CS2040S

Recitation 4
AY21/22S2

(a,b)-trees

(a,b)-tree overview

- Is an generalisation of Binary Search Trees ideas
- 2 parameters: a and b where $2 \leqslant a \leqslant (b+1)/2$
- Can store **more than one value per node** to **divide the range of its subtree's values** into more than two subranges

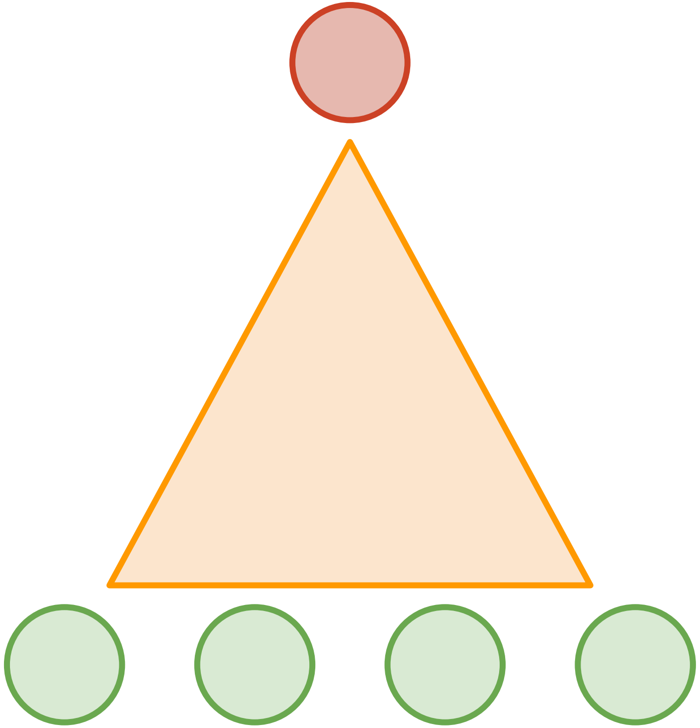
(a,b)-tree overview

BST	(a,b)-tree
Each node has *at most 2* children	Each node *can have more than 2* children
Each node stores *exactly 1* key	Each node *can store multiple* keys

(a,b)-tree rules

1. "(a,b)-child policy"
2. "Key ranges"
3. "Leaf depth"

Basic nomenclature



Root node

Internal nodes

Leaf nodes

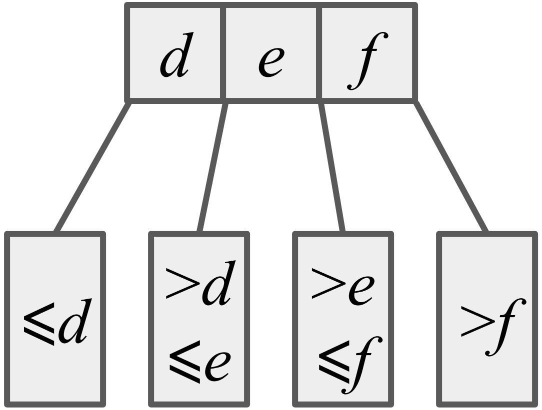
Rule 1 — "(a,b)-child policy"

Node:	**Root**	**Internal**	**Leaf**
#Keys	$[1, b-1]$	$[a-1, b-1]$	$[a-1, b-1]$
#Children	$[2, b]$	$[a, b]$	N.A

See next rule for why bounds on number of children is always 1 more than number of keys

Rule 2—"Key ranges"

- **Internal nodes** has one more child than its number of keys
- This ensures all value ranges defined by its keys are covered in its subtrees
- For instance, an internal node with 3 **ascending ordered** keys $[d, e, f]$ will entail 4 subranges
  1. $\leqslant d$
  2. $> d$ and $\leqslant e$
  3. $> e$ and $\leqslant f$
  4. $> f$



Working vocabulary

To have a working vocabulary, for any node z, we shall hereby refer to

- The range of keys covered in subtree rooted at z as *key range*
- The list of of keys within z as *keylist*
- The list of z's children (i.e. key ranges due to its keylist) as *treelist*

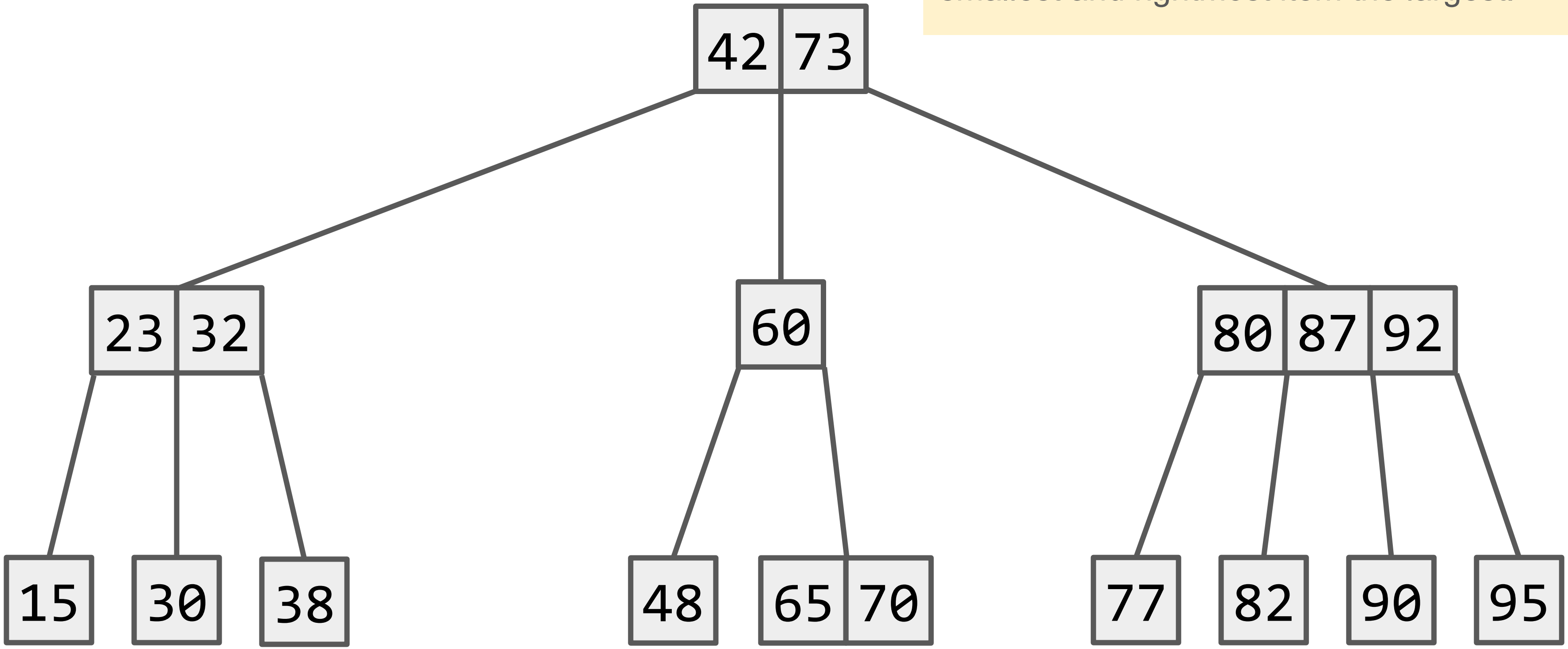
In addition, we shall always treat the keylist in ascending **sorted order**.

Rule 3 — "Leaf depth"

Leaves must be at the same depth (from the root)

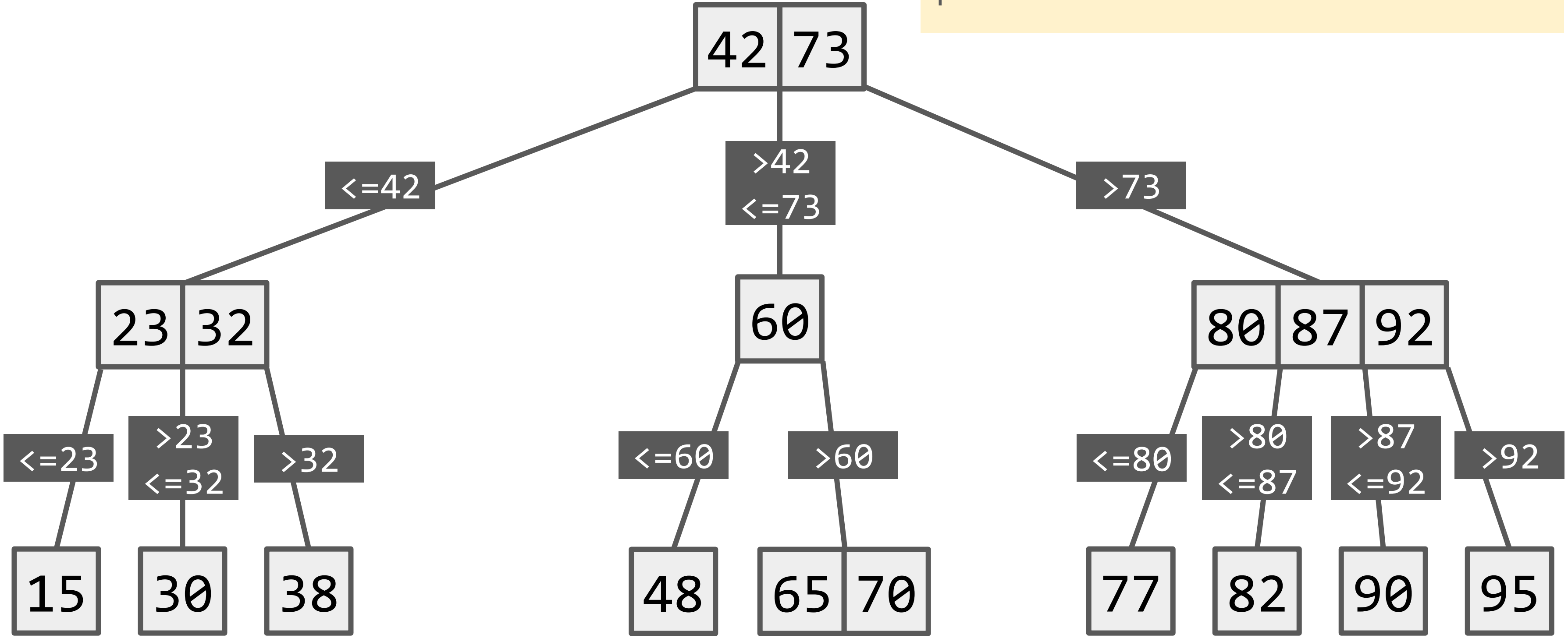
Example: (2,4)-tree

In our diagrams, all the keylists are in sorted order with leftmost item being the smallest and rightmost item the largest.



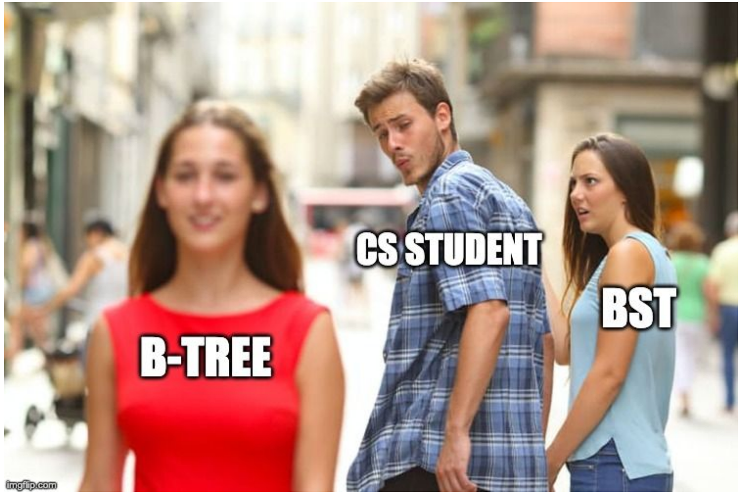
Example: $(2,4)$-tree

Explicitly denoting the key range of each subtree by labelling the edge from its parent.



B-trees

- Are simply a subcategory of (a,b)-trees
- When $a = \mathrm{B}$ and $b = 2\mathrm{B}$
- B is a number we specify
  - E.g. if $\mathrm{B} = 2$, we have a $(2,4)$-tree
- B **does not** stand for "Binary" (what does it stand for?)



History

B-trees were invented by Rudolf Bayer and Edward M.McCreight while working at Boeing Research Labs, for the purpose of efficiently managing index pages for large random access files. ...

Bayer and McCreight never explained what, if anything, the B stands for: Boeing, balanced, broad, bushy, and Bayer have been suggested. McCreight has said that **"the more you think about what the B in B-trees means, the better you understand B-trees."**

Source: Wikipedia

Did you know?

- B-trees are one of the most important data structures out there today
- Variants of B-trees used in all major databases
- Very fast! Not just asymptotic analysis, but in practice nearly impossible to beat a well-implemented B-tree
- Benefit comes both from good cache performance, low overhead, good parallelization, etc.

Problem 1.a.

Is a $(2,4)$-tree balanced?

If it is, which rule(s) ensures that?

If not, why?

Problem 1.a.ー Solution

For a binary tree, it is balanced *iff* for every node, its left and right subtree has height that differ by at most 1. Let's generalize this definition for (a,b)-trees.

Realize rule 3 ensures that the tree is balanced. If all leaves are at the same depth from the root, then it also means the child subtrees of every node share the same height.

Problem 1.b.

What is the minimum and maximum height of an (a,b)-tree with n **keys**?

Problem 1.b.

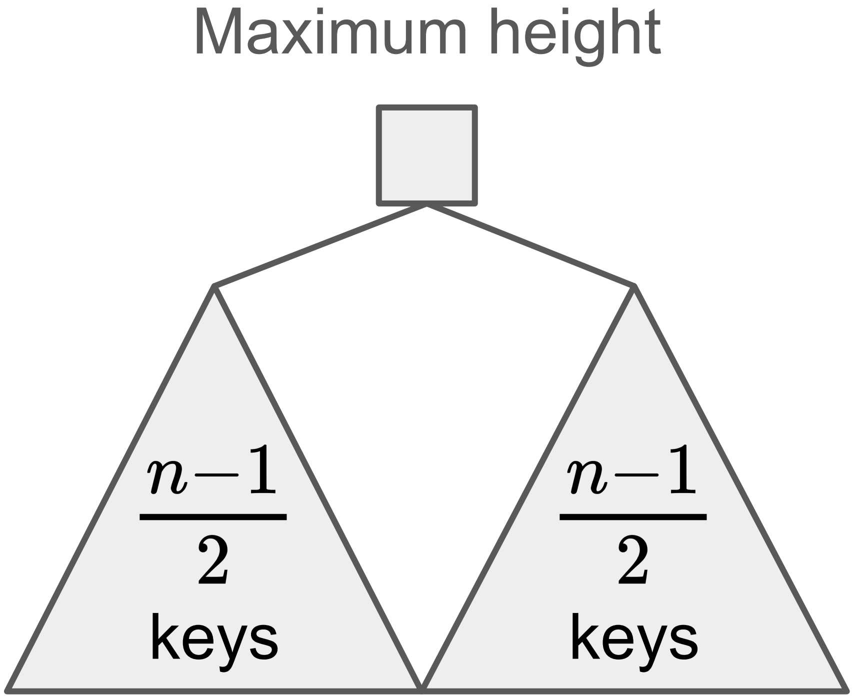
Maximum height when,

- Minimum branching factor of internal nodes (a children each)
- Equivalently, each non-root node has $a-1$ keys
- Root node has 1 key and 2 children

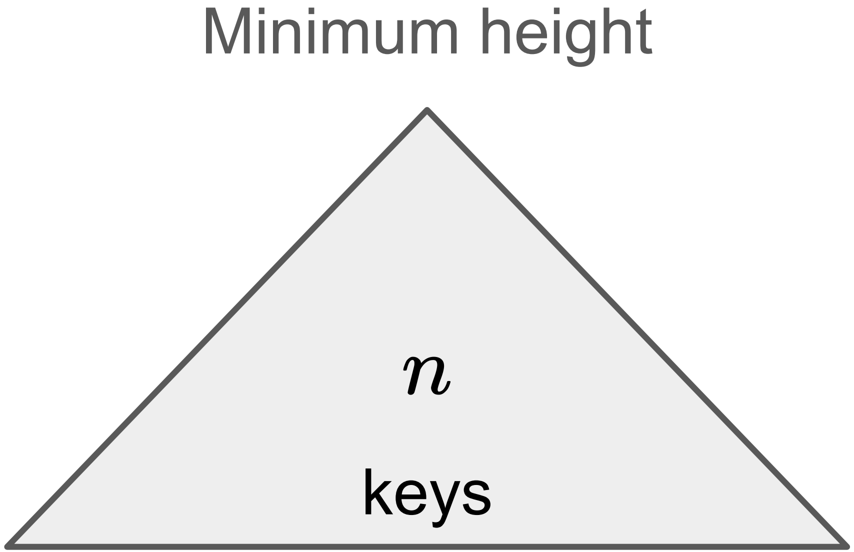
Minimum height when,

- Maximum branching factor of internal nodes (b children each)
- Equivalently, each non-root node has $b-1$ keys

Problem 1.b.



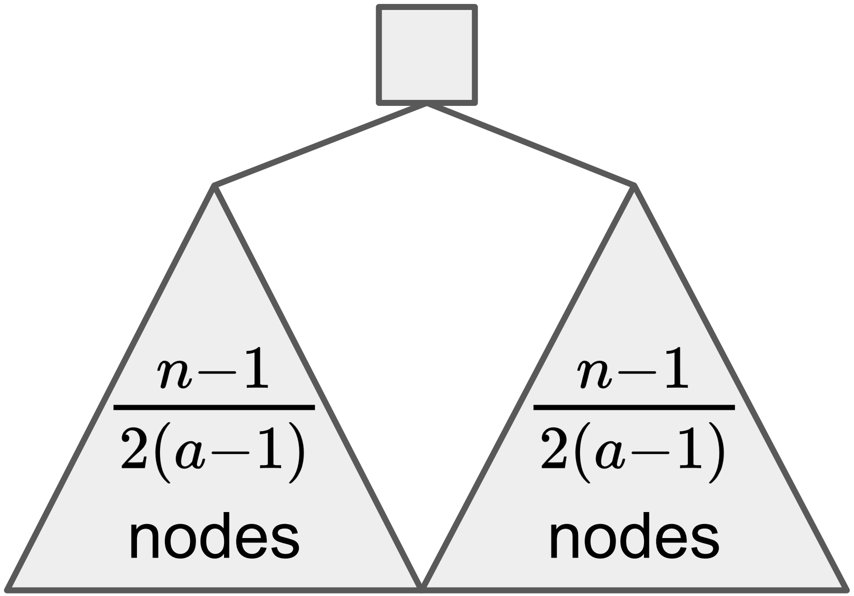
Maximum height
$\frac{n-1}{2}$
keys
$\frac{n-1}{2}$
keys



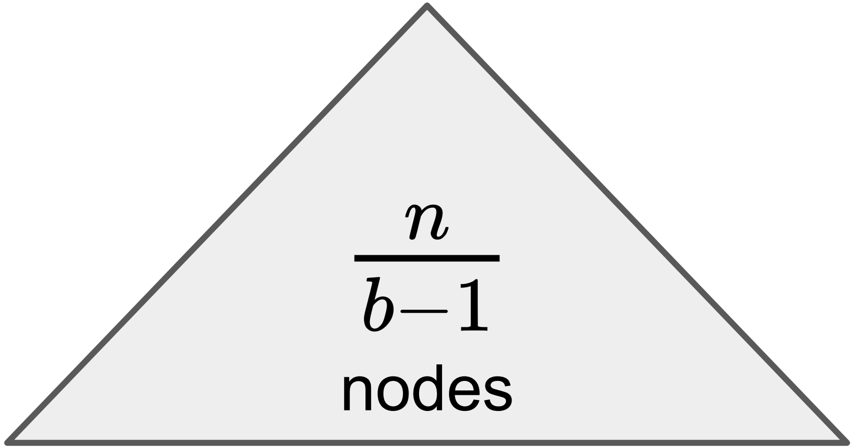
Minimum height
n
keys

Problem 1.b.

