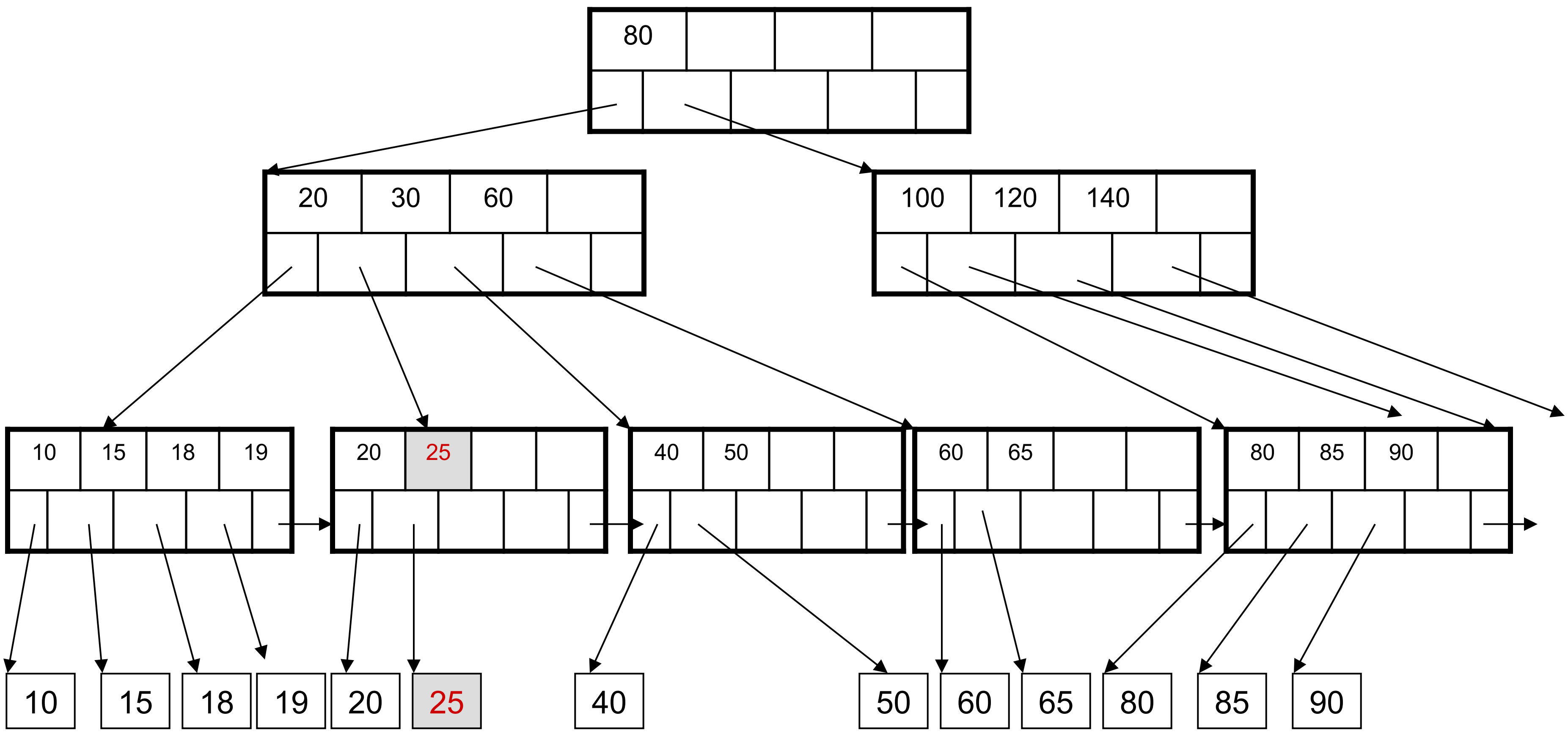
Deletion from a B+ Tree

After deleting 30



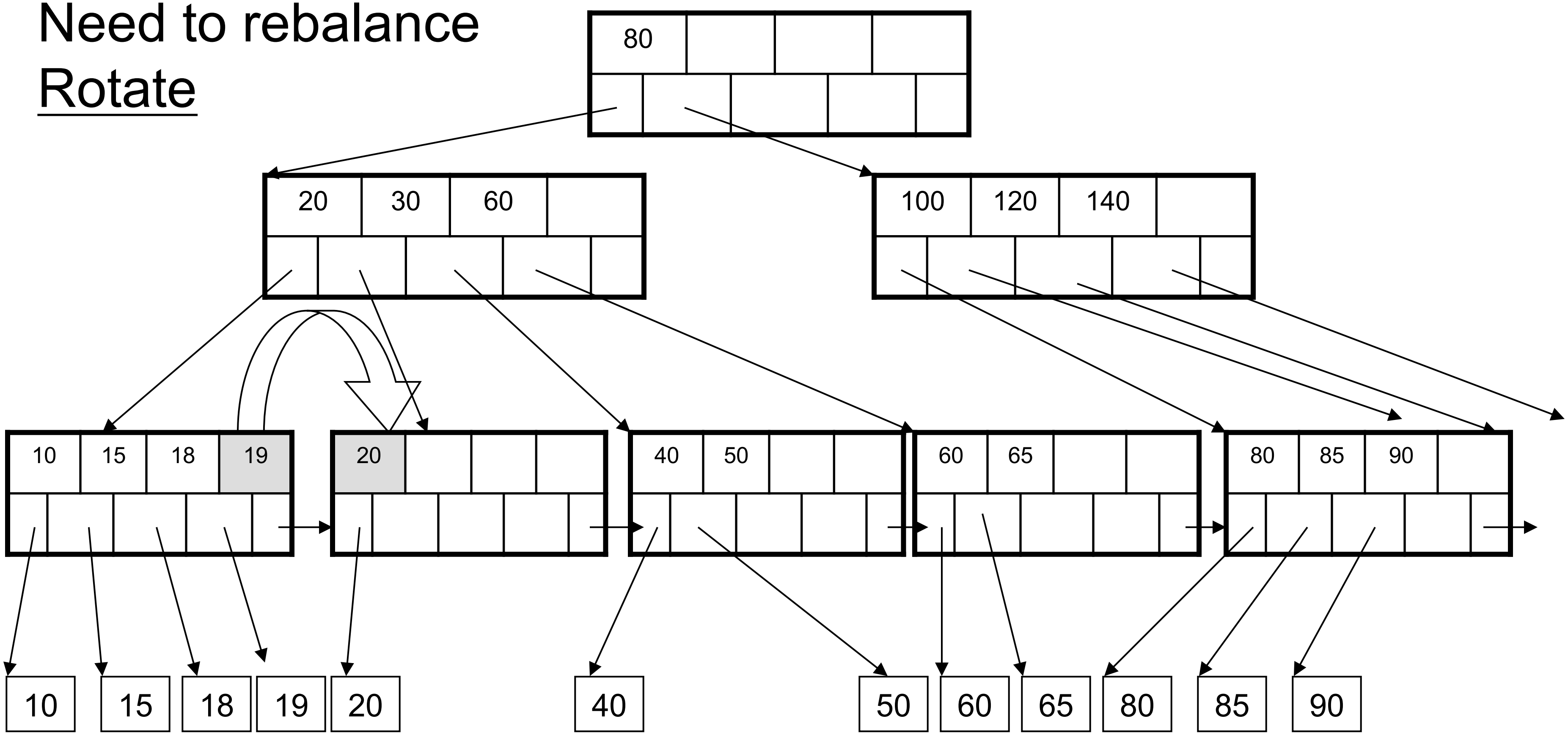
Deletion from a B+ Tree

Now delete 25

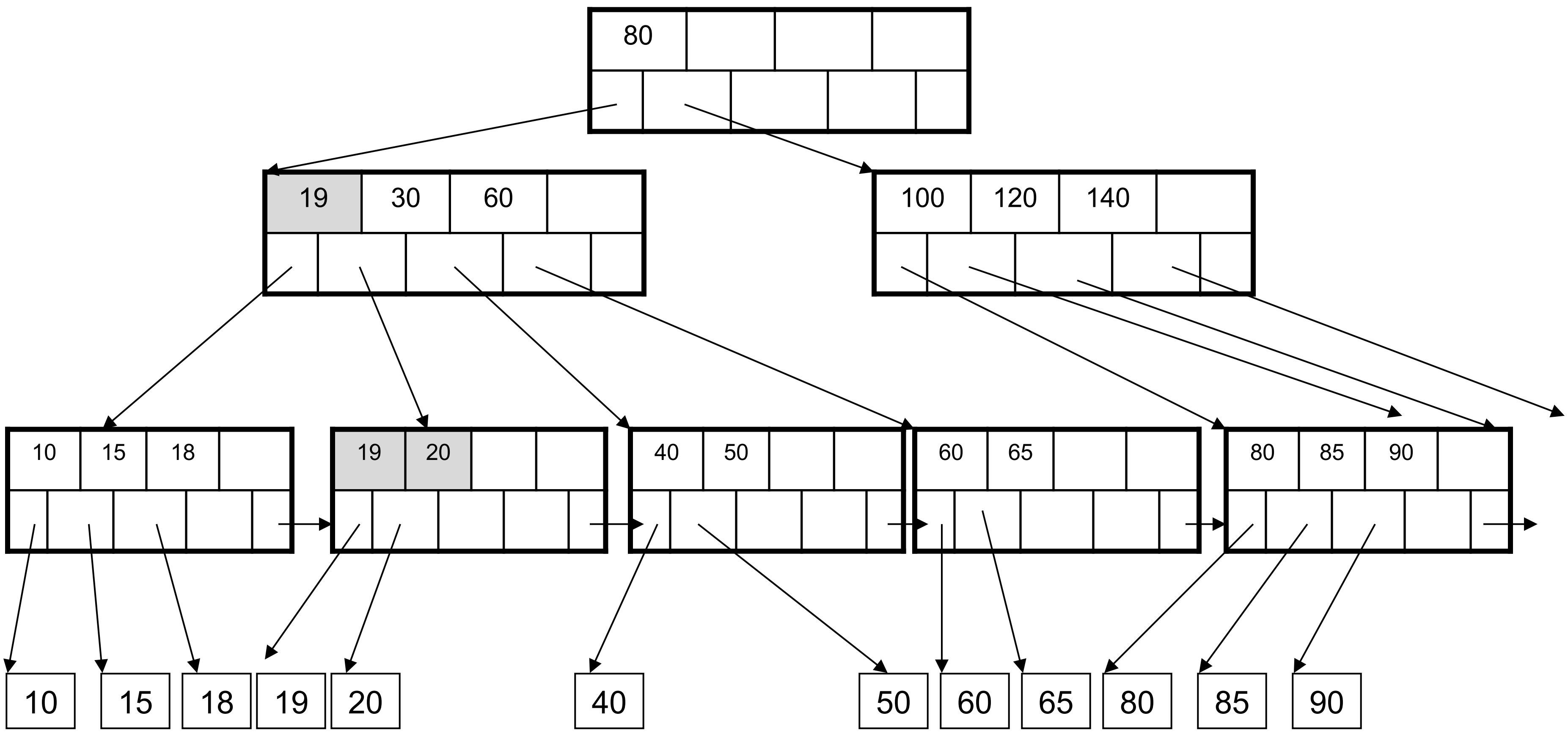


Deletion from a B+ Tree

After deleting 25
Need to rebalance
Rotate

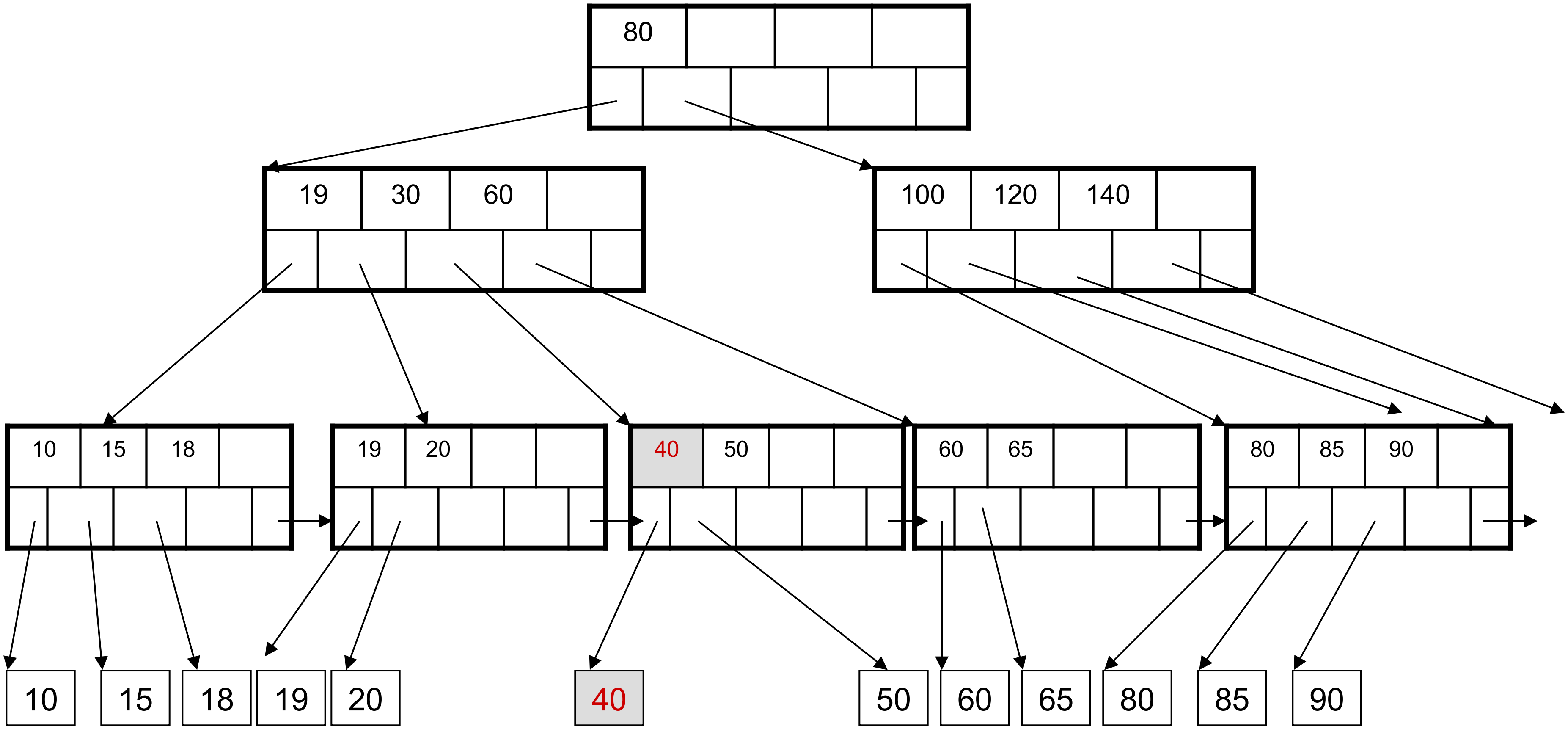


Deletion from a B+ Tree



Deletion from a B+ Tree

