CS 350 Algorithms and Complexity

Winter 2019

Lecture 12: Space & Time Tradeoffs.

Part 2: Hashing & B-Trees

Andrew P. Black

Department of Computer Science

Portland State University

Space-for-time tradeoffs

- Two varieties of space-for-time algorithms:
- input enhancement: preprocess the input (all or part) to store some info to be used later in solving the problem
  - counting sorts
  - string searching algorithms
- prestructuring — preprocess the input to make accessing its elements easier
  - hashing
  - indexing schemes (*e.g.*, B-trees)

Hashing

- A very efficient method for implementing a dictionary, *i.e.*, a mapping from *keys* to *values* with the operations:
  - find(k) — also called at(k), lookup(k), [k]
  - insert(k, v) — also called at(k) put(v), put (k, v), [k] := v
  - delete(k)
- Based on representation-change and space-for-time tradeoff ideas
- Important applications:
  - symbol tables
  - sets (values are not relevant)
  - databases (extendible hashing)

Hash tables and hash functions

- The idea of hashing is to map n keys into an array of size m, called the hash table, by using a predefined function, called the hash function,

  h: key → index in the hash table

  - Example: student records, key = StudentID.
  - Hash function: $h(K) = K \bmod m$ where m is some integer (typically, prime)
- Generally, a hash function should:
  - be easy to compute
  - distribute keys about evenly throughout the hash table
  - use all the data in the key

Question:

- If $m = 1000$,
  $h(k) = \text{k mod } m$,

  where is record with StudentId = 990159265 stored?

  - Numeric answer:

Collisions

- If $h(k_1) = h(k_2)$, there is a *collision*
  - Good hash functions result in fewer collisions than bad ones, but:
  - some collisions should be expected (birthday paradox)
- Two principal hashing schemes handle collisions differently:
  - Open hashing
    – each cell is a header of “chain” — a list of all keys hashed to it
  - Closed hashing
    - *one* key per cell
    - in case of collision, finds another cell *in the hash table itself* by
      - linear probing: use next free bucket, or
      - double hashing: use second hash function to compute increment

What Levitin didn’t tell you

- Hash functions usually designed in two parts:

1. the hash $Object \to [0..2^{32})$
2. the address calculation

$$[0..2^{32}) \to [0..hashTableSize)$$

A Real Hash Function:

```
function robertJenkins32bit5shiftHash(n) {
    var answer = n & 0xFFFFFFFF;
    answer = 0x479AB41D + answer + (answer << 8);
    answer = answer & 0xFFFFFFFF;
    answer = (0xE4AA10CE ⊻ answer) ⊻ (answer >> 5);
    answer = 0x09942F0A6 + answer - (answer << 14);
    answer = (0x5AEDD67D ⊻ answer) ⊻ (answer >> 3);
    answer = 0x17BEA992 + answer + (answer << 7);
    return (answer & 0xFFFFFFFF);
}
```

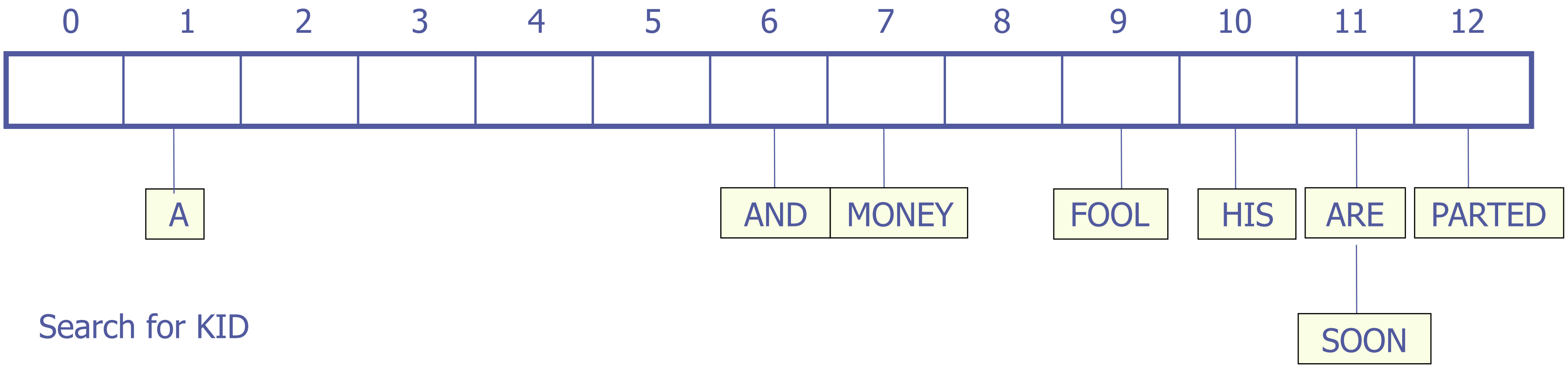
Returns 32 “scrambled” bits

Open hashing (Separate chaining)

- Keys are stored in linked lists *outside* a hash table whose elements serve as the lists' headers.
- Example: A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED

h(*k*) = sum of key's letters' positions in the alphabet **mod** 13

key	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
h(key)	1	9	6	10	7	11	11	12



Search for KID

h(KID) = 11

Open hashing (cont.)

- The ratio $\alpha = n/m$ is called the *load factor*
- If hash function distributes keys uniformly, average length of linked list will be α
- Average number of key comparisons in successful, S, and unsuccessful searches, U:

$$\boxed{S \approx 1+\alpha/2, \qquad U = \alpha}$$

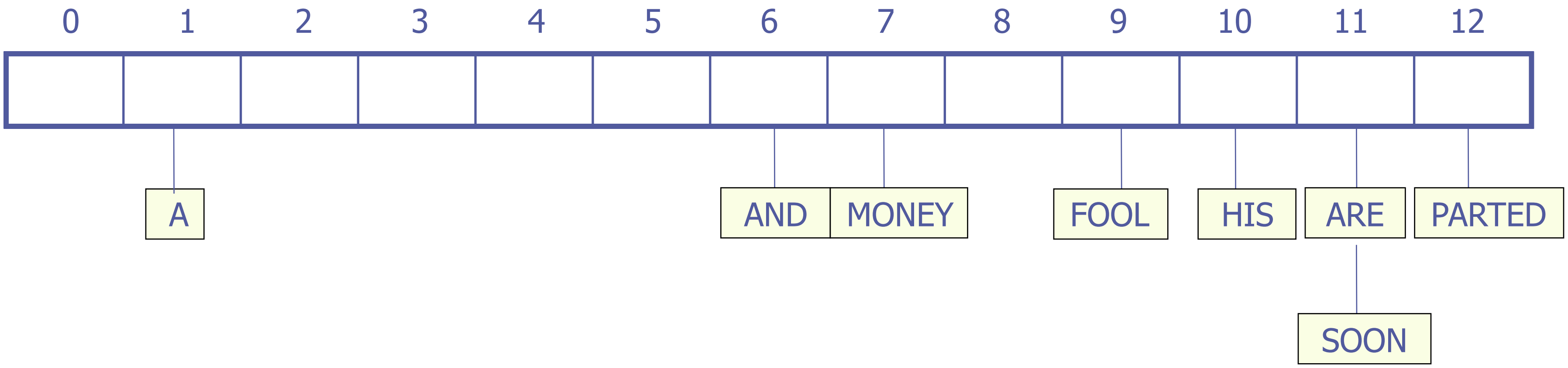
- α is typically kept small (ideally, about 1)
- Open hashing still works if $n > m$
  - but it's *much* less efficient

Efficiencies (open hashing)

	Successful	Unsuccessful
α	$S \approx 1+\alpha/2$	$U = \alpha$
0.33	1.17	0.33
0.50	1.25	0.5
0.75	1.38	0.75
0.90	1.45	0.9
0.95	1.48	0.95
1.00	1.5	1
1.50	1.75	1.5
2.00	2	2

Mnemonic

- This method is called *open hashing* because the hash table is open:
  - the stored keys and values are not *in* the table, but “hang” *out* of it ...
  - on separate chains (hence the alternative name)



0 1 2 3 4 5 6 7 8 9 10 11 12

A

AND

MONEY

FOOL

HIS

ARE

PARTED

SOON

Question:

- Suppose that you have 1000 records in memory, and you index them in a hash table with $\alpha = 1.$ How many words of *additional* memory will be needed if you use an open hash table with separate chaining?

  Each pointer occupies 1 word. *Additional* means beyond the table and the records.)

- Numeric answer:

Closed hashing (Open addressing)

- All keys are stored *inside* the hash table.

Key	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
h(K)	1	9	6	10	7	11	11	12

0	1	2	3	4	5	6	7	8	9	10	11	12

Closed hashing (cont.)

- Does not work if $n > m$
- Avoids pointer chains
- Deletions are not straightforward
- Number of probes to find/insert/delete a key depends on load factor $\alpha = n/m$ **and** on collision resolution strategy.
- For linear probing:

$\boxed{S \approx \frac{1}{2}(1 + 1/(1 - \alpha))}$ and $\boxed{U \approx \frac{1}{2}(1 + 1/(1 - \alpha)^2)}$

Efficiencies (closed hashing)

- As the table fills (α approaches 1), number of probes in linear probing increases dramatically:

	Successful	Unsuccessful
α	$S \approx \tfrac{1}{2}(1+1/(1-\alpha))$	$U \approx \tfrac{1}{2}(1+1/(1-\alpha)^2)$
0.33	1.25	1.625
0.50	1.5	2.5
0.75	2.5	8.5
0.90	5.5	50.5
0.95	10.5	200.5
0.99	50.5	5000.5

Question:

- Suppose that you have 1000 records in memory, and you index them in a hash table with $\alpha = 1/2$. How many words of *additional* memory will be needed if you use a *closed* hash table?