Maximum height



Minimum height



Problem 1.b.

Hence for sufficiently large n and $n \gg b$

Maximum height: $O(\log_a n)$

Minimum height: $O(\log_b n)$

(a,b)-tree properties

We have established the following properties for (a,b)-tree:

1. It is balanced
2. Has height: $O(\log_a n)$

Why are these properties important?

Searching

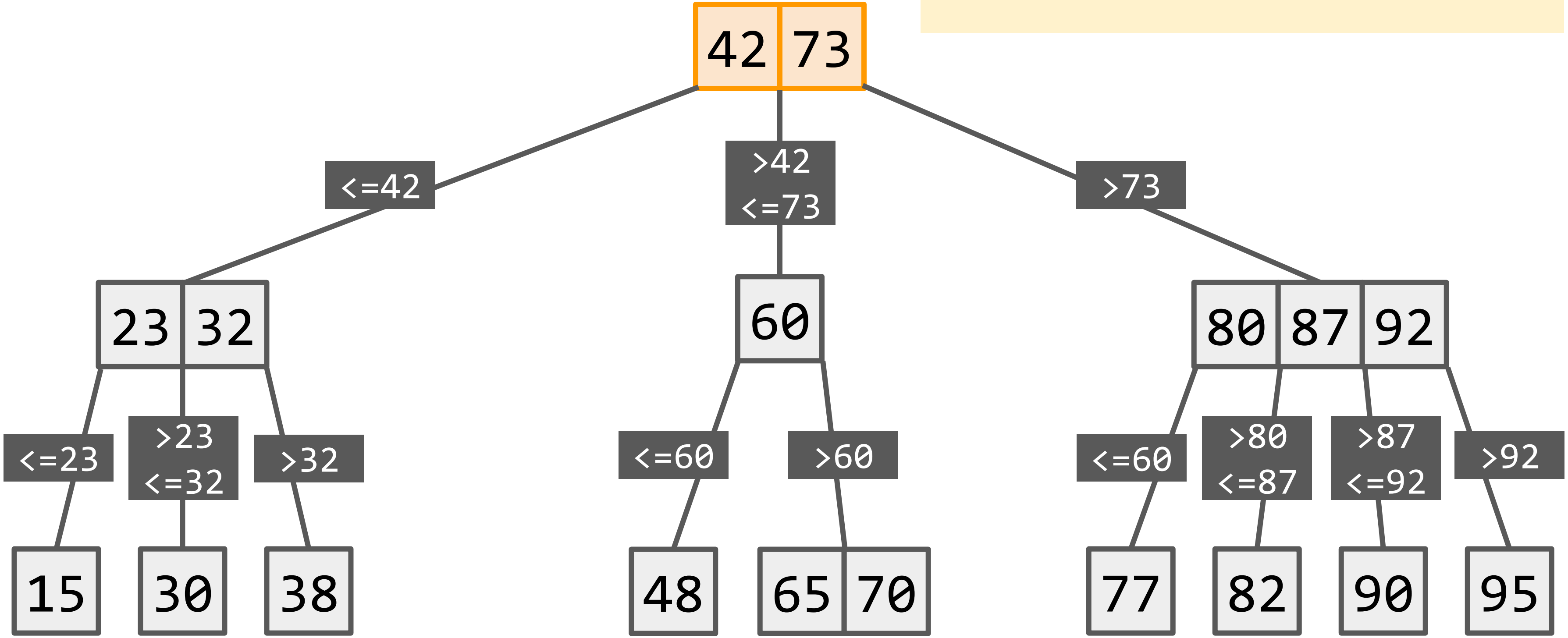
Because what else is a search tree good for?

Problem 1.c.

Write down the pseudocode for searching an (a,b)-tree.

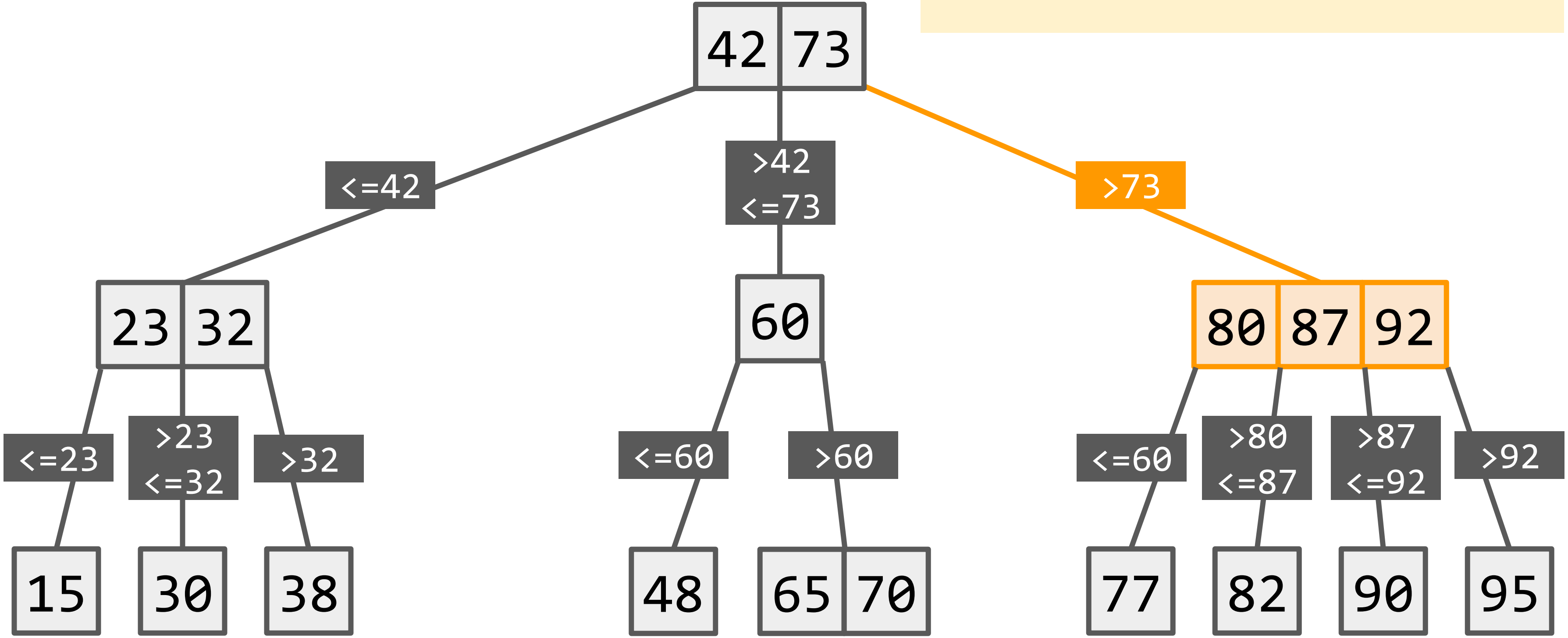
(2,4)-tree — `search(82)`

- Starting at the root
- `82` not in keylist: `[42,73]`
- Need to look in its subtrees



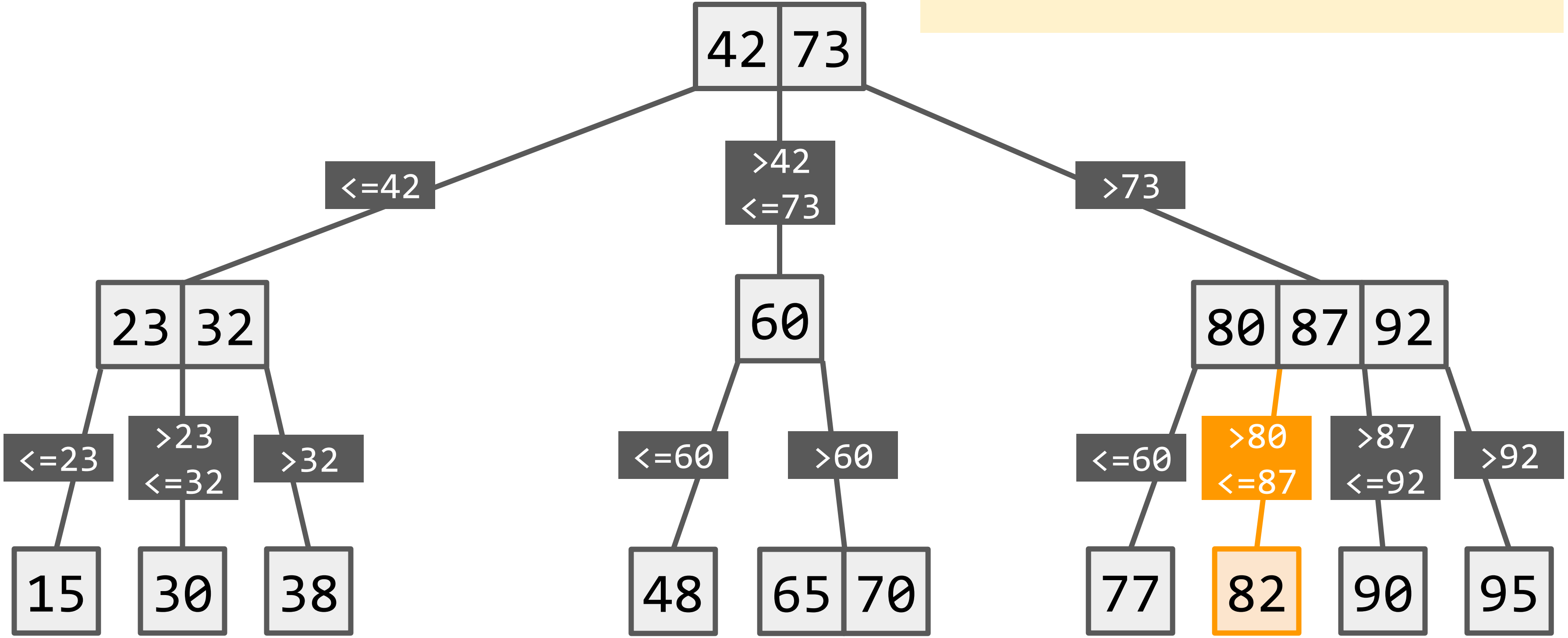
(2,4)-tree — `search(82)`

- `82>73`, go to rightmost child
- `82` not in keylist: `[80,87,92]`
- Need to look in its subtrees



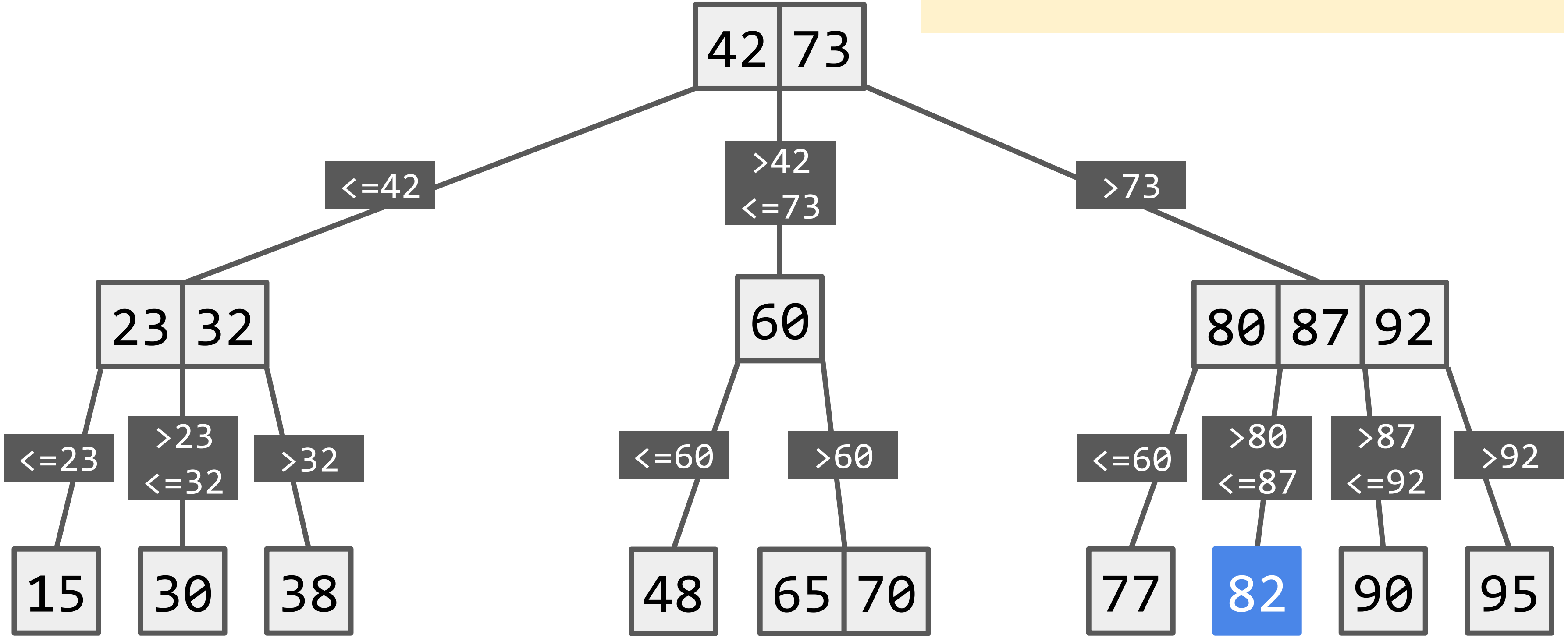
(2,4)-tree — `search(82)`

- `87>=82>80`, go to second left child



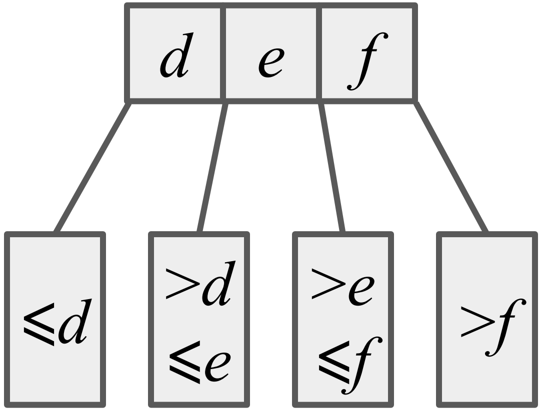
(2,4)-tree — `search(82)`

- `82` found in keylist: `[82]`
- Return node



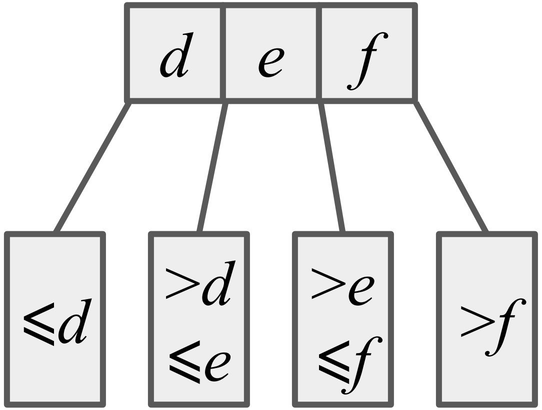
Problem 1.c.

What data structure should we use for storing the keys and children in a node?



Problem 1.c.

One possible way is to use an array of (key, subtree) pairs:



(d, t_1)	(e, t_2)	(f, t_3)	(INF, t_4)

$t_1 > -\infty$ $t_2 > d$ $t_3 > e$ $t_4 > f$

$t_1 \leqslant d$ $t_2 \leqslant e$ $t_3 \leqslant f$ $t_4 \leqslant +\infty$

Notice that for the key range of a subtree, the right-hand-bound (i.e. $\leqslant$) is explicit in the pairing, but the left-hand-bound (i.e. $>$) is implicit because you just need to look to the left neighbor in the sorted keylist to determine it.

Problem 1.d.

What is the cost of searching an (a,b)-tree with n **nodes**?

Problem 1.d.

- An (a,b)-B-tree with n **nodes** has $\mathrm{O}(\log_a n)$ height
- Binary search for a key at every node takes $\mathrm{O}(\log_2 b)$ time
- Hence total search cost:

$$\mathrm{O}(\log_2 b \cdot \log_a n)$$

$$= \mathrm{O}(\log_a n) \qquad \text{since } \log_2 b \text{ is a constant}$$

$$= \mathrm{O}(\log n)$$

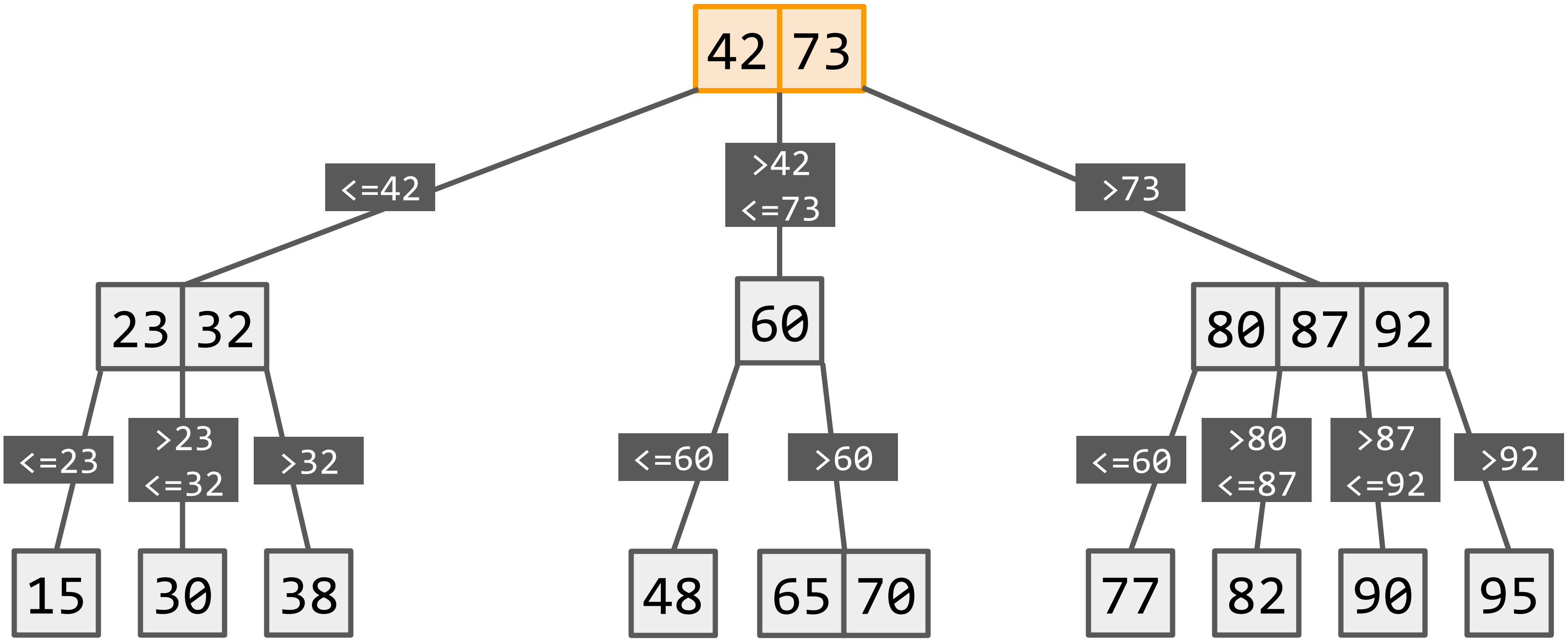
Insertion

Because how do you even search for something when you have nothing?

