CMPT 100
FALL 2017

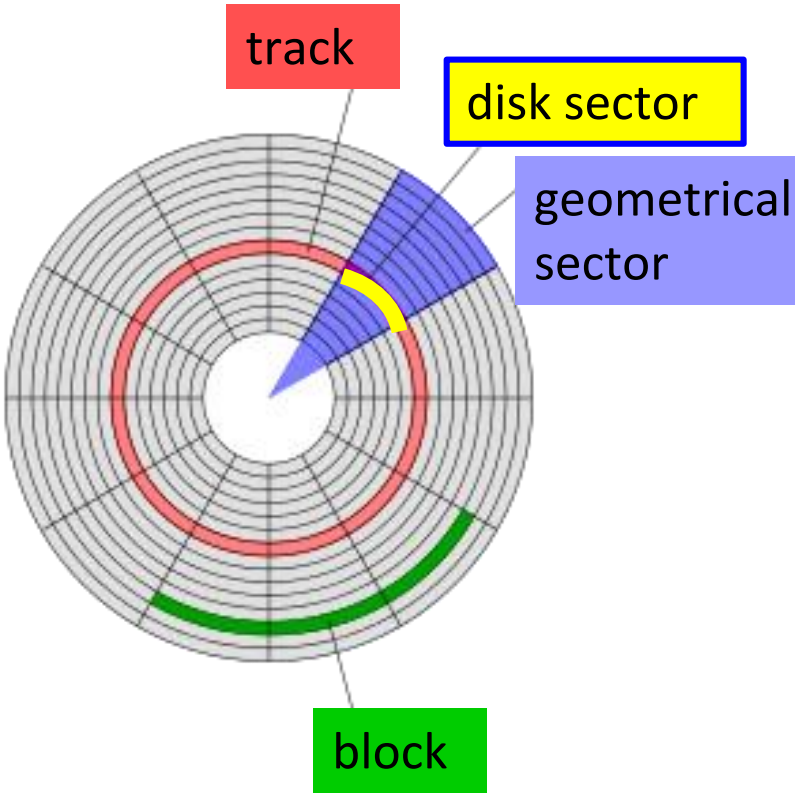
Indexes

Lecture 07.01

By Marina Barsky

Disk blocks

- The ***Block*** (transferred as a ***Page*** to RAM) is a fixed-size portion of secondary storage corresponding to the amount of data transferred in a single access and physically occupies one or more consecutive sectors
  - Typical block size: 1, 4, 8, 16, or 32 KB.
  - Has to be set before creating database
- The data is read and written in blocks



Auxiliary data structures for efficient search: indexes

- Goal: quickly locate the record given a key
- Idea 1:
  - The records are mapped to the disk blocks in specific ways: we deduce the disk location from a key, because records either sorted by key or the block is a hash of a key
- Idea 2:
  - Store records in a pile
  - Provide auxiliary data structures guiding the search (think library index/catalogue)

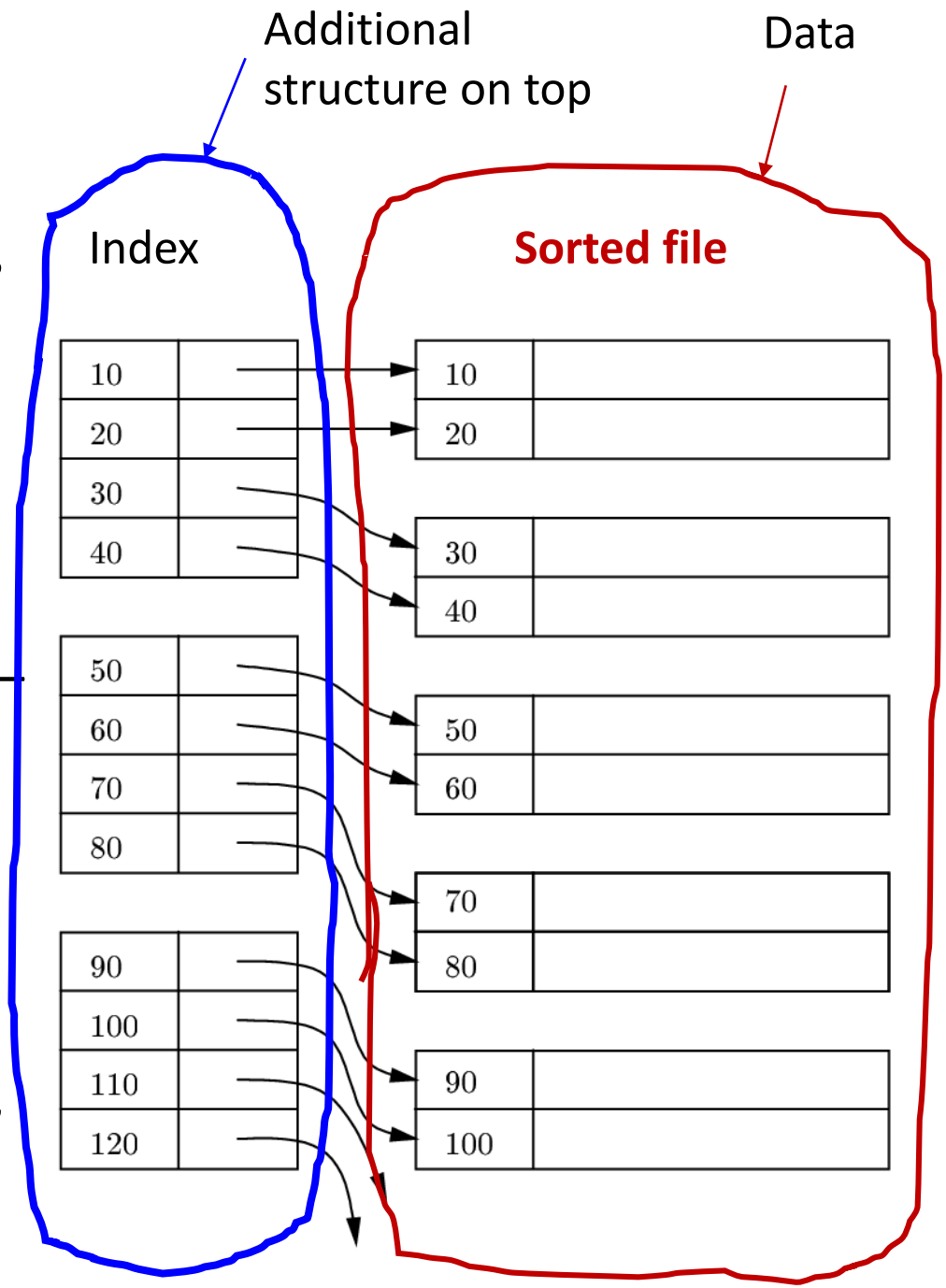
Flat indexes

- Have a catalog of search keys which is smaller than the entire table and can be searched more efficiently (in RAM or with less disk I/Os)

- Inside the index each value of a search key is associated with a unique, system-generated physical address of a corresponding tuple on disk: RID (file number, block number, slot within the data block)

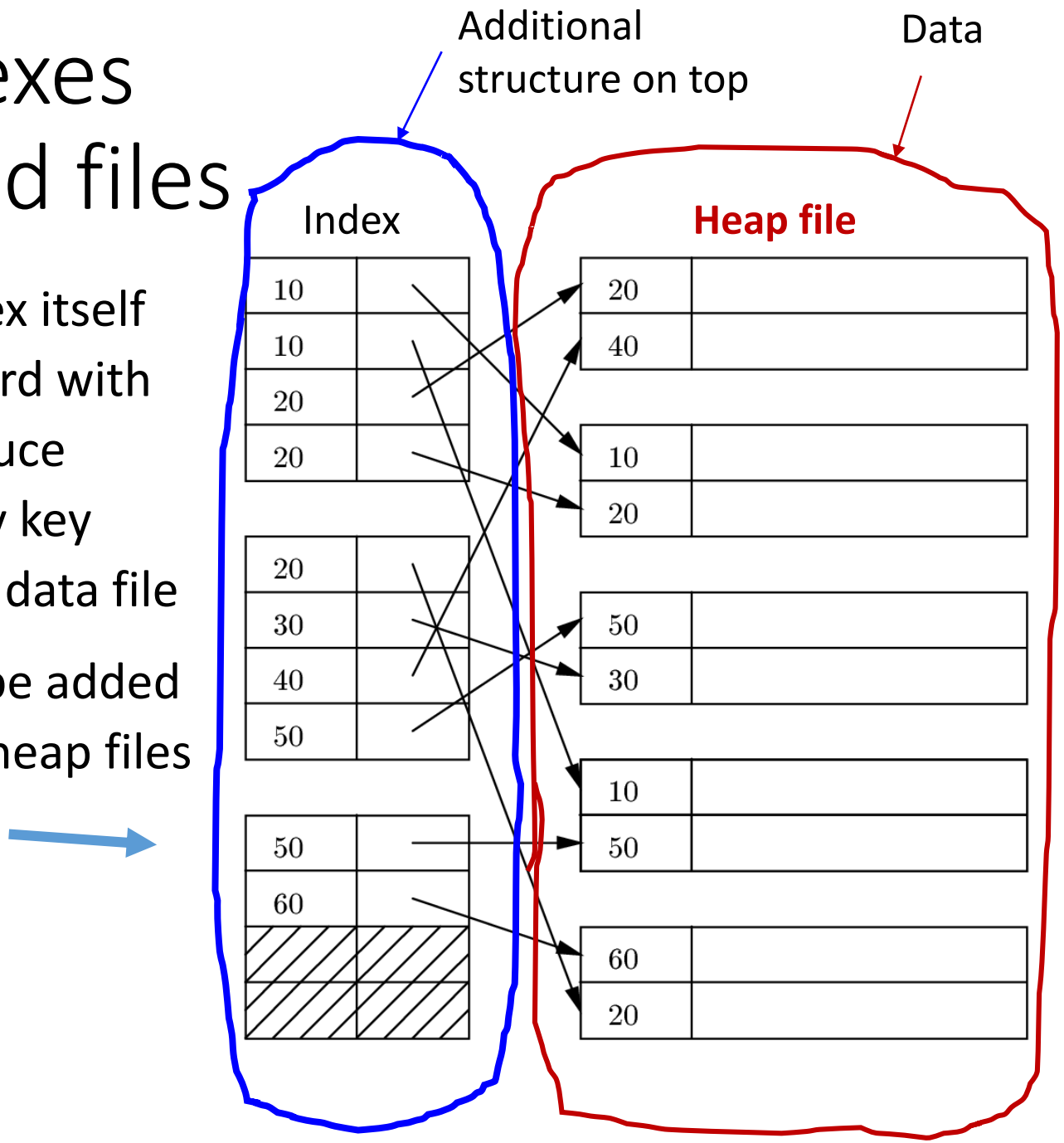
Dense indexes

- ***Dense index*** – each record has its representative inside an index
- If the table has multiple fields, the index – which stores only key-RID pair - is much smaller – may fit into RAM
- The keys in the index are sorted: use binary search, buffer guiding pointers at 1/2N, 1/4N, 3/4N, 1/8N, 3/8N, 5/8N, 7/8N –th positions to save disk I/Os



