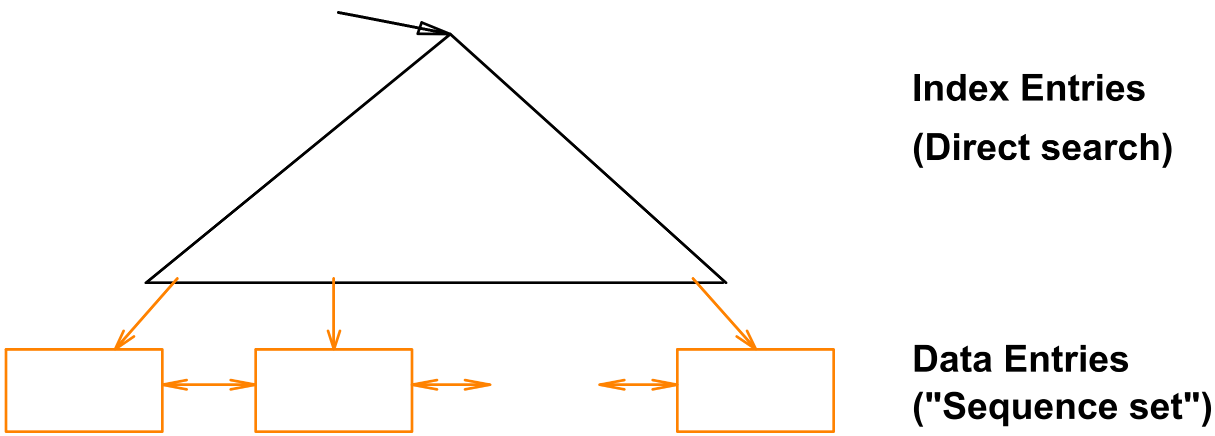
# B+ Review

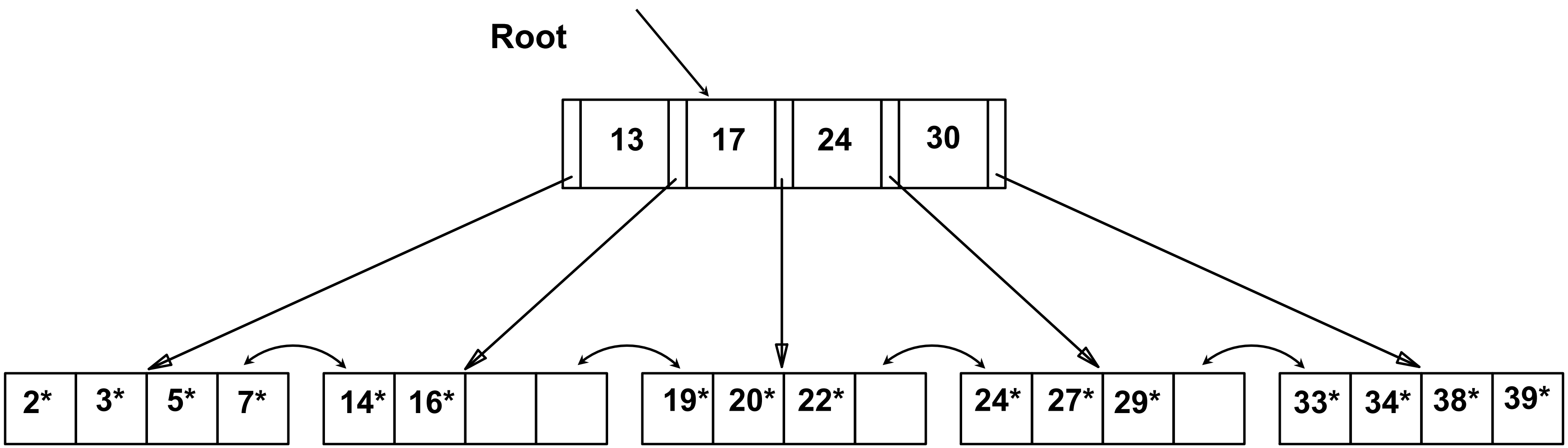
# B+ Tree: Most Widely Used Index

- Insert/delete at $\log_F N$ cost; keep tree *height-balanced*. (F = fanout, N = # leaf pages)
- Minimum 50% occupancy (except for root). Each node contains **d** <= $\underline{m}$ <= 2**d** entries. The parameter **d** is called the *order* of the tree.
- Supports equality and range-searches efficiently.



# Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- Search for 5*, 15*, all data entries >= 24* ...