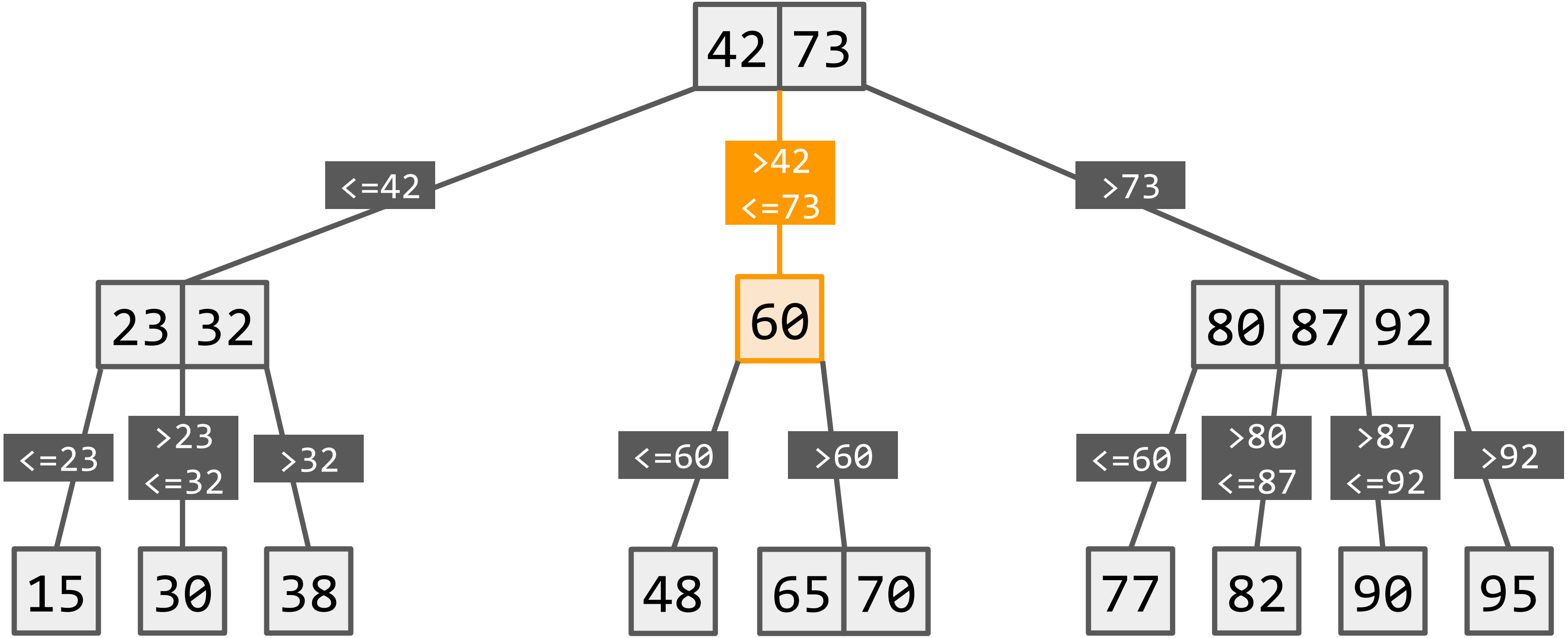
Insertion

- Just like BSTs, we only insert at leaves!
- So the idea is to navigate to a suitable leaf and insert the new key to its key list

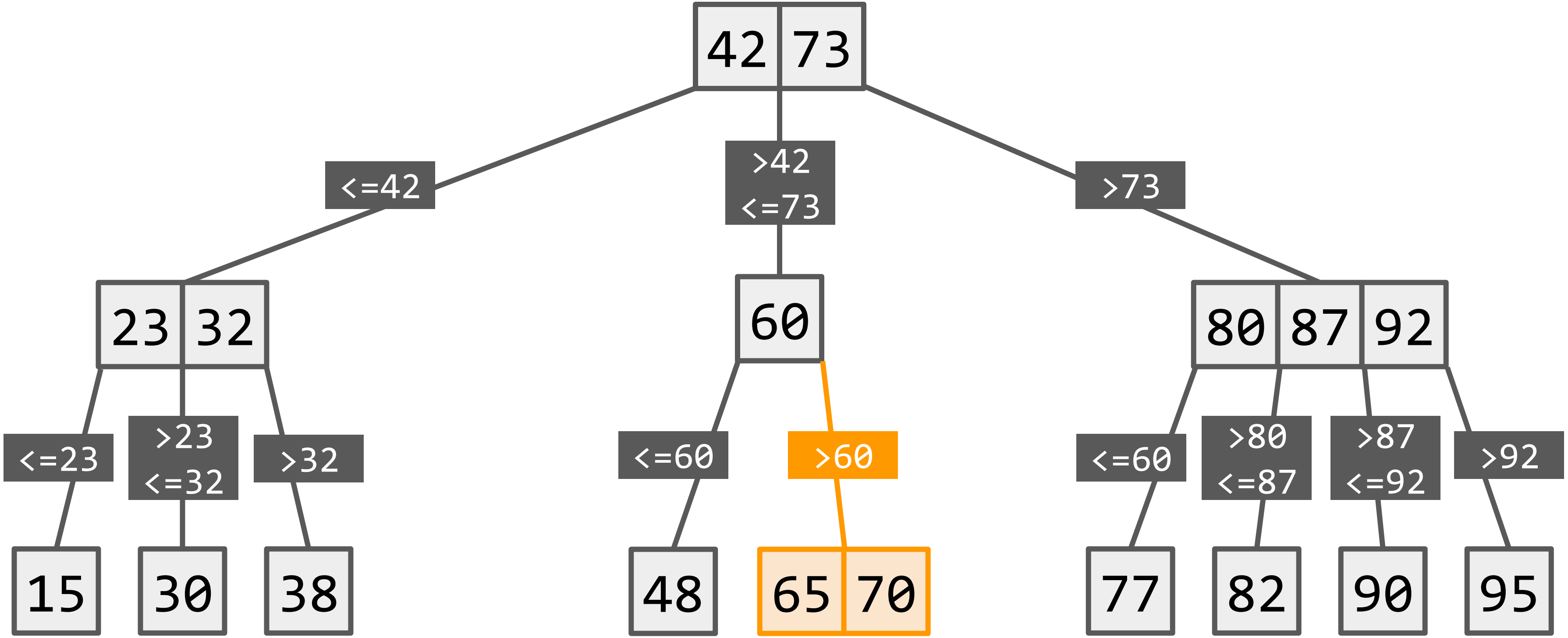
(2,4)-tree — insert(71)



(2,4)-tree — `insert(71)`

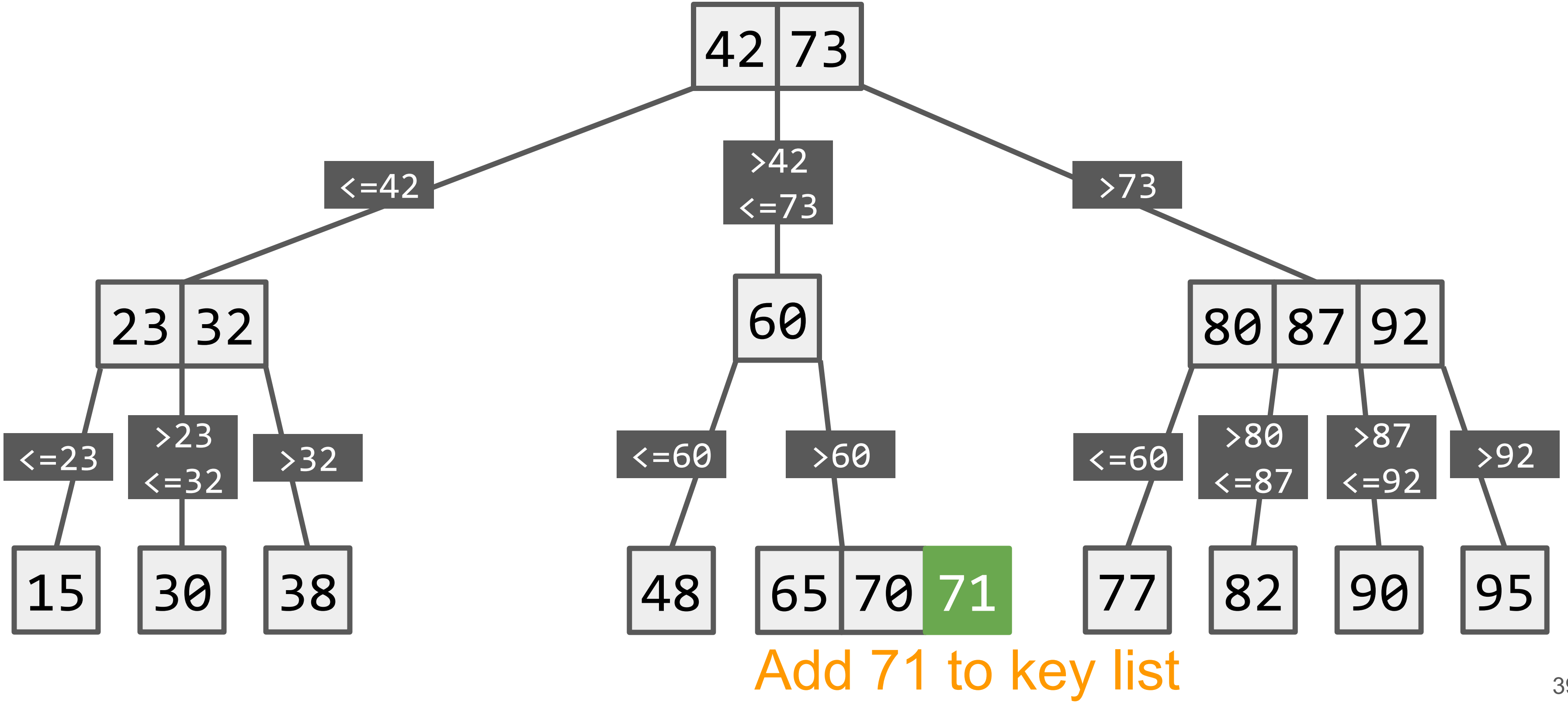


(2,4)-tree — `insert(71)`



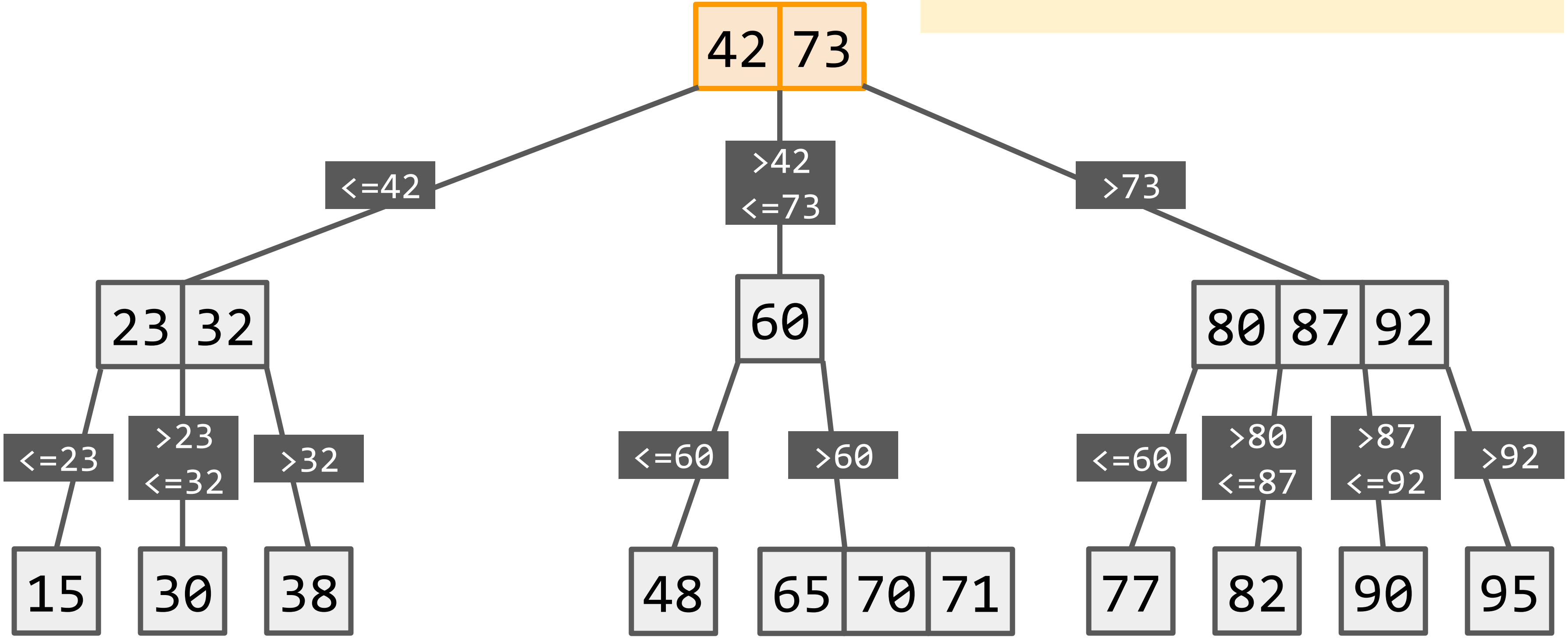
Found appropriate leaf node!

(2,4)-tree — `insert(71)`

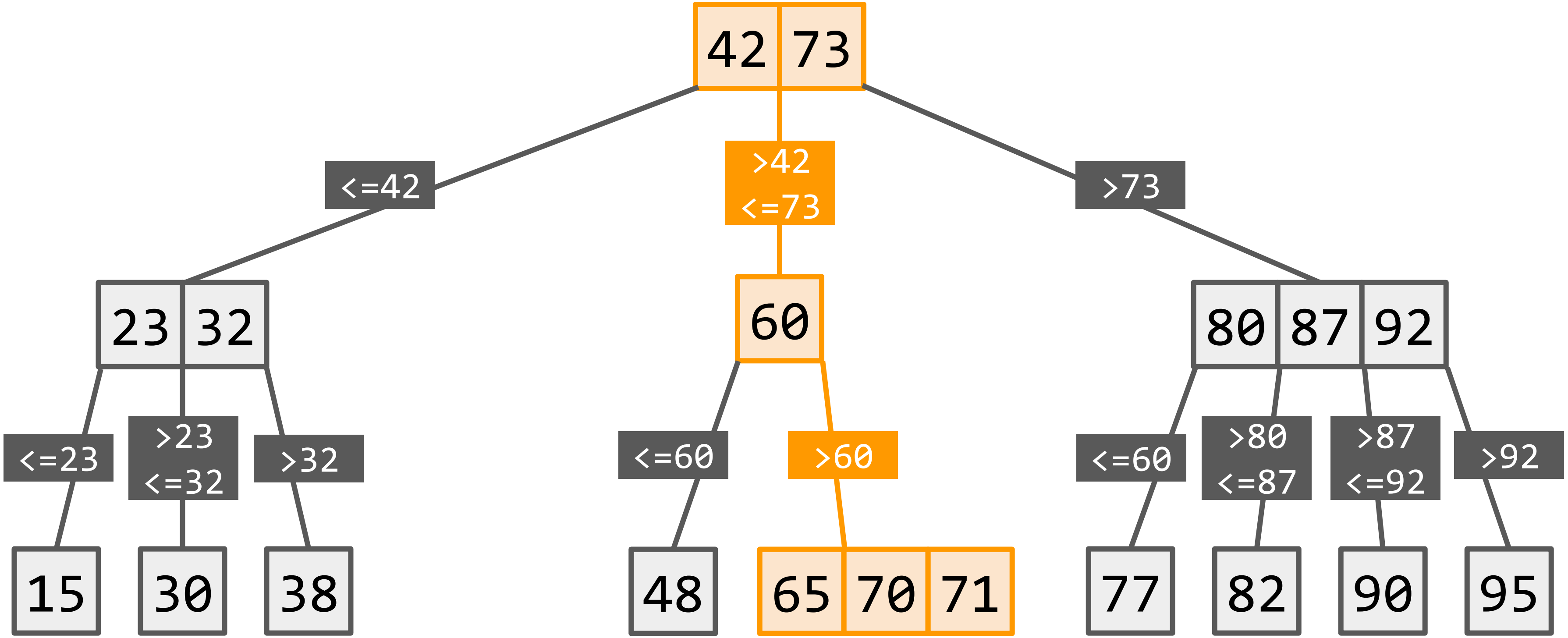


(2,4)-tree — `insert(72)`

Now let's insert `72`

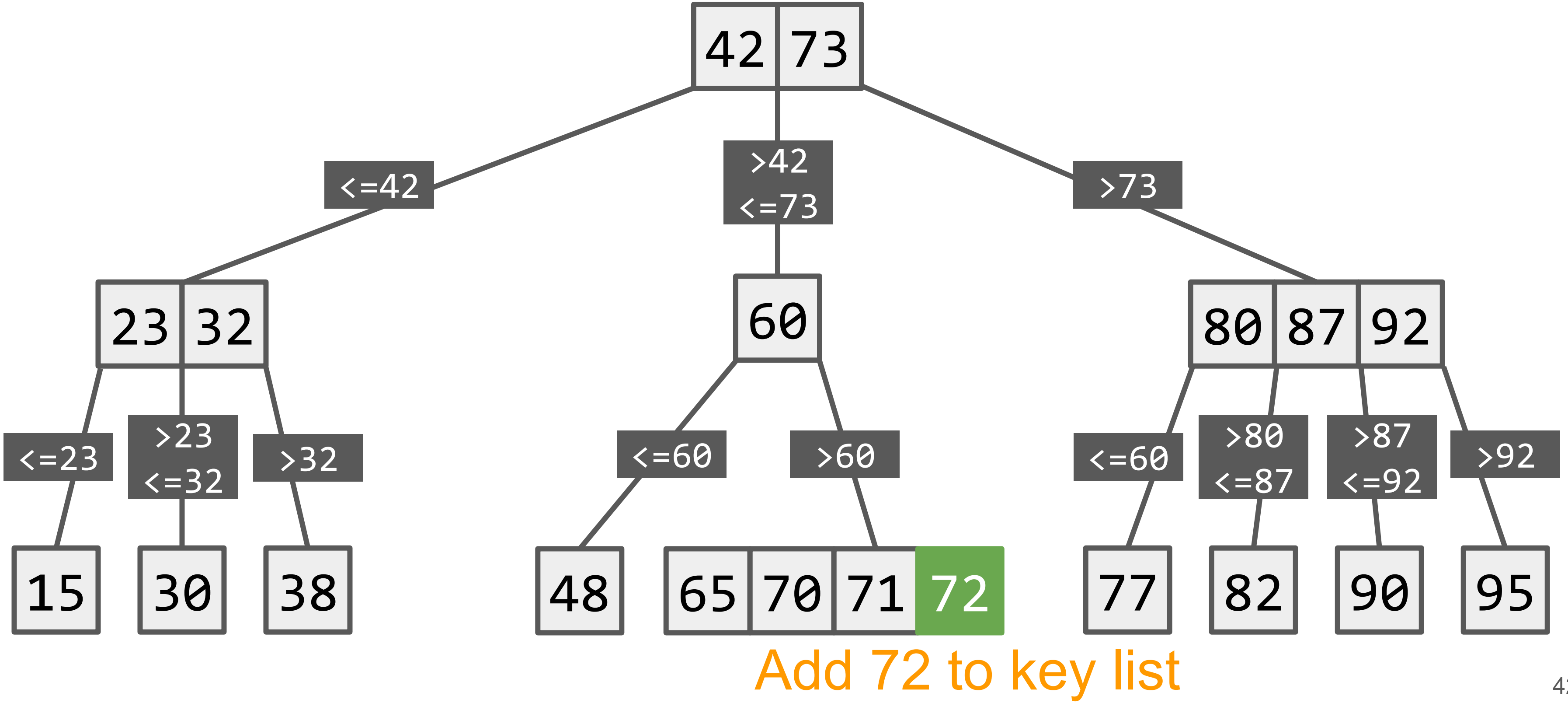


(2,4)-tree — insert(72)

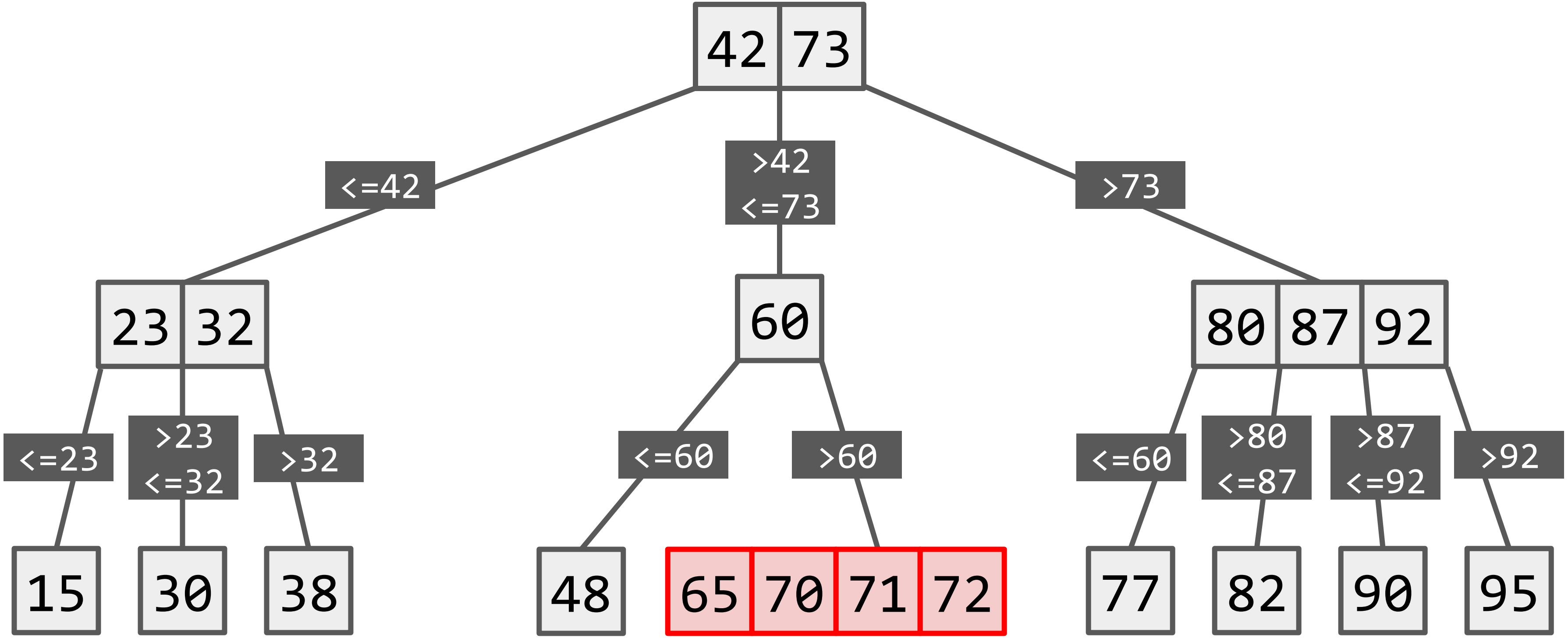


Found appropriate leaf node!