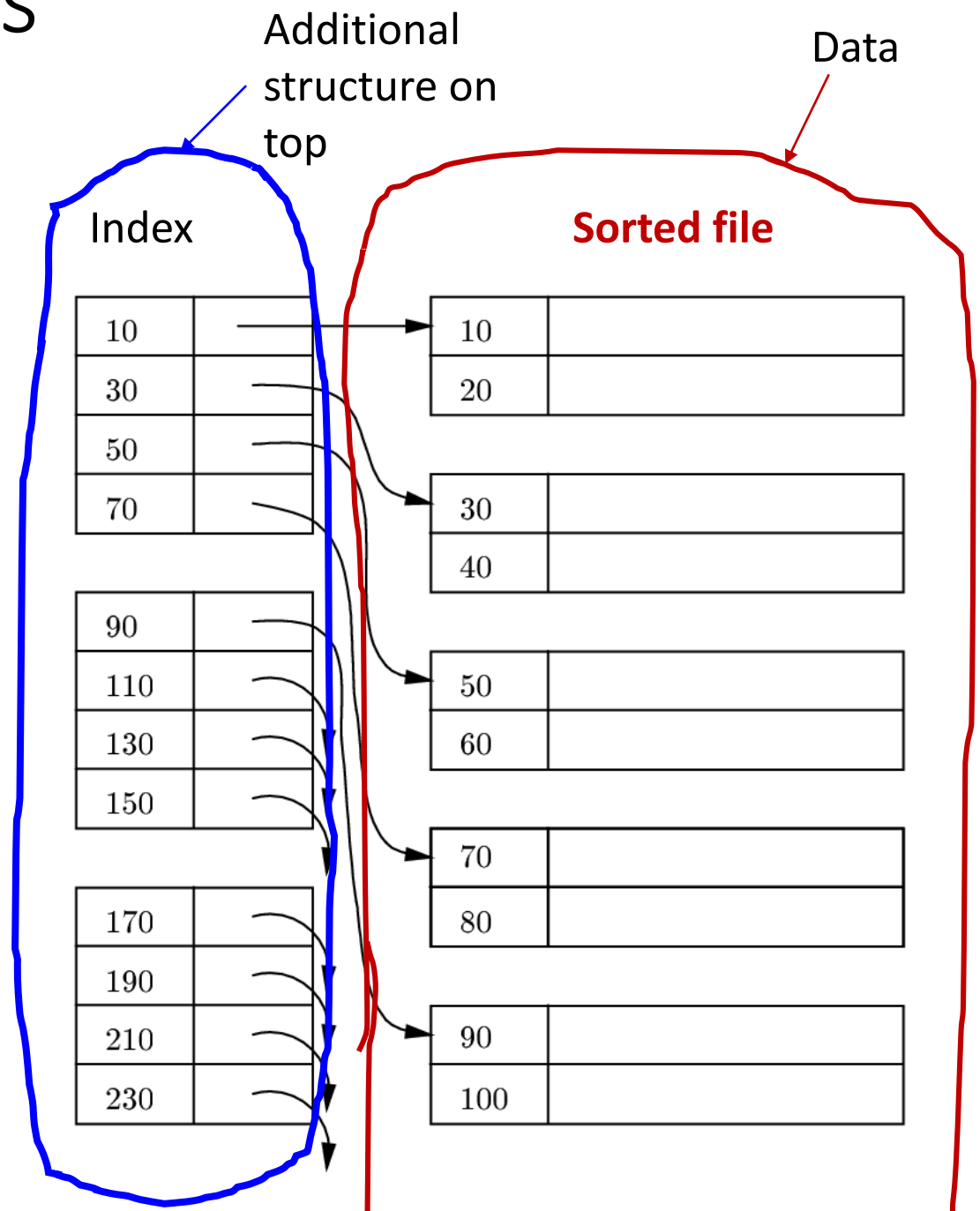
Dense indexes on unsorted files

- Search through index itself can answer if a record with key *A* exists or produce counts of records by key without accessing a data file
- Dense indexes can be added even to unordered heap files



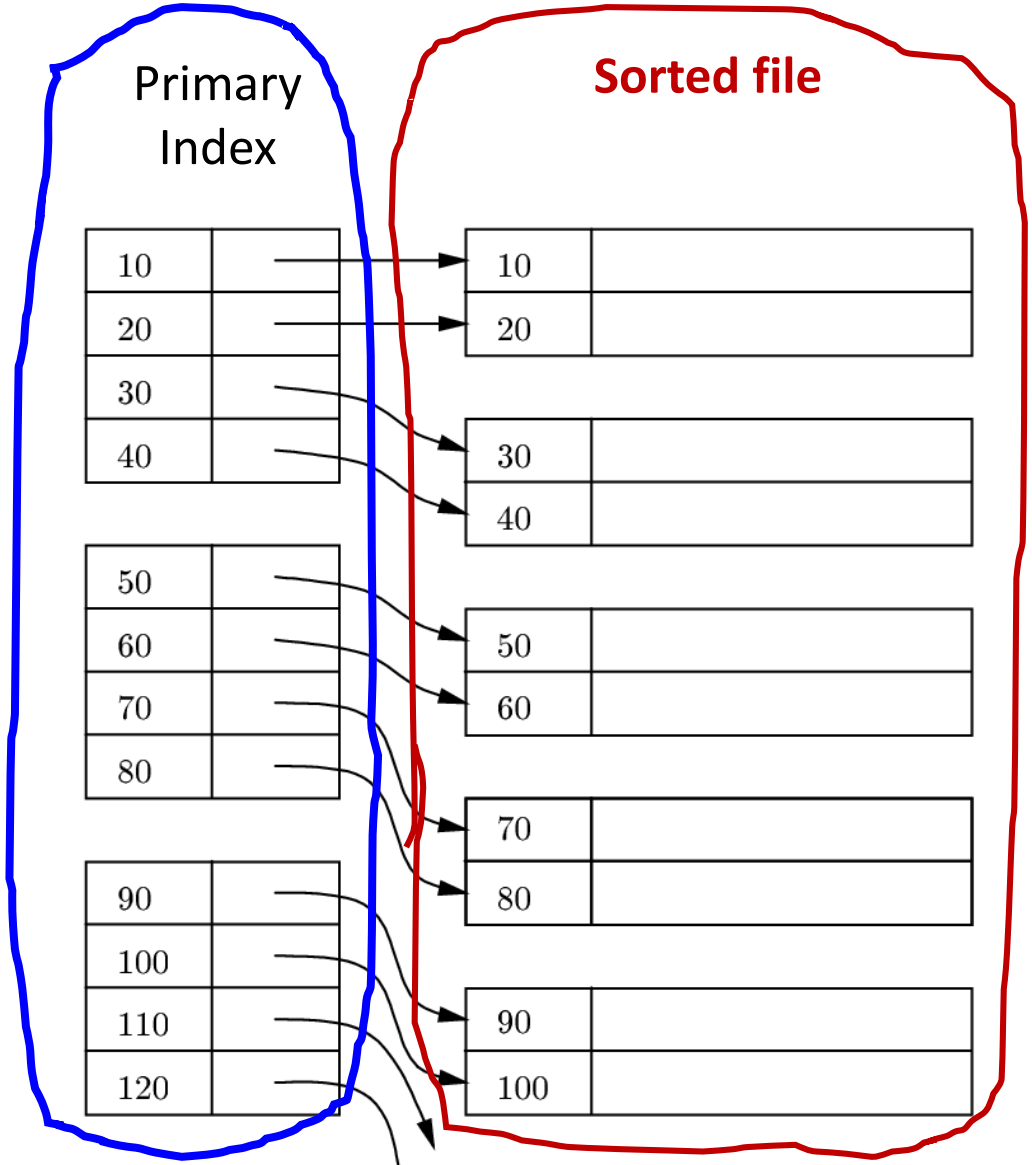
Sparse indexes

- ***Sparse index*** – contains key-RID pairs for only a subset of records, typically first in each block.
- Works only with sorted files – why?
- Allows for very small indexes - better chance of fitting in memory
- Tradeoff: *must* access the relation file even if the record is not present



Primary indexes

- ***Primary index*** – indexes on a sorted file for the sorting attribute
- Only one primary index per relation – otherwise needs to maintain several sorted copies of the same data



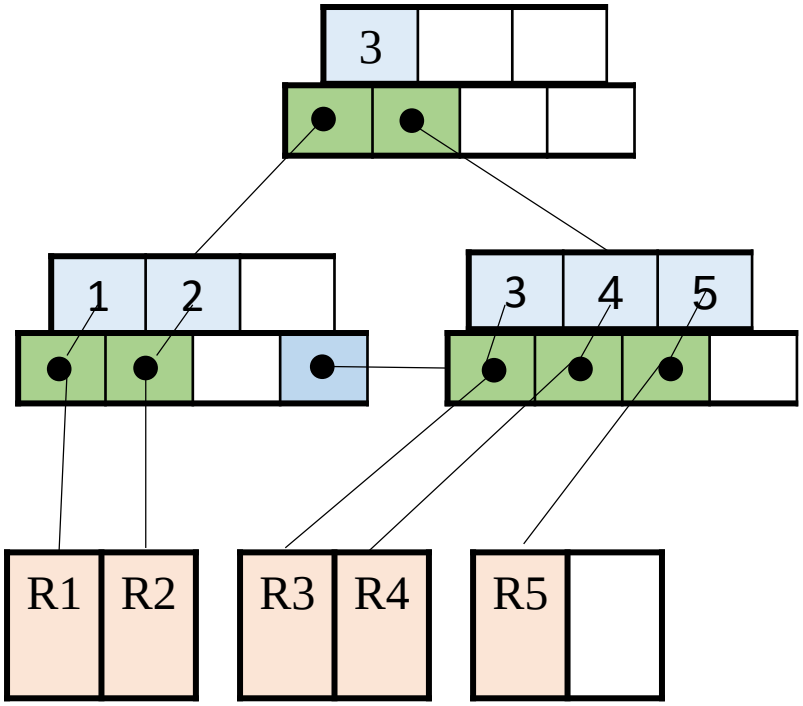
What if a flat index is too big?

Example:

- Relation of size: N = 500 GB = $5*10^{11}$ bytes
- 100 tuples per block: $5*10^{9}$ blocks to index
- Each key-blockID pair is at least 16 bytes
- So, even keeping one entry per page (sparse index) takes too much space - 8 GB

Solution: build an index on the index itself!