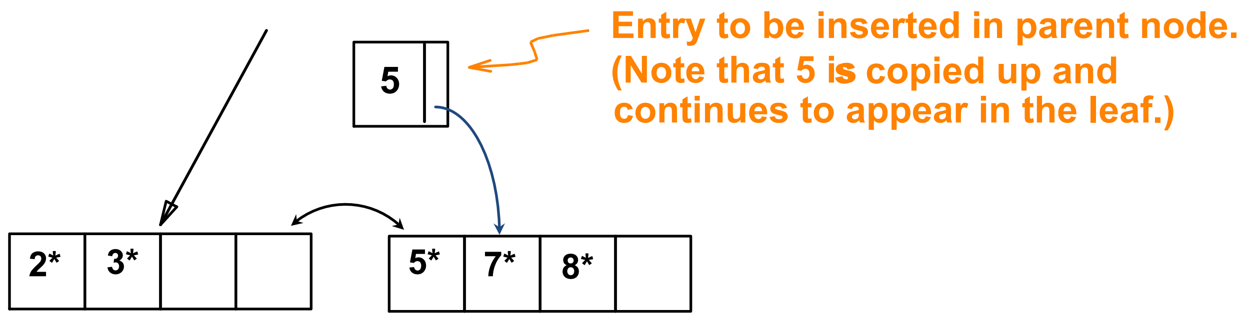

*Based on the search for 15*, we know it is not in the tree!*

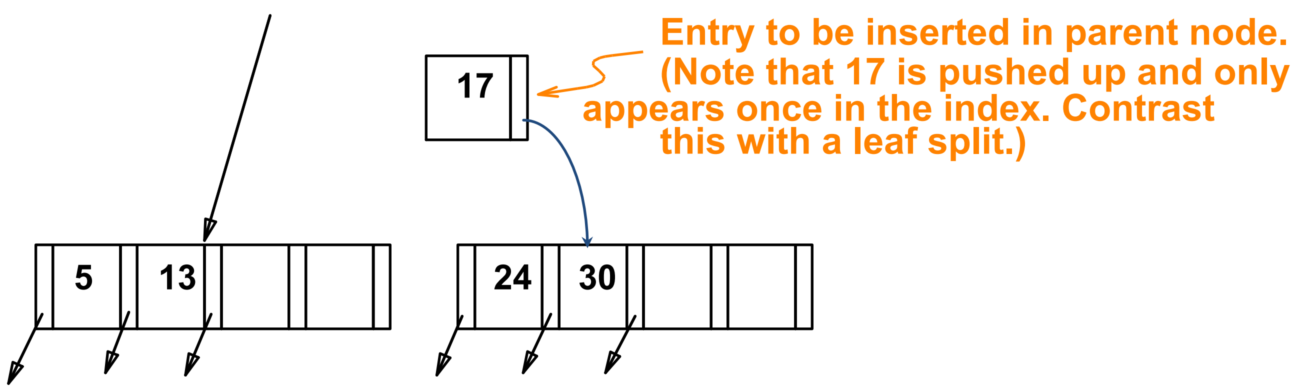
# Inserting a Data Entry into a B+ Tree

- Find correct leaf *L.*
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

# Inserting 8* into Example B+ Tree

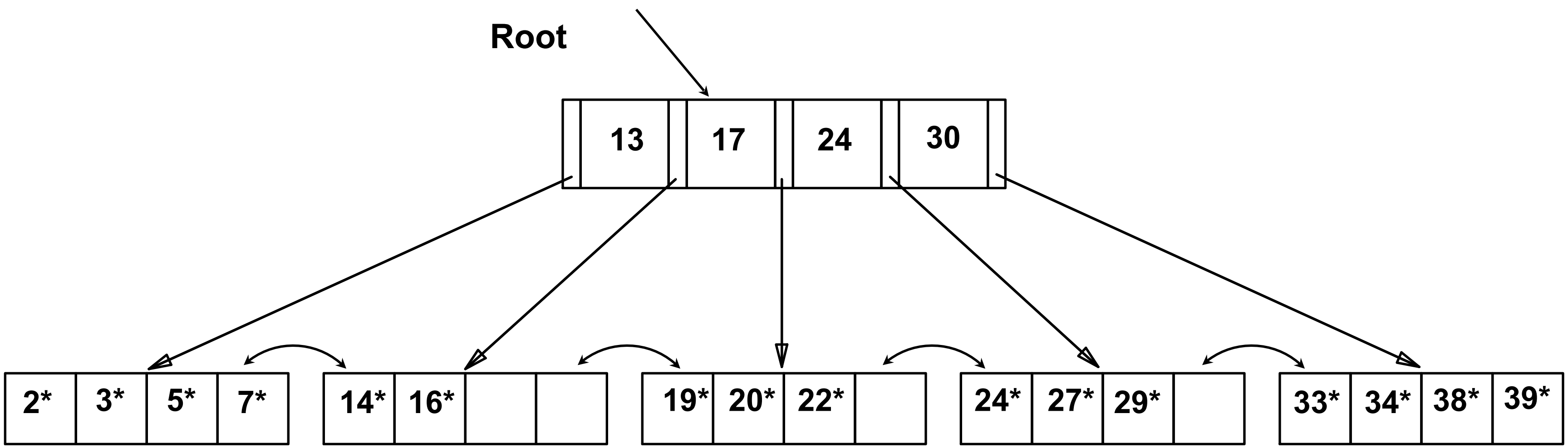
- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.
- Note difference between *copy-up* and *push-up*; be sure you understand the reasons for this.



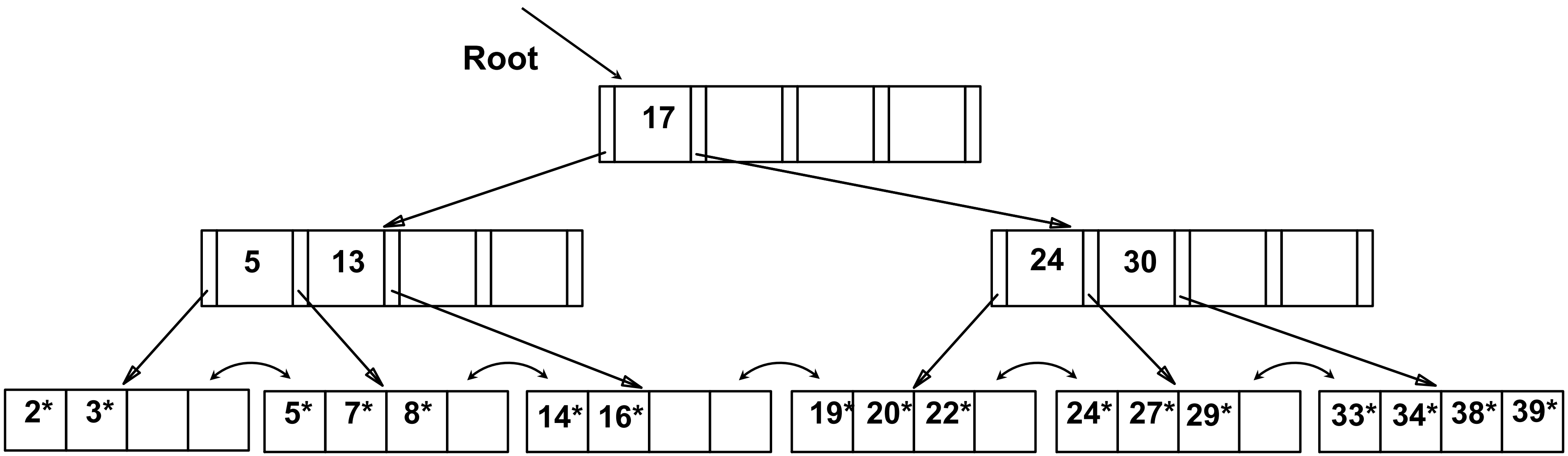
# Example B+ Tree

- We're going to insert 8.




? *Based on the search for 15*, we know it is not in the tree!*

# Example B+ Tree After Inserting 8*




❖ Notice that root was split, leading to increase in height.

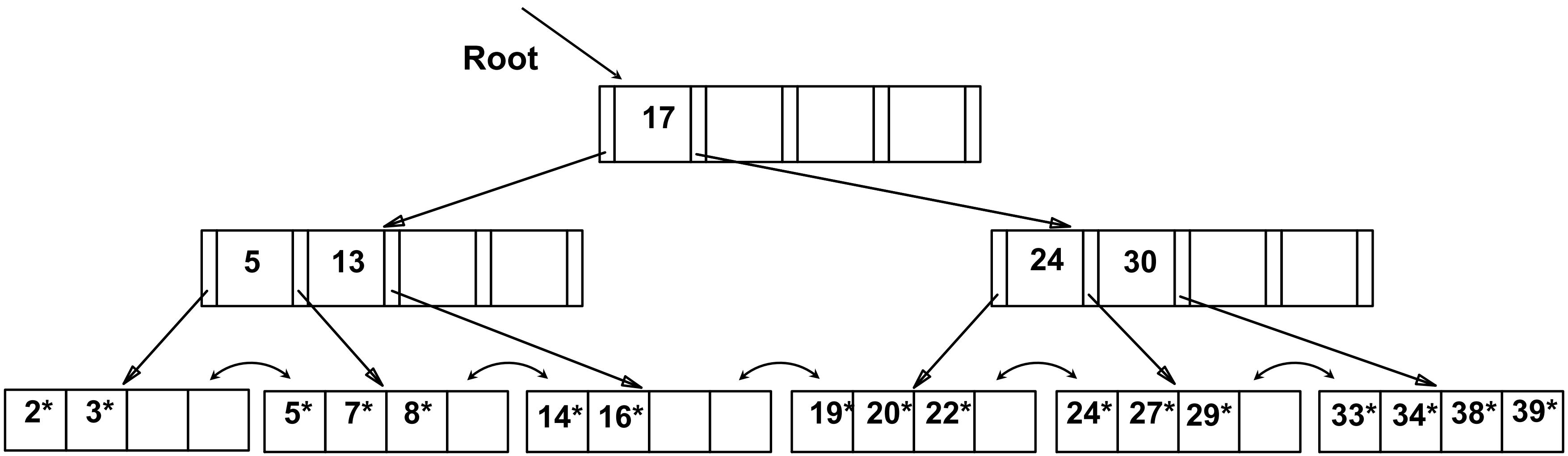
❖ In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

# Deleting a Data Entry from a B+ Tree

- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to re-distribute, borrowing from *sibling* *(adjacent node with same parent as L)*.
    - If re-distribution fails, *merge* *L* and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