(2,4)-tree — `insert(72)`



(2,4)-tree — `insert(72)`



What's the problem here?

Insert may violate rule 1!

- Recall rule 1 enforces that nodes can have at most $b - 1$ keys
- Insertion may cause the leaf nodes to grow too large
- We need to have an operation to handle such cases
- Key idea: Redistribute out the keys!

Key-redistribution strategy

How to redistribute out the keys?

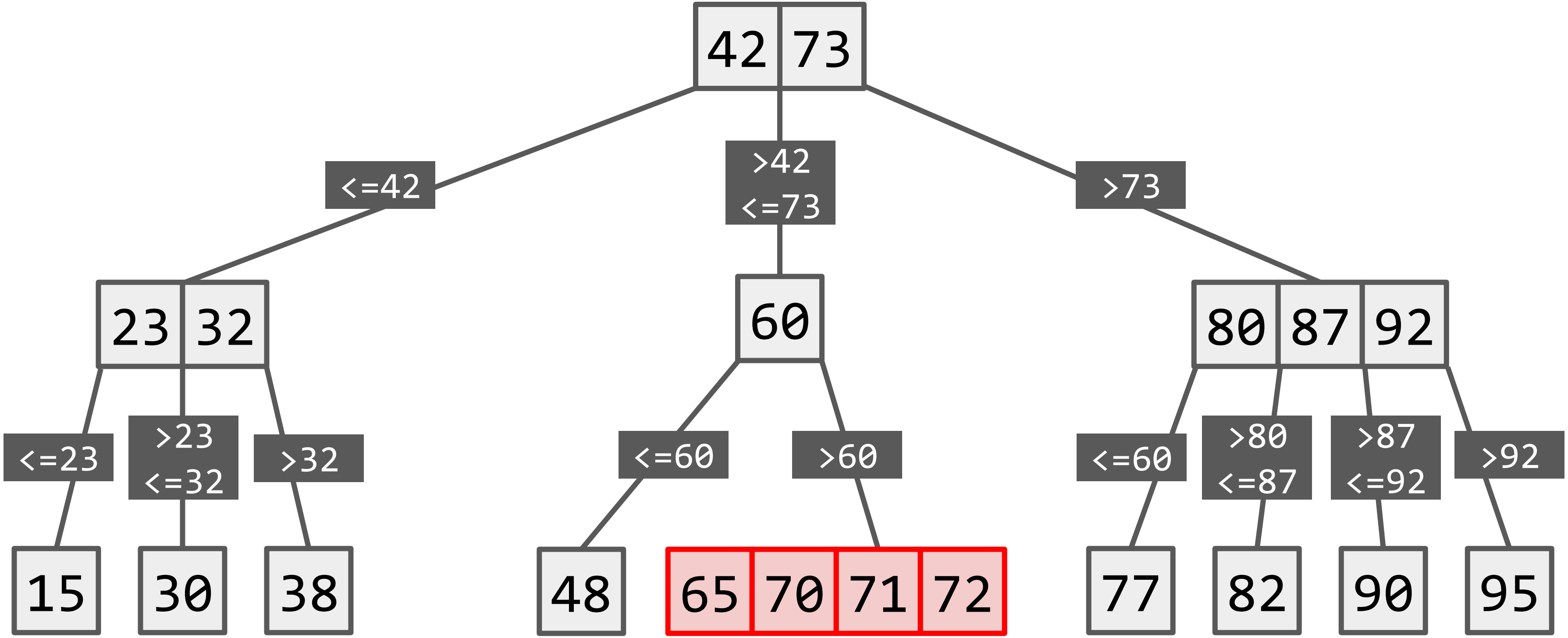
A. Over sibling nodes?
B. Over a new node?

Key-redistribution strategy

How to redistribute out the keys?

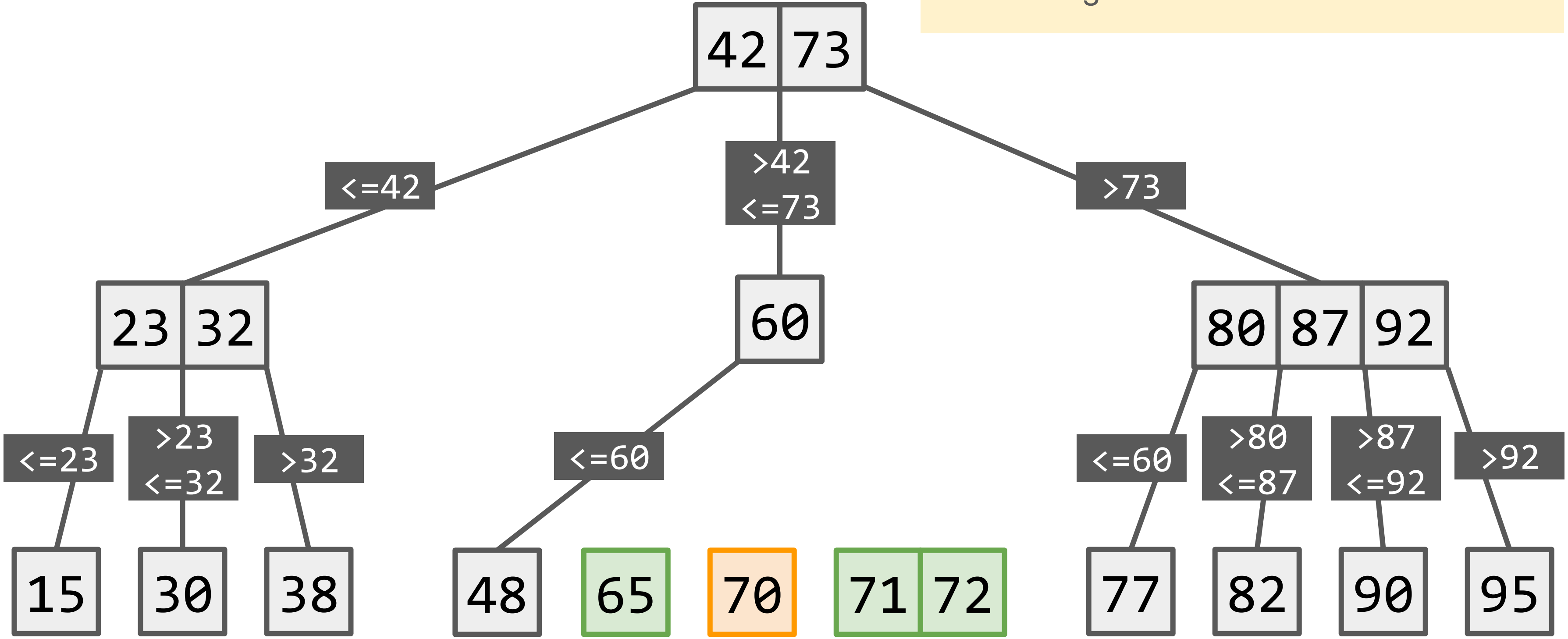
A. Over sibling nodes?
    - Cannot guarantee no further violations on current level!
B. **Over a new node**
    - Can always guarantee postcondition adheres to rule 1 on current level!

Key-redistribution



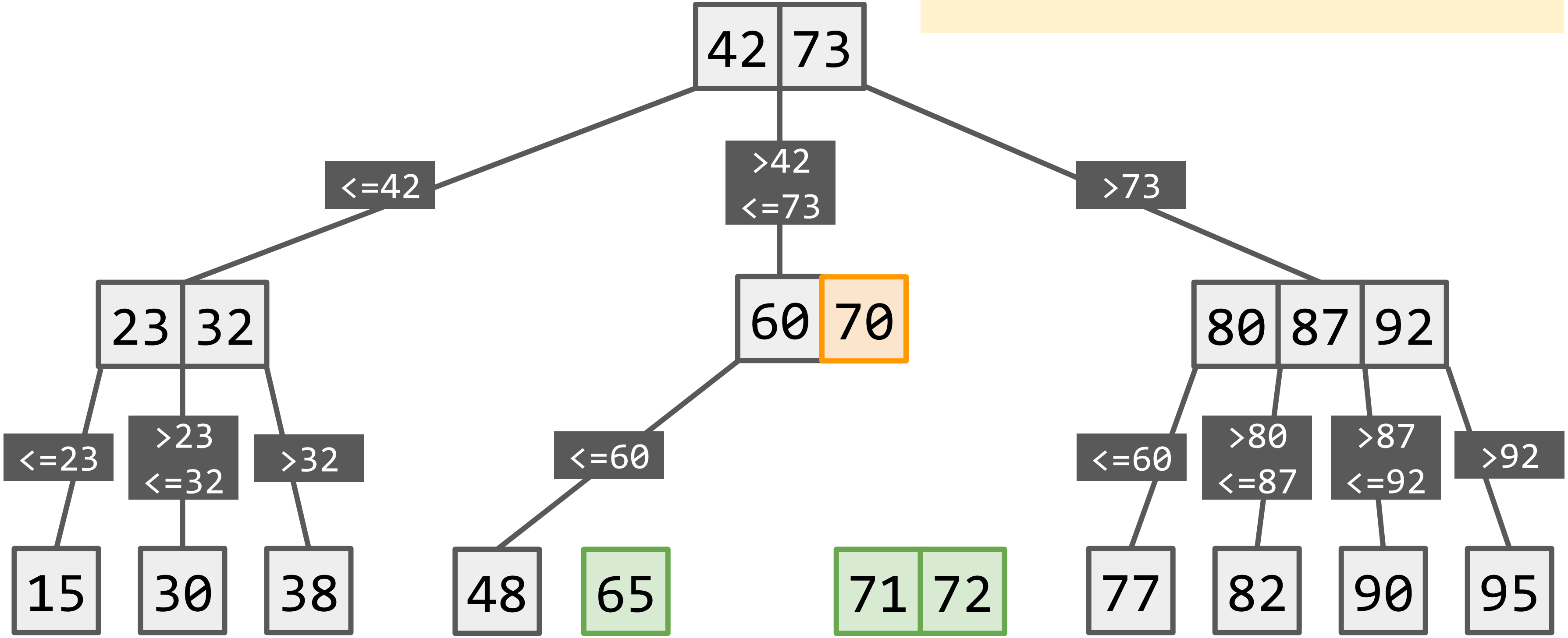
Key-redistribution

- Choose the median key `70`
- Use that to split keylist into 2 halves
- Left half goes into a new node



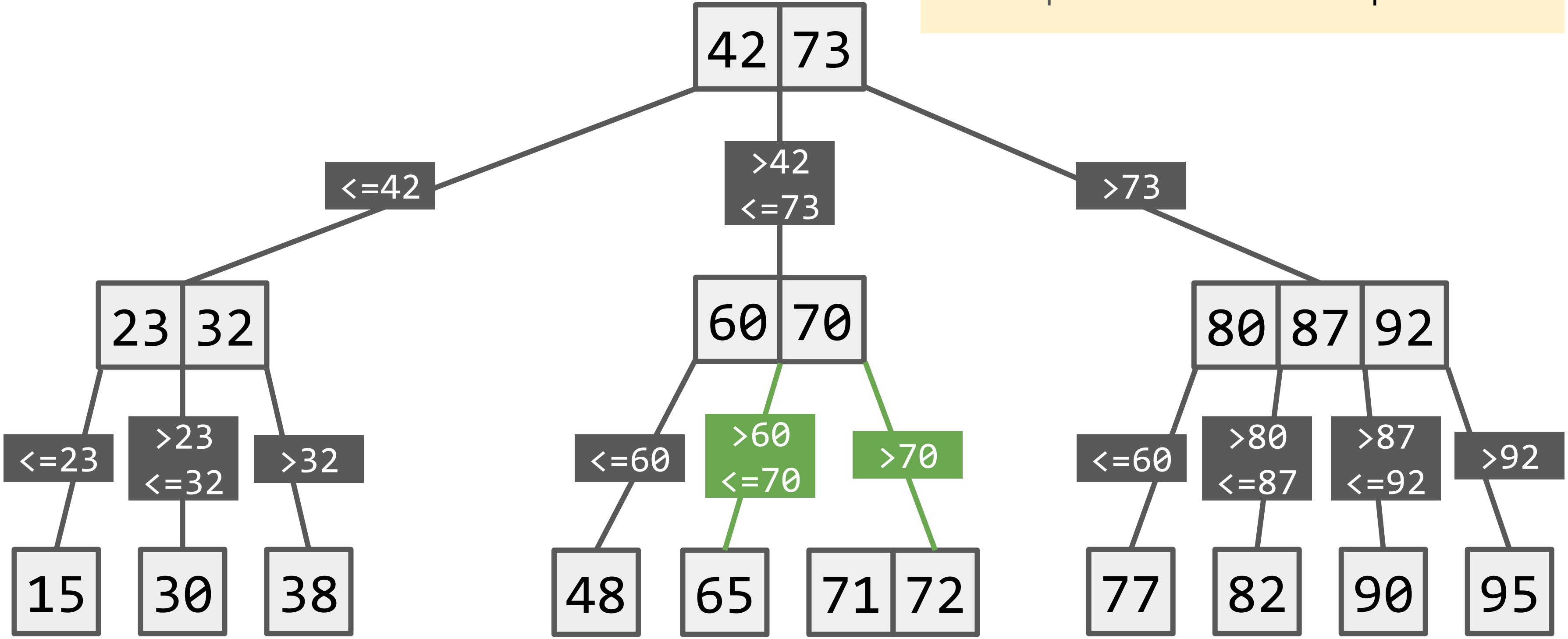
Key-redistribution

- Move median key to parent



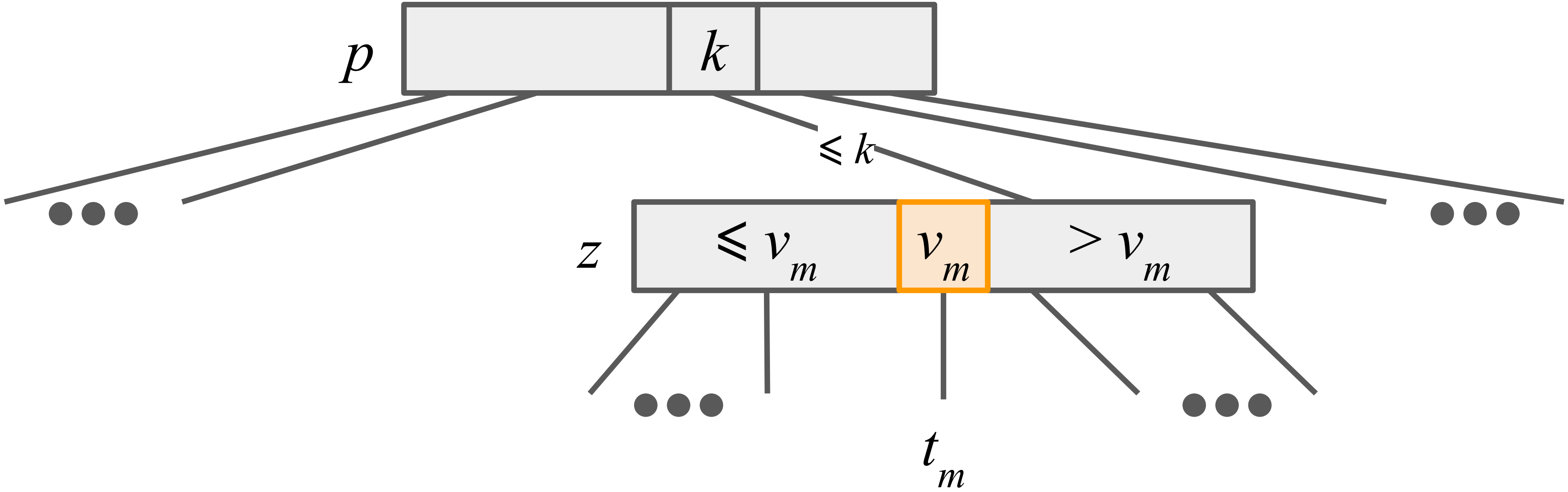
Key-redistribution

- Link nodes to parent
- All nodes now fulfil rule 1
- This operation is known as `split`