Now delete 40

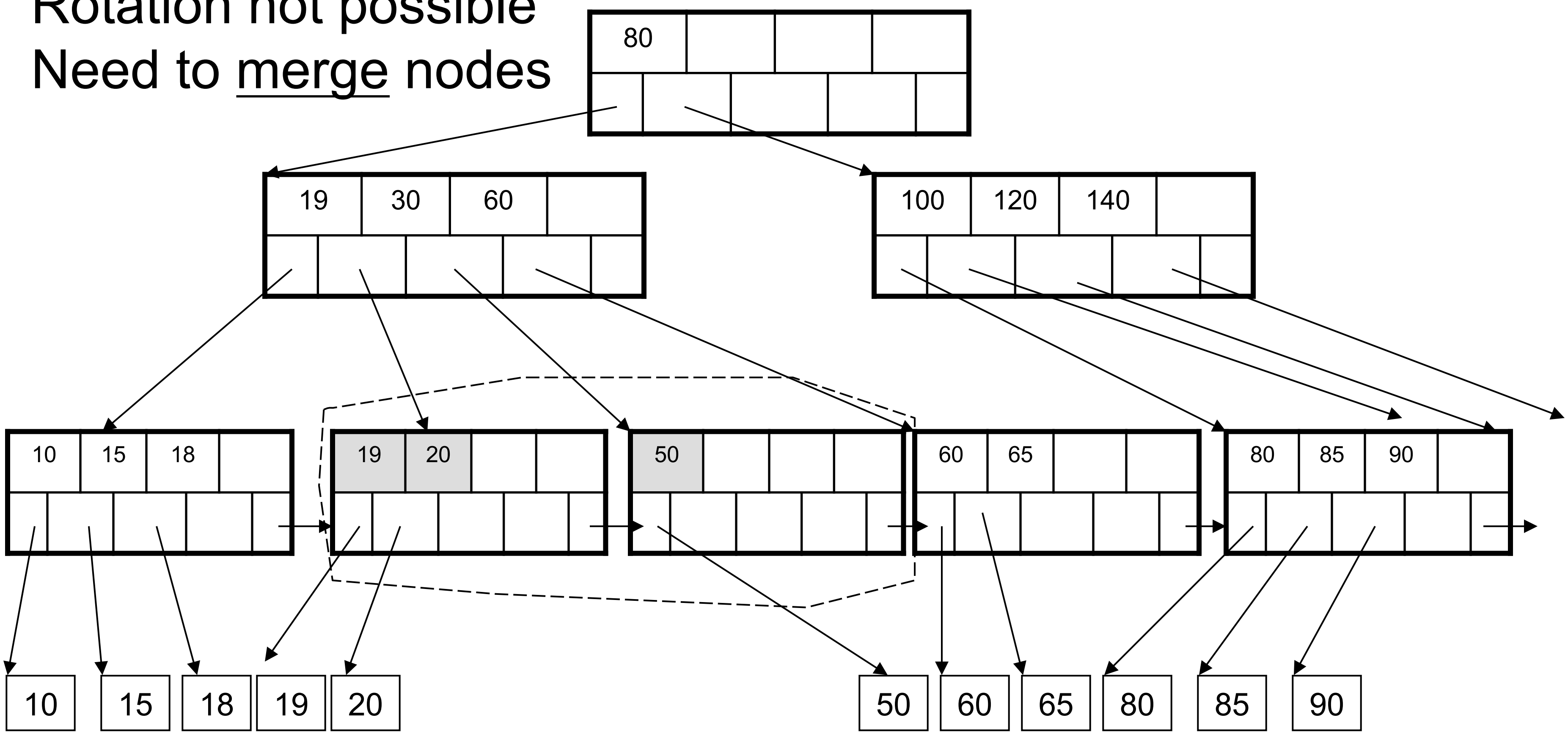


Deletion from a B+ Tree

After deleting 40

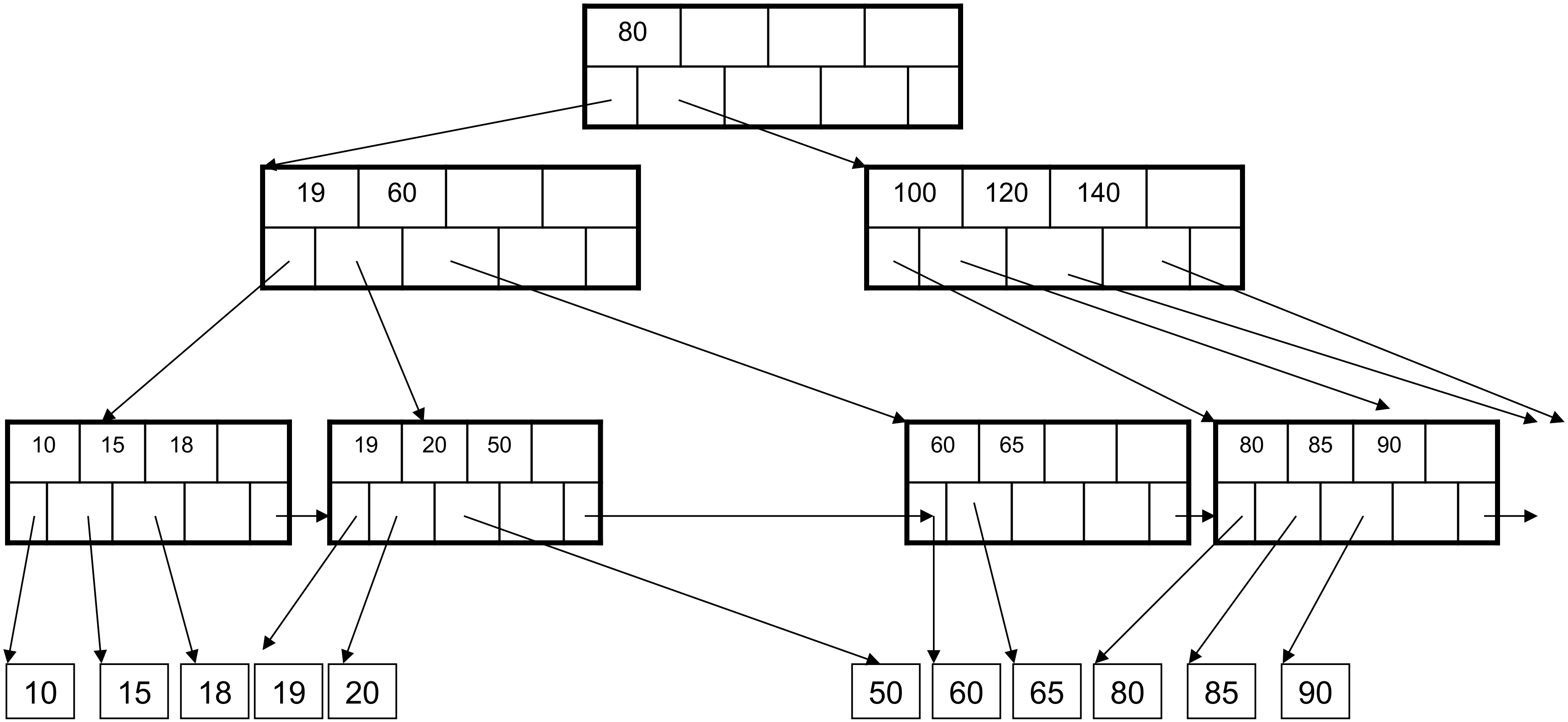
Rotation not possible

Need to merge nodes



Deletion from a B+ Tree

Final tree



Deletion: Summary

- Find key in the leaf node, delete it
- If current node p below min-capacity:
  - Try to rotate and **Stop**
  - merge with a neighbor, recurse on parent