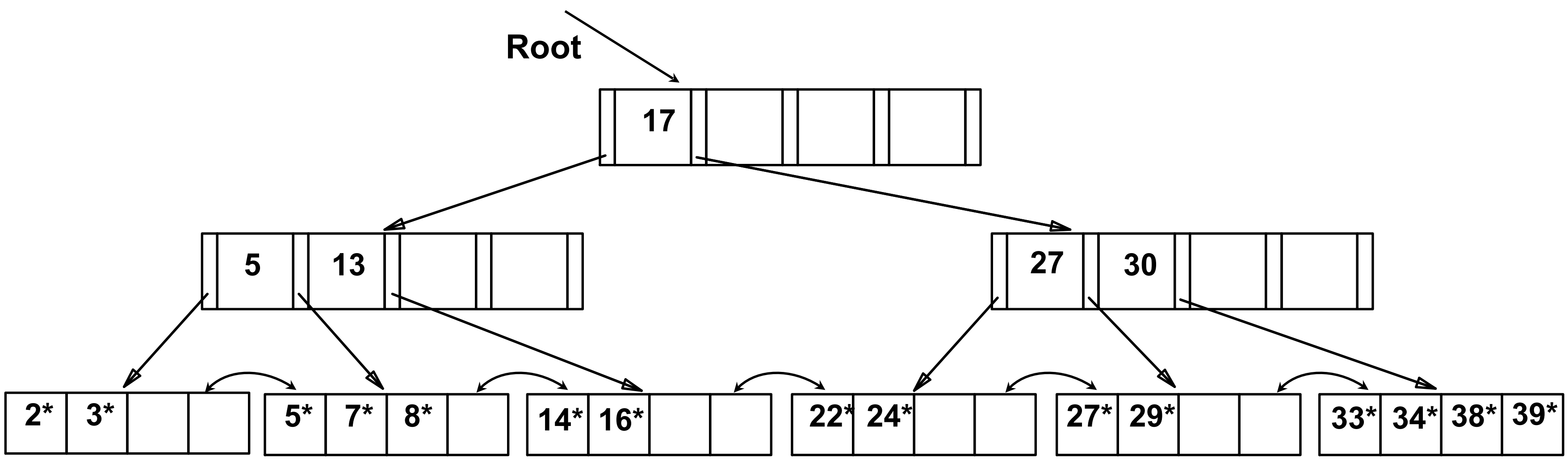
# Delete

# Example B+ Tree After Inserting 8*




- ❖ We're going to delete 19 and 20

# Example Tree After (Inserting 8*, Then) Deleting 19* and 20* ...

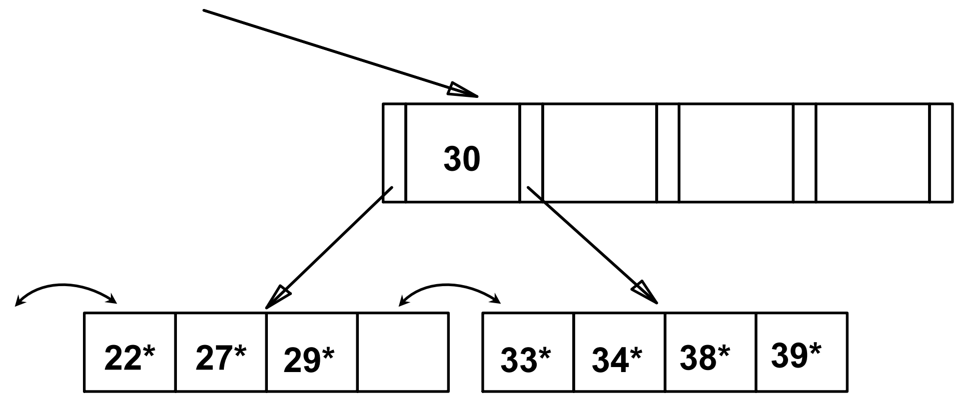


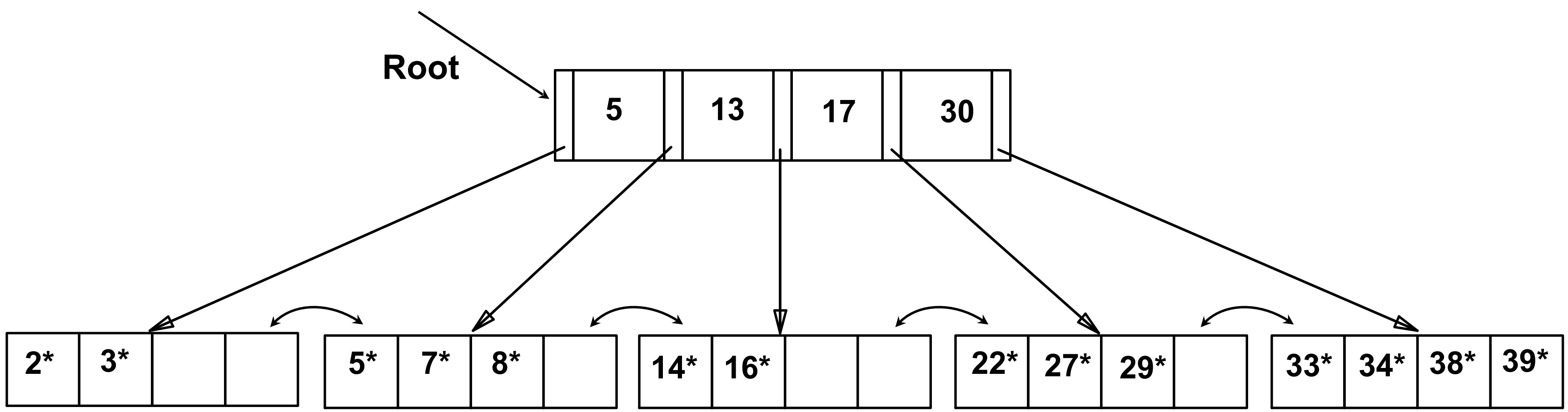

- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

Next, we delete 24

# ... And Then Deleting 24*

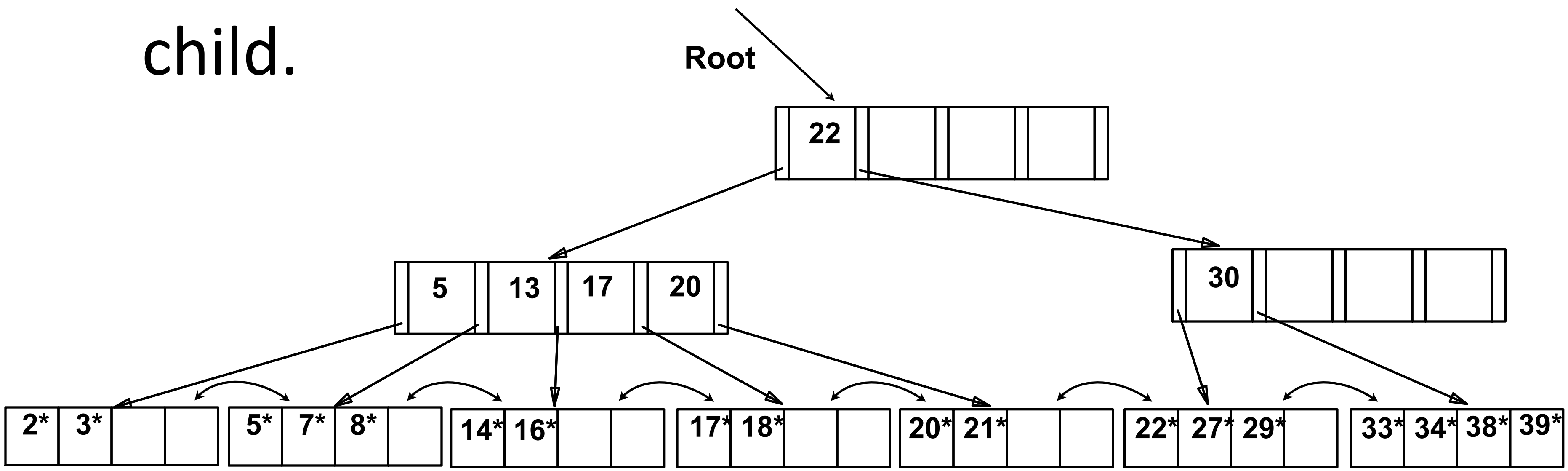
- Must merge.
- Observe *`toss'* of index entry (on right), and *`pull down'* of index entry (below).