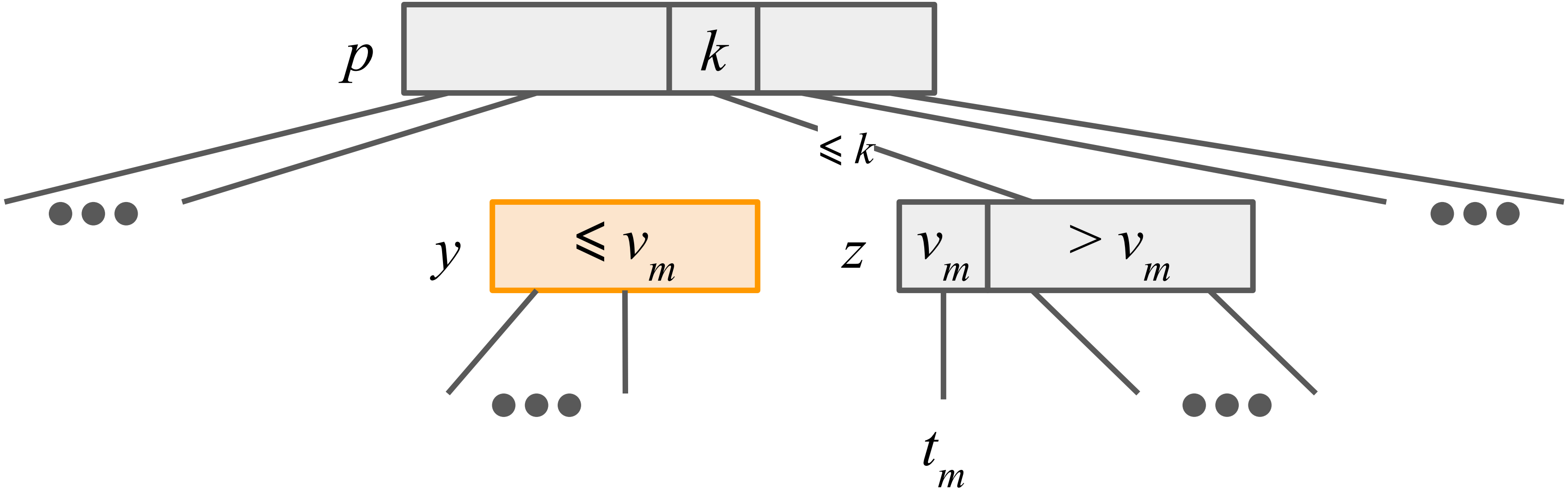
Split operation

Step 1



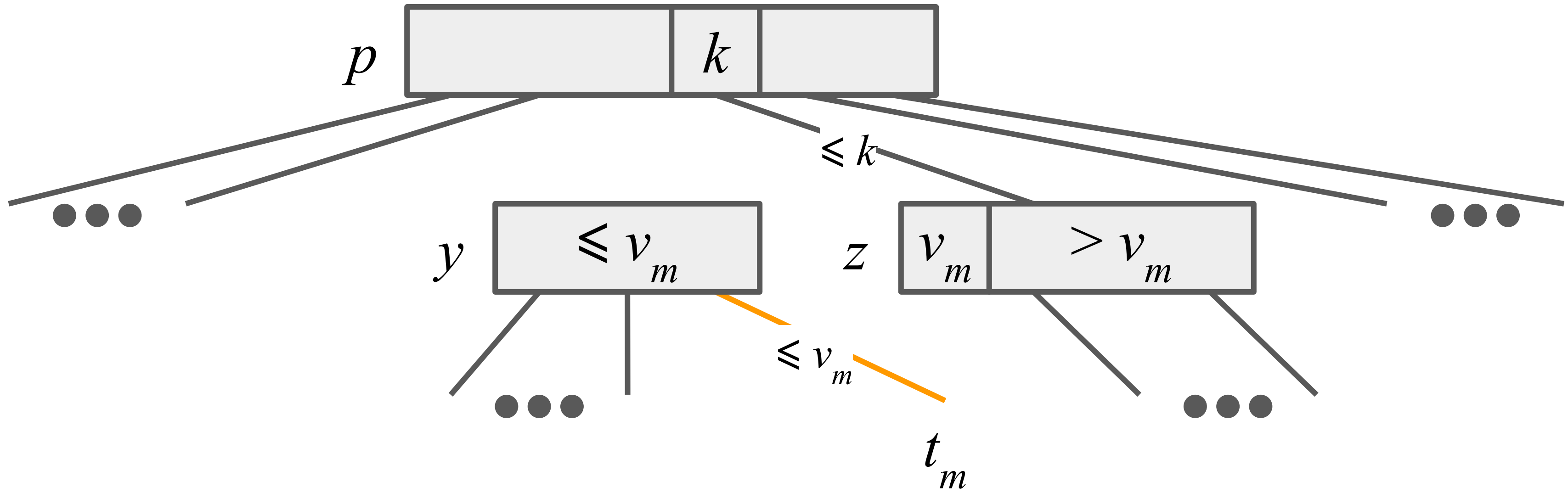
Find the **median key** v_m where the number of z's keys that are $\leqslant v_m$ (LHS) is $\lfloor (b-1)/2 \rfloor$ and $> v_m$ (RHS) is $\lceil (b-1)/2 \rceil$.

Step 2



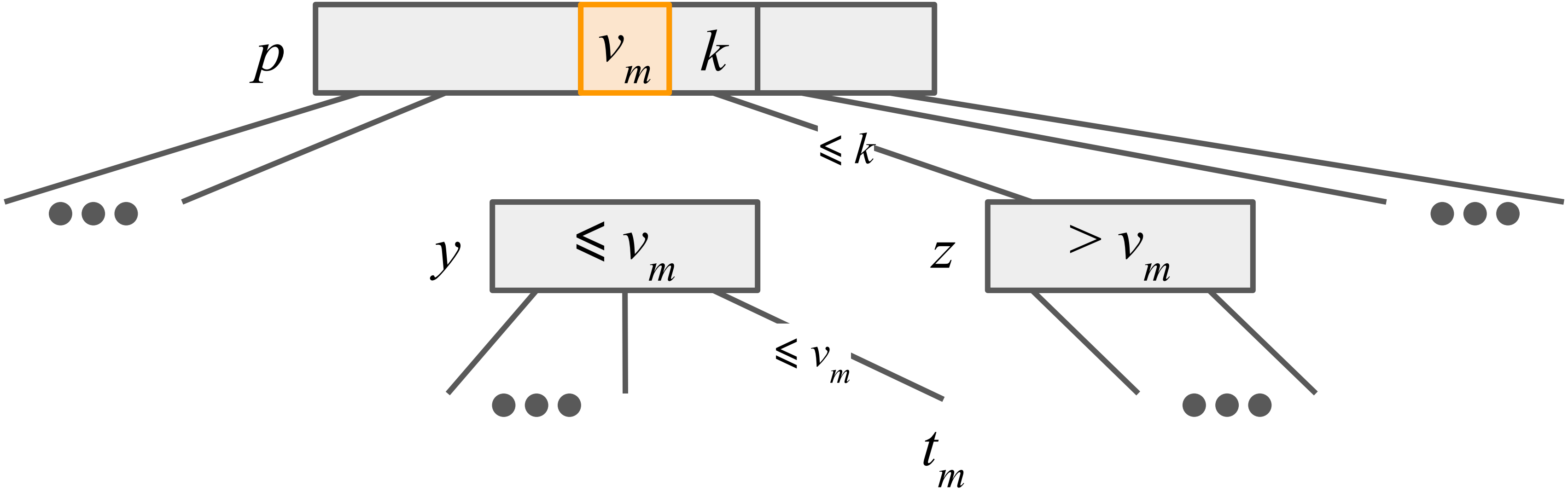
In z, separate all keys $\leqslant v_m$ as well as their associated subtree links to form a new node y.

Step 3



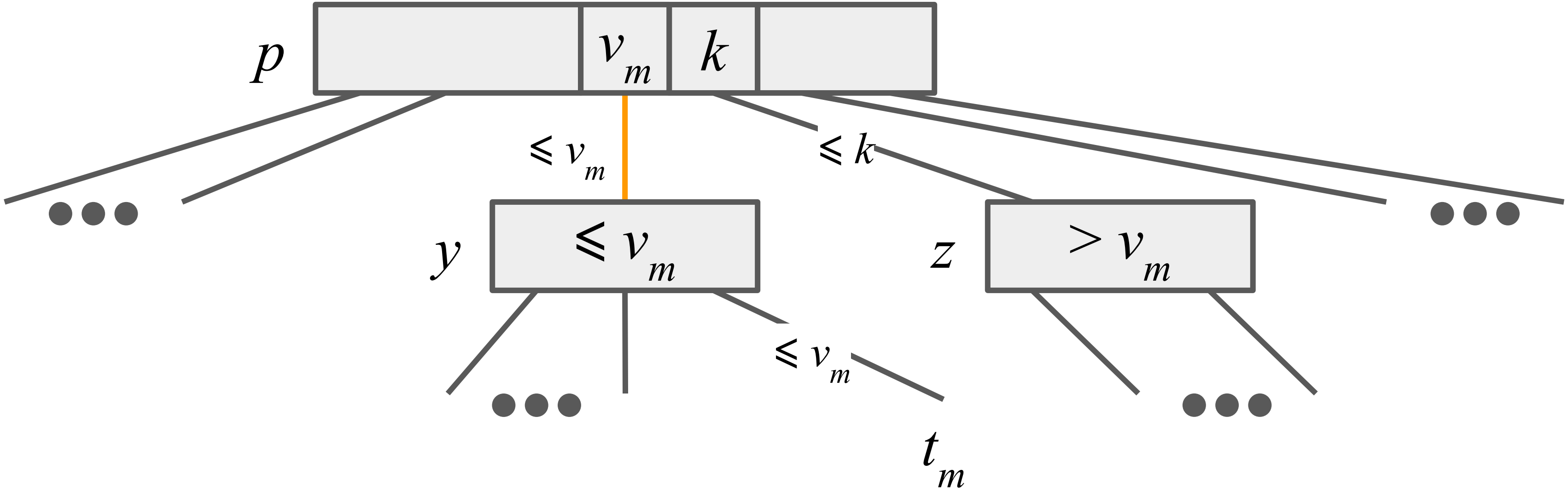
y needs to have a final child, so we assign it to be t_m, (the subtree previously corresponding to v_m). We of course also have to remove z as the parent of t_m.

Step 4



Move key v_m from z to p (z's parent). This newly inserted key should immediately precede the link to node z (i.e. key k).

Step 5



Add y as a child of p by inserting a link from v_m (i.e. immediately preceding the link to z)

Guiding question

Why must `split` “offer” one key to the parent?

Guiding question

Why must `split` “offer” one key to the parent?

Answer: After splitting, the parent will have one more child than before, therefore it must also have one more key. Taking that key from the node to be split is convenient.

Guiding question

Will the split procedure on an (a,b)-tree ever end up with nodes too small and violate rule 1?

Guiding question

Will the split procedure on an (a,b)-tree ever end up with nodes too small and violate rule 1?

Answer: No. This is because

- Precondition of split: node has keylist size at least b
- After splitting:
  - Keylist size of LHS is at least $\lfloor (b-1)/2 \rfloor$
  - Keylist size of RHS is at least $\lceil (b-1)/2 \rceil$
- Recall by definition, $2 \leqslant a \leqslant (b+1)/2$ for (a,b)-trees
- We can rewrite the inequality as $1 \leqslant a-1 \leqslant (b-1)/2$
- Therefore the keylist size of either LHS and RHS will be at least $a-1$ after the split

Guiding question

What is one potential problem split can cause?

How can we solve it?

Guiding question

What is one potential problem split can cause?

How can we solve it?

Answer: The parent can be over-capacity after being offered one key from the split. We therefore need to keep splitting upwards the tree until we resolve all over-capacities.

Guiding question

- The previous steps outlined the splitting procedure on a node z
- Which is the scenario (corner case) where these steps no longer apply exactly? I.e. Slight modifications required

Guiding question

- The previous steps outlined the splitting procedure on a node z
- Which is the scenario (corner case) where these steps no longer apply exactly? I.e. Slight modifications required

Answer: When z is the root

How to split a root node?

Key idea:

- Follow the same procedure as before
- Instead of elevating split key v_m to the parent (there's no parent for root), make it the new root!

How to split a root node?

1. Pick the median key v_m as the split key
2. Split z into LHS and RHS using v_m
3. Create a new node y
4. Move LHS split from z to y

unchanged steps

5. Create a new empty node r
6. Insert v_m into r
7. Promote r to new root node
8. Assign y and z to be the left and right child of r respectively
9. Assign previous subtree t_m associated with v_m to be the final child of y

New steps

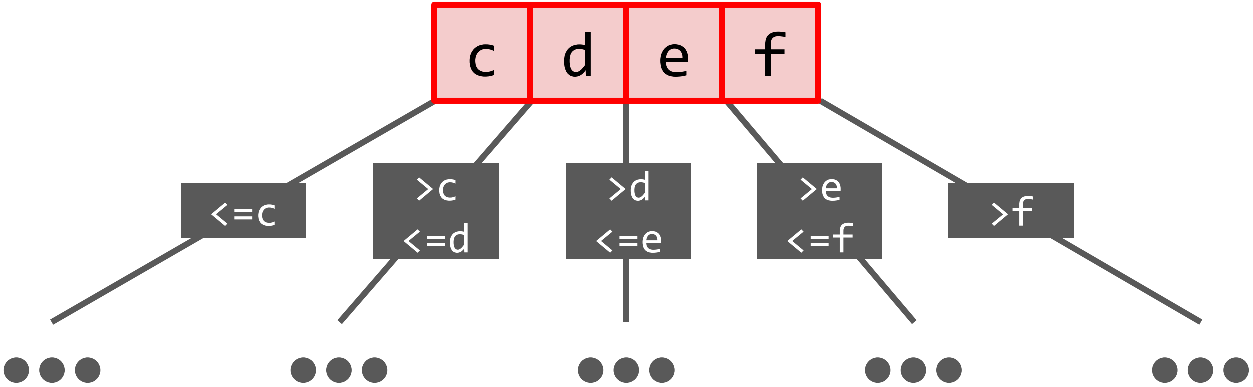
Problem 1.e.

Come up with an example each for

1. Splitting an internal node
2. Splitting a root node

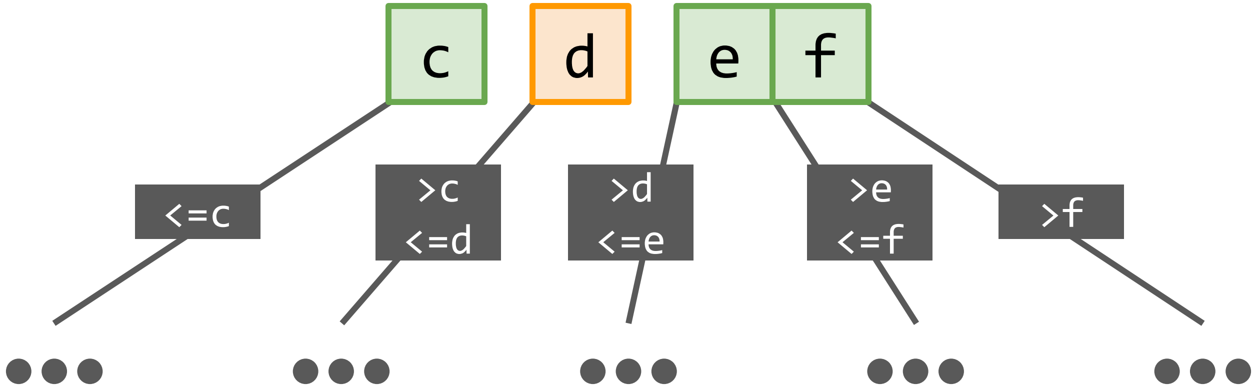
Corner case: splitting the root

- Root node has more than b - 1 keys
- Rule 1 violated



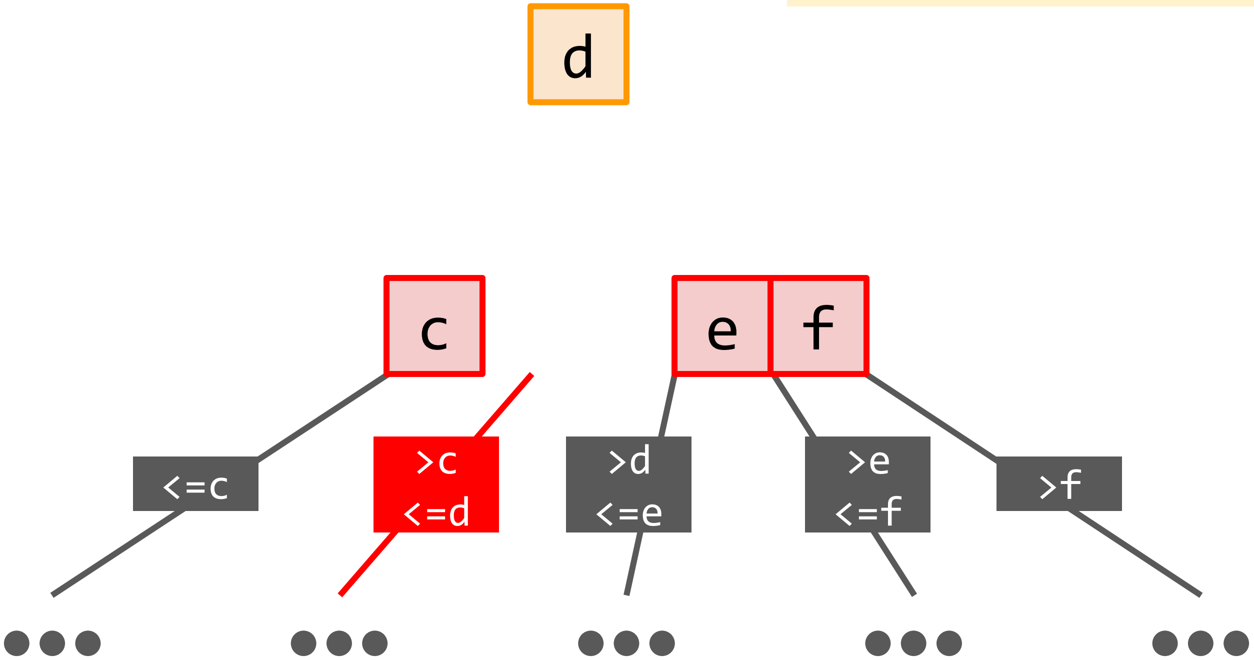
Corner case: splitting the root

- Split root node keys using median key



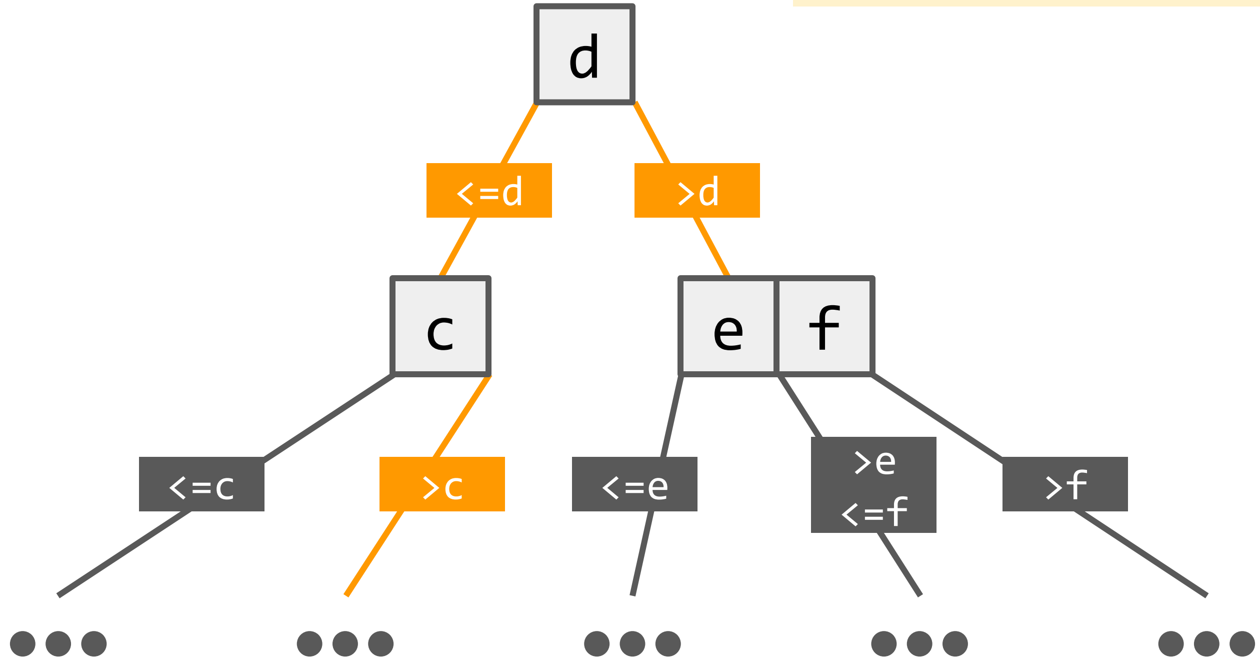
Corner case: splitting the root

- Promote median key to form new root with just this key
- 3 subtrees will be orphaned



Corner case: splitting the root

- Link old roots as children of new root
- Link previous child associated with `b` to be the last child of `a`



Guiding question

According to our scheme, immediately after we split the keylist of a node in half, which half will be lacking a child? Which child will it be lacking?

Guiding question

According to our scheme, immediately after we split the keylist of a node in half, which half will be lacking a child? Which child will it be lacking?

Answer: LHS will be lacking the rightmost child corresponding to the key range for everything greater than its last key. Note that RHS will always be a valid node immediately after split.

Guiding question

Why does root node have special treatment for rule 1 which allows it to have 1 key (as opposed to $a-1$ keys for internal nodes)?