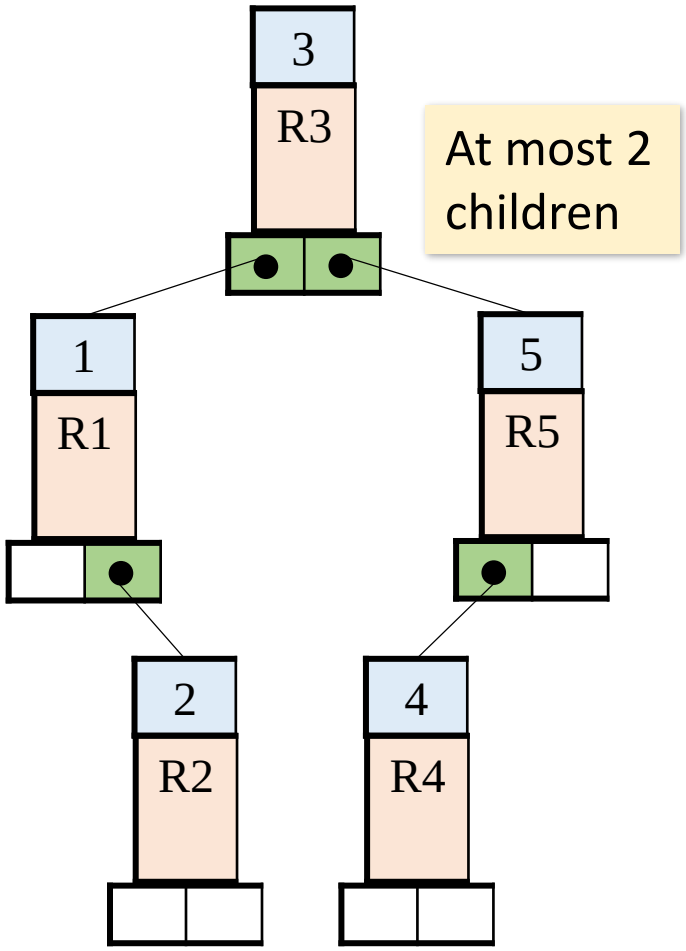
Dynamic indexes: B+ trees



Data stored separately

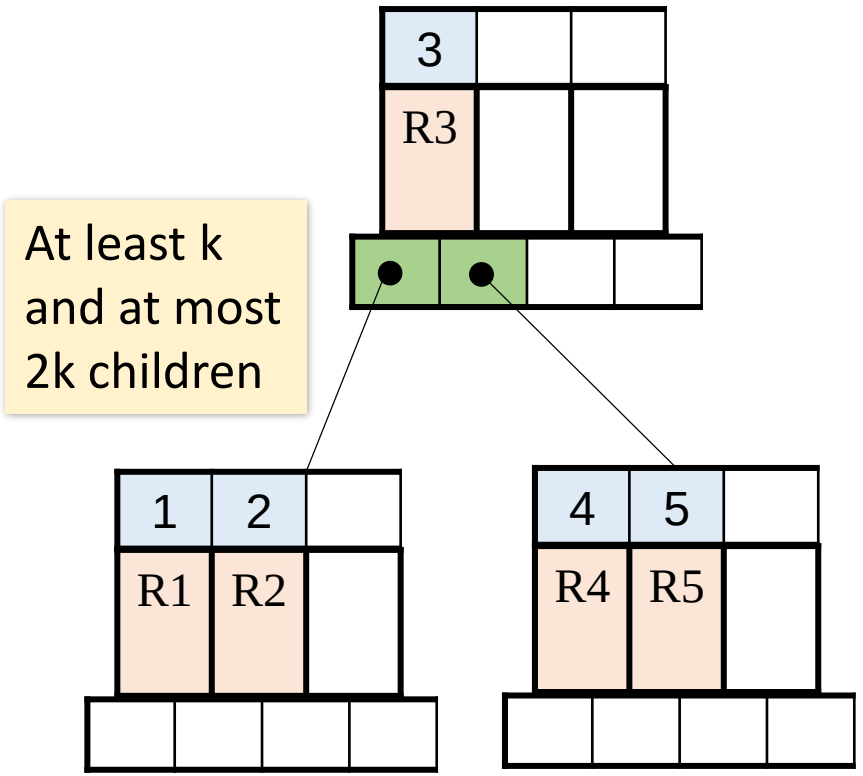
B+ tree

From binary search trees to k-2k B-trees



At most 2 children

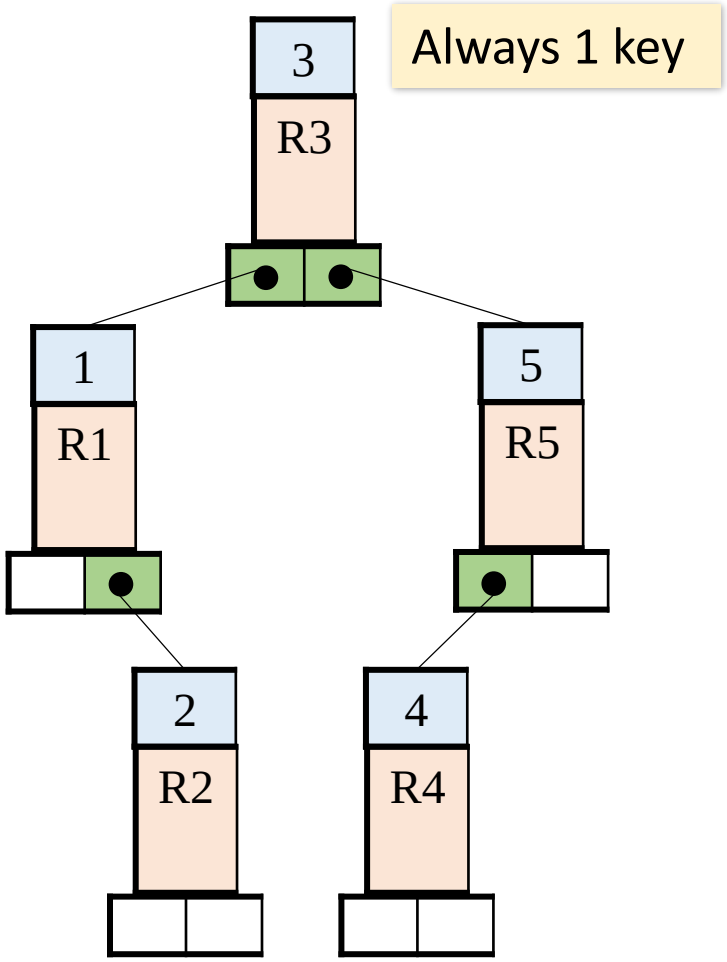
Binary search tree



At least k and at most 2k children

2-4 B-tree

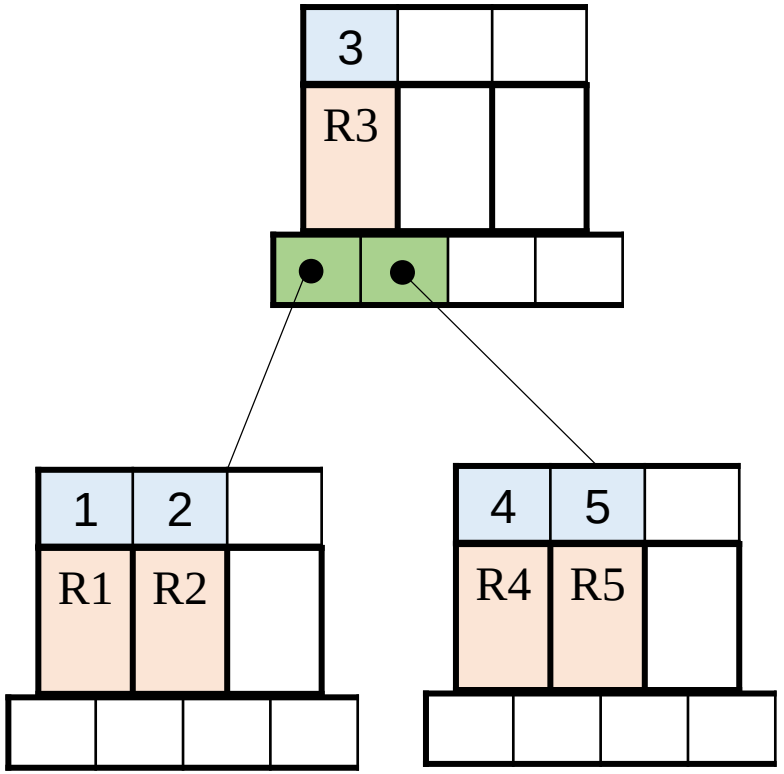
From binary search trees to k-2k B-trees



Always 1 key

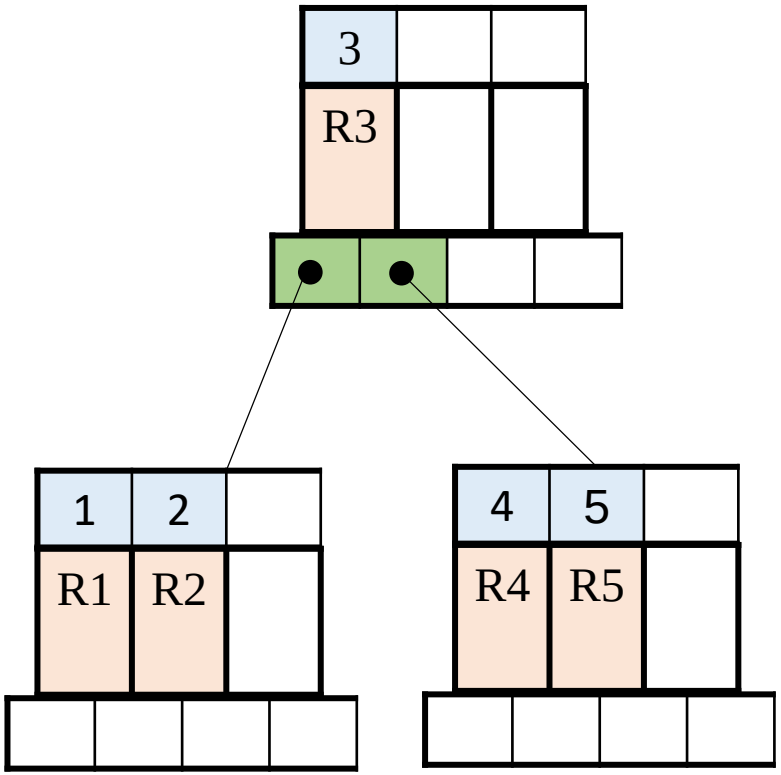
Binary search tree

At least k-1 and
at most 2k-1 keys



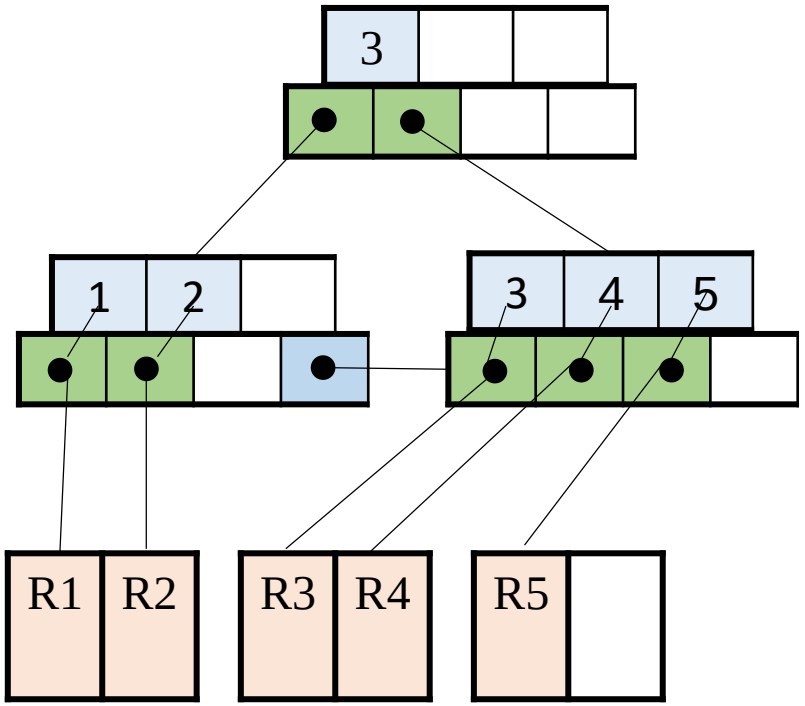
2-4 B-tree

From B-trees to B+ trees



Data stored together with the key

B-tree



Data stored separately

B+ tree

B+ tree vs. B-tree

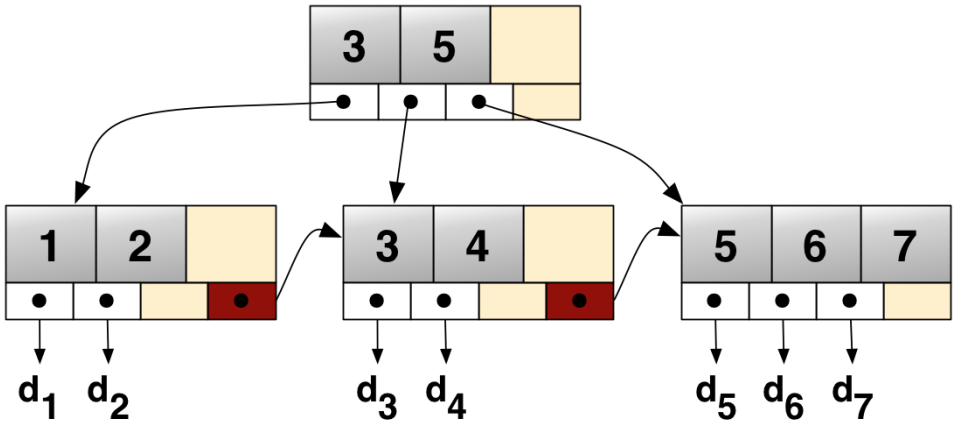
1. B+ -tree is a B-tree where internal nodes contain only keys and navigation pointers (not records, not pointers to records), and all the records (or pointers to records) are stored in leaves.
2. In B+ tree each internal node is stored in a page, and more keys fit in a single page. The navigational part of the index is overall smaller, and partly manageable in RAM.
3. The leaf nodes of B+ trees are linked, so doing a full scan of all objects in a tree requires just one linear pass through all the leaf nodes. A B tree, on the other hand, would require a traversal of every level in the tree (random access).

B+ tree (or simply B-tree)

- B+ tree is the only variant of B-trees used in DBMS
- In all research papers and implementations: we say B-tree - imply B+ -tree

k-2*k* B trees: properties I

- Each node contains p pointers: ***k <= p <= 2k***



k=2.

Each node has 2,3,or 4 child pointers and between 1 and 3 keys

k-2*k* B trees: properties II

- There are 2 types of nodes:
  - **Internal** (non-leaf) node:
    - all p pointers point to the child nodes
    - p-1 keys contain navigational info
  - **Leaf** node: 1 pointer points to the next leaf
    - p-1 key-value pairs (child pointers), where value can be RID or the entire record
    - the number of child pointers is at least k*

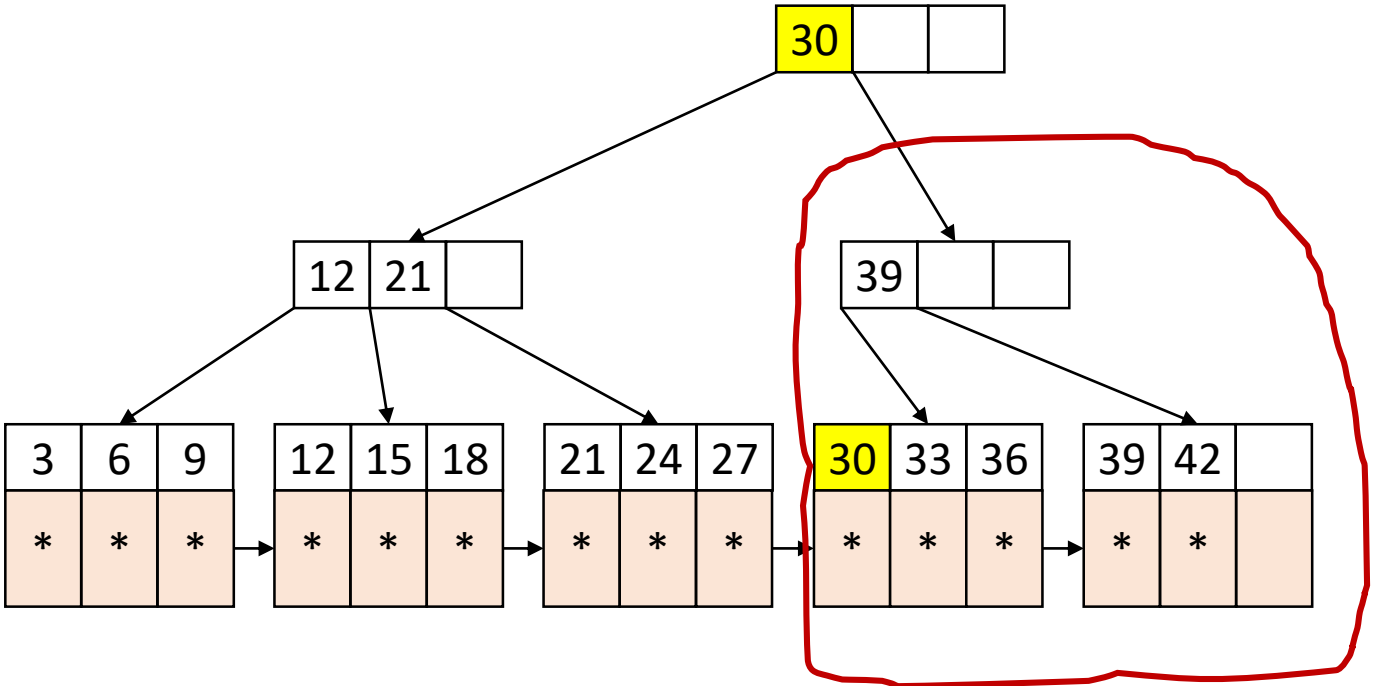
---

* In practice, the leaf node may have its own parameters on min and max key-value pairs, because the size of a value in key-value pair can be larger than the pointer used in the internal node. However, for the purposes of this lecture, we assume that min number of key-value pairs is k, and max is 2k-1 (1 pointer points to the next leaf).