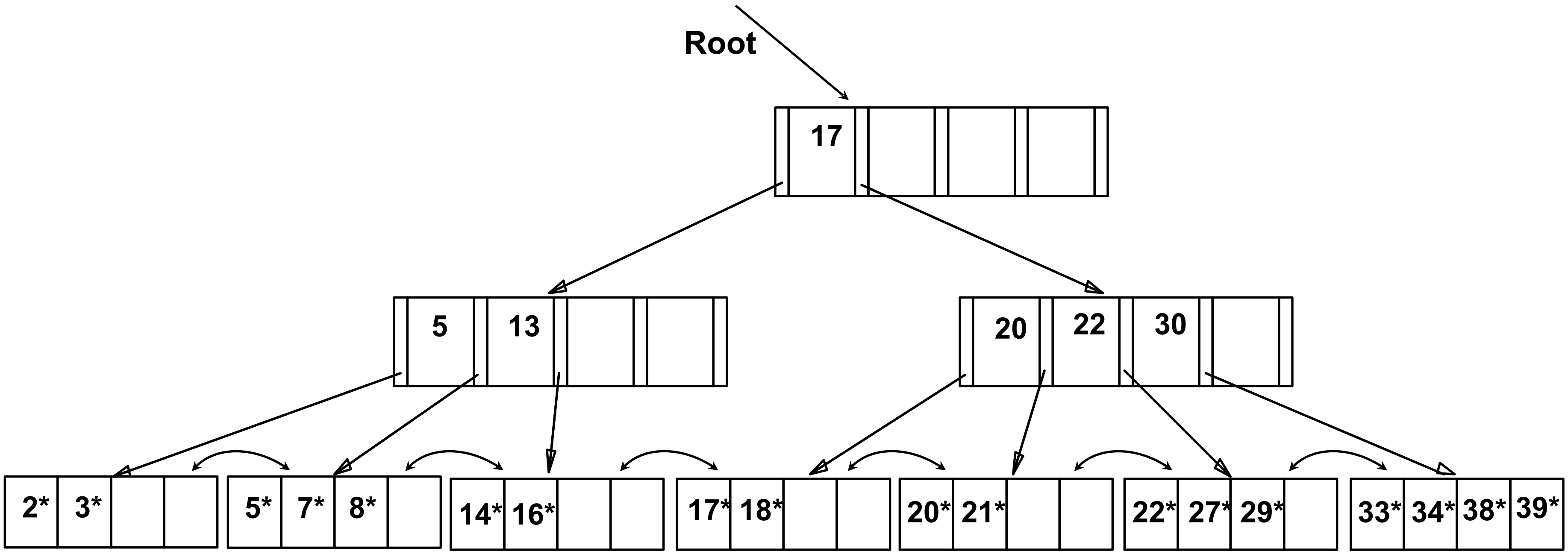
# Example of Non-leaf Re-distribution

- Tree is shown below *during deletion* of 24*. (What could be a possible initial tree?)
- In contrast to previous example, can re-distribute entry from left child of root to right child.



# After Re-distribution

- Entries are re-distributed by *`pushing through'* the splitting entry in the parent node.
- It suffices to re-distribute index entry with key 20; we've re-distributed 17 as well

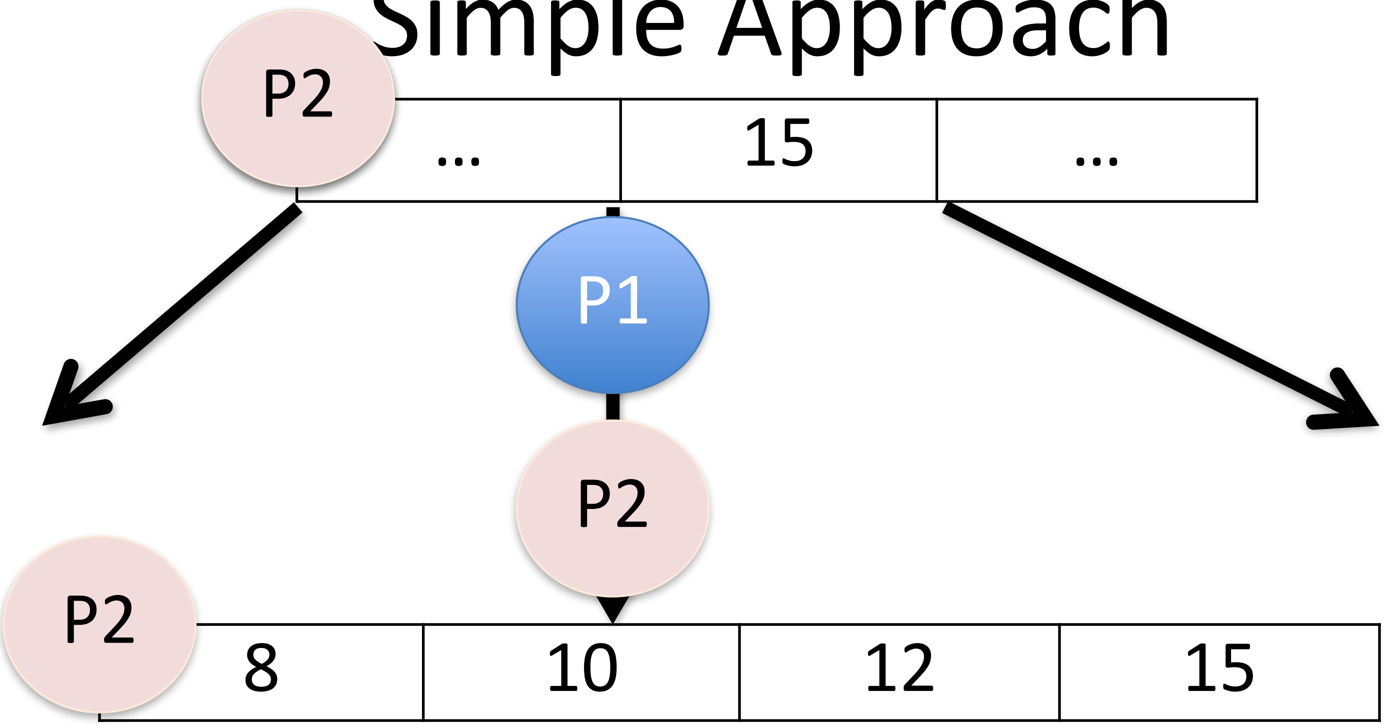


# B+ Concurrency

# Model

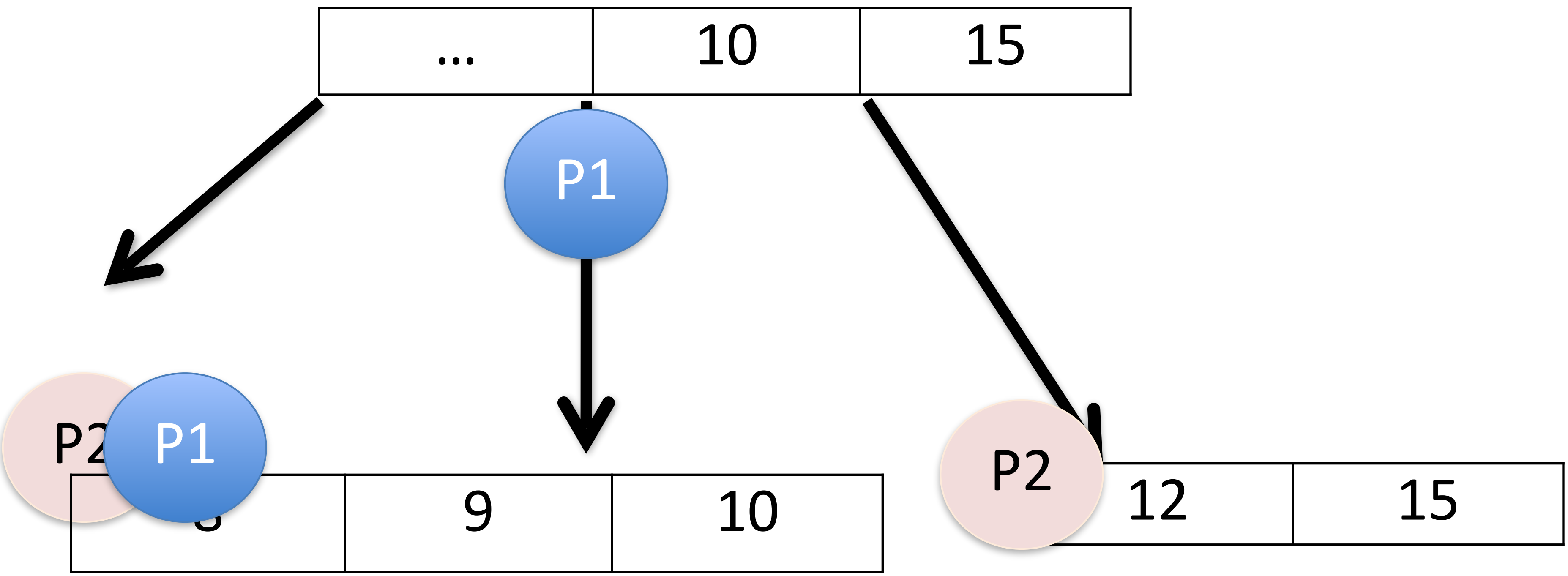
- We consider page lock(x)/unlock(x) of pages (only for writes!)
- We copy into our memory and then atomically update pages.

# Simple Approach




- P1 searches for 15
- P2 inserts 9

# After the Insertion




- P1 searches for 15
- P2 inserts 9

P1 Finds no 15!

How could we fix this?