Guiding question

Why does root node have special treatment for rule 1 which allows it to have 1 key (as opposed to $a-1$ keys for internal nodes)?

Answer: To permit split operation at root!

Proactive vs passive strategy

There are generally 2 approach for insertion:

1. *Proactive*: preemptively split any node at full capacity (i.e. $b-1$ keys) during search phase
2. *Passive*: lazier strategy whereby insertion is done first and then check parent for violation (potentially splitting all the way to the top)

Insertion algorithm

Insert($BTree$, x)

$w \leftarrow Btree$.root
while true **do**
    if w contains b - 1 keys **then**
        $v_m, y, z \leftarrow$ Split(w, v_m)
        if $x \leqslant v_m$ **then**
            $w \leftarrow y$
        else
            $w \leftarrow$ z
    if w is a leaf **then**
        break
    else
        $w \leftarrow$ GetSubtree(w, x)   // Subtree with keyrange containing x
InsertKey(w, x)

Proactive approach

Guiding question

Can you prove that the proactive strategy just presented only applies to (a,b)-trees with $b \geq 2a$?

Guiding question

Can you prove that the proactive strategy just presented only applies to (a,b)-trees with $b \geq 2a$?

Answer:

- Before proactive split, we have $b-1$ keys (max capacity)
- After split, we have $b-2$ keys across a sibling pair (after promoting one key)
- Left sibling keys: $\lfloor (b-2)/2 \rfloor = \lfloor b/2 \rfloor - 1$
- Left sibling (smaller half) has to have $\geq a-1$ keys to be a valid node after the split
- So explicitly the condition is: $\lfloor b/2 \rfloor - 1 \geq a-1$
- Add 1 to both sides: $\lfloor b/2 \rfloor \geq a$
- Multiply 2 to both sides: $b \geq 2a$

Problem 1.f.

Prove that:

1. If node w is split, then it satisfies the preconditions of the split routine
2. Before x is inserted into w (at the last step), there are $< b$ keys in w (so key x can be added)
3. All three rules of the (a,b)-tree are satisfied after an insertion

Problem 1.f. — Solution

[1] Precondition of the proactive split routine is that w must have at least $b - 1$ keys. After splitting, each side must have at least $a - 1$ keys. This means w must have at least $2a - 1$ keys before the split. Since $b = 2a$ for B-trees, this means w has at least $b - 1$ keys.

[2] Proactive split routine guarantees that every node having $b - 1$ or more keys will be split so w must have at most $b - 2$ keys prior to insertion. A passive approach would ensure that every node have at most $b - 1$ keys. There will be $< b$ keys in w in either case

[3] Inductively, [1] and [2] ensures that rules 1 and 2 will always be maintained at every node as we descend the tree. What about rule 3?

Guiding question

Will the split operation ever violate rule 3?

Guiding question

Will the split operation ever violate rule 3?

Answer: No! A split on node z will create a new node y, so

- If z is an *internal* node, y will be at the same level and nothing will be pushed down since y was split from z
- If z is a *leaf* node, y will also be also be a leaf node at the same level
- If z is the *root* node, everything will be pushed down 1 level

Guiding question

Will an (a,b)-tree ever violate rule 3 as the tree grows with new items being inserted at the leaves?

Guiding question

Will an (a,b)-tree ever violate rule 3 as the tree grows with new items being inserted at the leaves?

Answer: No! Unlike BSTs which grows the leaves downwards, (a,b)-trees **grows the root upwards**!

Therefore leaves are always guaranteed to be on the same level as one another!

Problem 1.g.

What is the cost of inserting into an (a,b)-tree with n nodes?

Problem 1.g.—Solution

- Worst case when insertion causes split operation to propagate up all the way till root
- So an additional $O(b)$ cost of splitting a node at each level.
- So insertion may cost $O(b \log n)$ for the worst case

Problem 1.g. — Discussion

One observation (which will make more sense later after we talk about hash tables growing and shrinking) is that (if we only have inserts), then after a split, there are $b/2$ keys at the node, so there will necessarily be at least $b/2$ more inserts before the next split.

Hence on average, each of those $b/2$ inserts has amortized cost (due to split) of $\mathrm{O}(\mathrm{b}/(b/2)) = \mathrm{O}(1)$.

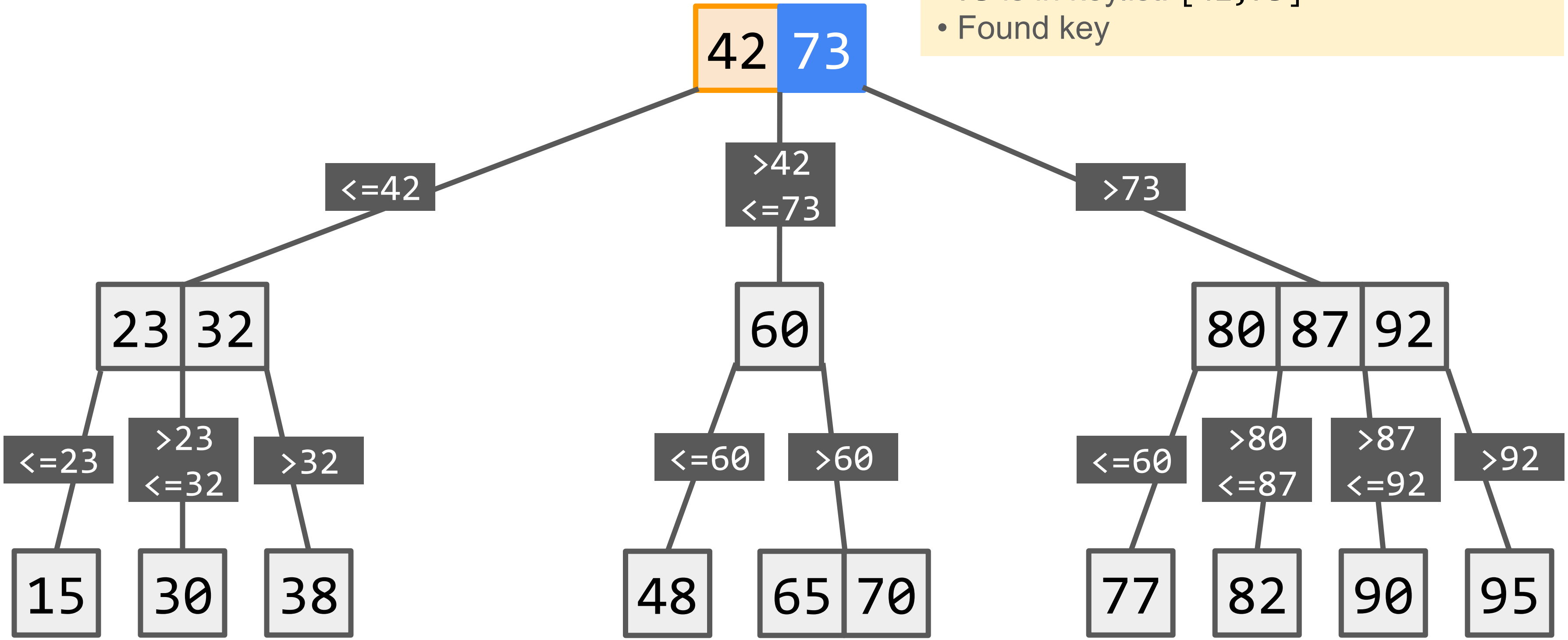
Realize however this amortization argument only checks out if we ignore deletion!

Deletion

Because nothing lasts forever

(2,4)-tree — `delete(73)`

- To delete an item, we first find it
- Starting at the root
- `73` is in keylist: `[42,73]`
- Found key



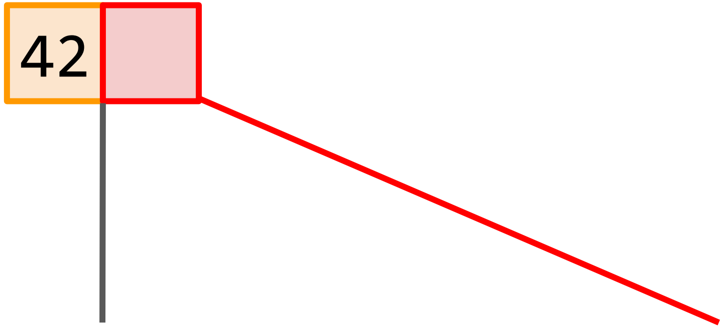
Guiding question

What is one immediate problem after deleting a key from a root or internal node?

Guiding question

What is one immediate problem after deleting a key from a root or internal node?

Answer: It will lead to an orphan.



Guiding question

How might we delete off the key and not leave behind orphans to resolve?

Hint: Think AVL trees

Guiding question

How might we delete off the key and not leave behind orphans to resolve?

Hint: Think AVL trees

Answer: Swap the key to be deleted off with its predecessor/successor which will be in a leaf node!

Guiding question

How to obtain the predecessor/successor of a key in a root/internal node?

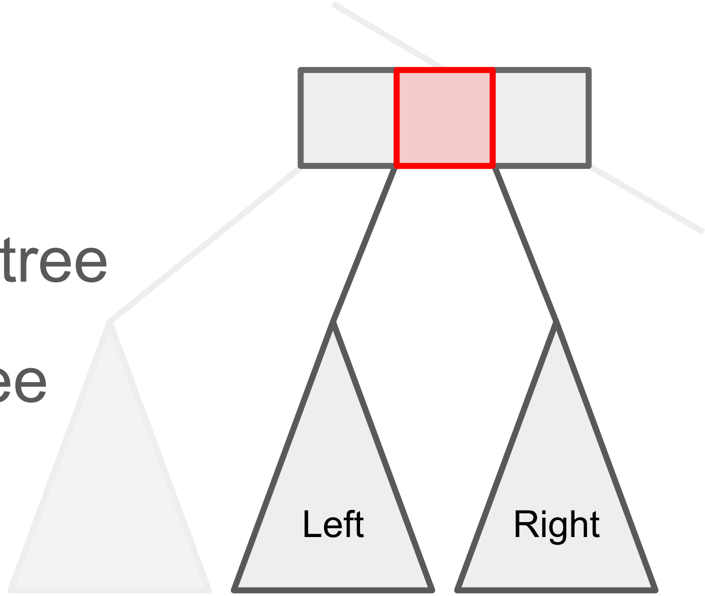
Guiding question

How to obtain the predecessor/successor of a key in a root/internal node?

Answer:

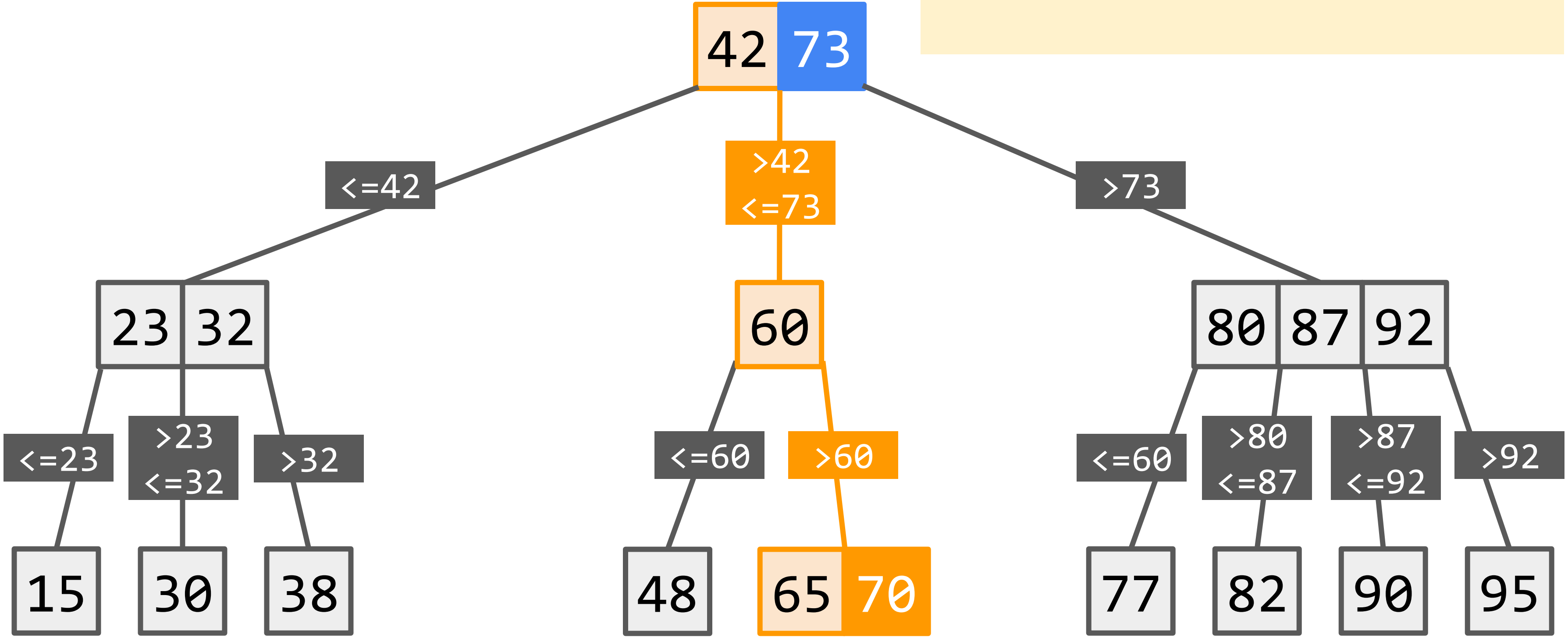
Predecessor: rightmost key in left subtree

Successor: leftmost key in right subtree



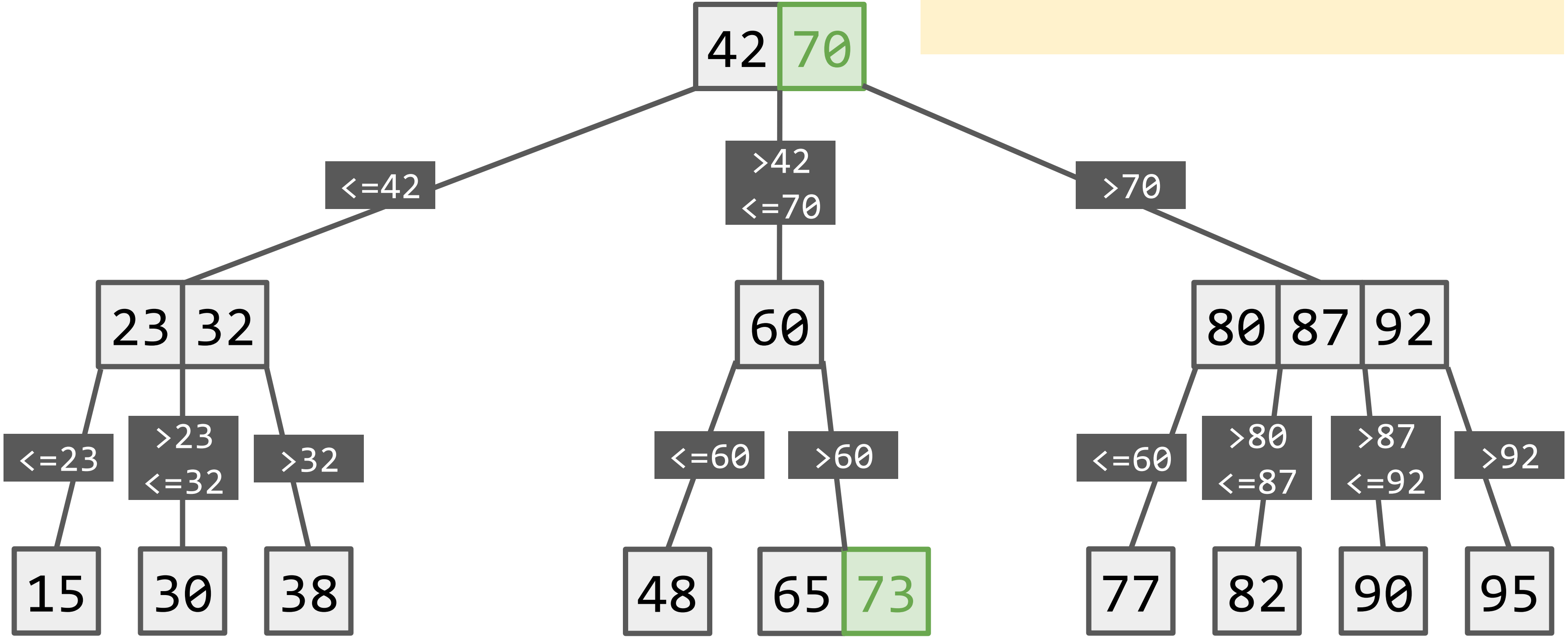
(2,4)-tree — `delete(73)`

- Predecessor of `73` is `70`



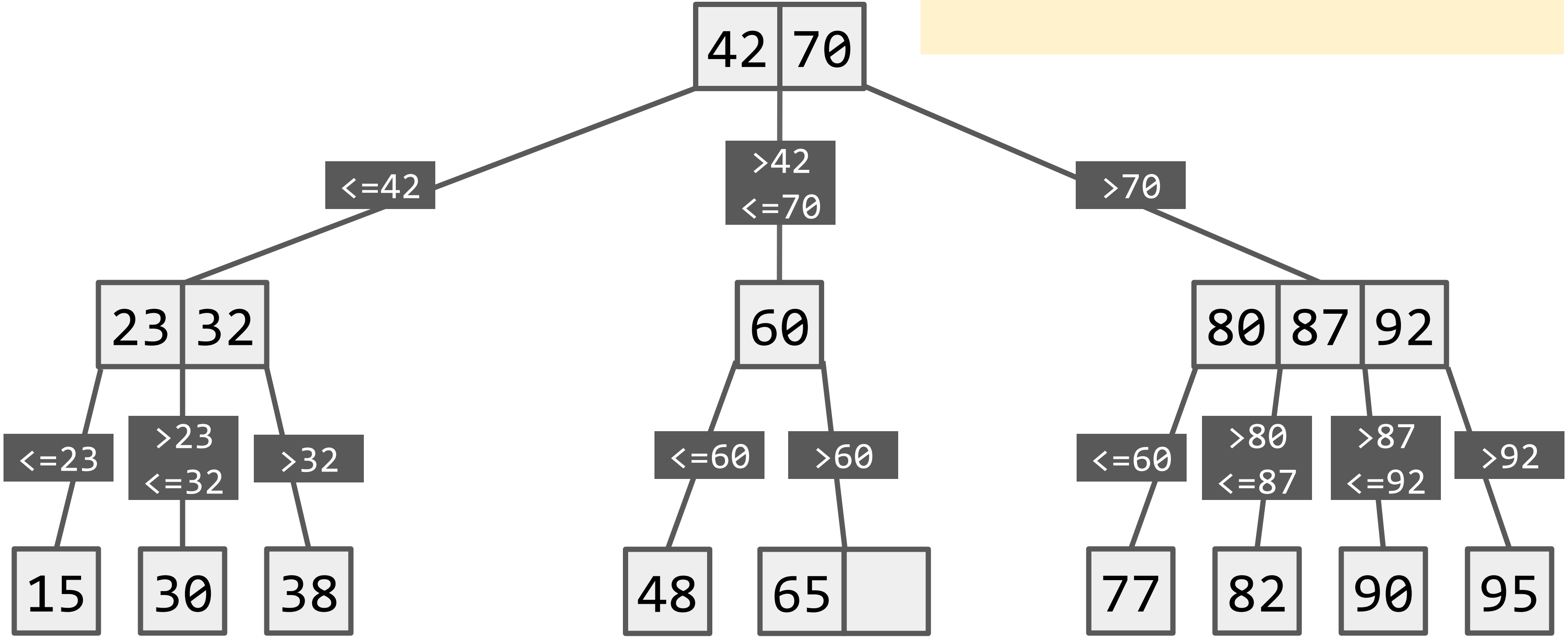
(2,4)-tree — `delete(73)`

- Swap `73` with `70`



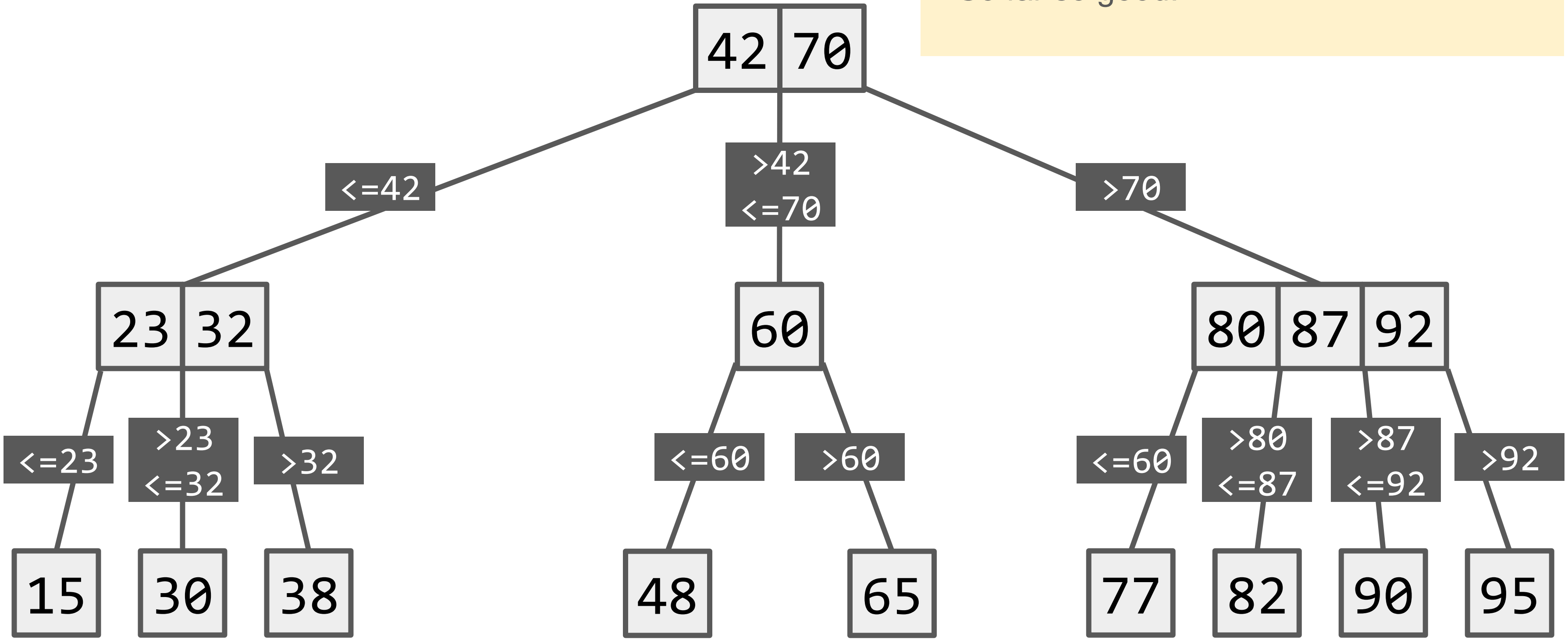
(2,4)-tree — delete(73)