- If root node p below min-capacity:
  - Delete the root node! (Has 1 child only)

All leaf nodes remain at the same depth

Discussion

- Reads are very fast
- Inserts are slow in two settings:
  - Initial data upload
  - Write-intensive workloads

Problem 1: Initial Data Upload

- Suppose you are inserting 10^6 records

- For each insert:
  - At least one random write
  - At worst O(log n) random writes

- Better: insert the data first, construct index later, using bulk index creation

Bulk Index Creation

Sort data first, then build the tree

50	15	65	20	80	18	60	90	10	85	19	…	…			∞

Bulk Index Creation

Sort data first, then build the tree

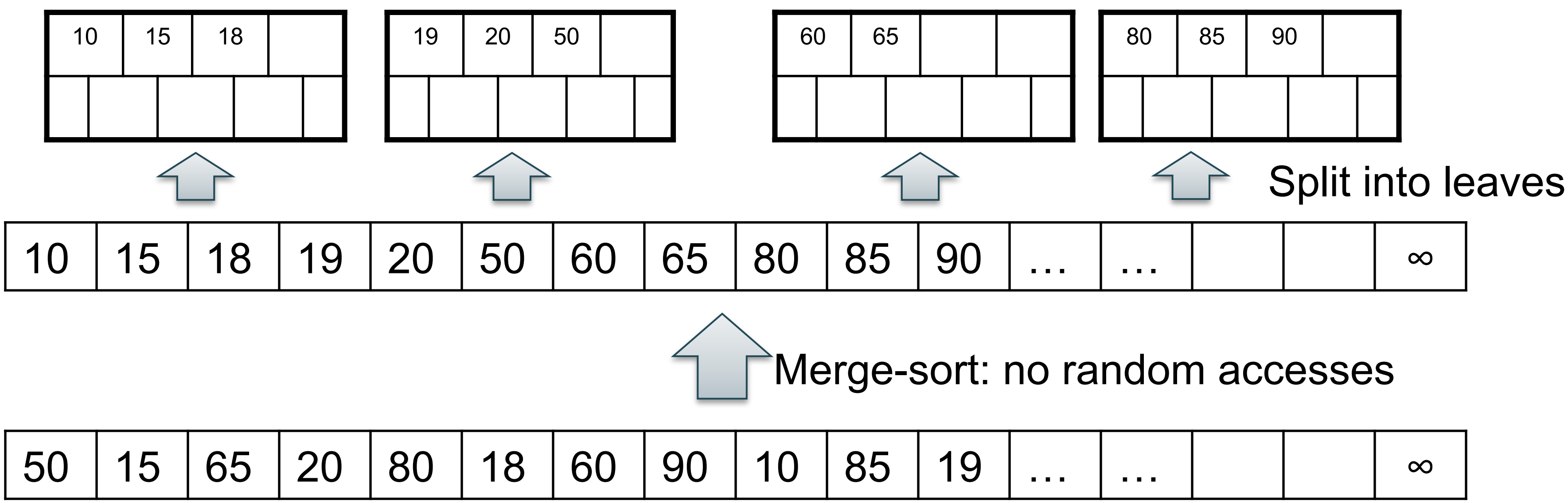
10	15	18	19	20	50	60	65	80	85	90	…	…			∞