# B-Link Trees

# Two important Conventions

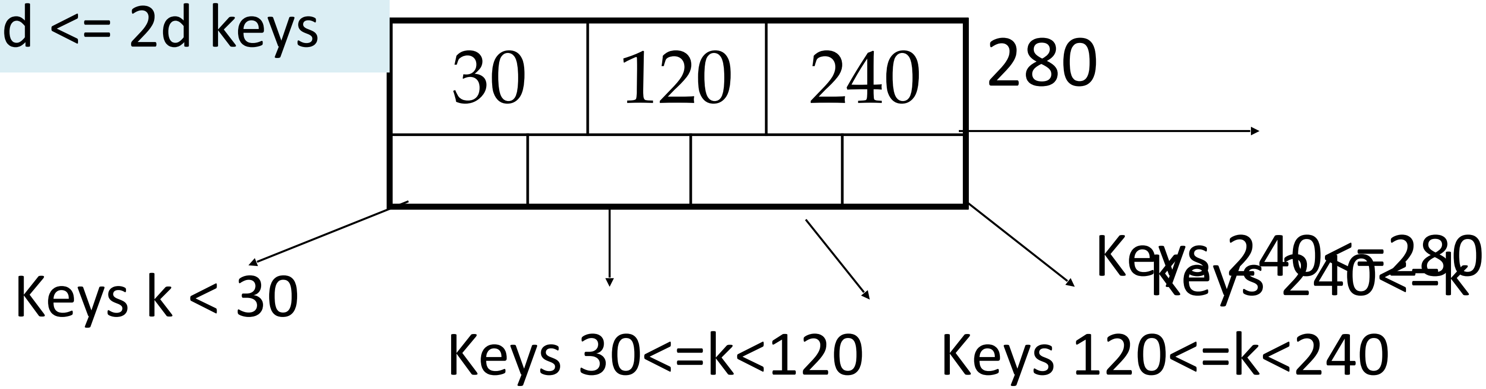
- Search for B-link trees root to leaf, left-to-right in nodes

- Insertions for B-link trees proceed bottom-up.

# Internal Nodes

- Parameter d = the <u>*degree*</u>

Internal Node has
s >= d and <= 2d keys




Add right pointers.

We add a High key

Idea: If we get to this page, looking for 300. What can we conclude happened?

# Valid Trees & Safe Nodes

- A node may not have a parent node, but it must have a left twin.
- We introduce the right links before the parent.
- A node is safe if it has [k,2k-1] pointers.

# Scannode

**scannode**(u, A) : examine the tree node in A for value u and return the appropriate pointer from A.

Appropriate pointer may be the right pointer.

# Searching for v

```
current = root;
A = get(current);
while (current is not a leaf) {
      current = scannode(v, A);
      A = get(current);}
while ((t = scannode(v,A)) == link pointer of A) {
      current = t;
      A = get(current);}
Return (v is in A) ? success :  failure;
```

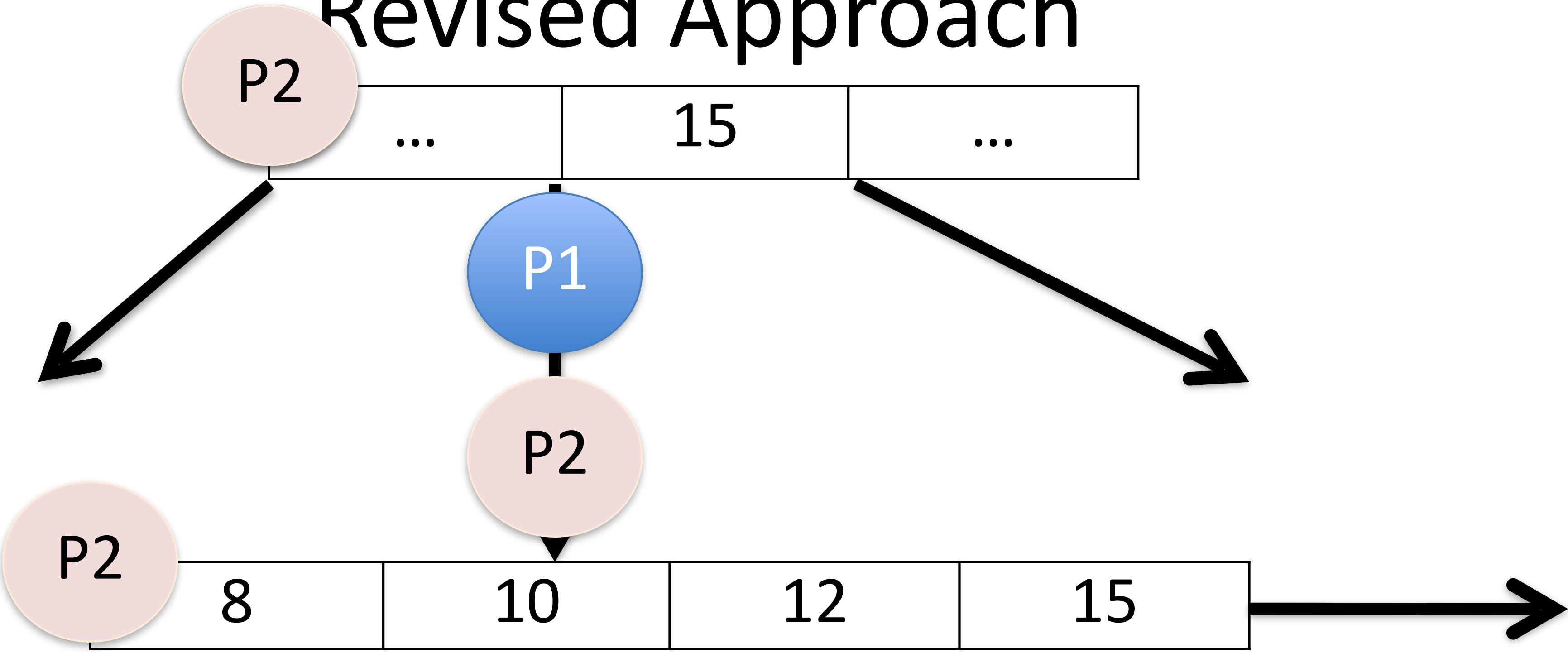
Find the leaf w/ v

Find the leaf w/ v

Only modify scannode – No locking?!?