- Delete `73` from keylist of leaf



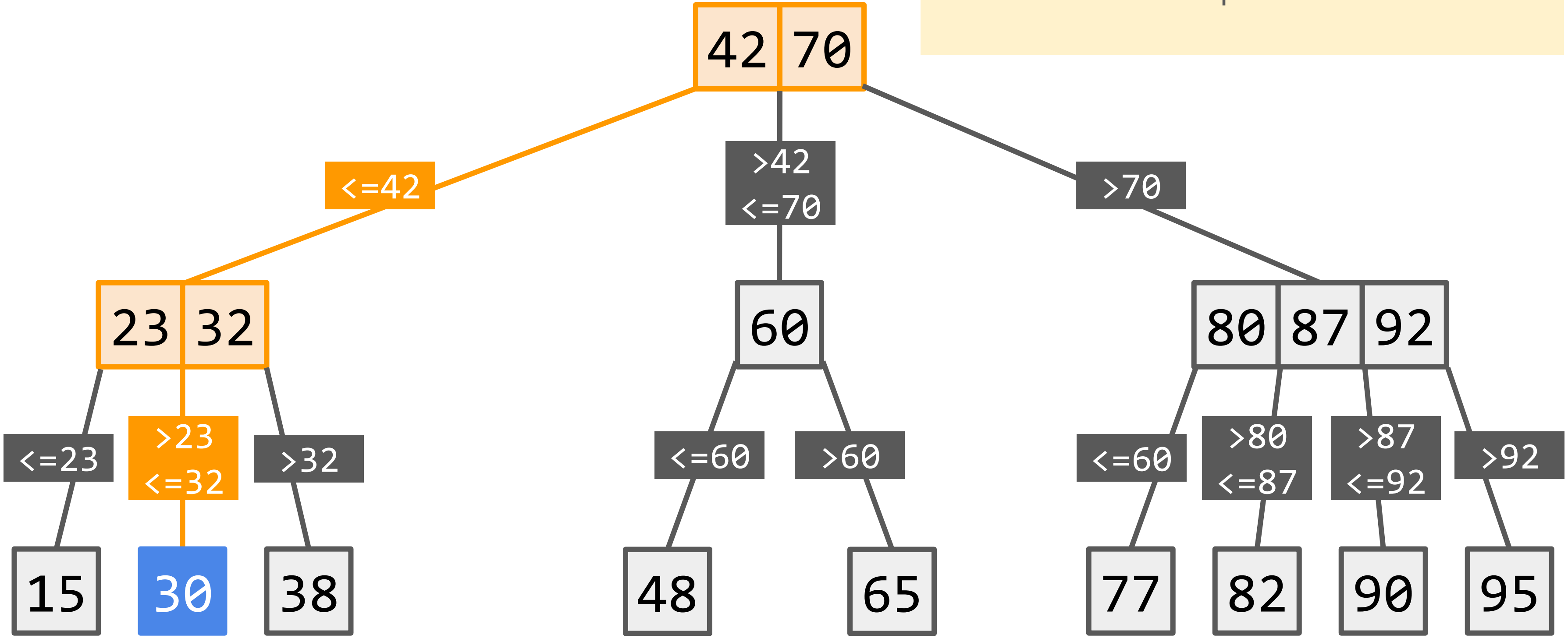
(2,4)-tree — `delete(73)`

- This simple search and delete approach *seems* to work
- So far so good!



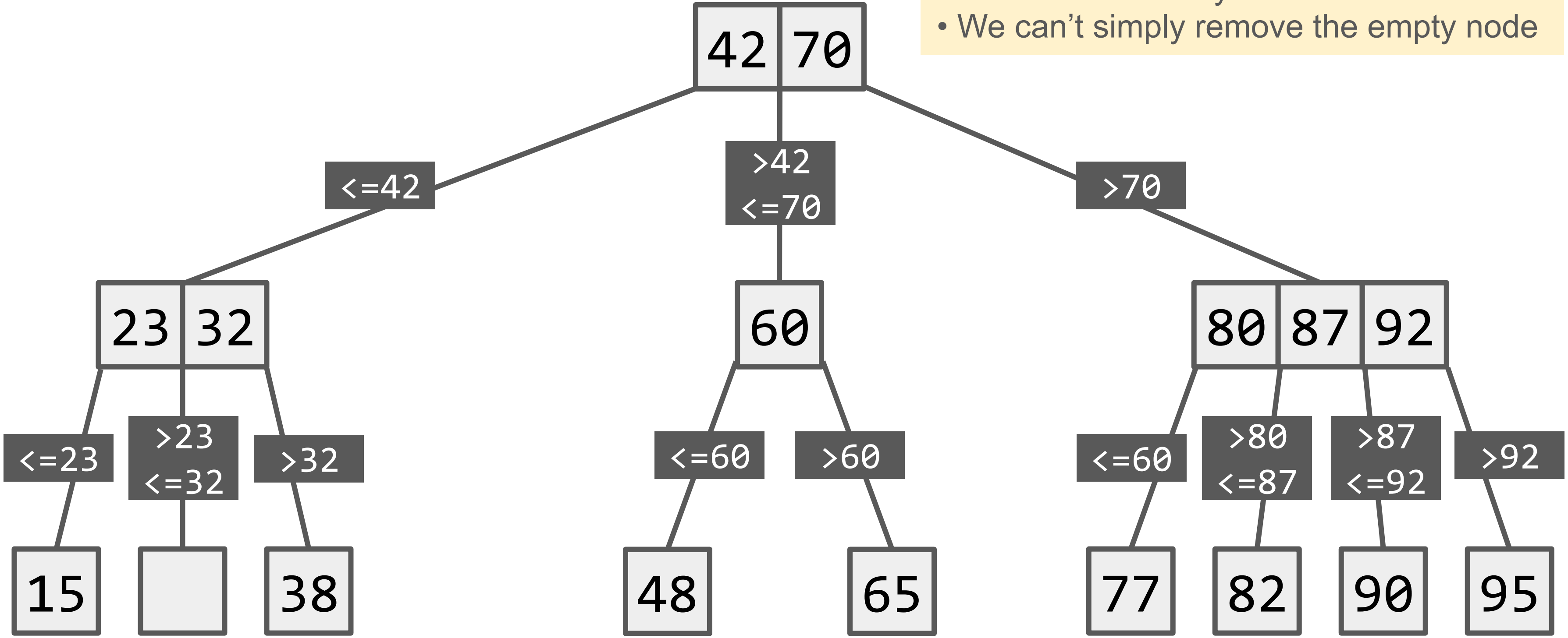
(2,4)-tree — `delete(30)`

- Let's now delete `30`
- This is the search path
- `30` is found so we proceed to its deletion



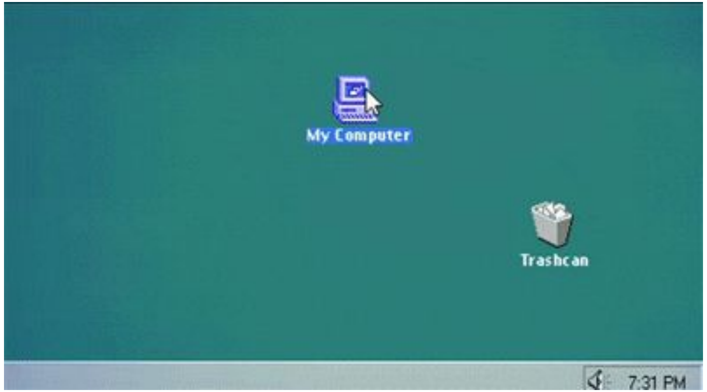
(2,4)-tree — delete(30)

- Ok now what?
- Now we have a node with 0 keys
- A node without keys is still a node!
- We can't simply remove the empty node



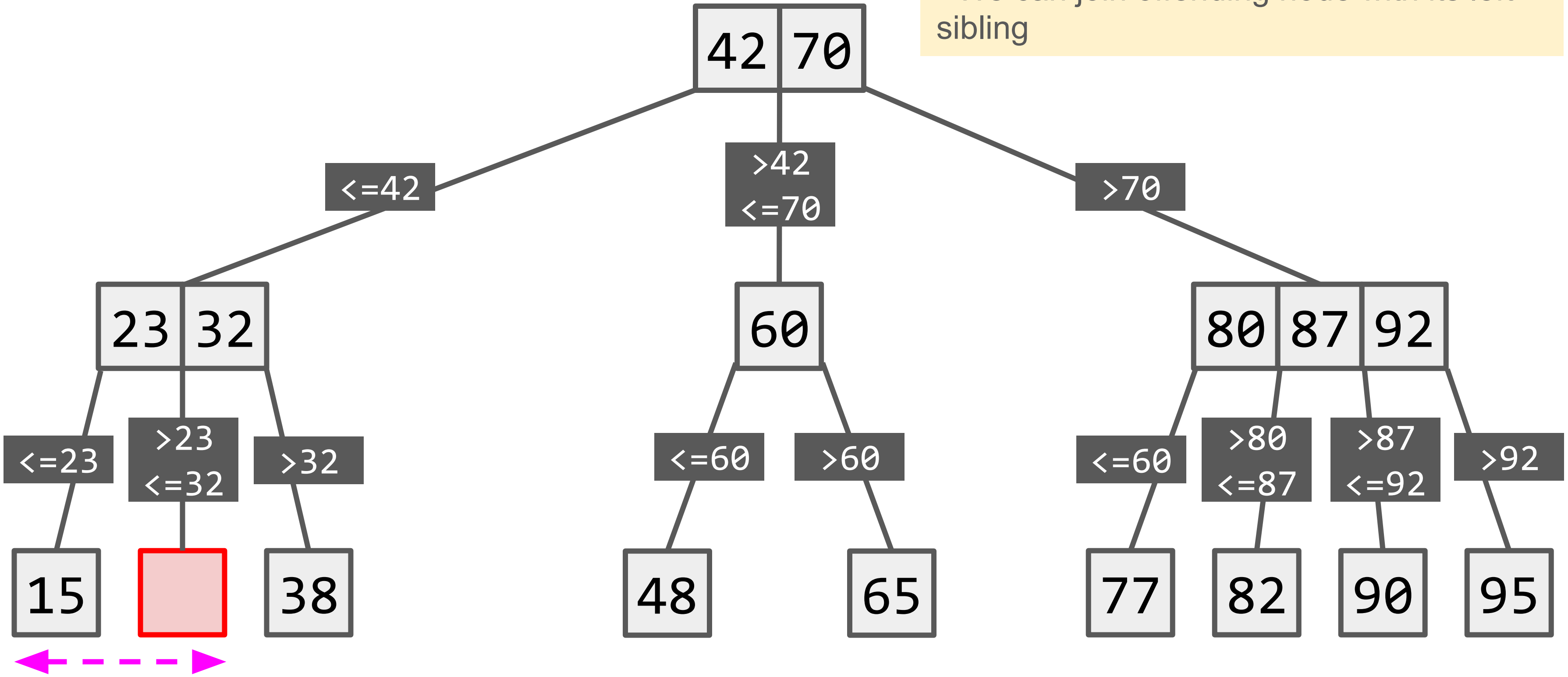
Deletion may violate rule 1!

- Recall rule 1 enforces that non-root nodes must have:
  - Number of children at least a
  - Or equivalently number of keys at least $a-1$
- Deletion may cause internal nodes to shrink too small
- We need to have an operation to handle such cases
- Key idea: Join with an adjacent sibling!



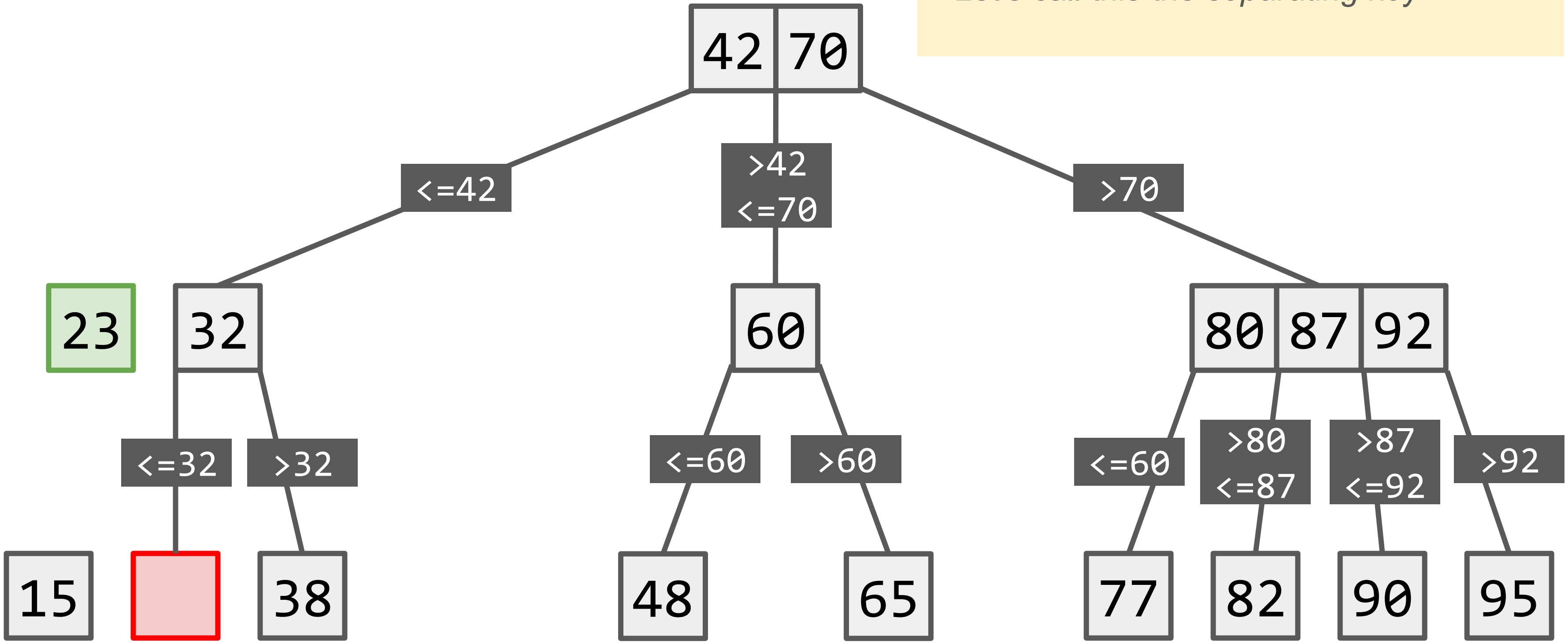
(2,4)-tree — delete(30)

- Offending node is the empty leaf node because it has less than $a-1$ keys
- We can join offending node with its left sibling



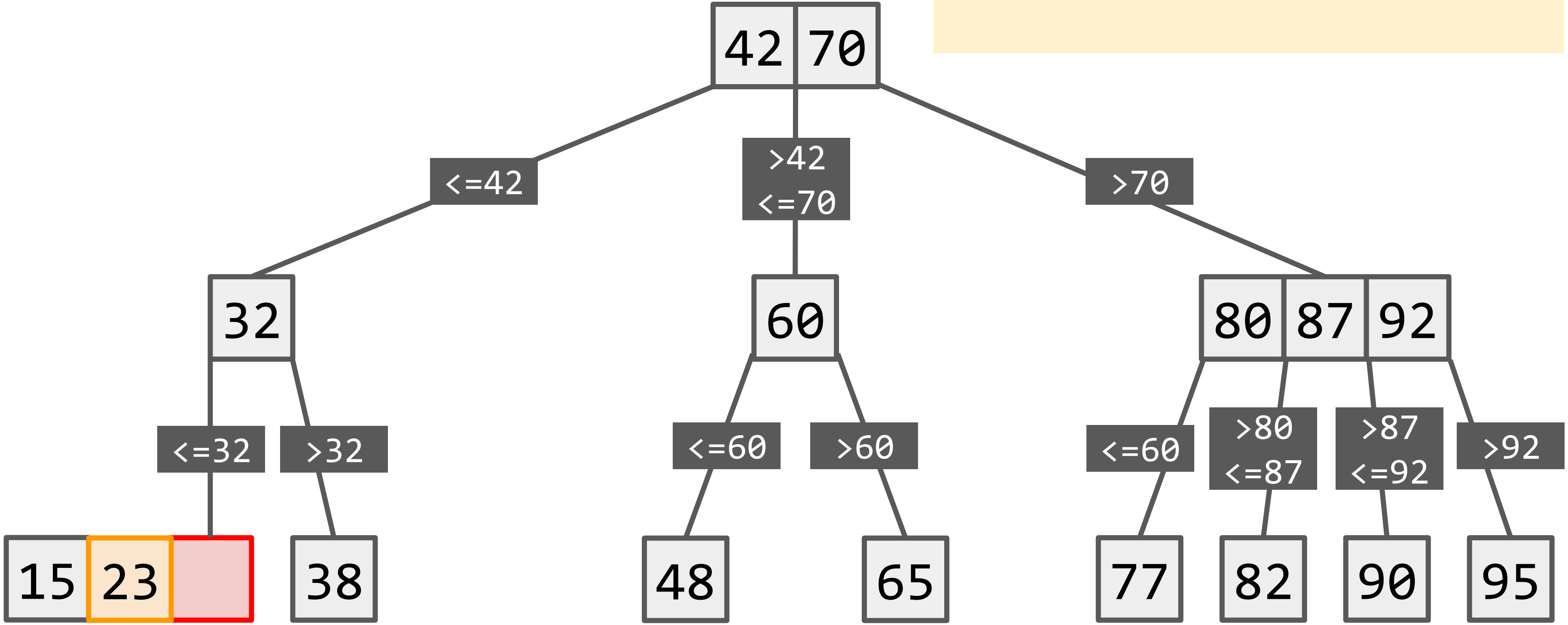
(2,4)-tree — `delete(30)`

- Remove the key separating the siblings from the parent's keylist
- Let's call this the *separating key*