k-2*k* B trees: properties III

- Each key in an internal node guides the search:

  All keys in the left sub-tree of a given key *X* have key value < *X*, all keys in the right sub-tree have key value >= *X*



Degree=order=fanout =branching factor

- Degree *d (=2*k)* means that all internal nodes have space for at most *d* child pointers

Example

- Each node is stored in 1 block of size 4096 bytes
- Let
  - key 4 Bytes,
  - pointer 8 Bytes.
- Let's solve for *d*:

$4(d-1)+8(d) \leq 4096$

$\Rightarrow \; d \leq 341$

B-tree capacity: example

$d \approx 300$

a typical node is 67% full (*fill factor*) ≈ 200 keys in each node

We have:

- 200 keys at the root
- At level 2 – for each key – another 200 keys – total 200^2 nodes
- At level 3: 200^3
- At level 4: $200^4 \approx 16 \times 10^8$ records can be indexed.

- Suppose each record = 1 KB - we can index a relation of size

  $16 \times 10^8 \times 10^3 \approx 1.6$ TB

- If the root and levels 2 and 3 are kept in main memory, then finding RID requires 1 disk I/O!

Buffering top-level nodes

- Often top levels are held in buffer pool:
  - Level 1 = 1 page = 4 KB
  - Level 2 = 200 pages = 800 KB
  - Level 3 = 40,000 pages = 160 MB

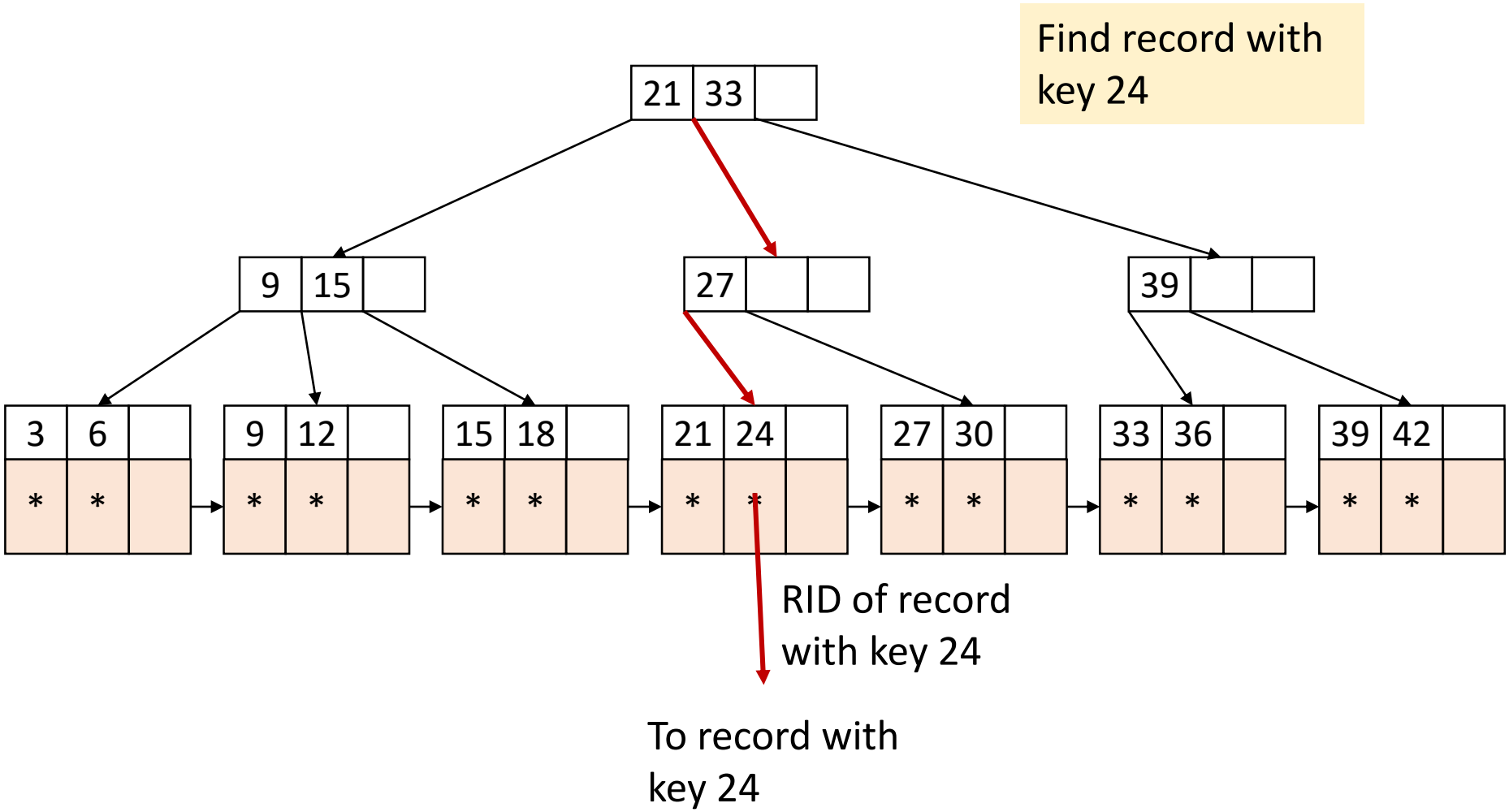
- In this case, in practice, lookup requires 1 disk I/O

B-tree lookup

Recursive procedure:

- Ends when we are at a leaf. In this case, look among the keys there. If the *i*-th key is K, then the *i*-th pointer will contain RID of the desired record.
- If we are at an internal node with keys $K_1, K_2, \ldots, K_d$, then if $K<K_1$ we call *lookup* with the first child node, if $K_1 \leq K < K_2$ we use the second child, and so on.

B-tree lookup example



B-tree: range search

- Query: select all records where key is in range [*x*,*y*]
  - Use *x* as a search key
  - Once at the leaf: scan the data entries to find *x* or the first key that is > *x* (if *x* is not there)
  - After that, data entries are retrieved sequentially until the first record with key > *y*

B-tree in action

- When data are inserted or removed from a node, its number of child nodes changes.
- In order to maintain the pre-defined capacity range, internal nodes must be joined or split.

- B-tree is a dynamic data structure with a guaranteed upper bound for lookup, insertion and deletion: **O (log $_d$ N)** disk I/Os

where

N – total number of leaf nodes (leaf blocks)