  Each pointer occupies 1 word. *Additional* means beyond the table and the records.

- Numeric answer:

Question:

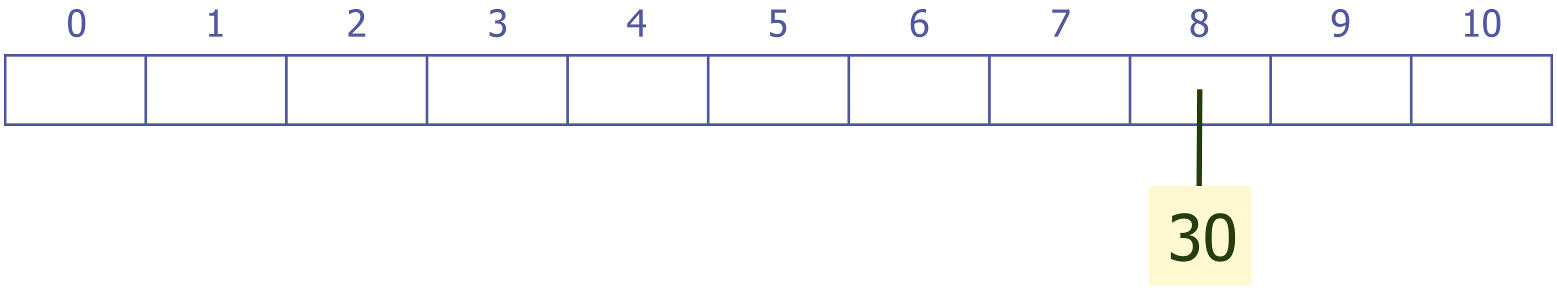
- For the input:

  30 20 56 75 31 19

30 mod 11 = 8

and hash function $h(k) = k \bmod 11$

1. Construct an open hash table of size 11



0 1 2 3 4 5 6 7 8 9 10

30

2. What is the largest number of key comparisons in a successful search in this table?

Question:

- For the input:

30 20 56 75 31 19

30 mod 11 = 8

and hash function $h(k) = k \bmod 11$

1. Construct a closed hash table of size 11

0	1	2	3	4	5	6	7	8	9	10
								30		

2. What is the largest number of key comparisons in a successful search in this table?

Question:

- Suppose that you have 1000 records in memory, and you have 3000 words of memory to use for indexing them. What’s the best way to use that memory if most searches are expected to be successful?

  (Assume that each pointer occupies 1 word)

  A. Open Hash table with separate chaining
  B. Closed Hash table with open addressing
  C. Binary search tree

Problem

- Fill in the following table with the average-case efficiencies for 5 implementations of Dictionary

	array	sorted array	Binary search tree	Open Hash w/separate chaining	Closed Hash w/linear probing
search					
insertion					
deletion					

2–3 trees

- 2–3 trees are perfectly-balanced search trees where each node has either:
  - 2 children (and 1 key), or
  - 3 children (and 2 keys).
- Details:
  - Levitin pp. 223–5.
  - Lyn Turbak on 2–3 Trees
  - Visualization of 2–3 trees

B+ Trees (slides based on those of Miriam Sy)

- The B+ Tree index structure is the most widely-used of several index structures that maintain their efficiency despite insertion and deletion of data.
- Balanced tree: every path from the root of the tree to a leaf is of the same length.
- All data are in leaves.
  - Hence: B+ tree

B Trees & B+ Trees

B Trees:

- Multi-way trees
- Dynamic growth
- Data stored with keys throughout the tree pages

B+ Trees:

- Contains features from B Trees
- Dynamic growth
- Separates index and data pages
- Internal nodes contain just keys
- All data are at the leaves
- leaves can be linked sequentially

B Tree *Order*

- The “order” of a B tree is the:

A. way the purchasing department buys it

B. *maximum* number of children any node can have

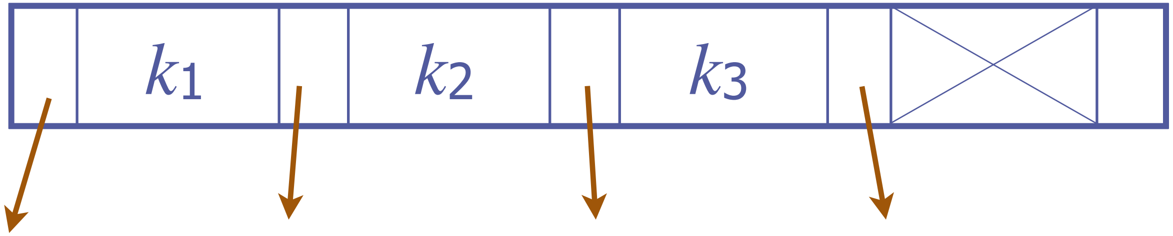
C. *minimum* number of children any node can have

D. number of keys in each index node

E. height of the tree

Fill Factor

- B+ Trees use a “fill factor” to control growth: fill factor = fraction of keys filled



- What is the minimum fill factor for a B+ Tree?

  A. 33.3%
  B. 50%
  C. 75%
  D. None of the above

B+ Tree Statistics

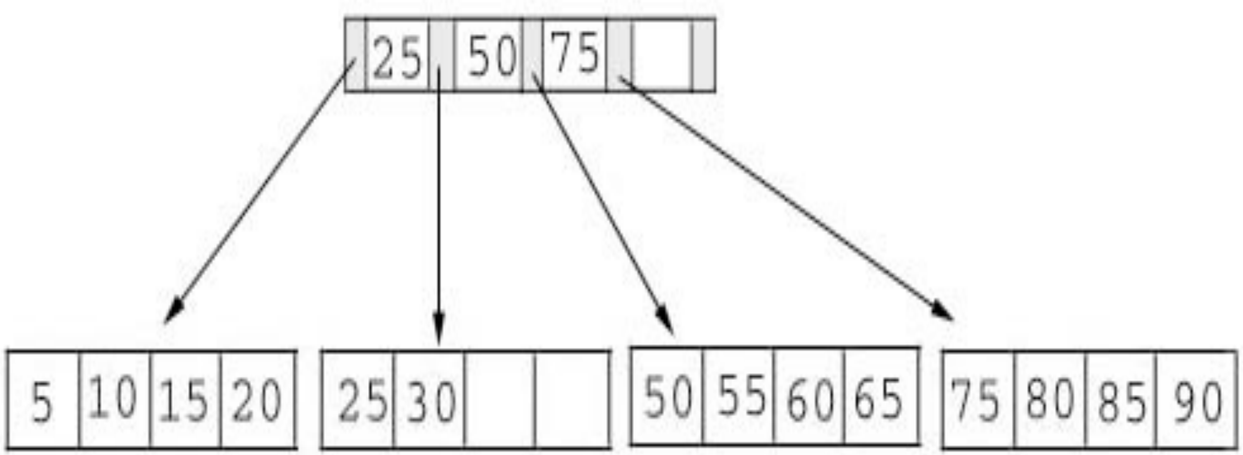
* Except the root

- Each node* has between $\lceil m/2 \rceil$ and m children
- 50% fill factor is the minimum for a B+ Tree.
- For tree of order 5, these guidelines must be met:

m, max pointers/page	5
$n = m-1$, max keys/page	4
$\lceil m/2 \rceil$, min pointers/page	3
min keys/page	2
minimum fill factor	50%

B+ Trees

- B+ Tree with $m=5$



- *Note:* B+ Tree in commercial database might have $m=2048$

Inserting Records

- The key determines a record’s placement in a B+ Tree
- The leaf pages are maintained in sequential order. A ***doubly-linked list*** (usually not shown) connects each leaf page with its sibling page(s).

Insertion Algorithm

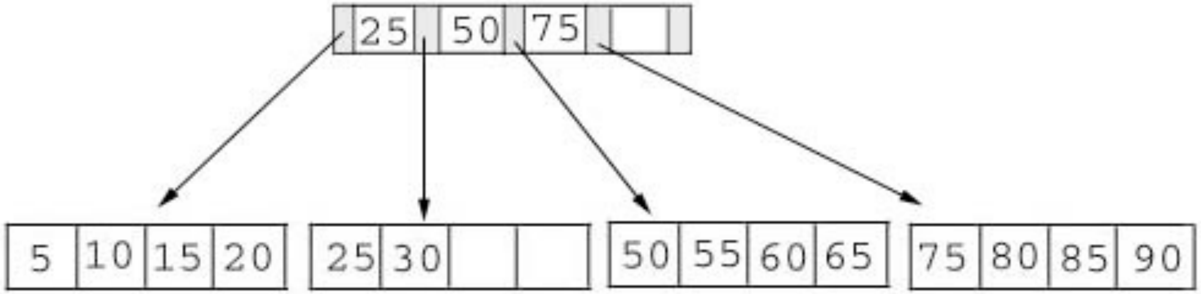
Leaf Page Full?	Index Page Full?	Action
No	No	Place the record in sorted position in the appropriate leaf page
Yes	No	1. Split the leaf page 2. Copy middle key into the index page in sorted order. 3. Left leaf page contains records with keys < the middle key. 4. Right leaf page contains records with keys ≥ than the middle key.