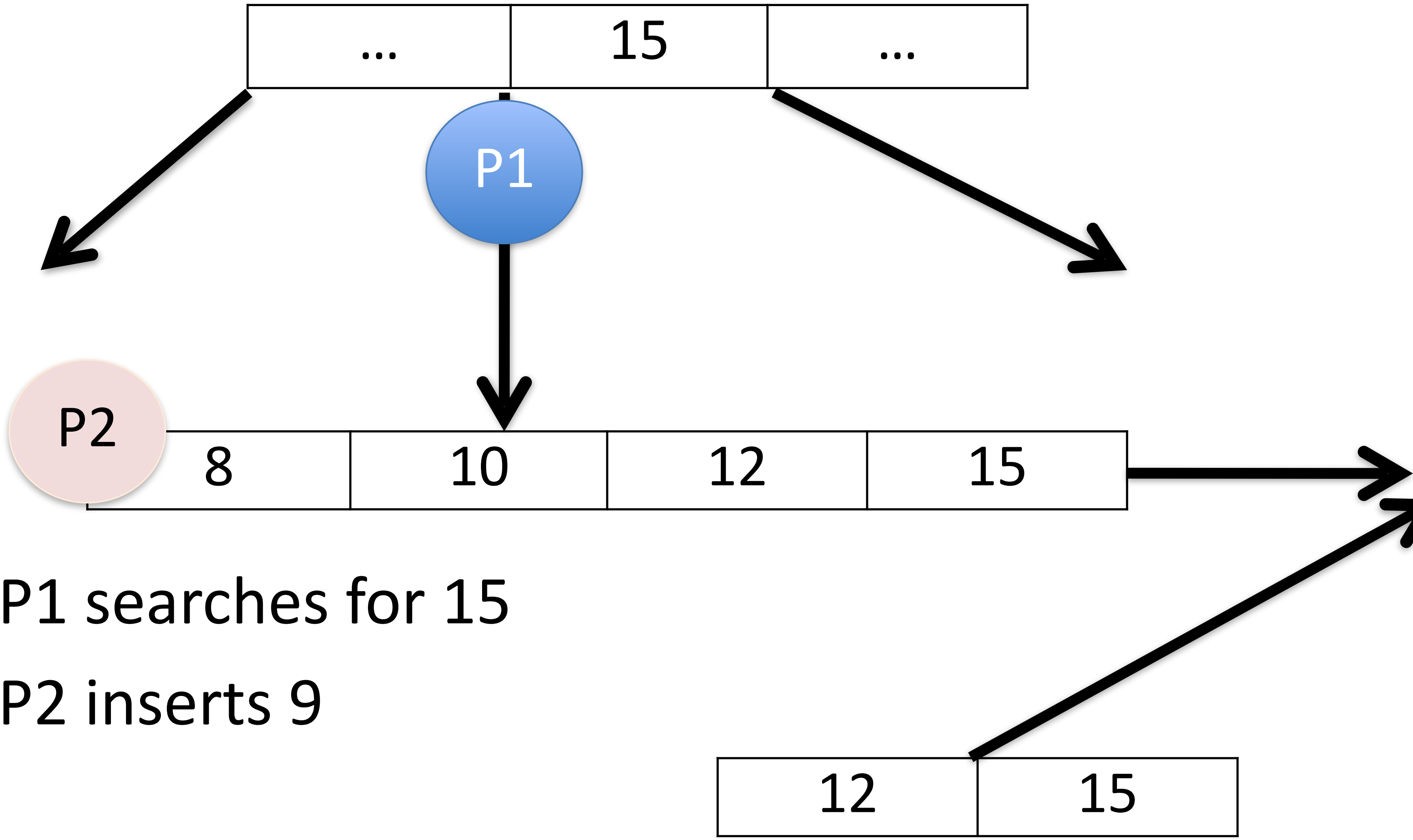
# Insert

# Revised Approach



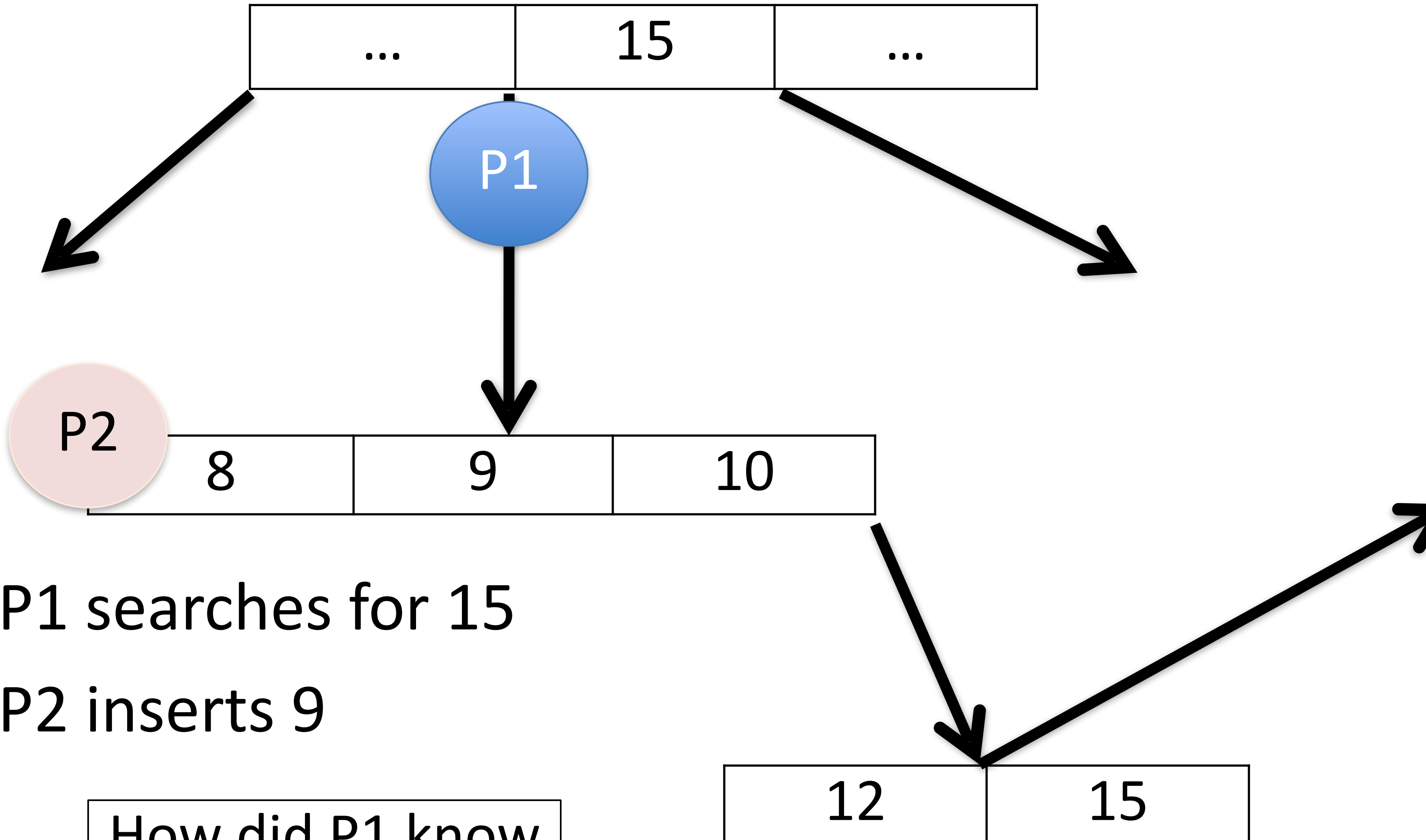

- P1 searches for 15
- P2 inserts 9

# Revised Approach: Build new page




- P1 searches for 15
- P2 inserts 9

# Revised Approach: Build new page




- P1 searches for 15
- P2 inserts 9

How did P1 know to continue?

# Start Insert

Keep a stack of the rightmost node we visited at each level:

```
initialize stack; current = root;
A = get(current);
while (current is not a leaf) {
      t = current;
      current = scannode(v,A);
      if (current not link pointer in A)
            push t;
      A = get(current);}
```

# A subroutine: move_right

```
While t = scannode(v,A) is a link pointer of A do
    Lock(t)
    Unlock(current)
    Current = t
    A = get(current);
end
```

How many locks held here?

The move_right procedure scans right across the leaves with lock coupling.

# Easy case:

**DoInsert:**

```
if A is safe {
        insert new key/ptr pair on A;
        put(A, current);
        unlock(current);
}
```

# Fun Case: Must split

u = allocate(1 new page for B);

redistribute A over A and B ;

y = max value on A now;

make high key of B equal old high key of A;

make right-link of B equal old right-link of A;

make high key of A equal y;

make right-link of A point to B;

# Insert

put (B, u);

put (A, current);

oldnode = current;

new key/ptr pair = (y, u); // high key of new page, new page

current = pop(stack);

lock(current); A = get(current);

move_right();

unlock(oldnode)

goto Doinsertion;

*may have 3 locks: oldnode, and two at the parent level while moving right*

# Deadlock Free

# Total Order < on Nodes

Consider pages a,b define a total order <

1. a < b if b is closer to the root than a (different height)
2. If a and b are at the same height, then a < b if b is reachable.

"Order is bottom-up"

Observation: Insert process only puts down locks satisfying this order. Why is this true?

# Deadlock Free

Since the locks are placed by every process in a total order, there can be no deadlock. Why?

Is it possible to get the cycle:
T1(A) T2(B) T1(B) T2(A)?