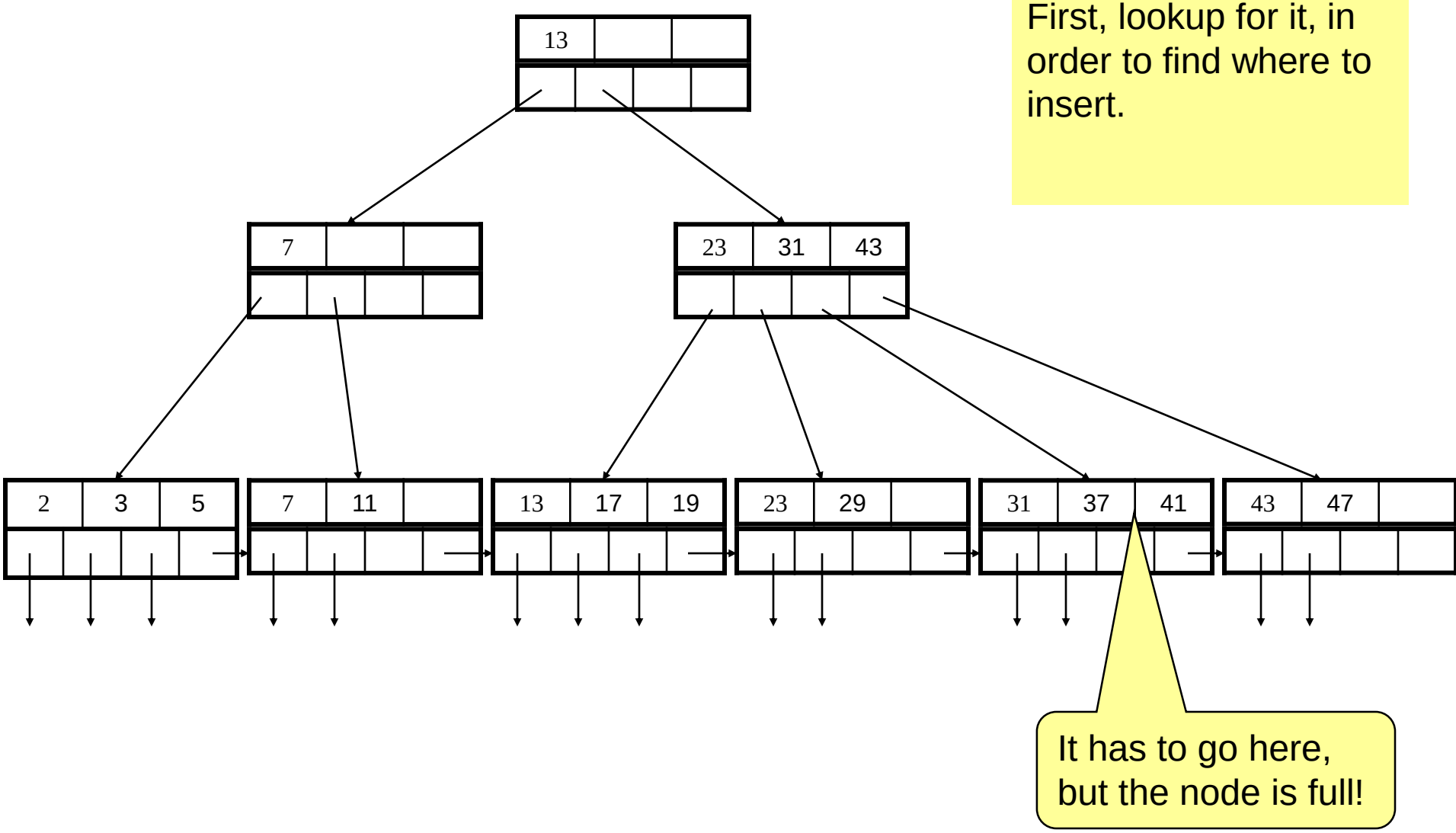
d – branching factor

B Trees: efficiency

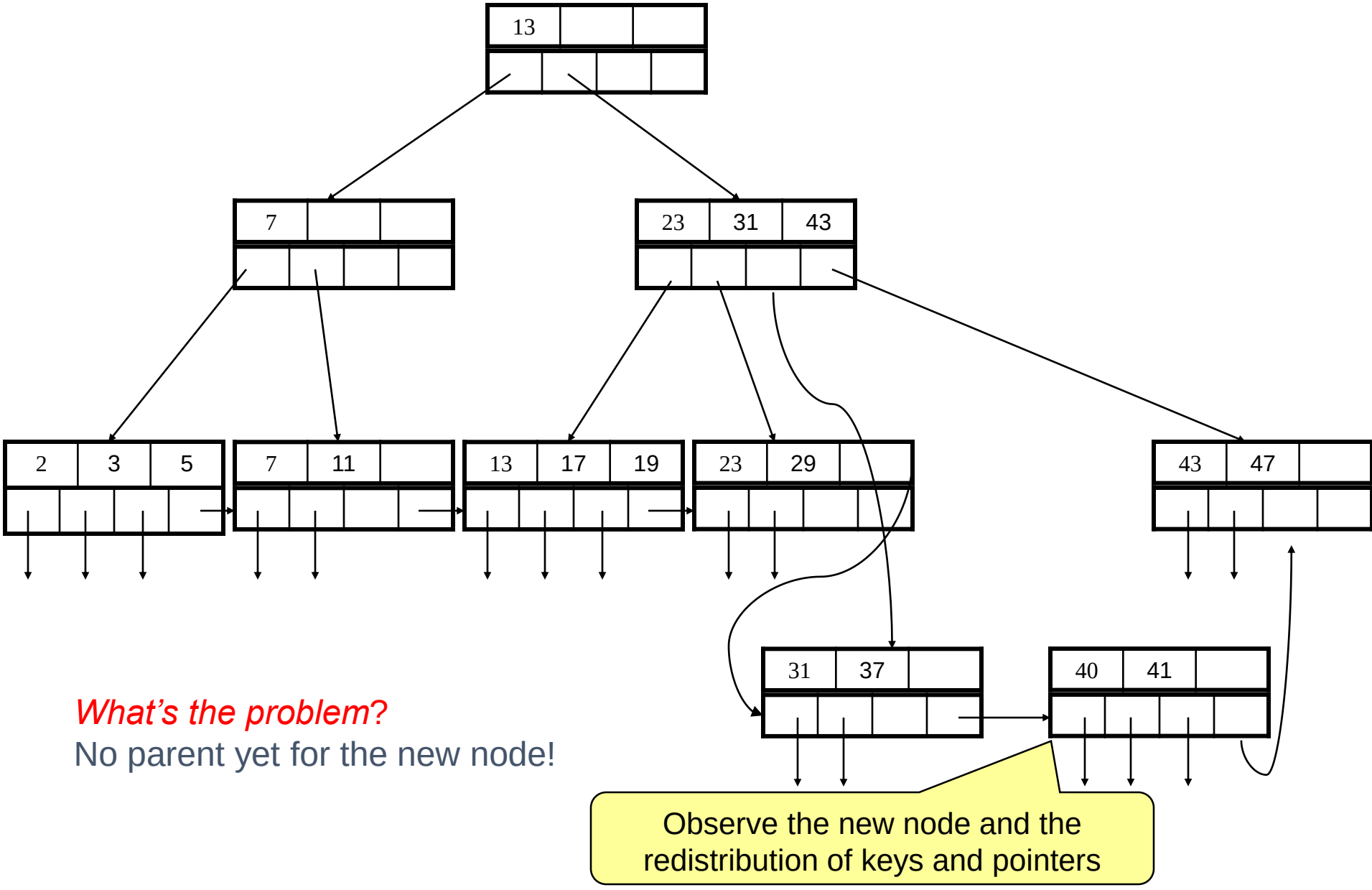
- Searching:
  - $log_d(N)$ – Where *d* is the order, and *N* is the maximum total number of entries in all the leafs
- Insertion:
  - Find the leaf to insert into
  - If full, split the node, and adjust index accordingly
  - Similar cost as searching
- Deletion
  - Find the leaf node
  - Delete
  - May not remain half-full; must adjust the index accordingly
    - Either borrow 1 key from the sibling
    - Or merge with the sibling if there are not enough keys to borrow

Insertion

Try to insert a search key = 40.
First, lookup for it, in order to find where to insert.



Beginning of the insertion of key 40

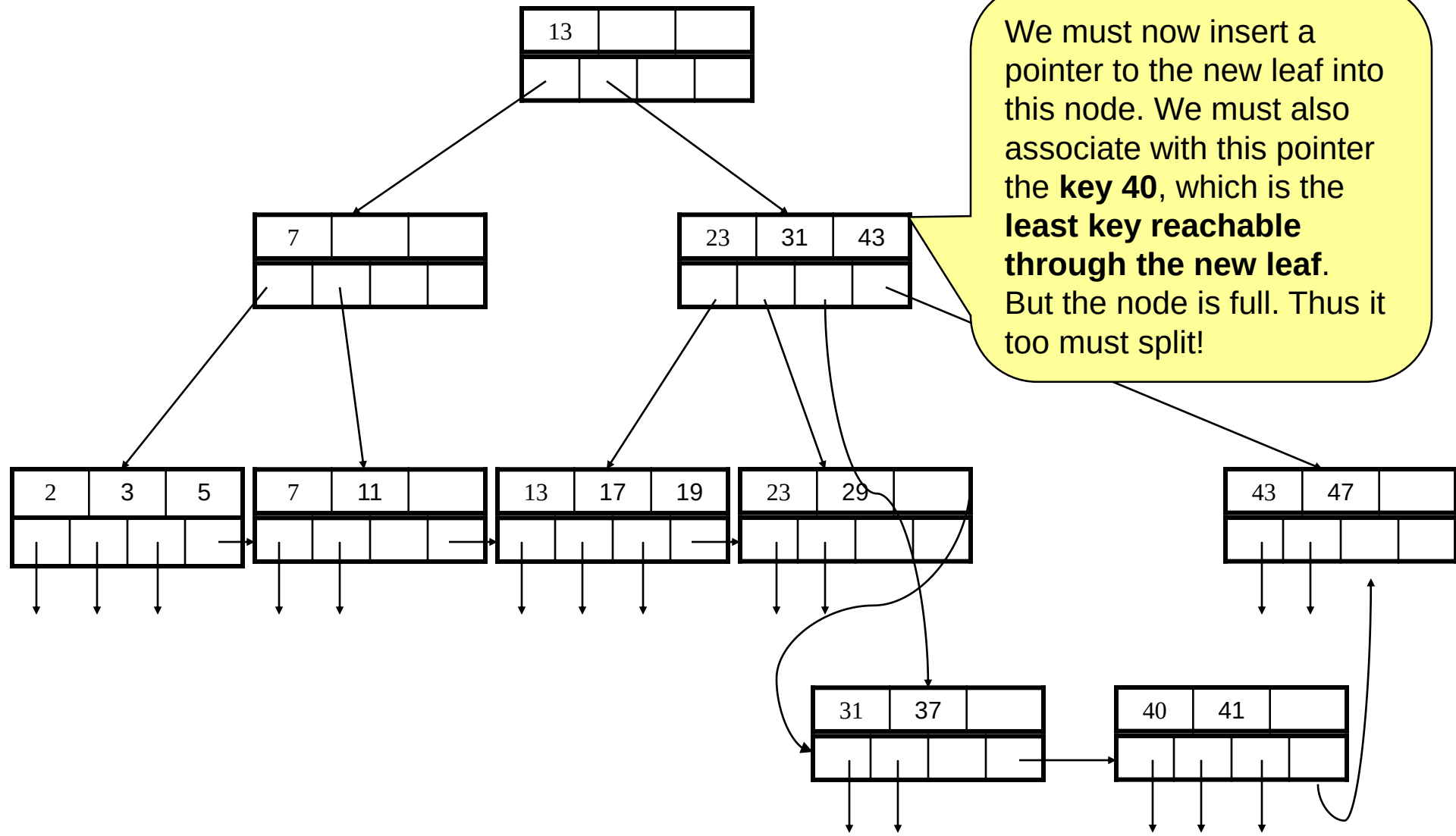


What's the problem?

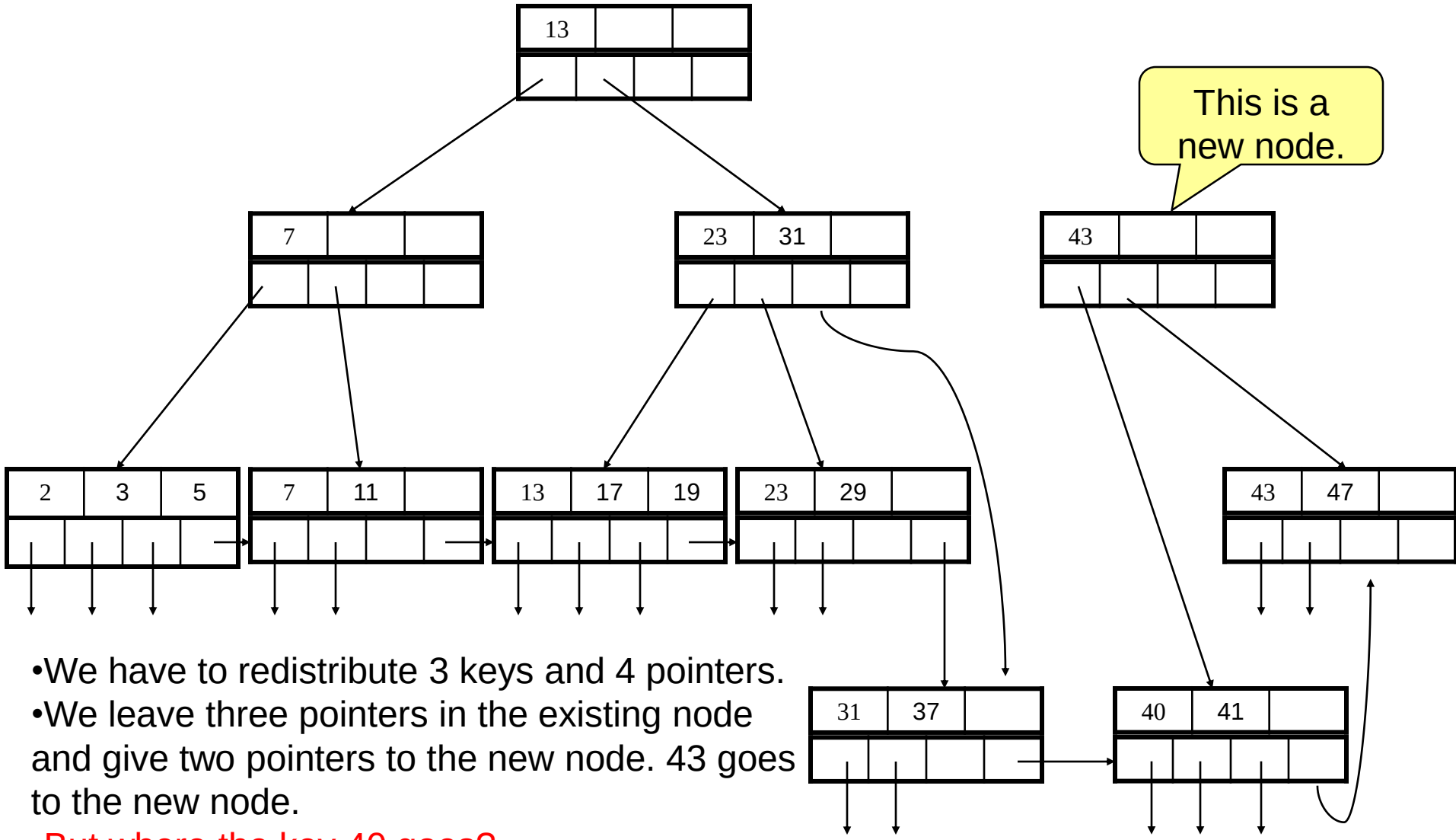
No parent yet for the new node!