Insertion Algorithm, cont…

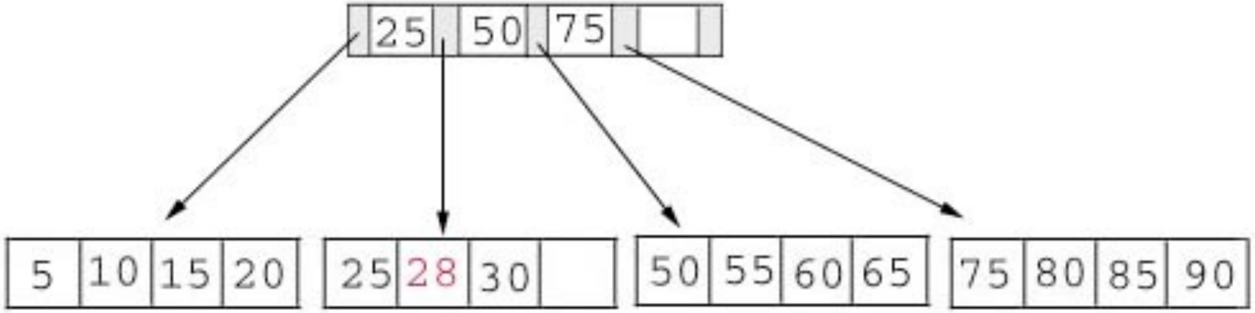
Leaf Page Full?	Index Page Full?	Action
Yes	Yes	1. Split the leaf page 2. Records with keys < middle key go to the left leaf page 3. Records with keys ≥ middle key go to the right leaf page. 4. Split the index page 5. Keys < middle key go to the left index page. 6. Keys > middle key go to the right index page. 7. The middle key goes to the next (higher) level. If the next level index page is full, continue splitting the index pages.

1st Case: leaf page & index page not full

- Original Tree:

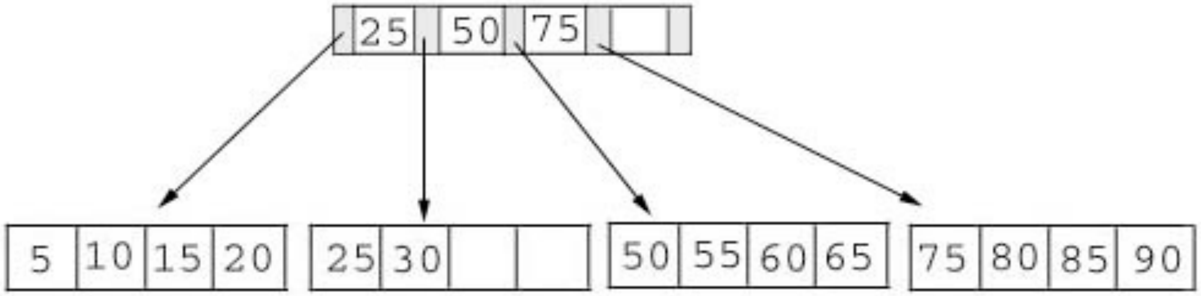


- Insert 28:



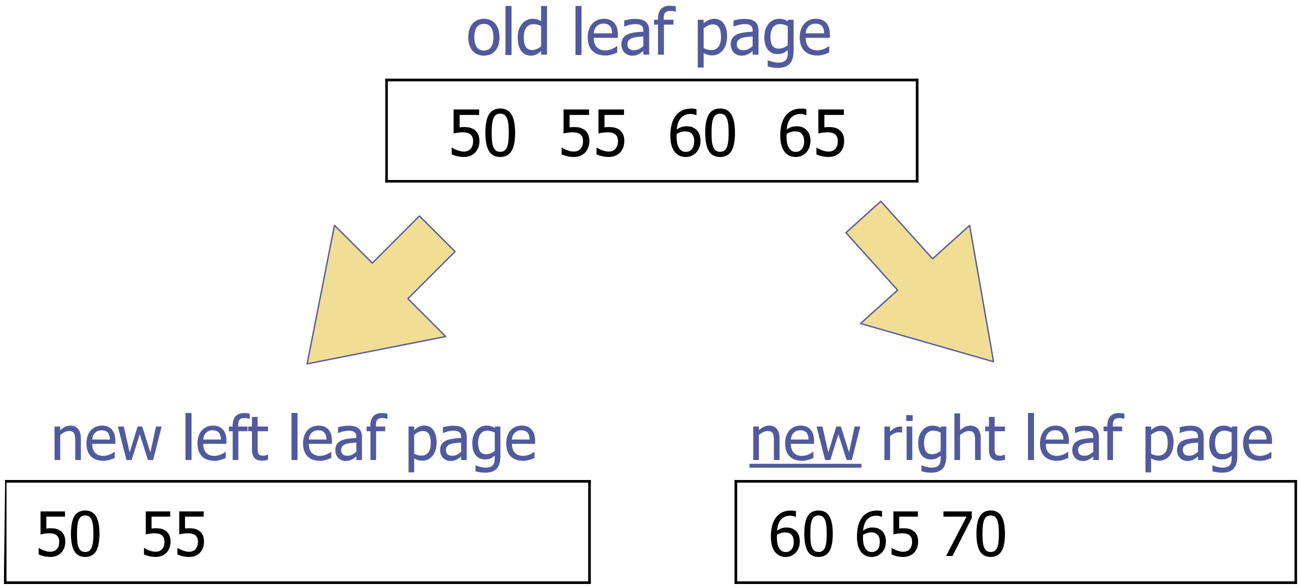
2nd Case: leaf page Full, index page Not Full

- Insert a record with a key value of 70.



2nd Case: leaf page Full, index page Not Full

- Insert a record with a key value of 70.
- This record should go in the leaf page with 50, 55, 60, and 65.
- This page is full, so we need to ***split*** it as follows:



old leaf page

50 55 60 65

new left leaf page

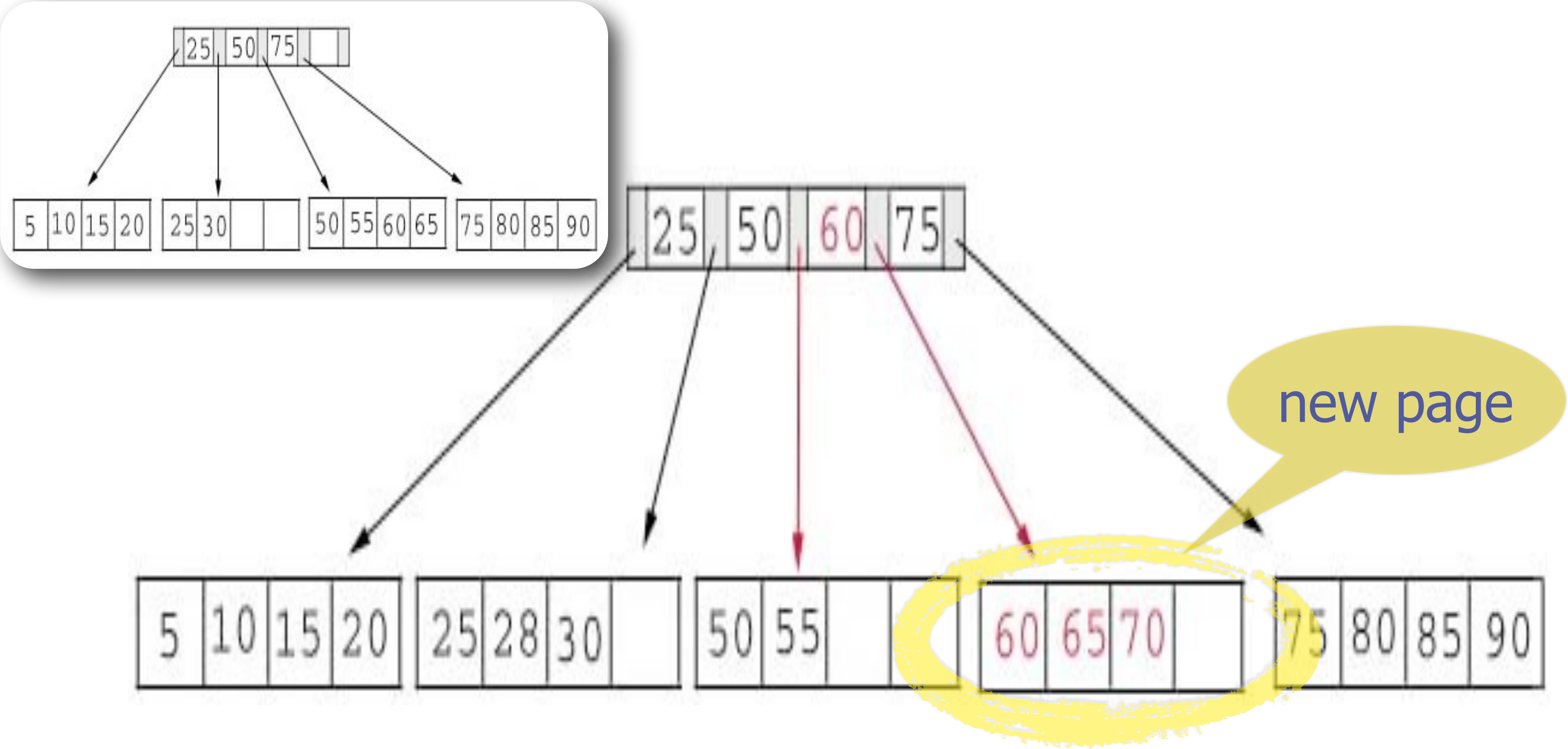
50 55

new right leaf page

60 65 70

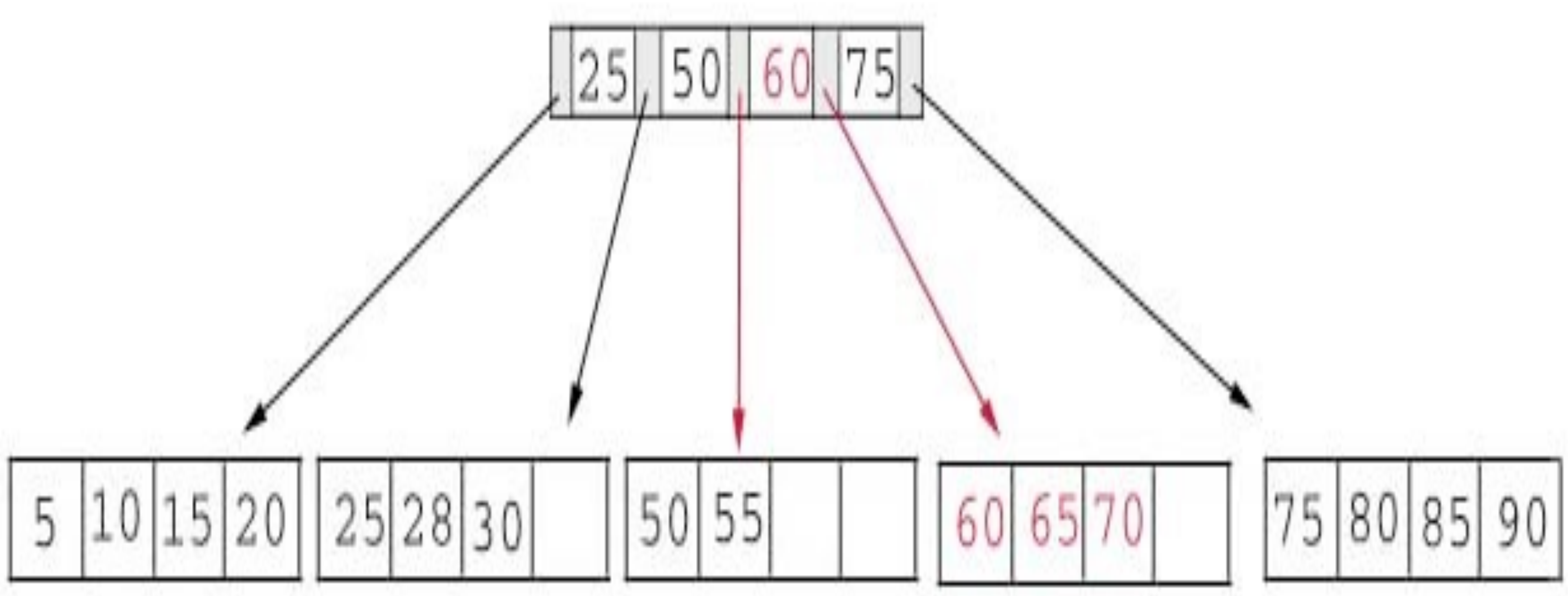
Now add the new leaf to the index:

- The middle key (60) is copied into the index page between 50 and 75.



3rd Case: leaf page & index page both full

- To prior example, we need to add 95:



3rd Case: leaf page & index page both full

- We need to add 95 to leaf with

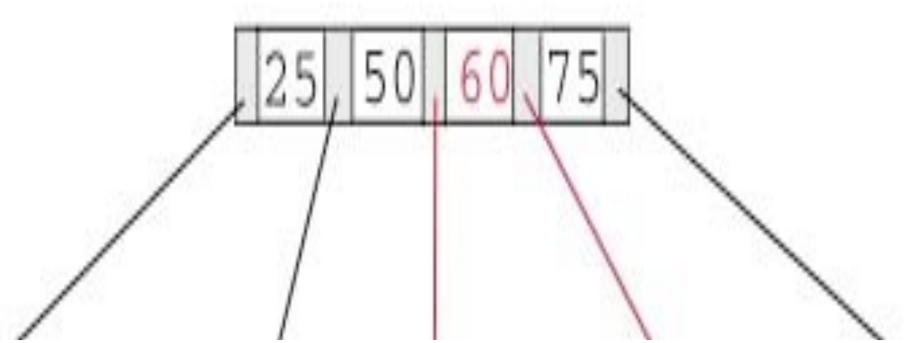
75 80 85 90

which is full. So we need to split it into two pages:

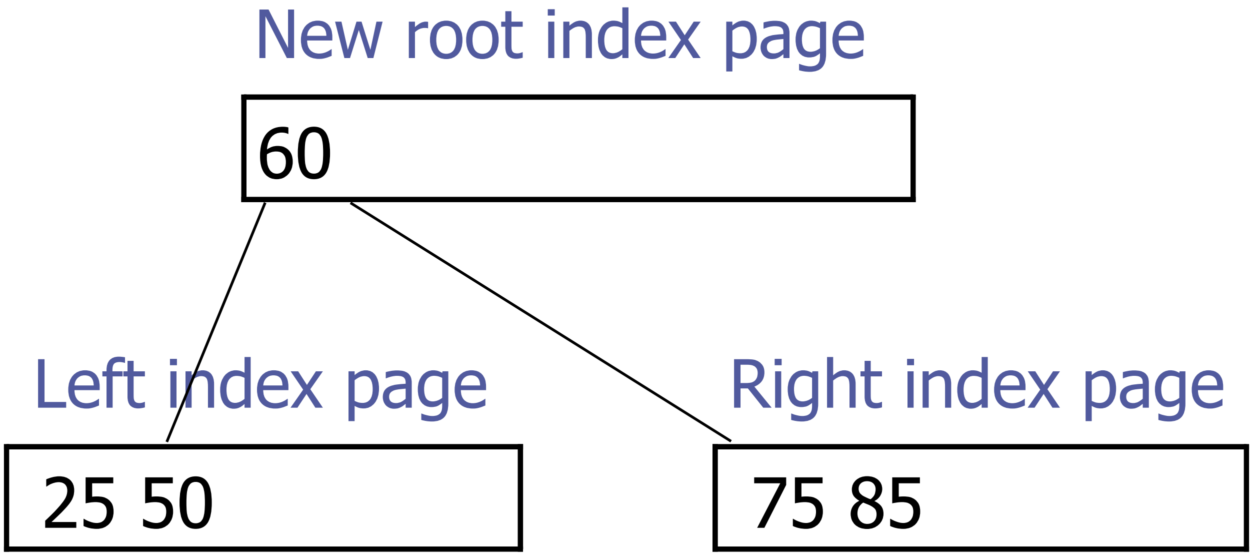
Left leaf page	Right leaf page
75 80	85 90 95

- The middle key, 85, rises to the index page.

3rd Case: leaf page & index page both full



- The index page is *also* full so we must split it:



New root index page

60

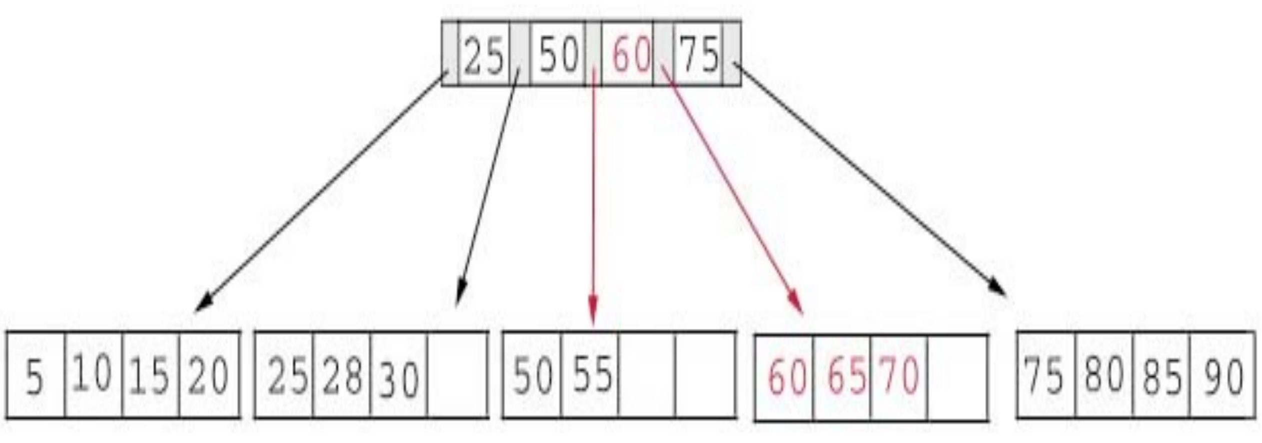
Left index page

25 50

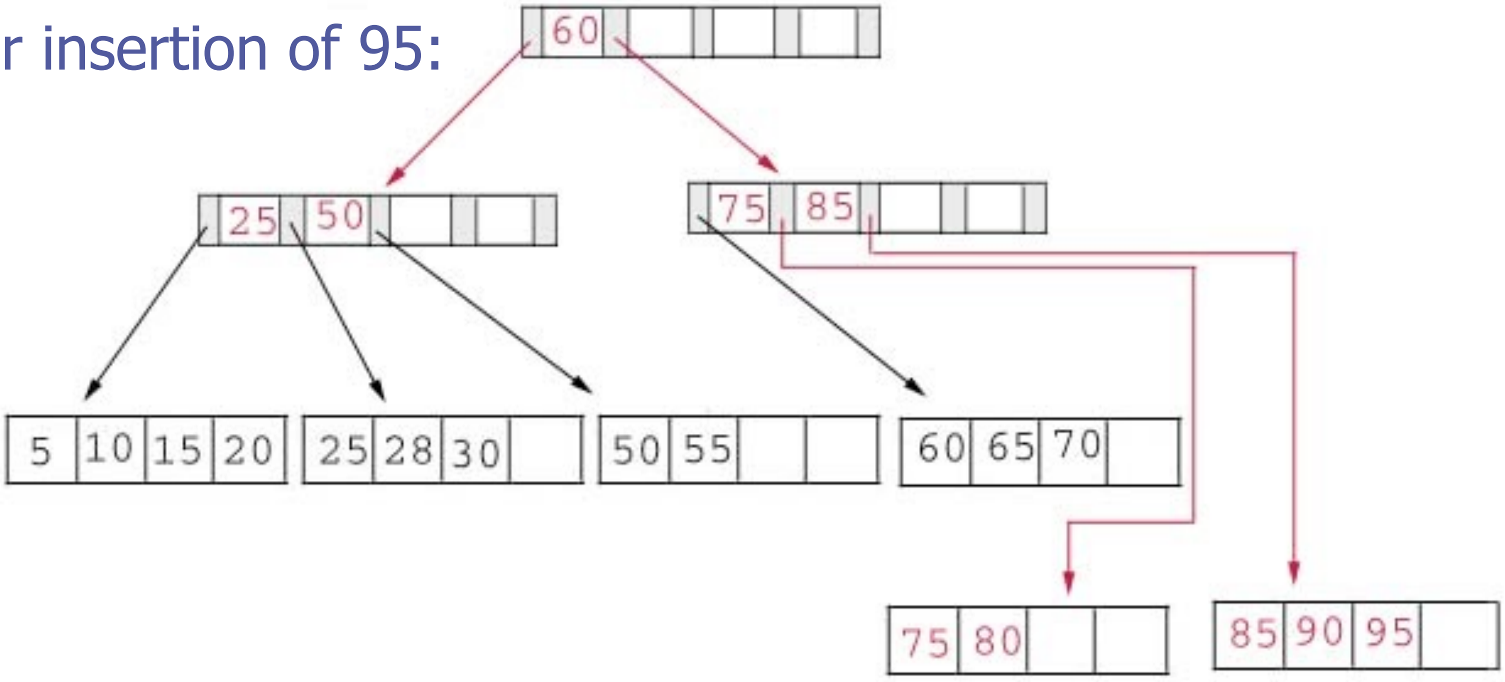
Right index page

75 85

Before…



and after insertion of 95:

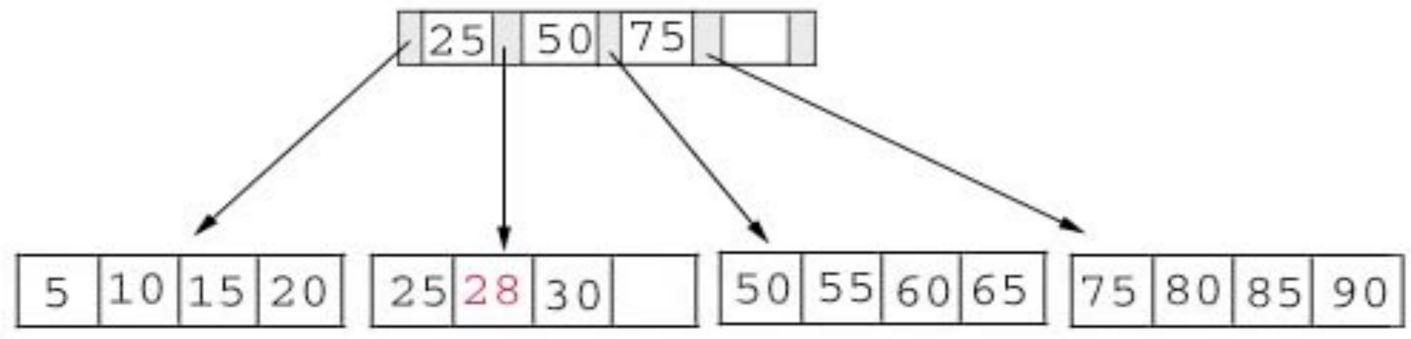


Rotation

- B+ Trees can incorporate *rotation* to reduce the number of page splits.
- A rotation occurs when a leaf page is full, but one of its siblings pages is not full. Rather than splitting the leaf page, we move a record to its sibling, adjusting indices as necessary.
- The left sibling is usually checked first, then the right sibling.

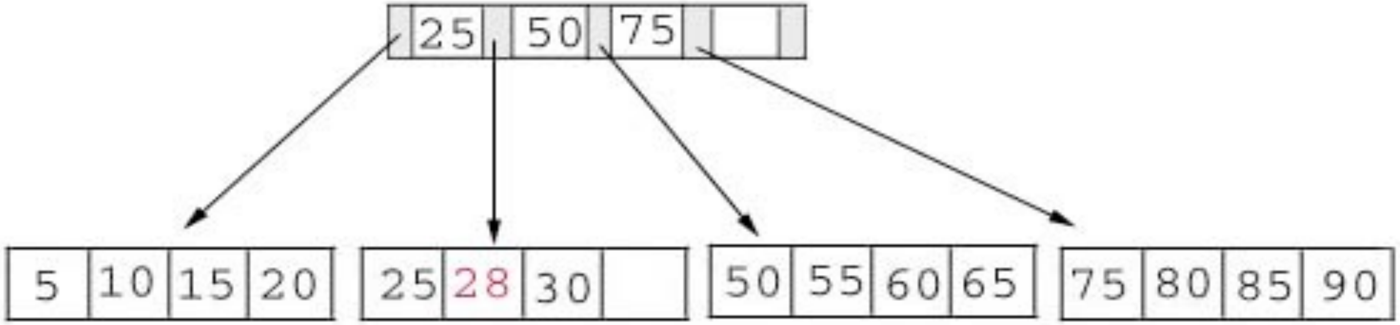
Rotation

- For example, consider again the B+ Tree before the addition of 70. This record belongs in the leaf node containing 50 55 60 65. Notice how this node is full but its left sibling is not.

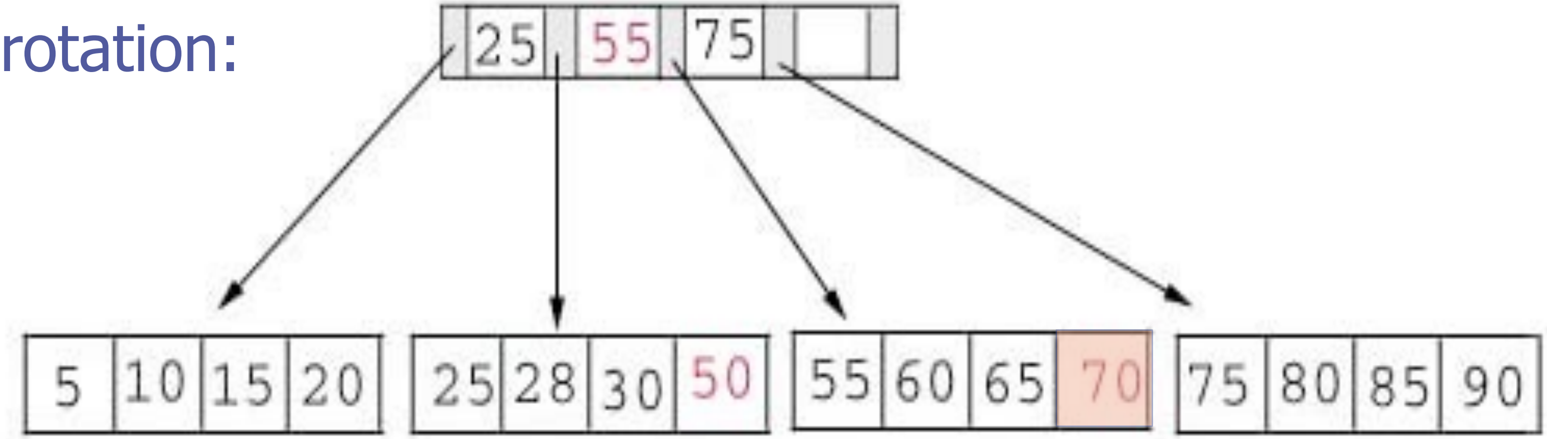


- Using rotation, we move the record with the lowest key to its sibling. We must also modify the index page:

Before rotation:



After rotation:



Deletion Algorithm

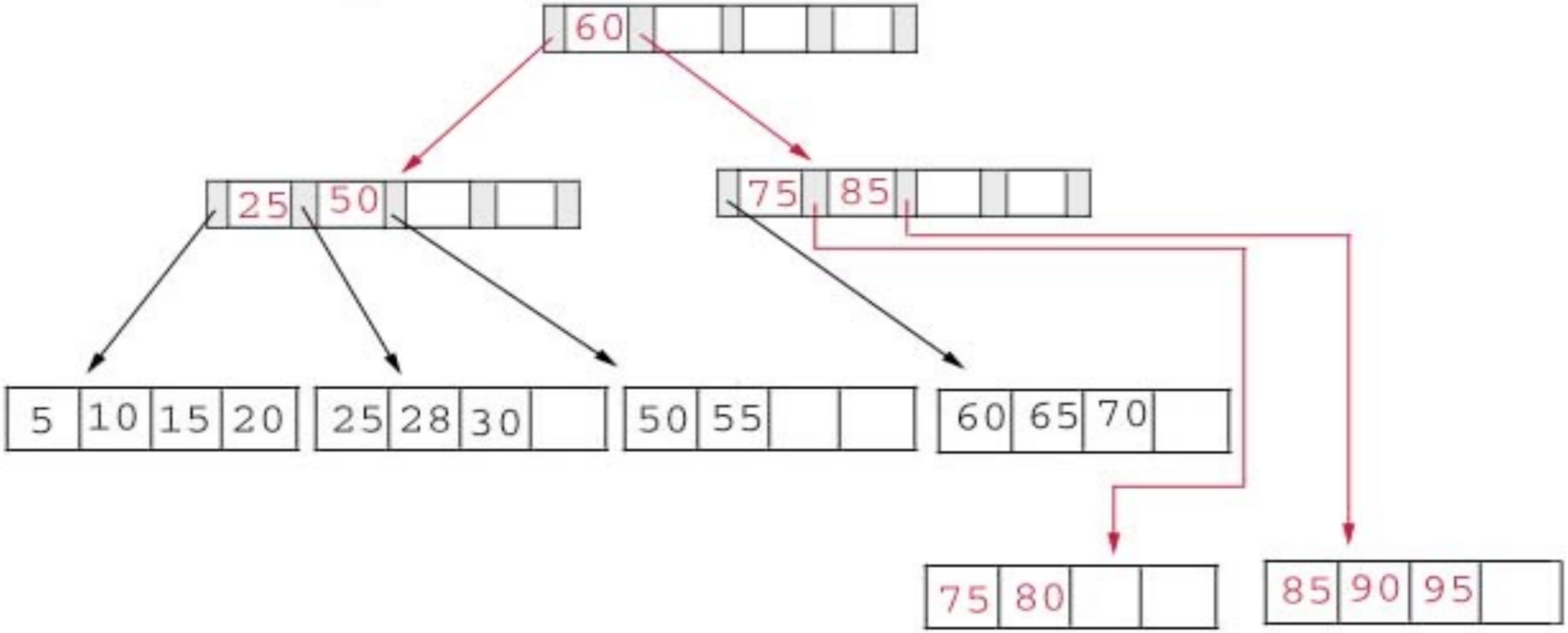
- Like the insertion algorithm, there are three cases to consider when deleting a record:

Leaf Page Below Fill Factor	Index Page Below Fill Factor	Action
No	No	Delete the record from the leaf page. Arrange keys in ascending order to fill void. If the key of the deleted record appears in the index page, use the next key to replace it.

Leaf Page Below Fill Factor ?	Index Page Below Fill Factor ?	Action
Yes	No	1. Combine the leaf page and its sibling. 2. Change the index page to reflect the deletion of the leaf.
Yes	Yes	1. Combine the leaf page and its sibling. 2. Adjust the index page to reflect the other change. 3. Combine the index page with its sibling. 4. Continue combining index pages until you reach a page with the correct fill factor, or you reach the root.

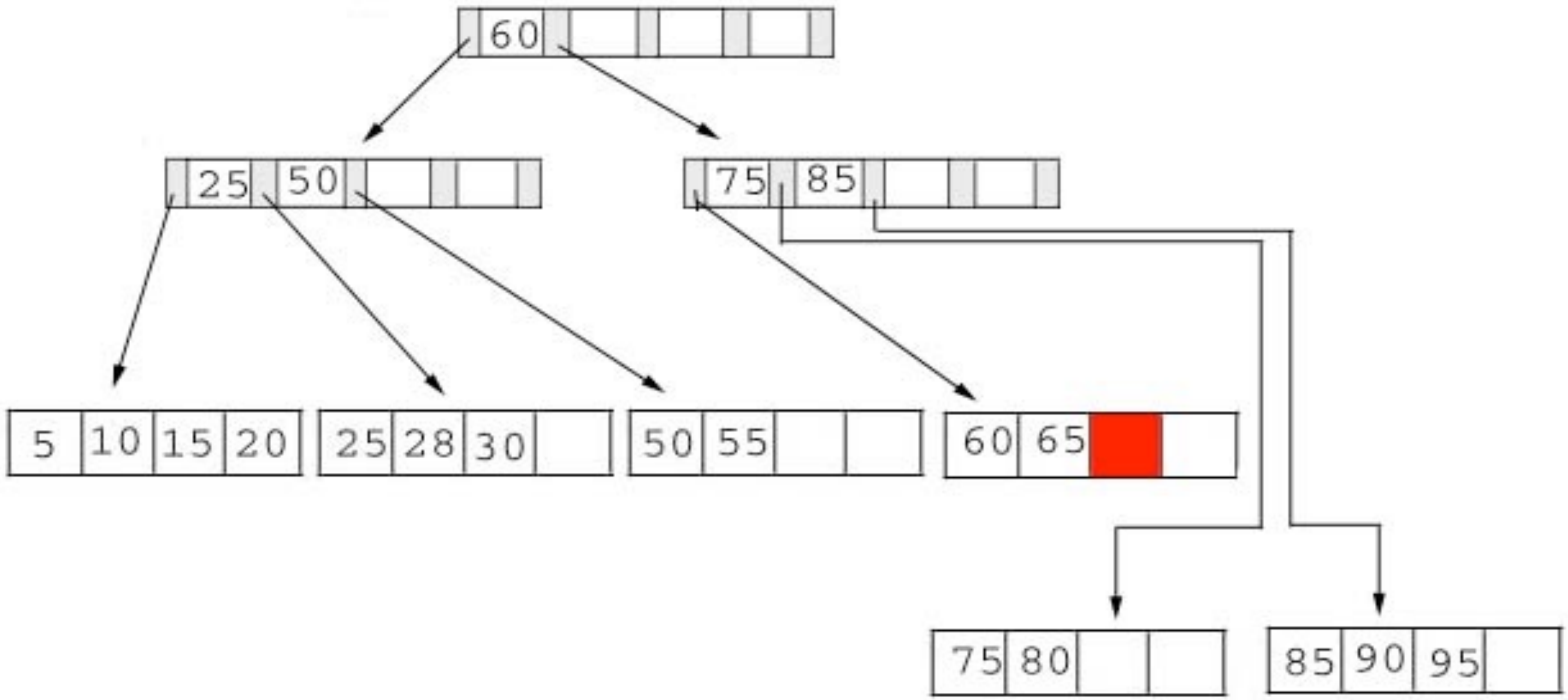
Deletion

- To refresh our memories, here's our B+ Tree right after the insertion of 95:



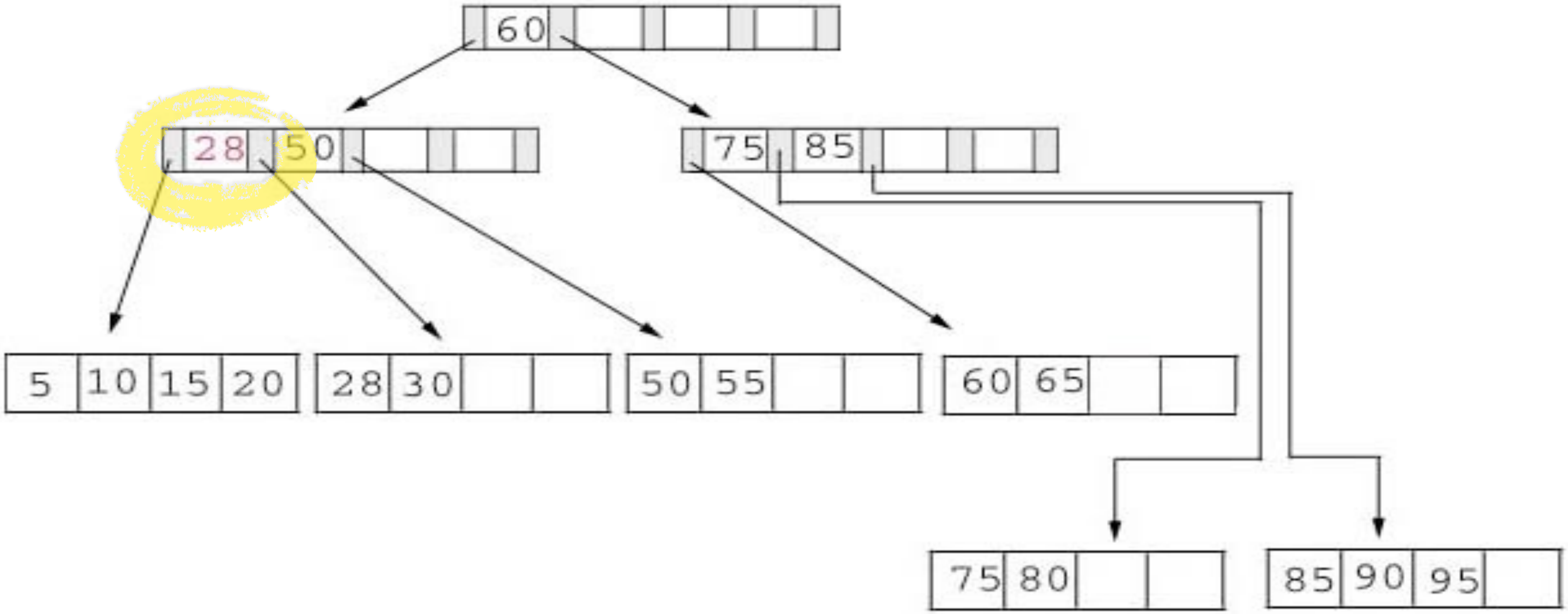
Deletion (1st case)