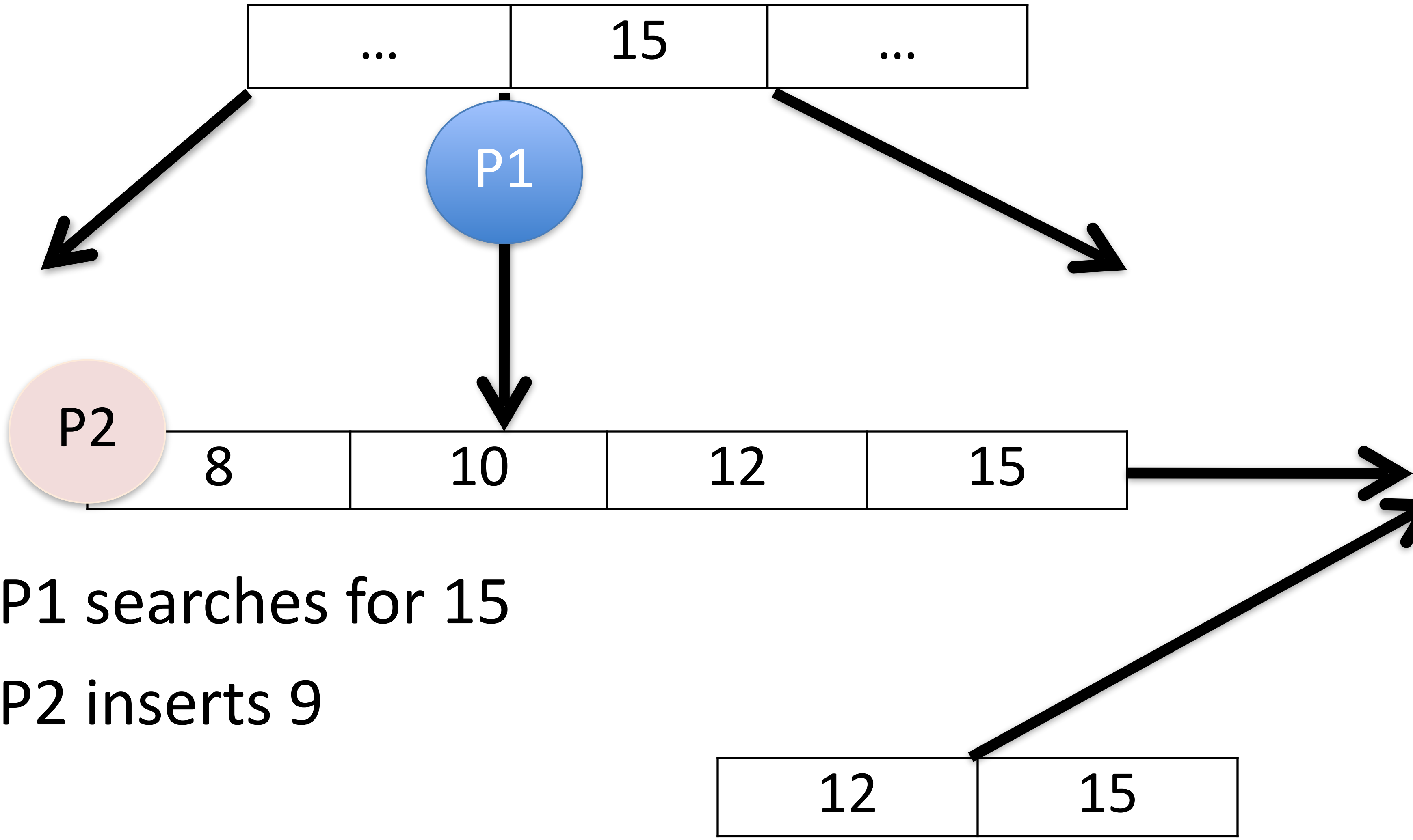
# Tree Modification

# Tree Modifications

Thm: All operations correctly modify the tree structure.

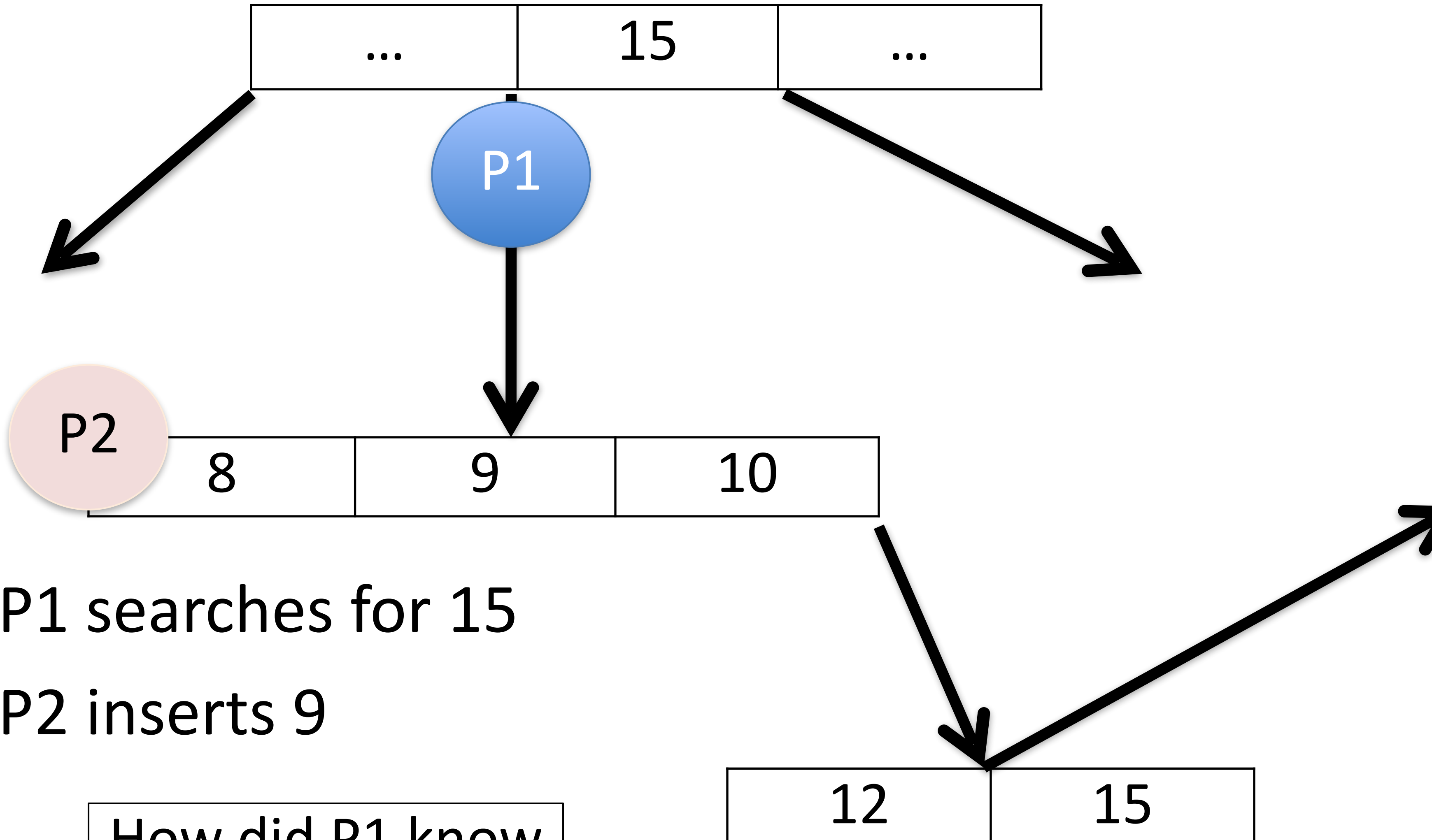
Observation 1: put(B,u) and put(A, current) are one operation (since put(B,u) doesn't change tree. Proof by pictures (again).

# Revised Approach: Build new page




- P1 searches for 15
- P2 inserts 9

# Revised Approach: Build new page




- P1 searches for 15
- P2 inserts 9

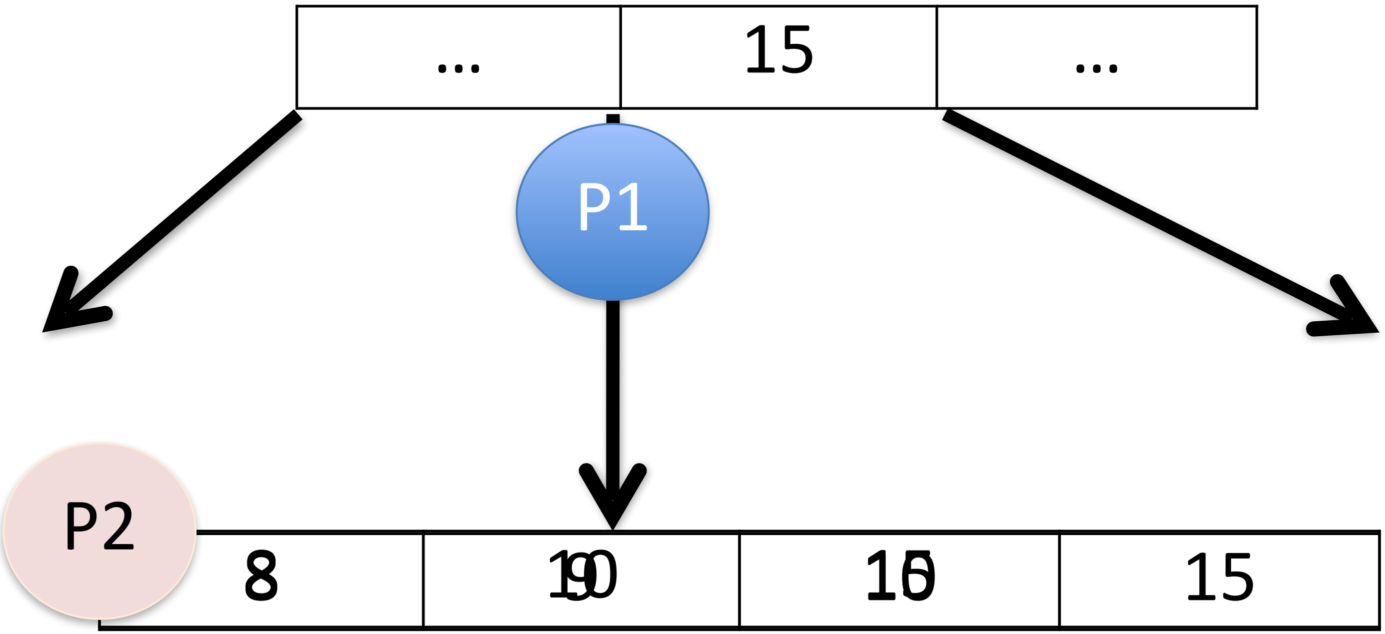
How did P1 know to continue?