Observe the new node and the redistribution of keys and pointers

Continuing the Insertion of key 40

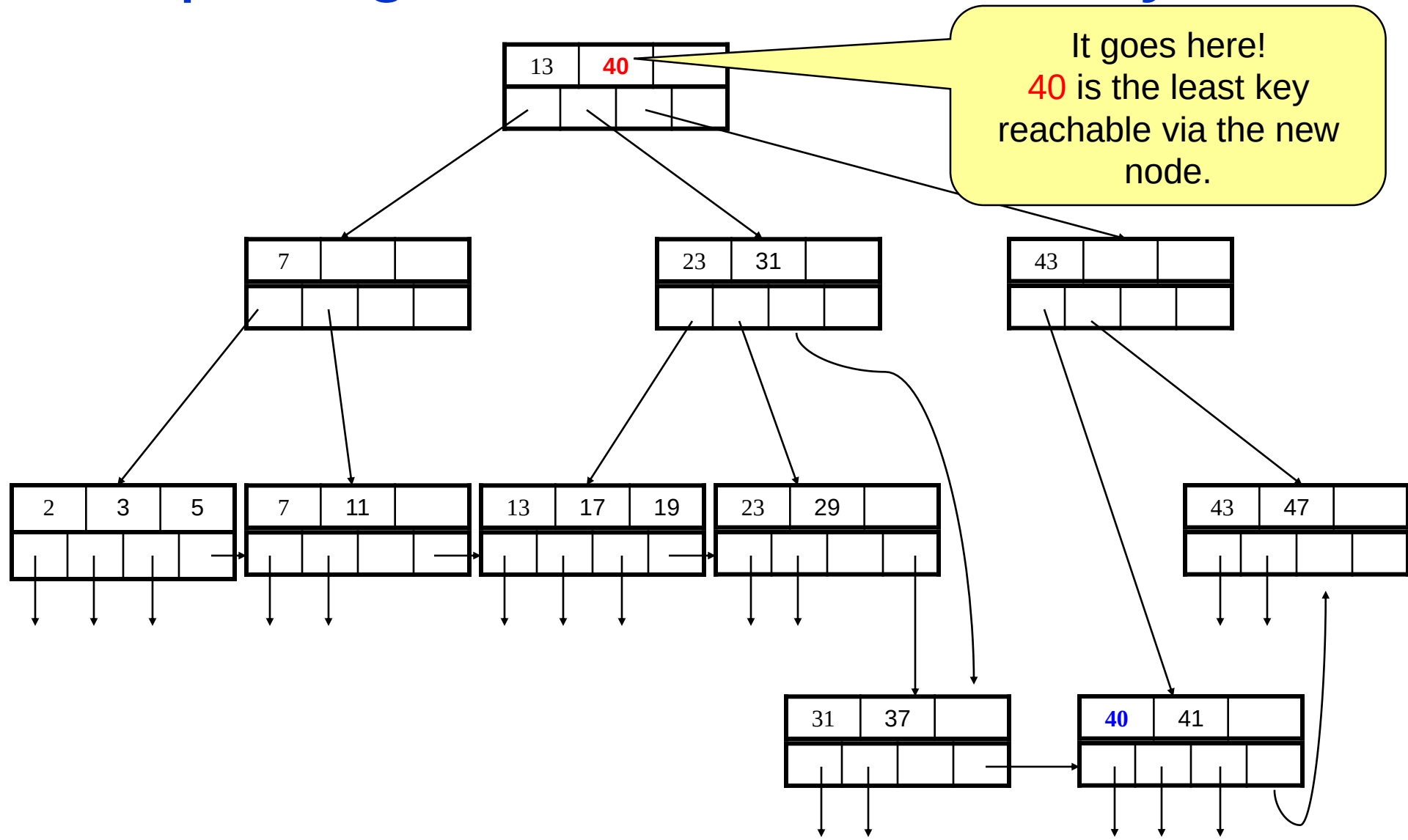


Completing of the Insertion of key 40



- We have to redistribute 3 keys and 4 pointers.
- We leave three pointers in the existing node and give two pointers to the new node. 43 goes to the new node.
- But where the key 40 goes?
- 40 is the least key reachable via the new node.

Completing of the Insertion of key 40

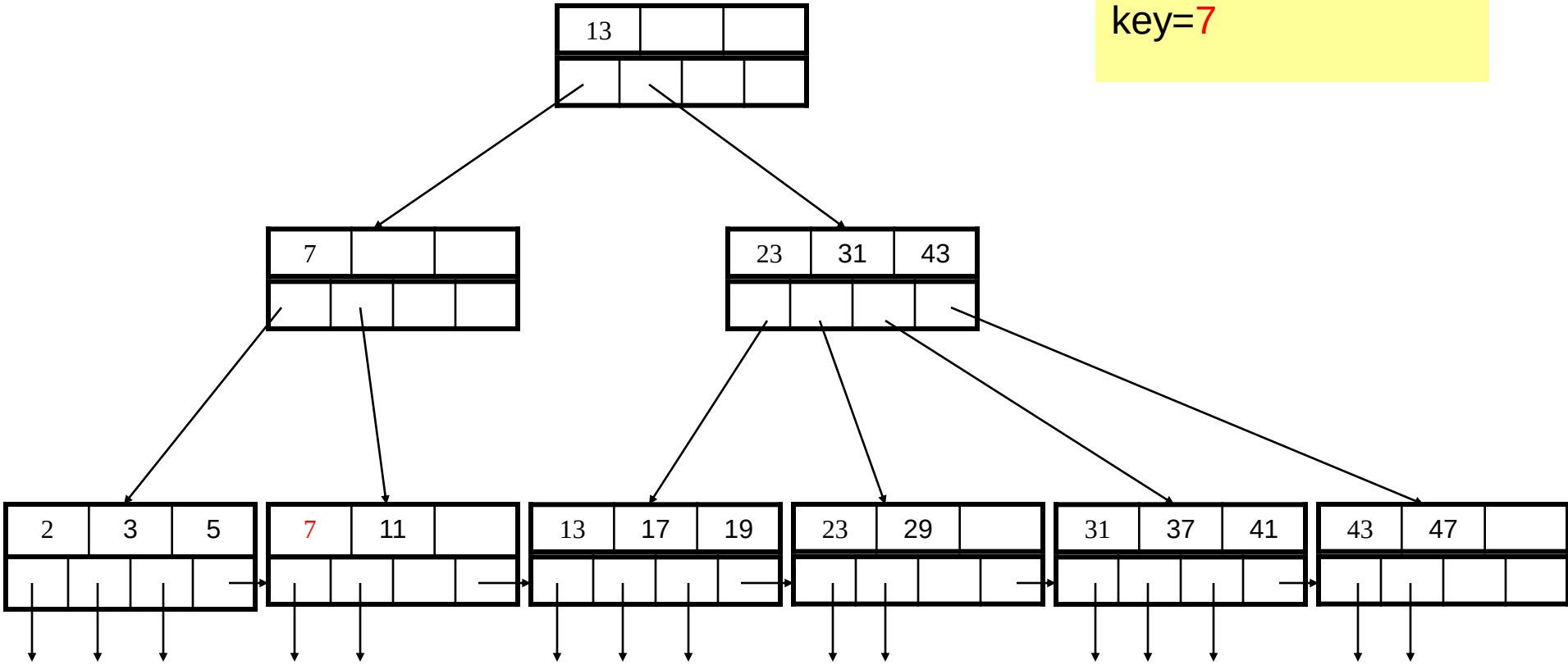


Insertion in words

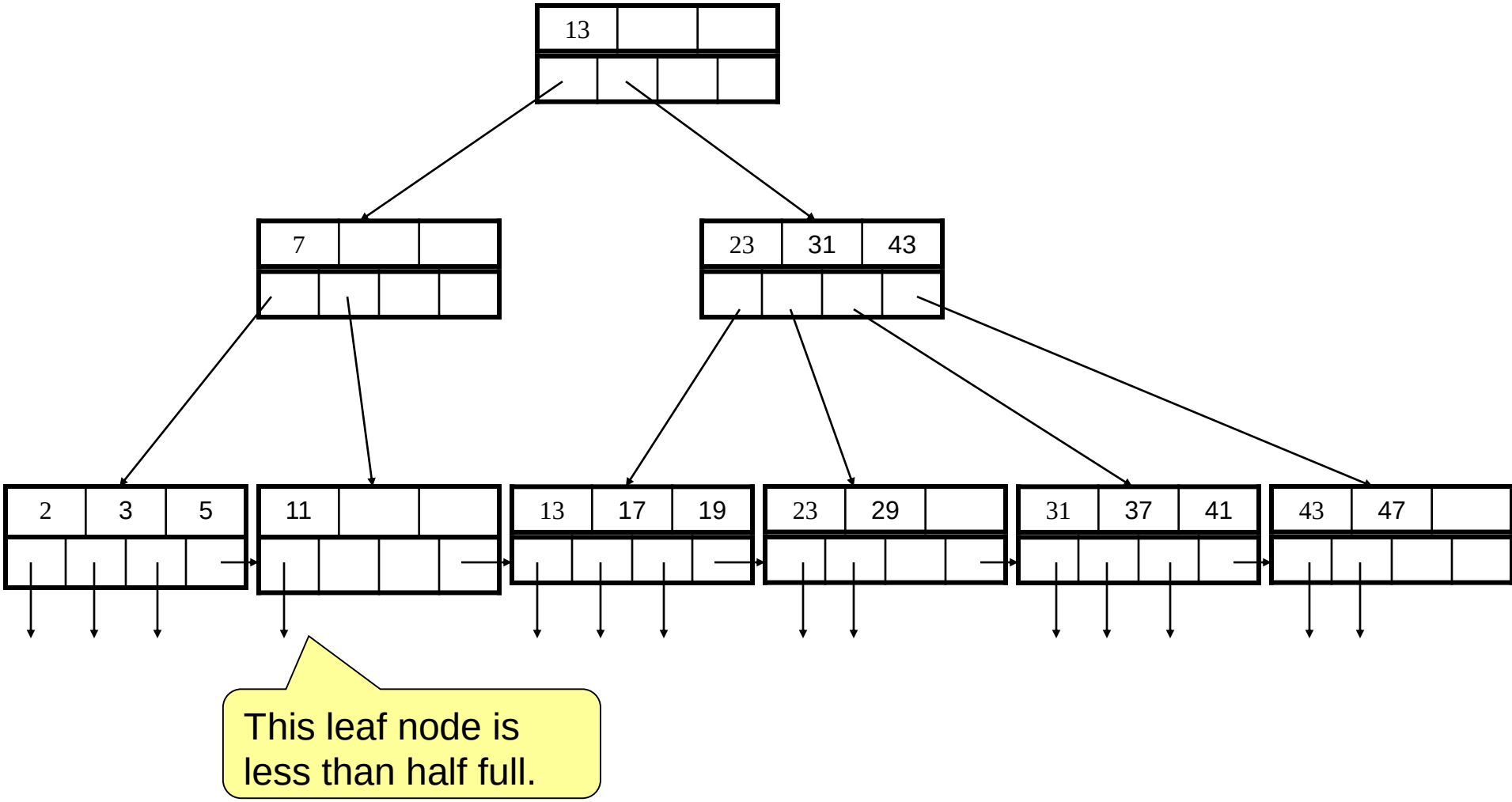
- We try to find a place for the new key in the appropriate leaf, and we put it there if there is room.
- If there is no room in the proper leaf, we “split” the leaf into two and divide the keys between the two new nodes, so each is half full or just over half full.
  - Split means “add a new block”
- The splitting of nodes at one level appears to the level above as if a new key-pointer pair needs to be inserted at that higher level.
  - We may thus apply this strategy to insert at the next level: if there is room, insert it; if not, split the parent node and continue up the tree.
- As an exception, if we try to insert into the root, and there is no room, then we split the root into two nodes and create a new root at the next higher level;
  - The new root has the two nodes resulting from the split as its children.

Deletion

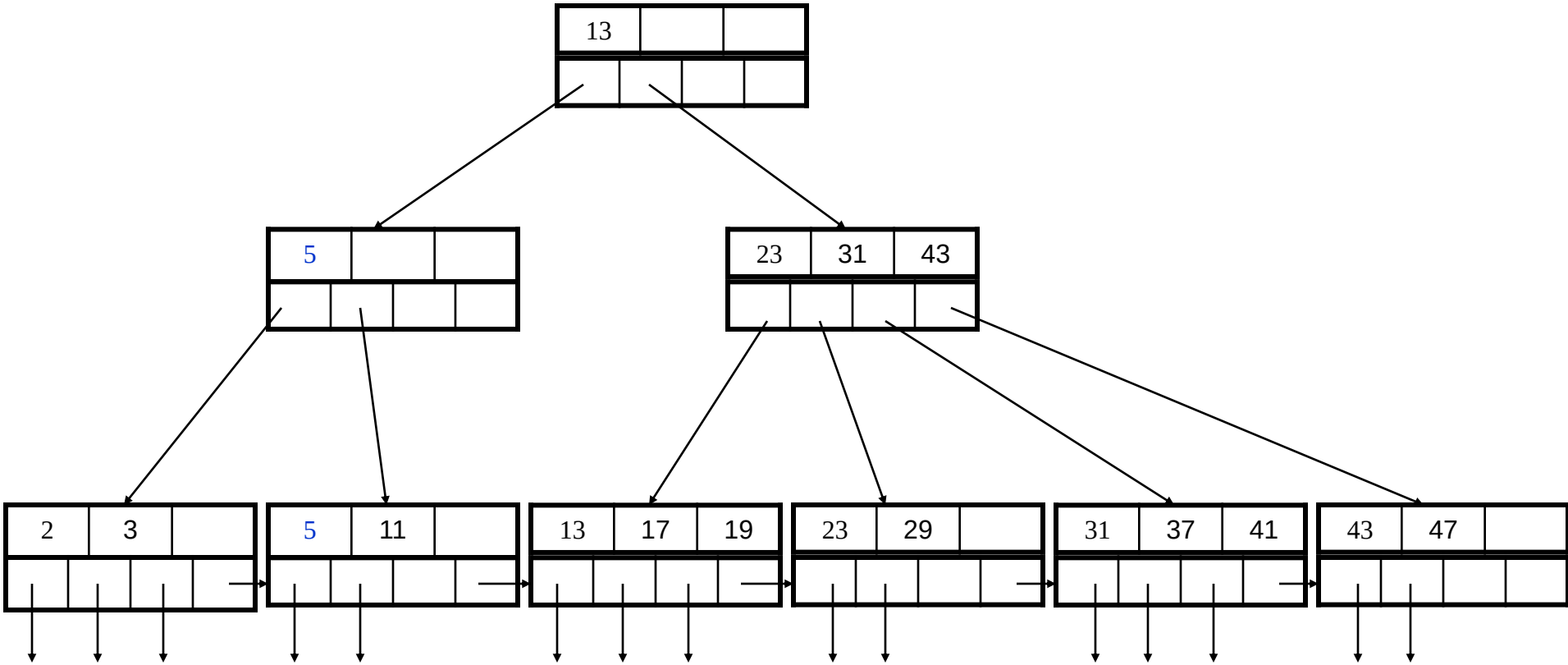
Suppose we delete key=7



Deletion


13
7
23
31
43
2
3
5
11
13
17
19
23
29
31
37
41
43
47
This leaf node is less than half full.