Merge-sort: no random accesses

50	15	65	20	80	18	60	90	10	85	19	…	…			∞

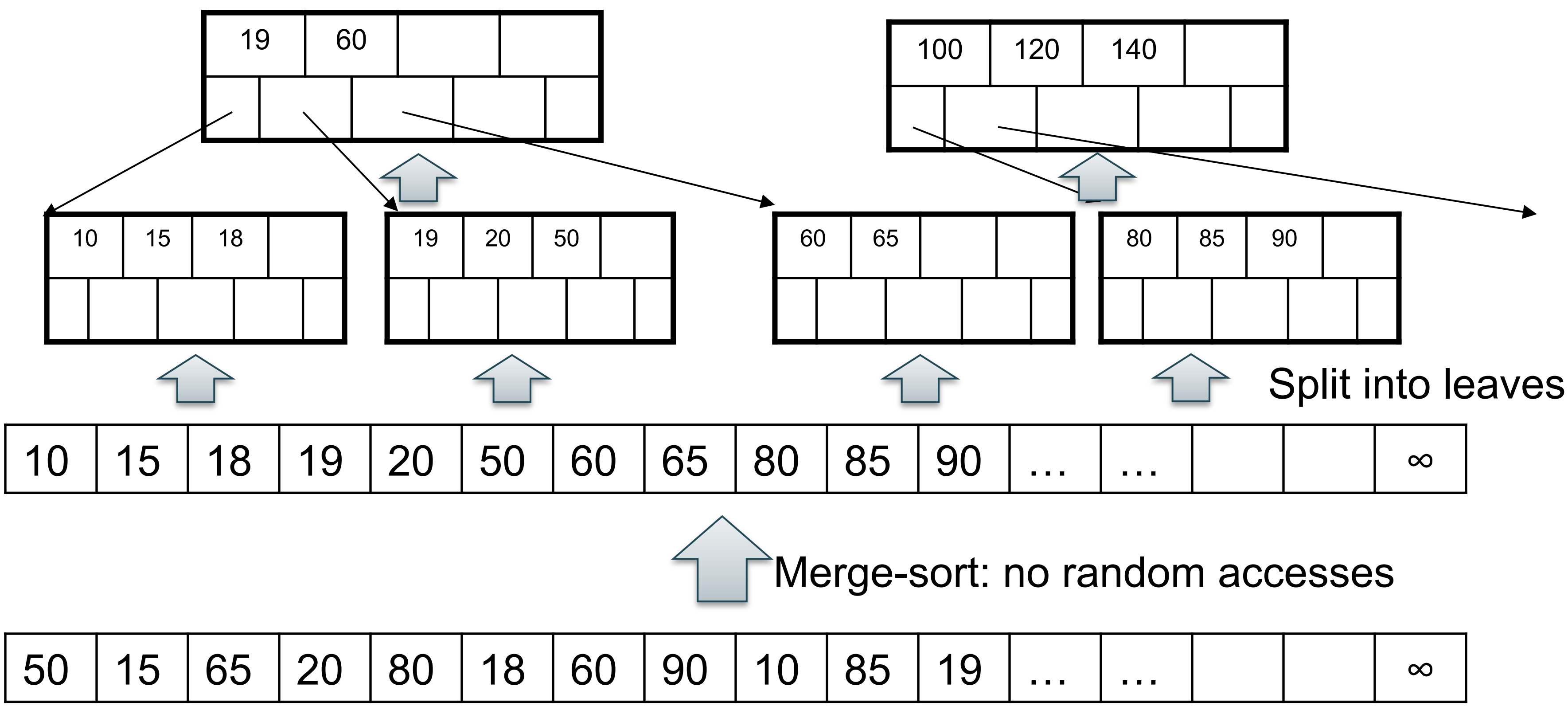
Bulk Index Creation

Sort data first, then build the tree



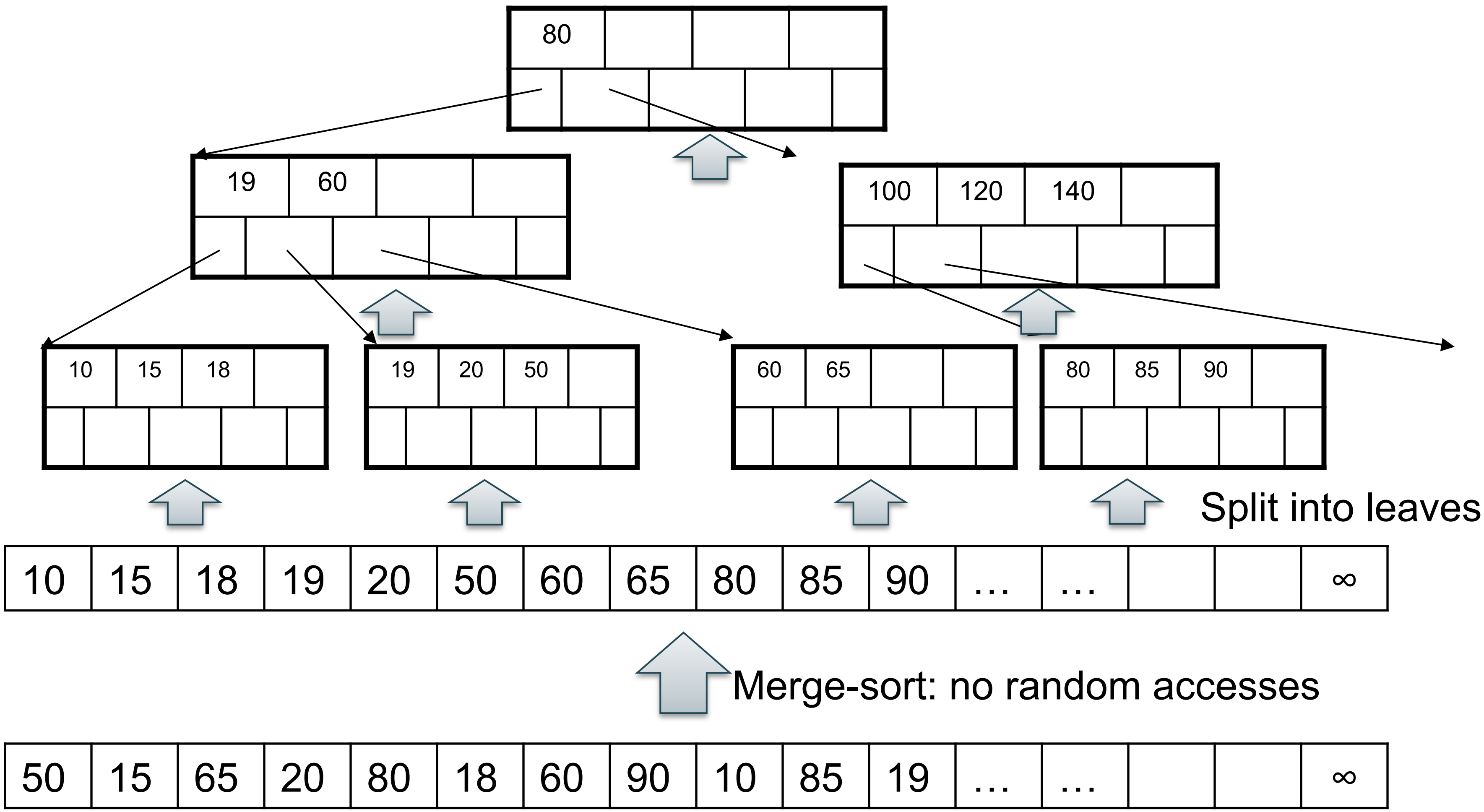
Bulk Index Creation

Sort data first, then build the tree



Bulk Index Creation

Sort data first, then build the tree



Problem 2: Write-intensive workloads:

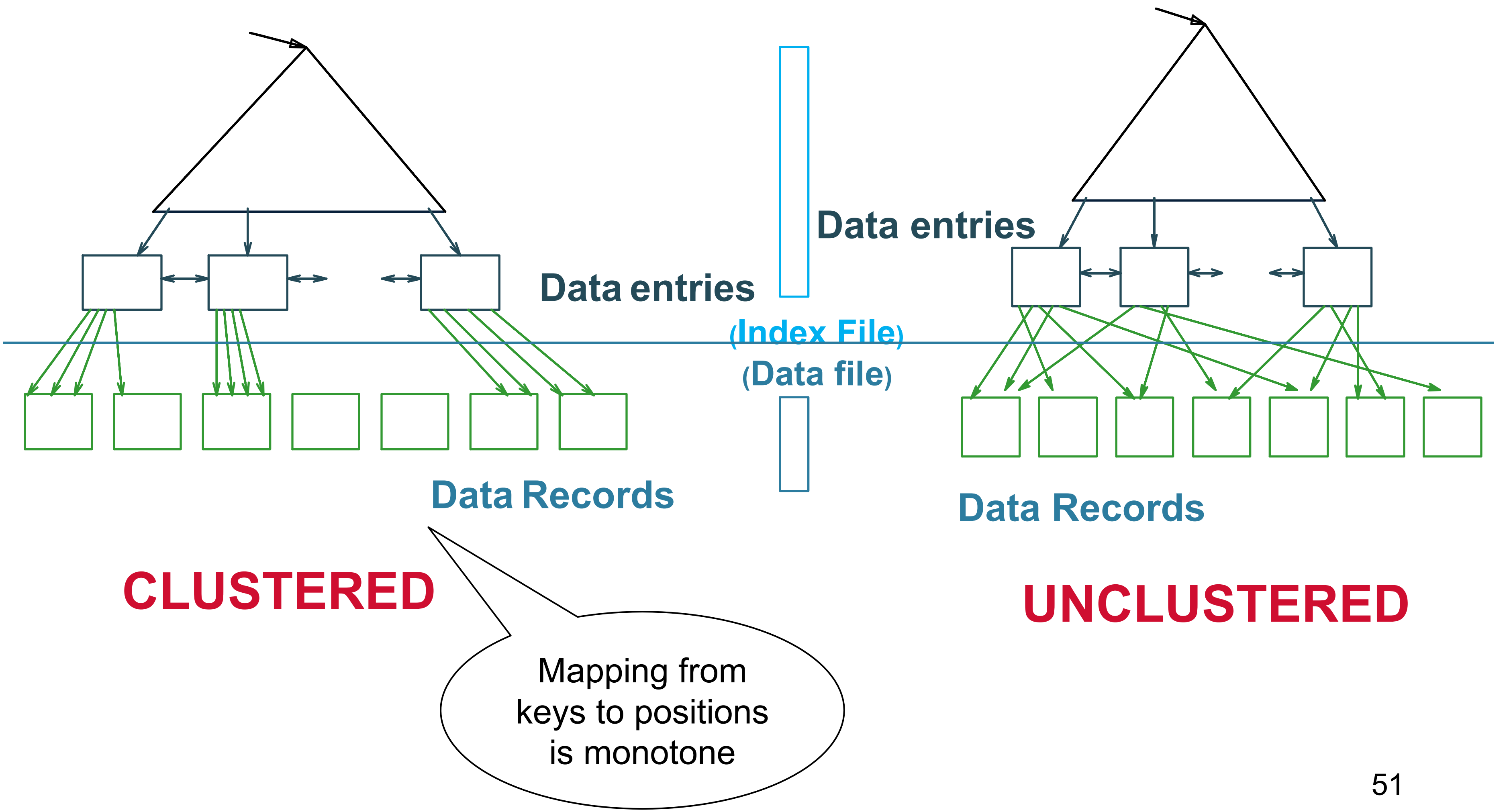
- Company inserts 1000 orders/second
- Adding 1000 records to a log file: fast
- Inserting 1000 in a B+ tree:
  - 1000 random writes (or more)
  - Slow
- LSM tree: buffer new records, then bulk insert.

Reading for Wednesday

Learned indexes

- Idea: if the index is *clustered* then it is a monotone function from keys to positions in the sorted file; replace the B+ tree with a regression model for this mapping

Clustered v.s. Unclustered



Outline

- B+ Trees
- Bloom Filters
- Next time:
  - Learned indexes (paper!), LSM trees
  - Note: Tim Kraska's talk May 24, 9am

Slides on Bloom Filters

Based in part on:

- Broder, Andrei; Mitzenmacher, Michael (2005), "Network Applications of Bloom Filters: A Survey", Internet Mathematics 1 (4): 485–509
- Bloom, Burton H. (1970), "Space/time trade-offs in hash coding with allowable errors", Communications of the ACM 13 (7): 422–42

Problem Setting

- Want a *very small*, and *very fast* dictionary H
  - Insert(k,H), member(k,H)
  - No values, just membership test

Problem Setting

- Want a <u>*very small*</u>, and <u>*very fast*</u> dictionary H
  - Insert(k,H), member(k,H)
  - No values, just membership test
- False positives are OK
  - find(k,H) = true: k may or may not be in H
  - find(k,H) = false: k is not in H

Problem Setting

- Want a *very small*, and *very fast* dictionary H
  - Insert(k,H), member(k,H)
  - No values, just membership test
- False positives are OK
  - find(k,H) = true: k may or may not be in H
  - find(k,H) = false: k is not in H
- Goal: minimize false positive rate, FPR

Bit Map

- Let $S = \{x_1, x_2, \ldots, x_n\}$ be a data set
- Hash function $h : S \rightarrow \{1, 2, \ldots, m\}$
  - Typically, $m=8n$

$$S = \{x_1, x_2, \ldots, x_n\}$$

	1	2										m
H=	0	0	1	0	1	1	0	0	1	1	0	0

$S = \{x_1, x_2, \ldots, x_n\}$

0	0	1	0	1	1	0	0	0	1	0	1

Bit Map = a Set

- Insert(x, H) = set bit h(x) to 1
  - Collisions are possible
- Member(y, H) = check if bit h(y) is 1
  - False positives are possible
- No deletions

$S = \{x_1, x_2, ..., x_n\}$

0	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**0**	**1**

Analysis

- Insert S into H
- Check membership of some y:
  - What is the probability member(y,H)=true?
  - This is the False Positive Rate, FPR

$S = \{x_1, x_2, \ldots, x_n\}$

0	0	1	0	1	1	0	0	0	1	0	1

Analysis

- Insert S into H

- Check membership of some y:
  - What is the probability member(y,H)=true?
  - This is the False Positive Rate, FPR
- Will compute in two steps
  - Will denote j=h(y)
  - FPR = Prob(bit(j)=true)

$S = \{x_1, x_2, \ldots, x_n\}$

0	0	0	0	1	0	0	0	0	0	0	0

Analysis

- Recall $|H| = m$
- Let's insert only x_i into H

- What is the probability that bit j is 0 ?

$S = \{x_1, x_2, \ldots, x_n\}$

0	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Analysis

- Recall $|H| = m$
- Let's insert only x_i into H
- What is the probability that bit j is 0 ?
- Answer: $p = 1 - 1/m$

$S = \{x_1, x_2, \ldots, x_n\}$

0	0	1	0	1	1	0	0	0	1	0	1

Analysis

- Recall $|H| = m$, $S = \{x_1, x_2, ..., x_n\}$
- Let's insert all elements from S in H

- What is the probability that bit j is 0 ?

$S = \{x_1, x_2, ..., x_n\}$

0	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**0**	**1**

Analysis

- Recall $|H| = m$, $S = \{x_1, x_2, \ldots, x_n\}$
- Let's insert all elements from S in H

- What is the probability that bit j is 0 ?

- Answer: $p = (1 - 1/m)^n$

$S = \{x_1, x_2, ..., x_n\}$

0	0	1	0	1	1	0	0	0	1	0	1

Analysis

- Recall $|H| = m$, $S = \{x_1, x_2, ..., x_n\}$
- Let's insert all elements from S in H

- What is the probability that bit j is 0 ?

- Answer: $p = (1 – 1/m)^n$

1/m very small!