# Correct Interaction of Readers and Writers

# Correct Interaction

Thm: Actions of an insertion process do not impair the correctness of the actions of other processes.

# Type 1: No split



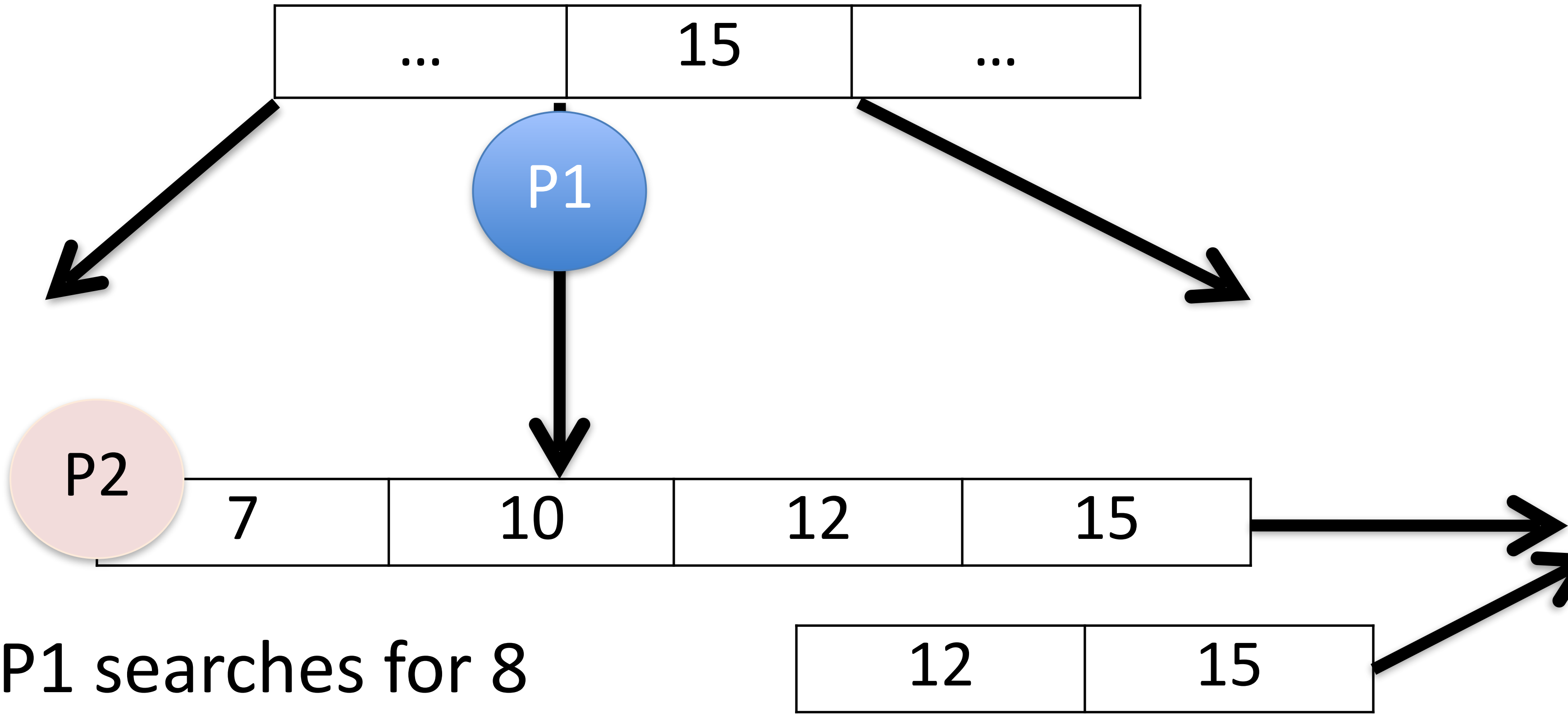

- P1 searches for 15
- P2 inserts 9

P2 reads the page.
What schedule is this?
Why can't P1,P2 conflict again?

What if P2 reads after P1?

# Type 2: Split. insert into left Node

# Type 2: Split. Insert LHS.



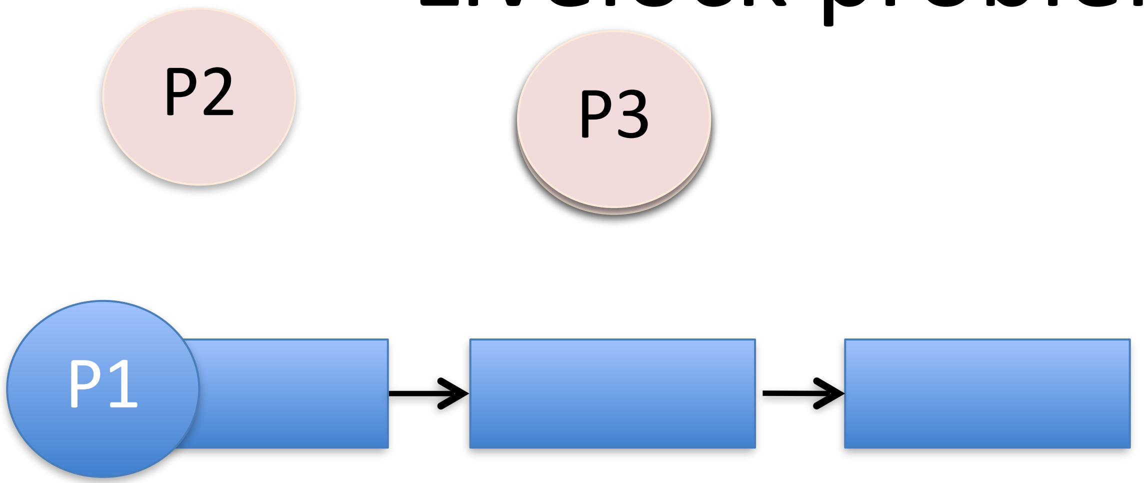

- P1 searches for 8
- P2 inserts 9

Notice that P1 would have followed 9s pointer!

How will P1 find 8?

# Livelock

# Livelock problem




Poor P1 never gets its value!
P1 is livelocked!

# Chaining Example

# Can we get down below 3 locks?

Consider the Alternative Protocol
(without lock coupling)

read A;

find out that there is room;

lock and re-read A;

find there is still room, and insert 9

unlock A;

Large # of inserts. A splits and after there is room!

What prevents this in Blink?

| | | |
|---|---|---|
| 5 | 6 | |

A

| | | |
|---|---|---|
| 12 | 15 | |

# Further Reading

- Recent HP Tech Report is great source (Graefe)

http://www.hpl.hp.com/techreports/2010/HPL-2010-9.pdf

- Extensions: R-trees and GiST


Marcel Kornacker, Douglas Banks: High-Concurrency Locking in R-Trees. VLDB 1995: 134-145

Marcel Kornacker, C. Mohan, Joseph M. Hellerstein: Concurrency and Recovery in Generalized Search Trees. SIGMOD Conference 1997: 62-72