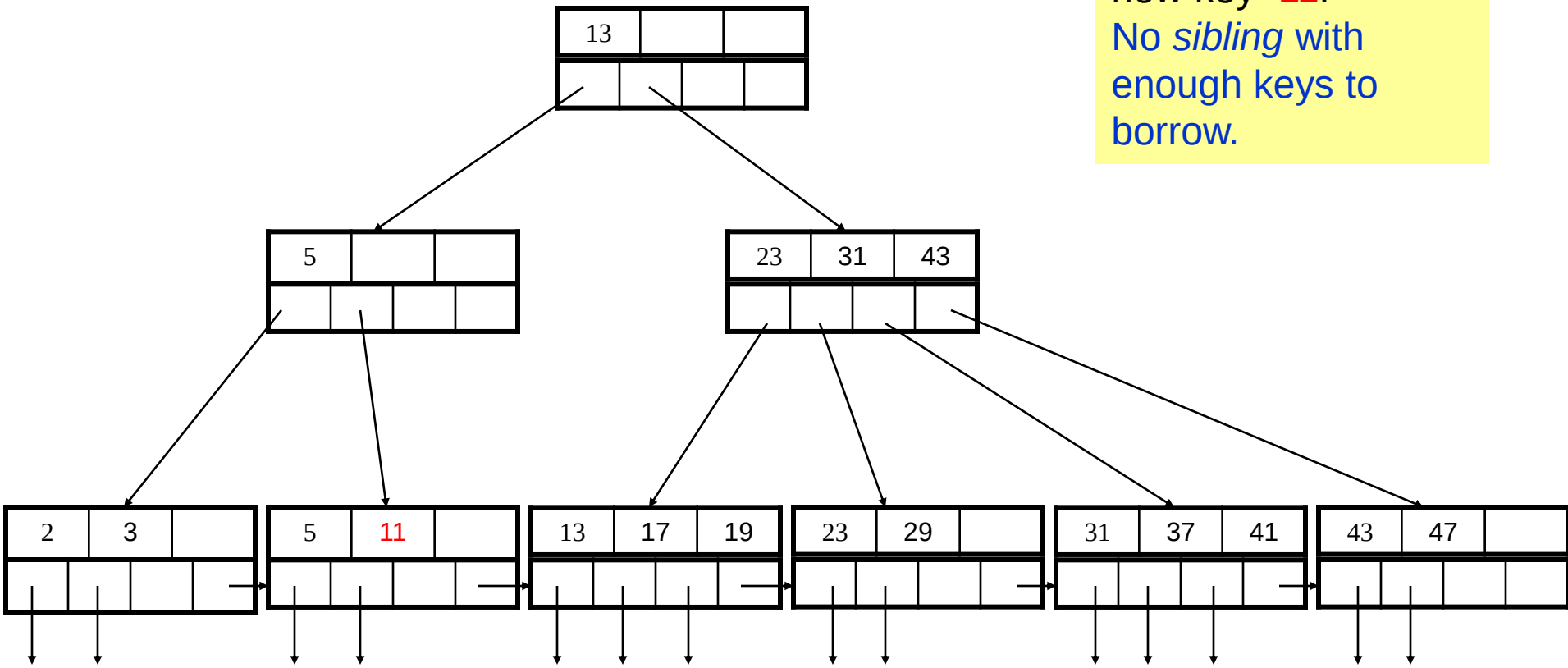
Deletion (Raising a key to parent)



This node is less than half full. So it borrows key 5 from the sibling, and updates parent node

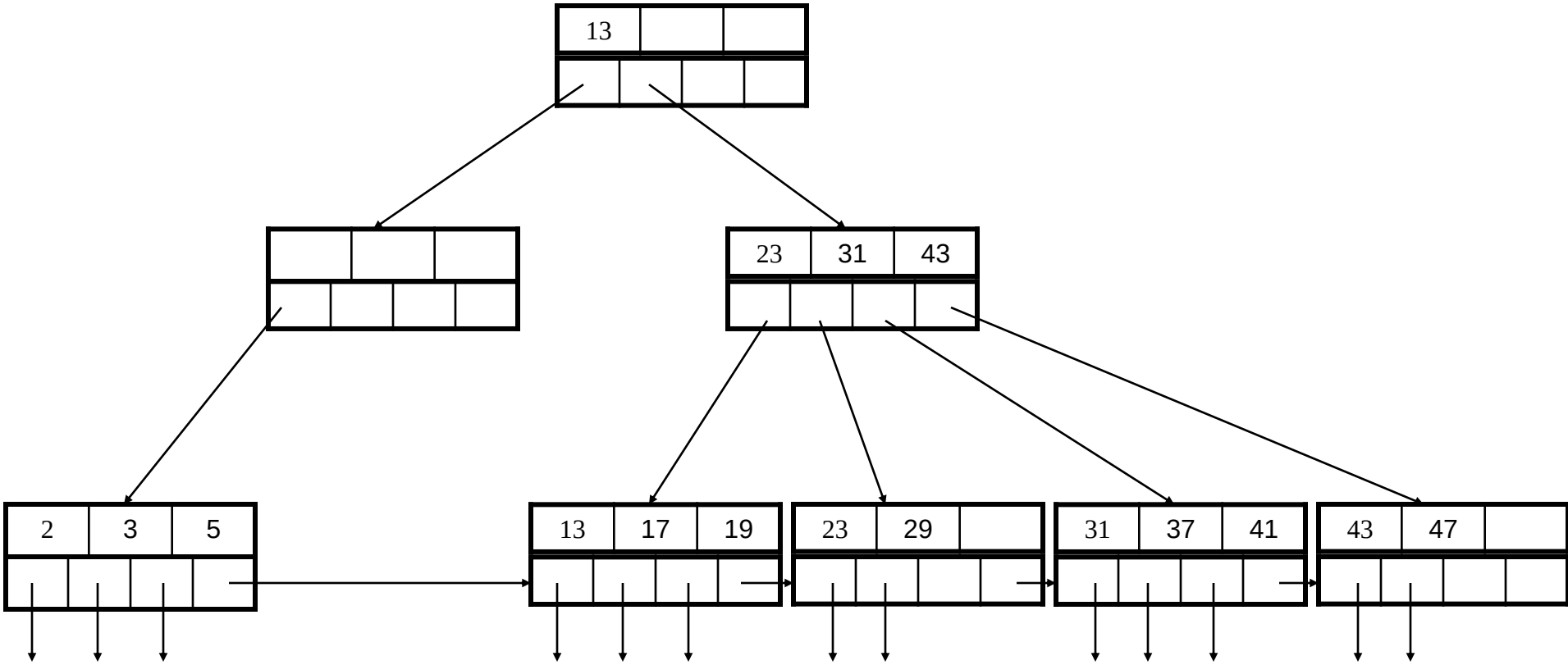
Deletion

Suppose we delete now key=11.
No *sibling* with enough keys to borrow.



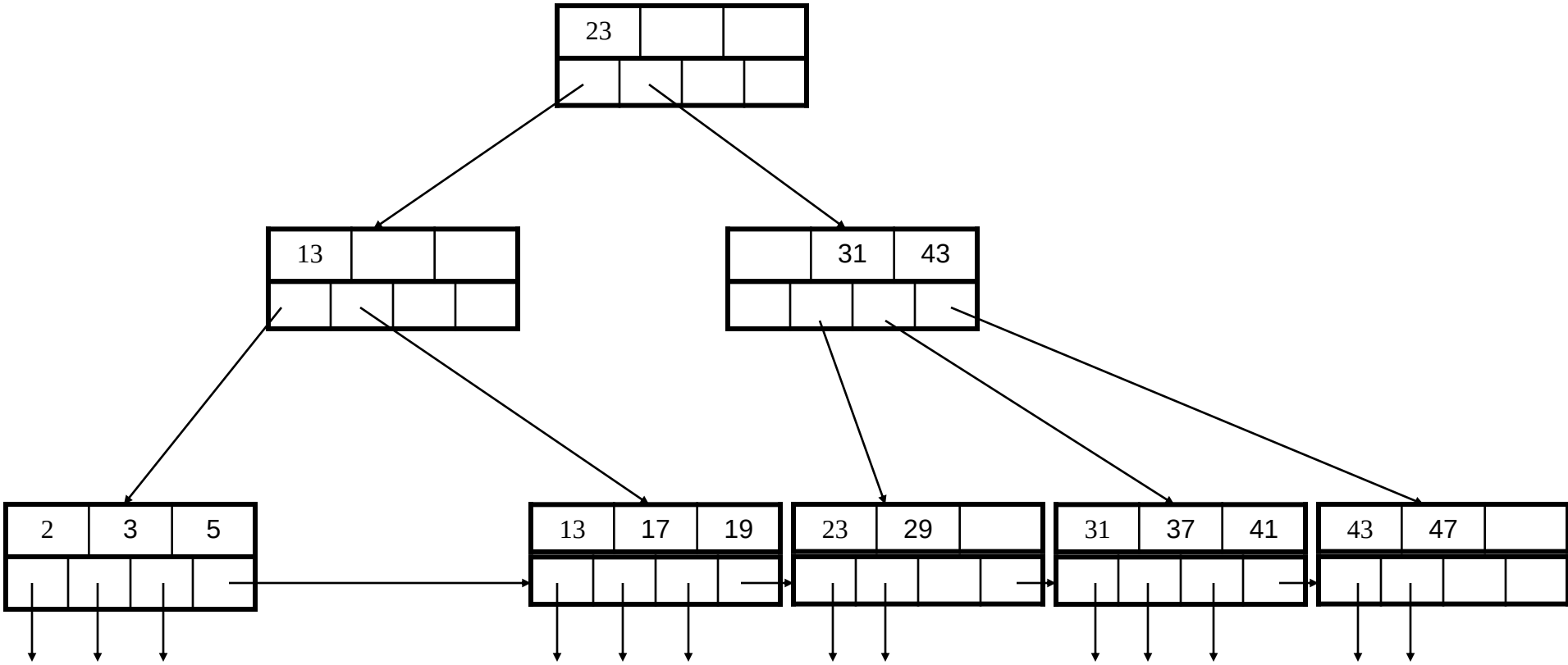
Note that node (13,17,29) is not a sibling – because it has a different parent

Deletion



We merge, i.e. delete a block from the index. However, the parent ends up having 1 pointer and zero keys

Deletion



Parent: Borrow pointer from sibling!

Deletion in words

- We find a place of the deleted key in the appropriate leaf, and remove the corresponding entry
- If the leaf node was at a minimum capacity before the deletion, it is now below minimum
  - If its most populous sibling contains more than d/2 children – borrow one and update parent pointer
  - Else if there are no nodes to borrow – merge current node with its sibling
- Update parent pointer. If there are less than d/2 children – borrow key from right sibling

Motivation for Indexes

Consider:

```
SELECT title
FROM Movies
WHERE studioName = 'Disney' AND year =1995;
```

- There might be 10,000 Movies tuples, of which only 200 were made in 1995.
  - Naive way to implement this query is to get all 10,000 tuples and test the condition of the WHERE clause on each.
  - Much more efficient if we had some way of getting only the 200 tuples from the year 1995 and testing each of them to see if the studio was Disney.
  - Even more efficient if we could obtain directly only the 10 or so tuples that satisfied both the conditions of the WHERE clause.