$S = \{x_1, x_2, \ldots, x_n\}$

0	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**0**	**1**

Analysis

- Recall $|H| = m$, $S = \{x_1, x_2, \ldots, x_n\}$
- Let's insert all elements from S in H

- What is the probability that bit j is 0 ?

- Answer: $p = (1 - 1/m)^n \approx e^{-n/m}$

1/m very small!

$S = \{x_1, x_2, \ldots, x_n\}$

0	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**0**	**1**

False Positive Rate

- FPR = Prob(member(y,H)=true) is:

$$1 - (1 - 1/m)^n \approx 1 - e^{-n/m}$$

$S = \{x_1, x_2, \ldots, x_n\}$

0	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**0**	**1**

Analysis: Example

- Example: $m = 8n$, then

  FPR $\approx 1 - e^{-n/m} = 1\text{-}e^{-1/8} \approx 0.11$

- 11% false positive rate
- Bloom filters improve that (next)

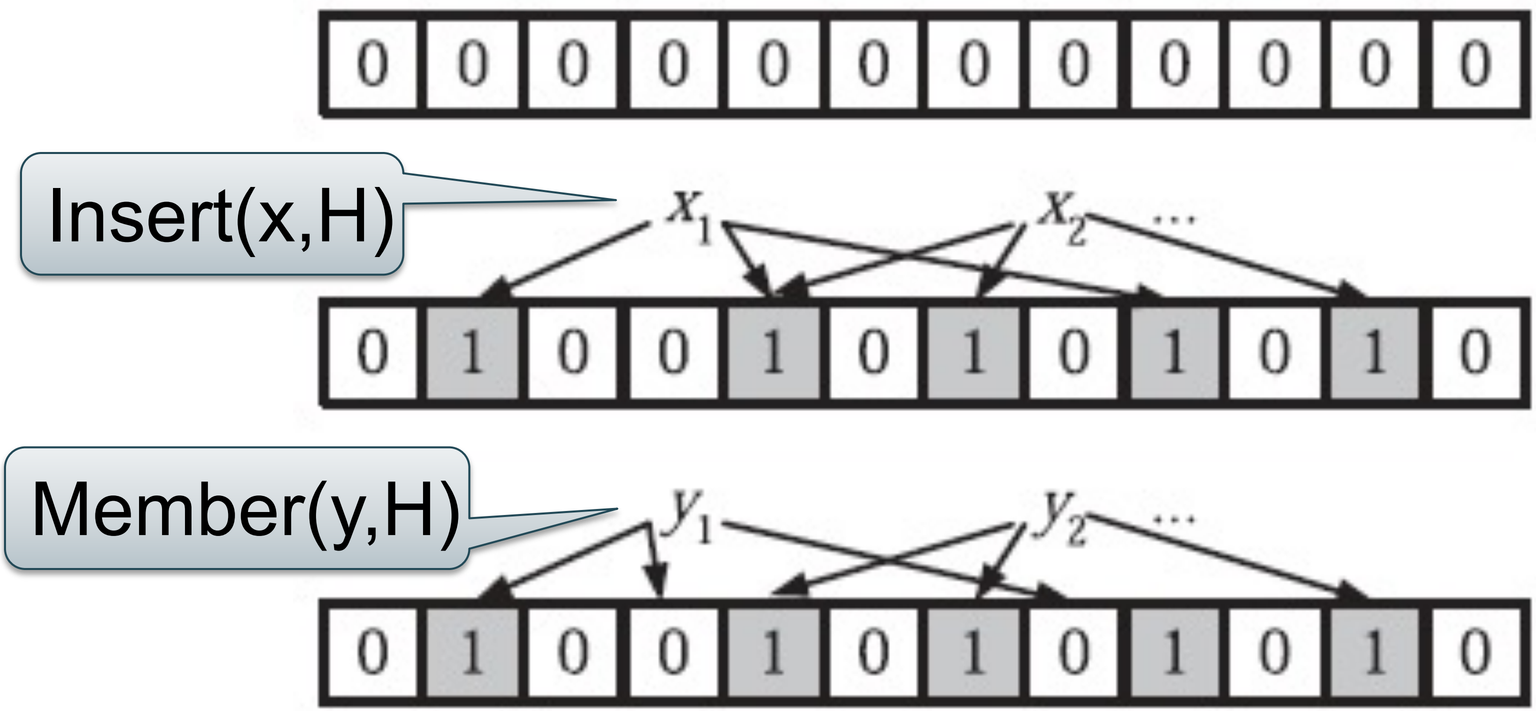
Bloom Filters

- Introduced by Burton Bloom in 1970
- Improve the false positive ratio
- Idea: use k independent hash functions

Bloom Filter = Dictionary

- Insert(x, H):
  - set bits $h_1(x), \ldots, h_k(x)$ to 1
- Member(y, H):
  - check if all bits $h_1(y), \ldots, h_k(y)$ are 1

Example Bloom Filter k=3



y_1 = is not in H (why ?); y_2 may be in H (why ?)

Choosing k

Two competing forces:

- If k = large
  - Test more bits for member(y,H) ➔ low FPR
  - More bits in H are 1 ➔ high FPR
- If k = small
  - More bits in H are 0 ➔ lower FPR
  - Test fewer bits for member(y,H) ➔ high FPR

$S = \{x_1, x_2, \ldots, x_n\}$

0	0	1	0	1	1	0	0	0	1	0	1

Analysis

- Recall $|H| = m$, #hash functions = k
- Let's insert only x_i into H
- What is the probability that bit j is 0 ?

$S = \{x_1, x_2, \ldots, x_n\}$

0	0	1	0	1	1	0	0	0	1	0	1

Analysis

- Recall $|H| = m$, #hash functions = k
- Let's insert only x_i into H
- What is the probability that bit j is 0 ?
- Answer: $p = (1 - 1/m)^k$

$S = \{x_1, x_2, \ldots, x_n\}$

0	0	1	0	1	1	0	0	0	1	0	1

Analysis

- Recall $|H| = m$, #hash functions = k
- Let's insert only x_i into H

- What is the probability that bit j is 0 ?

- Answer: $p = (1 - 1/m)^k \approx e^{-k/m}$

$S = \{x_1, x_2, \ldots, x_n\}$

0	0	1	0	1	1	0	0	0	1	0	1

Analysis

- Recall $|H| = m$, #hash functions = k
- Let's insert all elements from S in H
- What is the probability that bit j is 0 ?

$S = \{x_1, x_2, \ldots, x_n\}$

0	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**0**	**1**

Analysis

- Recall $|H| = m$, #hash functions = k
- Let's insert all elements from S in H

- What is the probability that bit j is 0 ?

- Answer:
  $\text{Prob}(\text{bit}(j)=0) = (1 - 1/m)^{kn} \approx e^{-kn/m}$

$$\mathrm{Prob}(\mathrm{bit}(j)=0) = (1 - 1/m)^{kn} \approx e^{-kn/m}$$

False Positive Rate

- What is the probability that member(y,H)=true?

$\mathrm{Prob(bit(j)=0)} = (1 - 1/m)^{kn} \approx e^{-kn/m}$

False Positive Rate

- What is the probability that
  member(y,H)=true?