- Delete record with Key 70



Deletion (2nd case)

- Delete 25 from the B+ Tree
- Because 25 appears in the index page, when we delete 25, we must replace it with another key (here, with 28)

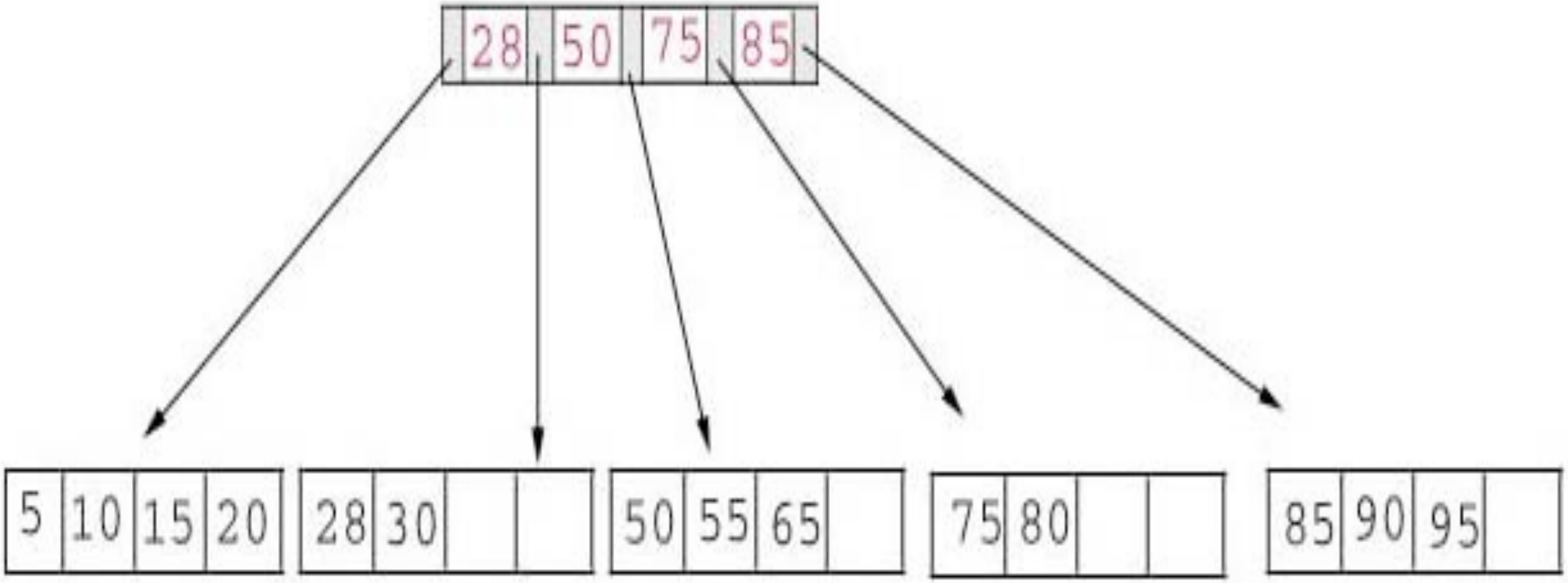


Deletion (3rd case)

- Deleting 60 from the B+ Tree
- The leaf page containing 60 (60 65) will be below the fill factor after the deletion. Thus, we must combine *leaf* pages.
- The recombination removes one key from the index page. Hence, it will also fall below the fill factor. Thus, we must combine *index* pages.
- 60 appears as the only key in the root index page. Obviously, it will be removed with the deletion.

Deletion (3rd case)

- B+ Tree after deletion of 60



B+ Trees in Practice

- Typical Order: 200 to 1000
- Typical Fill Factor: 67%
- Typical Capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records

B^+ Trees in Practice

- Fill factor for newly created tree is a design parameter
- Suppose I create a new B^+ tree with a fill factor of 100%

A. Index occupies minimum number of nodes

B. Adding a new entry will *always* require recursive splitting

C. Both of the above

D. None of the above

B^+ Trees in Practice

- Fill factor for newly created tree is a design parameter
- Suppose I create a new B^+ tree with a fill factor of 50%

A. Index occupies more nodes

B. Adding a new entry is unlikely to require recursive splitting

C. Both of the above

D. None of the above

Example: IBM Red Brick Warehouse 6.3

Each index node is 8172 bytes and corresponds to one 8 kB file system block minus 20 bytes overhead, and key size is the size in bytes of the key itself, plus 6 bytes of address.

Example

Assume a key size of 10 bytes and a table with 50 000 rows. To calculate the number of 8 kbyte blocks required to store the index for various fill factors, use the formulas in Estimating the size of indexes. The results are:

Fill factor	keys per node	Blocks
10%	81	627
50%	409	124
100%	817	63

If this table is to be loaded once and you anticipate no additions, use a fill factor of 100%.

Concluding Remarks

- Tree structured indexes are ideal for range-searches, also good for equality searches
- B^+ Tree is a dynamic structure
  - Insertions and deletions leave tree height-balanced
  - High fanout means depth is often just 3 or 4
  - Root can be kept in main memory
  - Almost always better than maintaining a sorted file

Problem

- Find the minimum order of the B+ tree that guarantees that no more than 3 disk reads will be necessary to access any record in a file of 100 million records.

main memory.

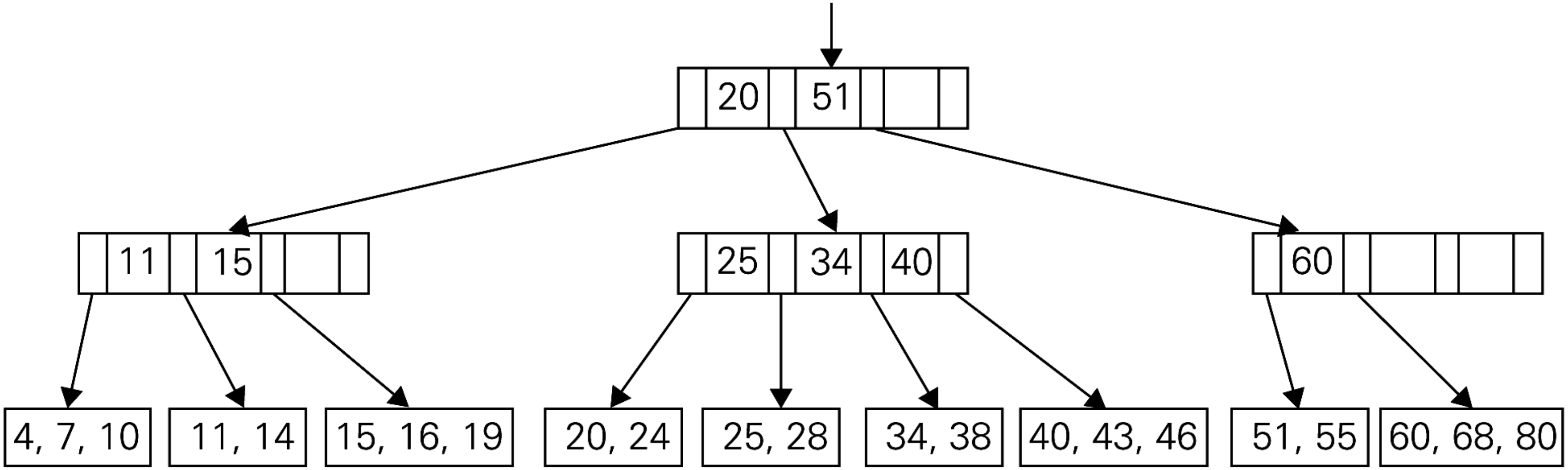
- Hint: look at inequality 17.7

$$h \le \lfloor \log_{\lceil m/2 \rceil} \frac{n+1}{4} \rfloor + 1.$$

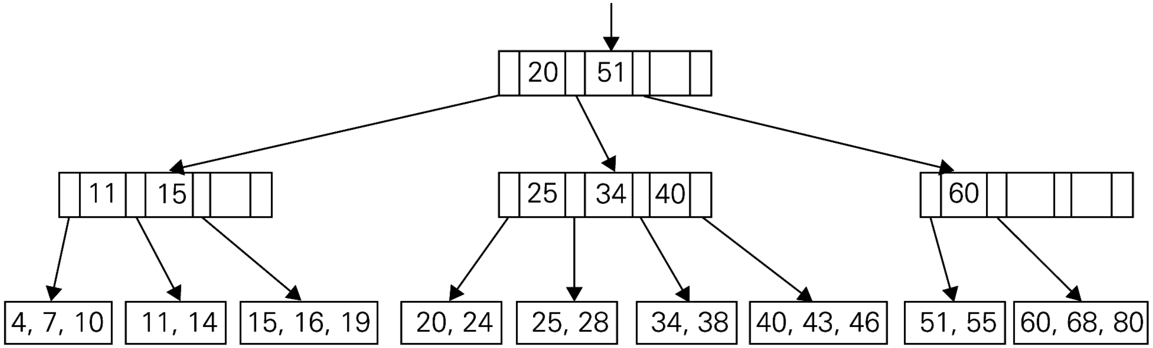
Numeric Answer:

Problem

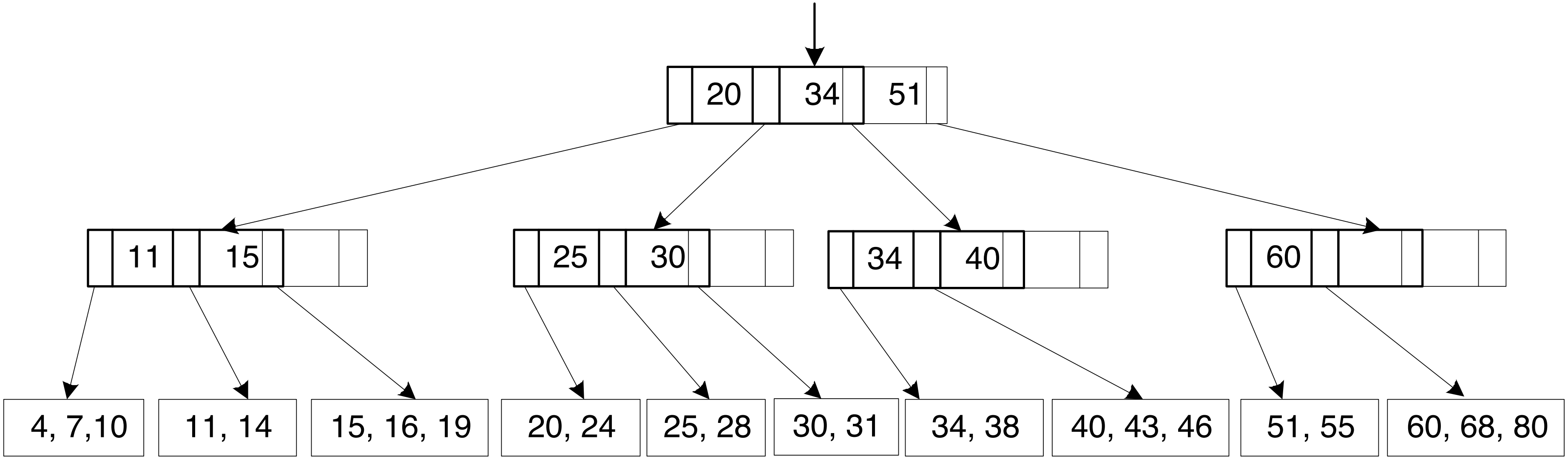
Draw the B-tree obtained after inserting 30 and then 31 in the B-tree in Figure 7.8. Assume that a leaf cannot contain more than three items.



Solution

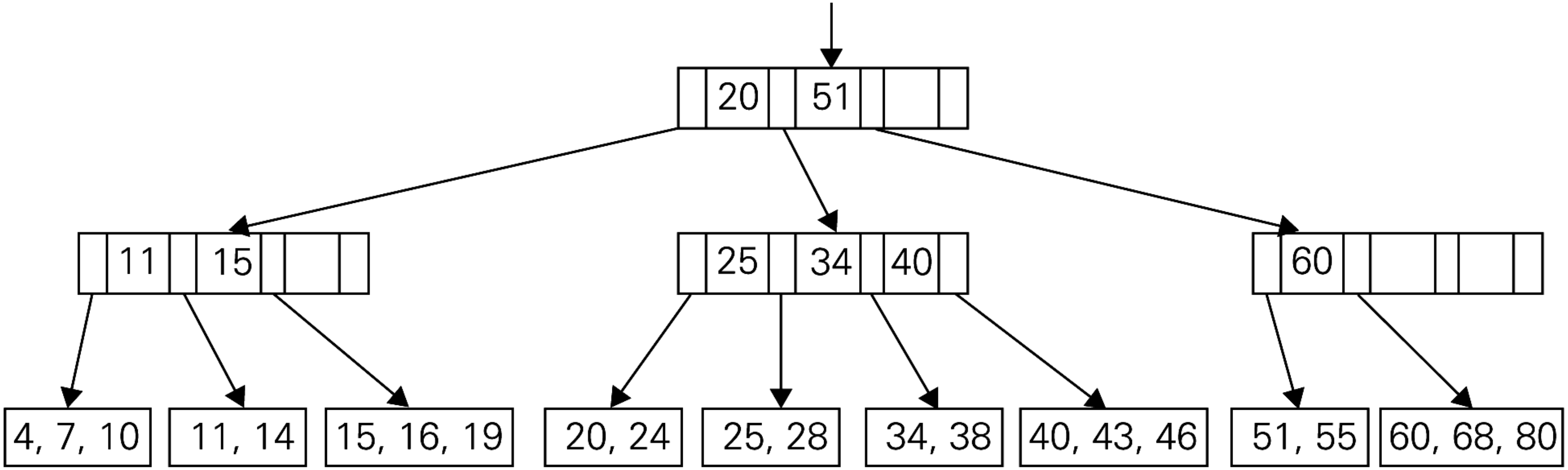


Inserting 31 will require the leaf's split and then its parent's split:



Problem

Outline an algorithm for finding the largest key in a B^{+}-tree.



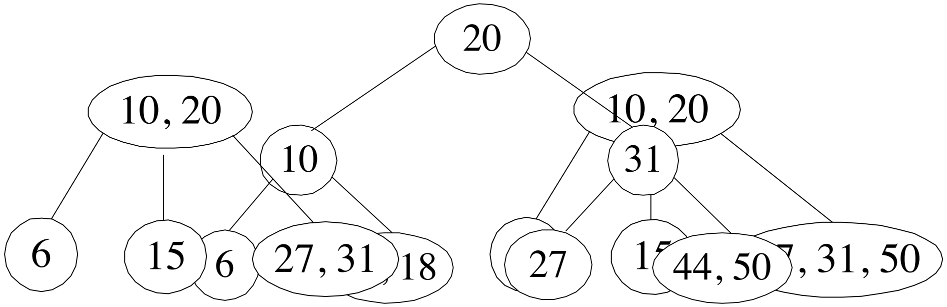
Problem

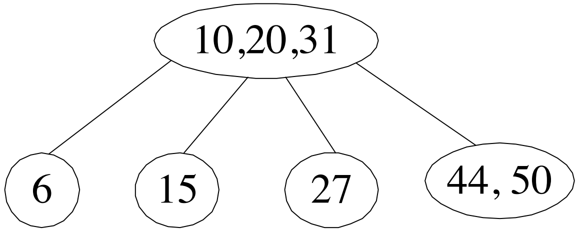
- Why are “recursive splits” bad news in a real database index?

Problem

- Levitin mentions that recursive splits can be avoided by “splitting on the way down”.
- Construct a B tree of order 4 by inserting (in order) into an empty tree

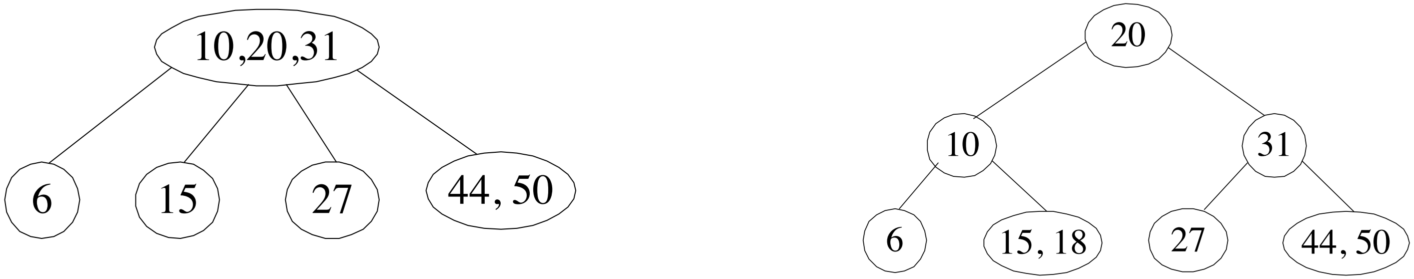
~~10~~, ~~6~~, ~~15~~, ~~31~~, ~~20~~, ~~27~~, ~~50~~, ~~44~~, ~~18~~





Problem

- Levitin mentions that recursive splits can be avoided by “splitting on the way down”
- What are the advantages and disadvantages of this variation (called a top-down B tree)?


10,20,31
6
15
27
44, 50
20
10
31
6
15, 18
27
44, 50