Pointer Intersection example

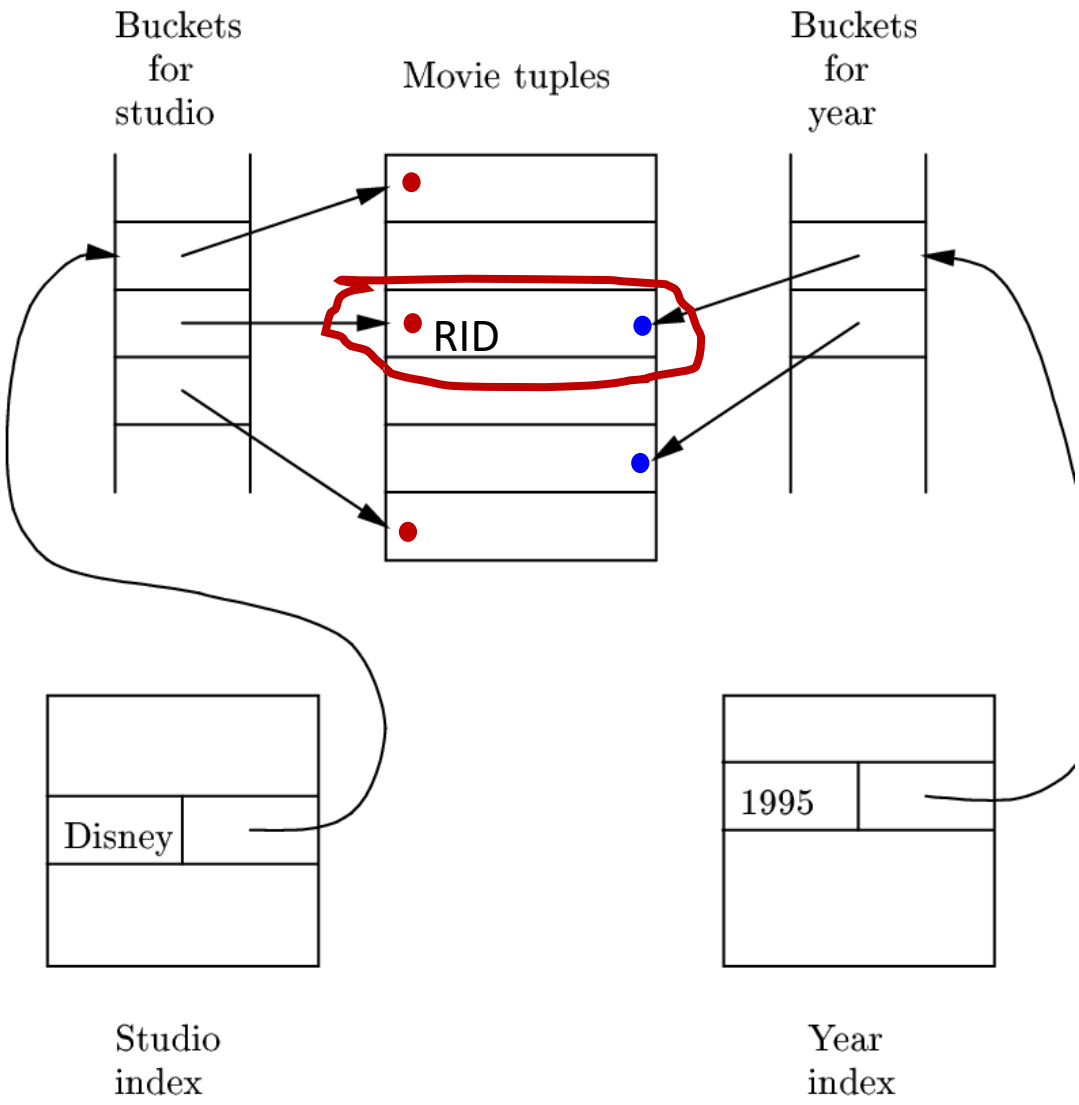
```
Movies(
  title,
  year,
  length,
  studioName);
```

Schema

Assume secondary indexes on **studioName** and **year**.

Query

```
SELECT title
FROM Movies
WHERE
  studioName='Disney'
  AND year = 1995;
```



Intersection of results from 2 indexes gives RID

Create Index

CREATE INDEX index_name ON table_name (column_name);

Created implicitly if you declare the column to be a PRIMARY KEY or UNIQUE.

Once an index exists, the user does not have to open or use it with a command, it is used automatically (by query optimizer)

All indexes are stored separately from the table on which they are based

Examples

1. CREATE INDEX YearIndex ON Movies(year);
2. CREATE INDEX KeyIndex ON Movies(title, year);
3. CREATE INDEX KeyIndex ON Movies (year, title);

When would it be beneficial to create the third vs. second?

Dropping an index:

DROP INDEX YearIndex;

Index on primary key

- Often, the most useful index we can put on a relation is an index on its key.
- Two reasons:
  - Queries in which a value for the key is specified are common.
  - Since there is at most one tuple with a given key value, the index returns either nothing or one location for a tuple.
    - Thus, at most one page of the relation must be retrieved to get that tuple into main memory

Example

```
SELECT name
FROM Movie, MovieExec
WHERE title = 'Star Wars'
   AND year='1994'
   AND producerC =cert;
```

Index on primary key: efficiency

Without Key Indexes

- Read each of the blocks of **Movies** and each of the blocks of **MovieExec** at least once.
  - In fact, since these blocks may be too numerous to fit in main memory at the same time, we may have to read each block from disk many times (to form a Cartesian product).

With Key Indexes

- Only two block reads.
  - Index on the key **(title, year)** for **Movies** helps us find the one Movie tuple for 'Star Wars' quickly.
    - Only one block - containing that tuple - is read from disk.
  - Then, after finding the producer-certificate number in that tuple, an index on the key **cert** for **MovieExec** helps us quickly find the one tuple for the producer in the MovieExec relation.
    - Only one block is read again.

Non-Beneficial Indexes

- When the index is not on a key, it may or may not be beneficial.

Example (of not being beneficial)

Suppose the only index we have on Movies is one on **year**, and we want to answer the query:

```
SELECT *
FROM Movie
WHERE year = 1990;
```

- Suppose the tuples of Movie are stored alphabetically by title.
- Then this query gains little from the index on year. If there are, say, 100 movies per page, there is a good chance that any given page has at least one movie made in 1990.

Beneficial Indexes

- There are two situations in which an index can be effective, even if it is not on a key.
    1. If the attribute is almost a key; that is, relatively few tuples have a given value for that attribute.

       Even if each of the tuples with a given value is on a different page, we shall not have to retrieve too many pages from disk.

Example

- Suppose Movies had an index on **title** rather than (title, year).

```
SELECT name
FROM Movie, MovieExec
WHERE title = 'King Kong' AND producerC =cert;
```

Beneficial Indexes (cont.)

2. If the tuples are "clustered" on the indexed attribute. We cluster a relation on an attribute by grouping the tuples with a common value for that attribute onto as few pages as possible.
   - Then, even if there are many tuples, we shall not have to retrieve nearly as many pages as there are tuples.

Example

- Suppose Movies had an index on **year** and tuples are clustered on year.

```
SELECT *
FROM Movie
WHERE year = 1990;
```

When to create indexes

Trade-off

- The existence of an index on an attribute may speed up greatly the execution of those queries in which a value, or range of values, is specified for that attribute, and may speed up joins involving that attribute as well.

- On the other hand, every index built for one or more attributes of some relation makes insertions, deletions, and updates to that relation more complex and time-consuming.

Cost Model

1. Tuples of a relation are stored in many pages (blocks) of a disk.
2. One block, which is typically several thousand bytes at least (e.g. 16K), will hold many tuples.
3. To examine even one tuple requires that the whole block be brought into main memory.

The *cost* of a query is dominated by **the number of block accesses**. Main memory accesses can be neglected.

Introduction to selection of indexes

StarsIn(movieTitle, movie Year , starName)

Q1:

```
SELECT movieTitle, movieYear
FROM StarsIn
WHERE starName = s;
```

Q2:

```
SELECT starName
FROM StarsIn
WHERE movieTitle = t AND movieYear= y;
```

I:

```
INSERT INTO StarsIn VALUES(t, y, s);
```

Before we begin: answer

1. How many pages for StarsIn: 10 pages
   If we need to examine the entire relation the cost is 10.

2. Average number of movies per star: on the average, a star has appeared in 3 movies

3. Average number of stars per movie: a movie has 3 stars on average

Analysis

1. Since the tuples for a given star or a given movie are likely to be spread over the 10 pages of **StarsIn**, even if we have an index on starName or on the combination of movie title and movieYear, it will take 3 disk accesses to retrieve the 3 tuples for a star or movie. If we have no index on the star or movie, respectively, then 10 disk accesses are required.

2. One disk access is needed to read a page of the index every time we use that index to locate tuples with a given value for the indexed attribute(s). If an index page must be modified (in the case of an insertion), then another disk access is needed to write back the modified page.

3. Likewise, in the case of an insertion, one disk access is needed to read a page on which the new tuple will be placed, and another disk access is needed to write back this page. We assume that, even without an index, we can find some page on which an additional tuple will fit, without scanning the entire relation.

Average cost for NO INDEX

Action	NO INDEX
Q1	10
Q2	10
Ins	2
Average	$10p_1 + 10p_2 + 2(1 - p_1 – p_2) = 2 + 8p_1 + 8p_2$

Q1: SELECT movieTitle, movieYear FROM StarsIn WHERE starName = s;
Q2: SELECT starName FROM StarsIn WHERE movieTitle = t AND movieYear= y;
I: INSERT INTO StarsIn VALUES(t, y, s);

p_1	is the fraction of times Q1 is executed
p_2	is the fraction of times Q2 is executed
$1-p_1-p_2$	is the fraction of times I is executed

Average cost for Star INDEX

Action	NO INDEX	INDEX on Star attribute of StarsIn
Q1	10	4 (1 to read an index, 3 to load corresponding blocks
Q2	10	10
Ins	2	4 (read/write 1 page for the index and for the data)
Average	$2 + 8p_1 + 8p_2$	$4p_1 + 10p_2 + 4(1 - p_1 - p_2) = 4 + 6p_2$

Q1: SELECT movieTitle, movieYear FROM StarsIn WHERE starName = s;
Q2: SELECT starName FROM StarsIn WHERE movieTitle = t AND movieYear= y;
I: INSERT INTO StarsIn VALUES(t, y, s);

p_1	is the fraction of times Q1 is executed
p_2	is the fraction of times Q2 is executed
$1-p_1-p_2$	is the fraction of times I is executed

Average cost for Movie INDEX

Action	NO INDEX	Star INDEX	INDEX on Movies attributes only
Q1	10	4	10
Q2	10	10	4
Ins	2	4	4
Average	$2 + 8p_1 + 8p_2$	$4 + 6p_2$	$4 + 6p_1$

Q1: SELECT movieTitle, movieYear FROM StarsIn WHERE starName = s;
Q2: SELECT starName FROM StarsIn WHERE movieTitle = t AND movieYear= y;
I: INSERT INTO StarsIn VALUES(t, y, s);

p_1	is the fraction of times Q1 is executed
p_2	is the fraction of times Q2 is executed
$1-p_1-p_2$	is the fraction of times I is executed

Average cost for both indexes

Action	NO INDEX	Star INDEX	Movies INDEX	INDEX on Movies attributes AND Star
Q1	10	4	10	4 (1 page to read an index, 3 to read data)
Q2	10	10	4	4 (1 page to read an index, 3 to read data)
Ins	2	4	4	6 (read/write 1 page for each index and 1 for data)
Average	$2 + 8p_1 + 8p_2$	$4 + 6p_2$	$4 + 6p_1$	$4p_1 + 4p_2 + 6(1 - p_1 - p_2) = 6 - 2p_1 - 2p_2$

Q1: SELECT movieTitle, movieYear FROM StarsIn WHERE starName = s;
Q2: SELECT starName FROM StarsIn WHERE movieTitle = t AND movieYear= y;
I: INSERT INTO StarsIn VALUES(t, y, s);

p_1	is the fraction of times Q1 is executed
p_2	is the fraction of times Q2 is executed
$1-p_1-p_2$	is the fraction of times I is executed

What solution is the best?

Action	NO INDEX	Star INDEX	Movies INDEX	INDEX on both
Q1	10	4	10	4
Q2	10	10	4	4
Ins	2	4	4	6
Average	$2 + 8p_1 + 8p_2$	$4 + 6p_2$	$4 + 6p_1$	$6 - 2p_1 - 2p_2$

If p1 = p2 = 0.1:
If p1 = p2 = 0.4:
If p1 = 0.5, p2 = 0.1:
If p1 = 0.1, p2 = 0.5:

p_1	is the fraction of times Q1 is executed
p_2	is the fraction of times Q2 is executed
$1-p_1-p_2$	is the fraction of times I is executed

Reasoning

- If $p_1 = p_2 = 0.1$, then the expression $2 + 8p_1 + 8p_2$ is the smallest, so we would prefer not to create any indexes.

- If $p_1 = p_2 = 0.4$, then the formula $6 - 2p_1 - 2p_2$ turns out to be the smallest, so we would prefer indexes on both starName and on the (movieTitle, movieYear) combination.

- If $p_1 = 0.5$ and $p_2 = 0.1$, then an index on stars-only gives the best average value, because $4 + 6p_2$ is the formula with the smallest value.

- If $p_1 = 0.1$ and $p_2 = 0.5$, then create an index on movies-only.

Quiz 1 (at home)

- First, build B-tree from records with sorted keys:

2,3,5,7,11,13,17,19,23,29,31,37,41,43,47

- Insert key 1
- Insert keys 14,15,16
- Delete key 23
- Delete all keys >23 (in turn)

Quiz 2 (at home)

- Suppose that the relation StarsIn requires now 100 pages rather than 10, but all other assumptions of this example continue to hold. Give formulas in terms of p1 and p2 to measure the cost of queries Q1 and Q2 and insertion I under the four combinations of index/non index discussed in the example.

- Compare with the results in the example for:

p1 = p2 = 0.1

p1 = p2 = 0.4

p1 = 0.5, p2 = 0.1

p1 = 0.1, p2 = 0.5