- Answer: it is the probability that all k bits $h_1(y), \ldots, h_k(y)$ are 1, which is:

$$\boxed{f = (1-p)^k \approx (1 - e^{-kn/m})^k}$$

$$\text{FPR} = (1\text{-}p)^k \approx (1 - e^{-kn/m})^k$$

Optimizing k

- m, n are fixed
- We choose k to minimize FPR:

$$\boxed{k = \ln 2 \times m/n}$$

Proof:

$ln(\text{FPR}) = k \cdot \ln\left(1 - e^{-\frac{kn}{m}}\right) = \frac{m}{n} \cdot \frac{kn}{m} \ln\left(1 - e^{-\frac{kn}{m}}\right) = -\frac{m}{n} \ln x \cdot \ln(1-x)$, where $x = e^{-\frac{kn}{m}}$.

We need to maximize the function $\text{g}(\text{x}) = \ln x \cdot \ln(1-x)$

Notice that $f(x) \stackrel{\text{def}}{=} \ln\ln x$ is concave, hence:

$\ln\big(g(x)\big) = \ln(\ln x \cdot \ln(1-x)) = f(x) + f(1-x) \leq 2 \cdot f\left(\frac{x+(1-x)}{2}\right) = 2 \cdot \text{f}\left(\frac{1}{2}\right)$,

Thus, $g(x)$ is maximized when $x = 1 - x$, hence $x = \frac{1}{2}$

$$FPR = (1-p)^k \approx (1 - e^{-kn/m})^k$$

Bloom Filter Summary

m, n are fixed → choose $k = \ln 2 \times m / n$

$FPR = (1\text{-}p)^k \approx (1 - e^{-kn/m})^k$

Bloom Filter Summary

m, n are fixed → choose $k = \ln 2 \times m/n$

Probability that some bit j is 1 $p \approx e^{-kn/m} = ½$

$$\boxed{FPR = (1-p)^k \approx (1 - e^{-kn/m})^k}$$

Bloom Filter Summary

m, n are fixed → choose $k = \ln 2 \times m / n$

Probability that some bit j is 1 $\boxed{p \approx e^{-kn/m} = ½}$

Expectation: $\boxed{m/2 \text{ bits } 1,\ m/2 \text{ bits } 0}$

$$FPR = (1-p)^k \approx (1 - e^{-kn/m})^k$$

Bloom Filter Summary

m, n are fixed → choose $k = \ln 2 \times m / n$

Probability that some bit j is 1 $p \approx e^{-kn/m} = ½$

Expectation: m/2 bits 1, m/2 bits 0

$$FPR = (1-p)^k \approx (½)^k = (½)^{(\ln 2)m/n} \approx (0.6185)^{m/n}$$

$\boxed{FPR = (1-p)^k \approx (1 - e^{-kn/m})^k}$

Bloom Filter Summary

m, n are fixed → choose $k = \ln 2 \times m/n$

Probability that some bit j is 1 $\boxed{p \approx e^{-kn/m} = ½}$

Expectation: $\boxed{\text{m/2 bits 1, m/2 bits 0}}$

FPR= $\boxed{(1-p)^k \approx (½)^k = (½)^{(\ln 2)m/n} \approx (0.6185)^{m/n}}$

Another way: $1-p \approx e^{-p} = e^{-\ln 2}$ $\boxed{(1-p)^k \approx e^{-\frac{m}{n}(\ln^2 2)}}$

Bloom Filter Summary

- In practice one sets $m = cn$, for some constant c
  - Thus, we use c bits for each element in S
  - Then $f \approx (0.6185)^c$ = constant

- Example: $m = 8n$, then

  $k = 8(\ln 2) = 5.545$ (use 6 hash functions)

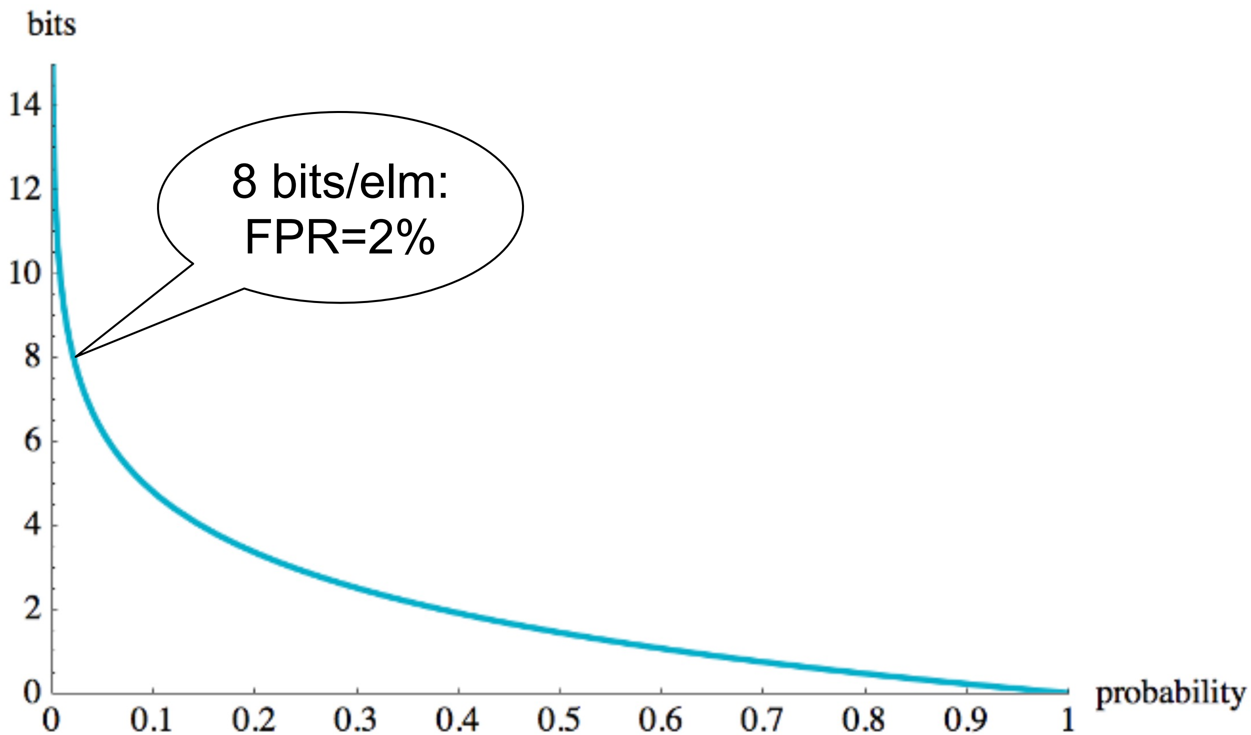
  $f \approx (0.6185)^{m/n} = (0.6185)^8 \approx 0.02$ (2% false positives)

  Compare to a hash table: $f \approx 1 - e^{-n/m} = 1-e^{-1/8} \approx 0.11$

FPR v.s. #bits/element

From https://corte.si/posts/code/bloom-filter-rules-of-thumb/

c=



Bits per element vs. false positive probability