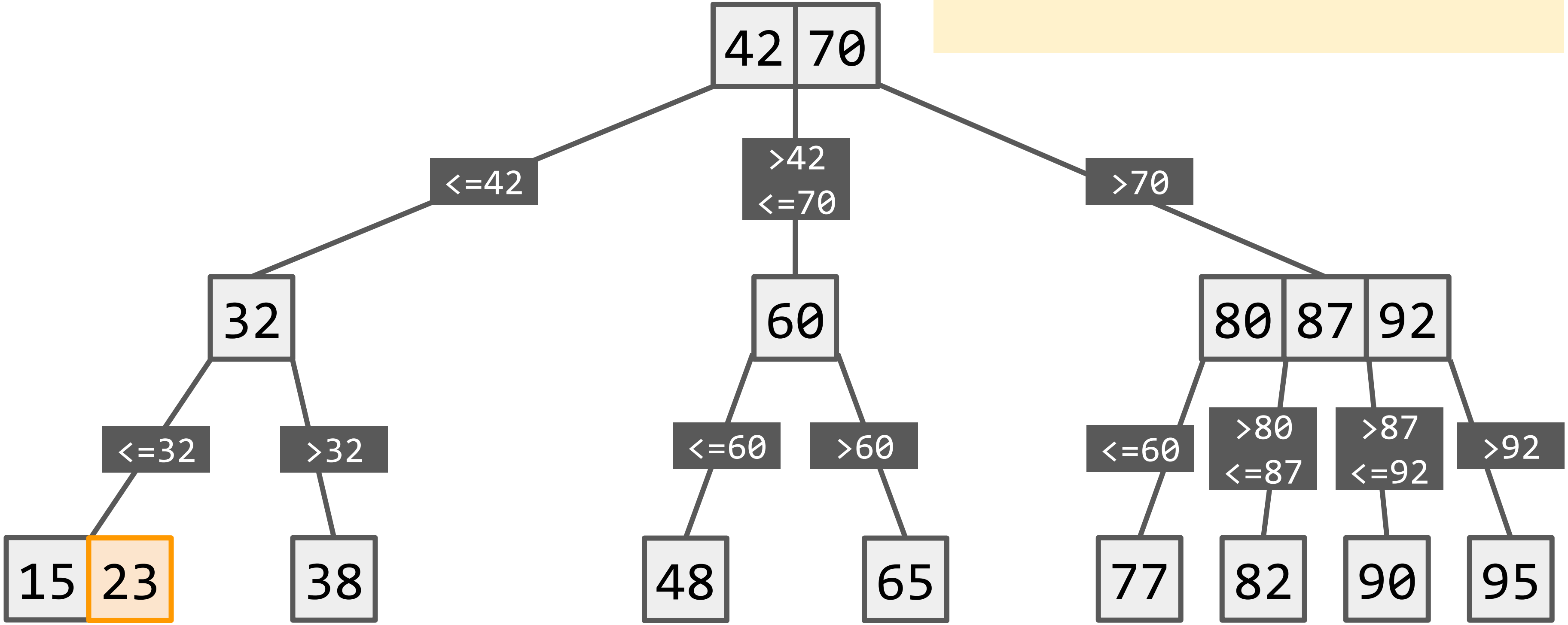
(2,4)-tree — delete(30)

- Join siblings using the separating key



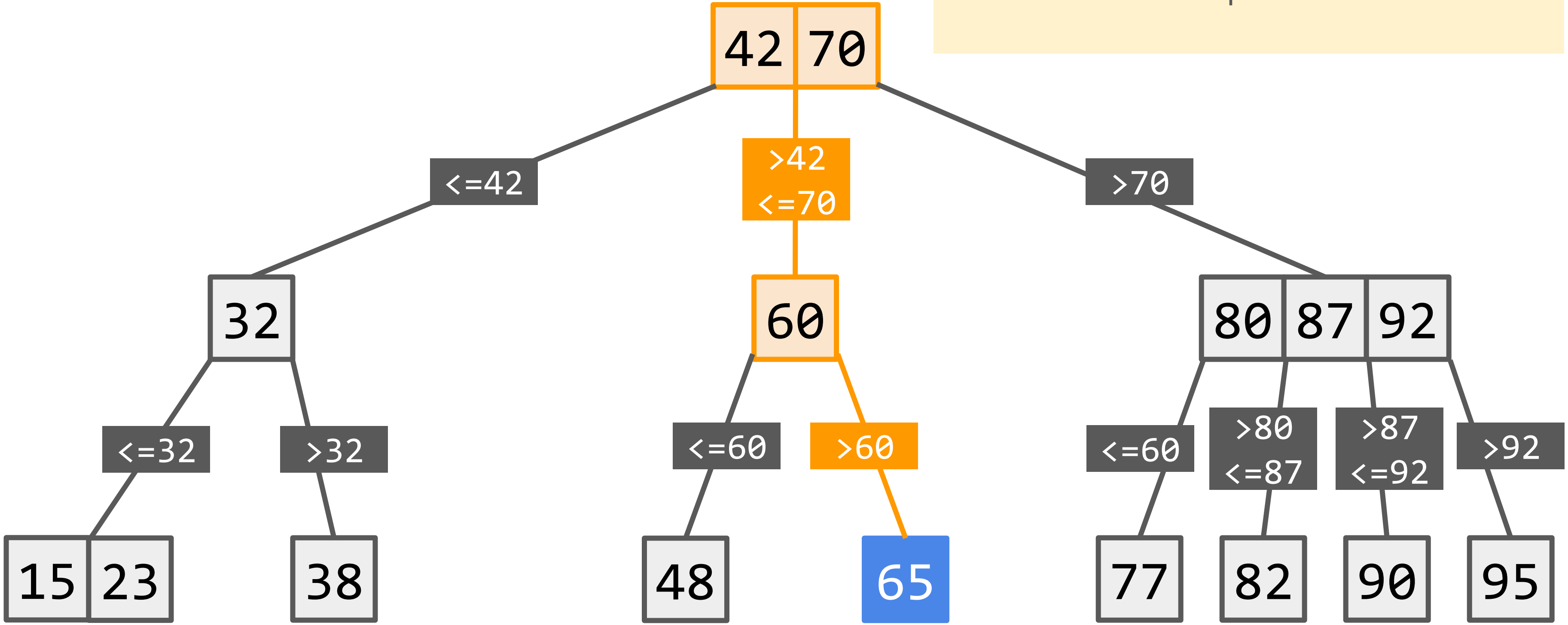
(2,4)-tree — delete(30)

- Since keylist of offending node is empty in this case, we can omit it in the drawing



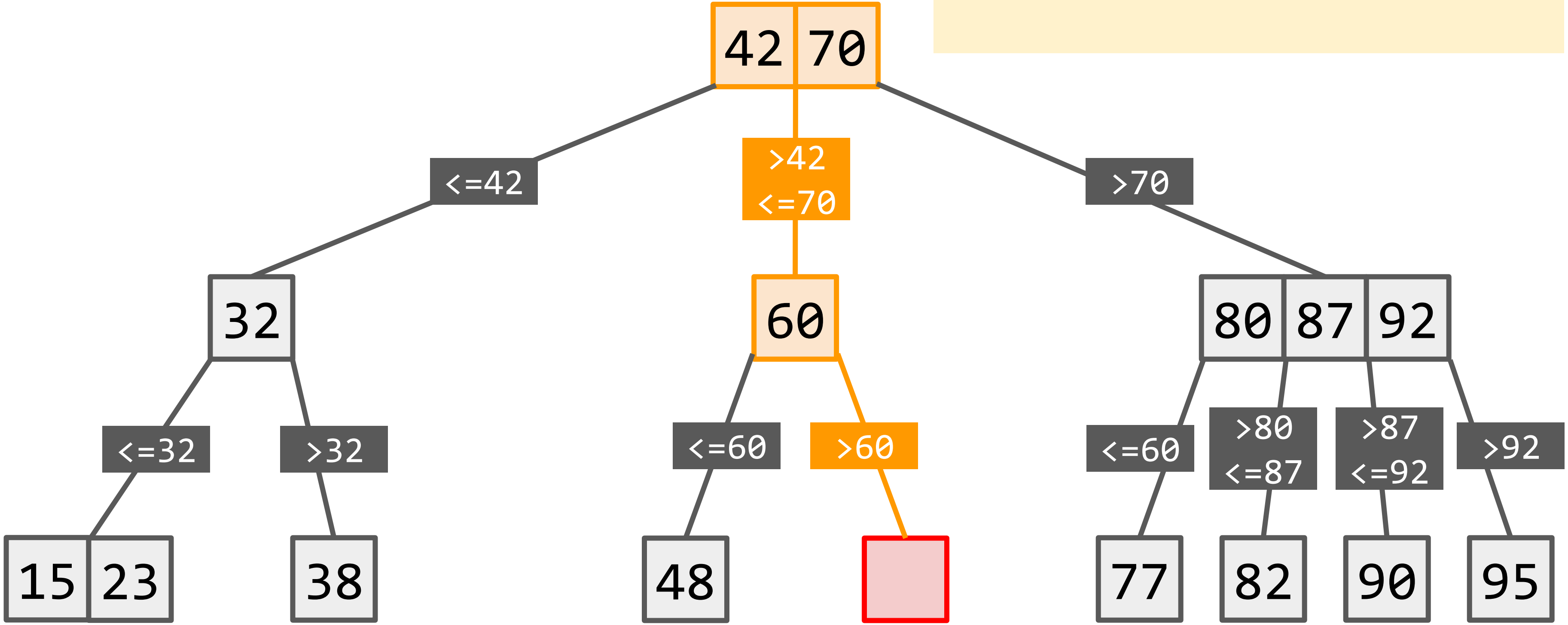
(2,4)-tree — `delete(65)`

- Let's now delete `65`
- This is the search path
- `65` is found so we proceed to its deletion



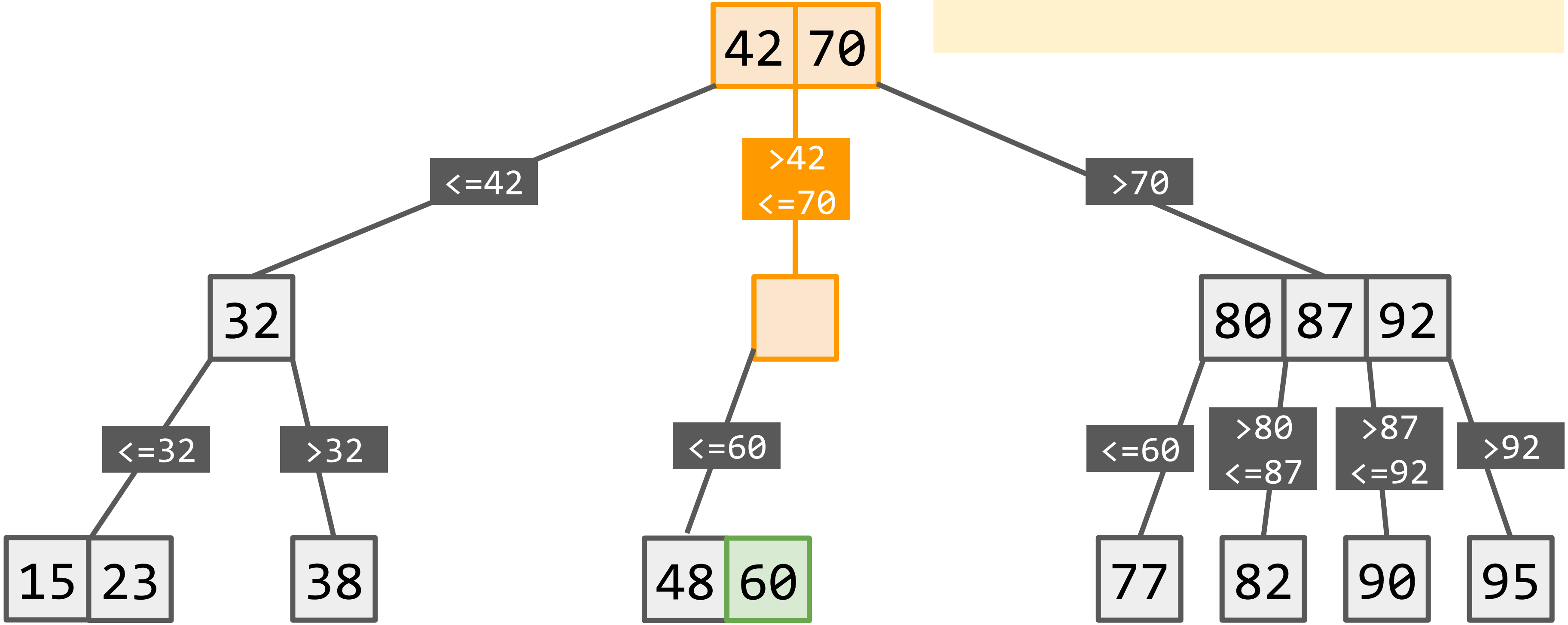
(2,4)-tree—delete(65)

- Delete 65 and we found that it’s offending rule 1



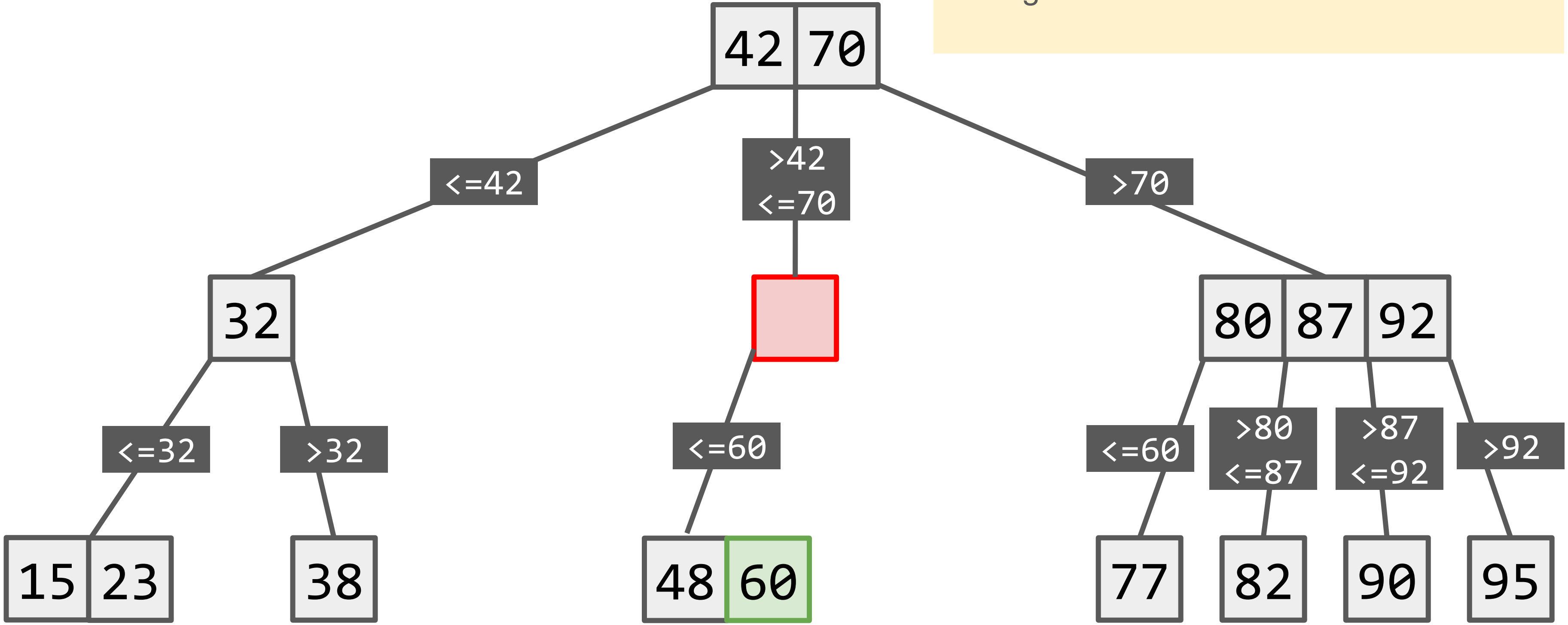
(2,4)-tree — delete(65)

Now merge the offending node with its sibling and demote a key from its parent



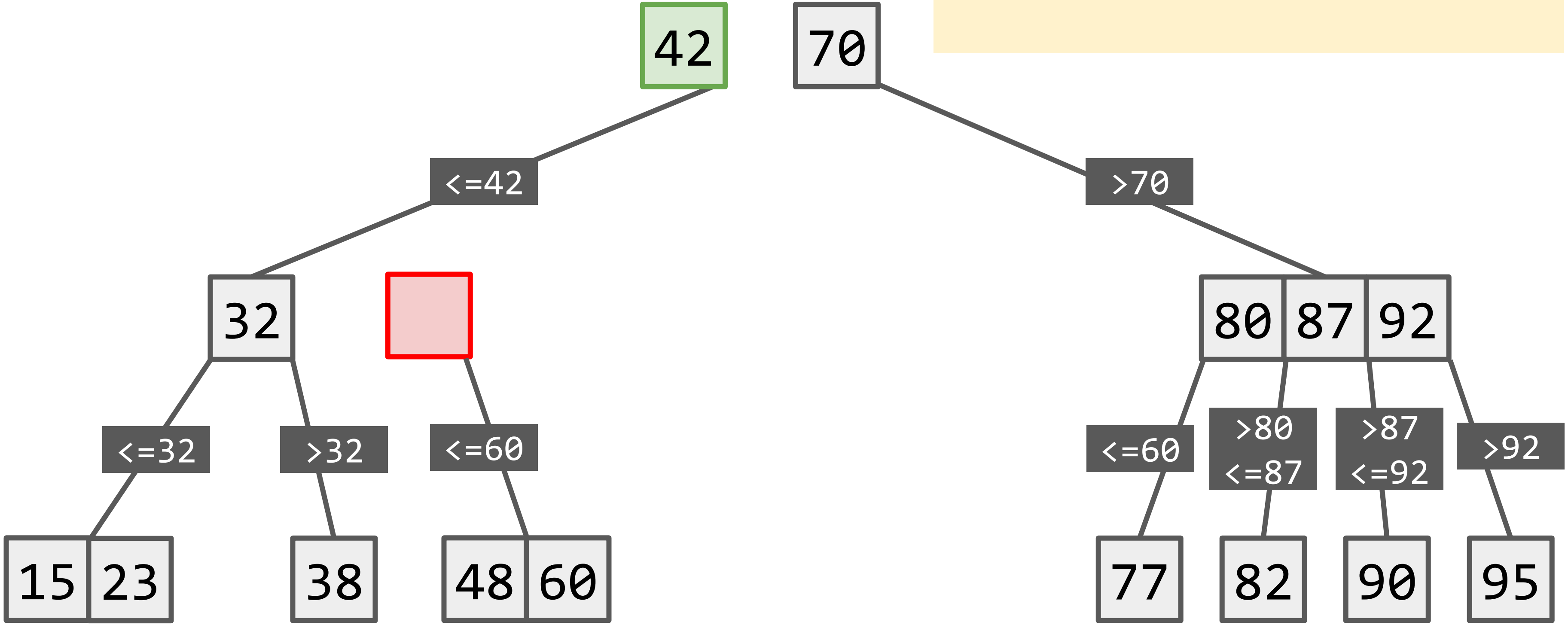
(2,4)-tree — delete(65)

But the parent now offends rule 1.
Need to merge the offending node with its sibling



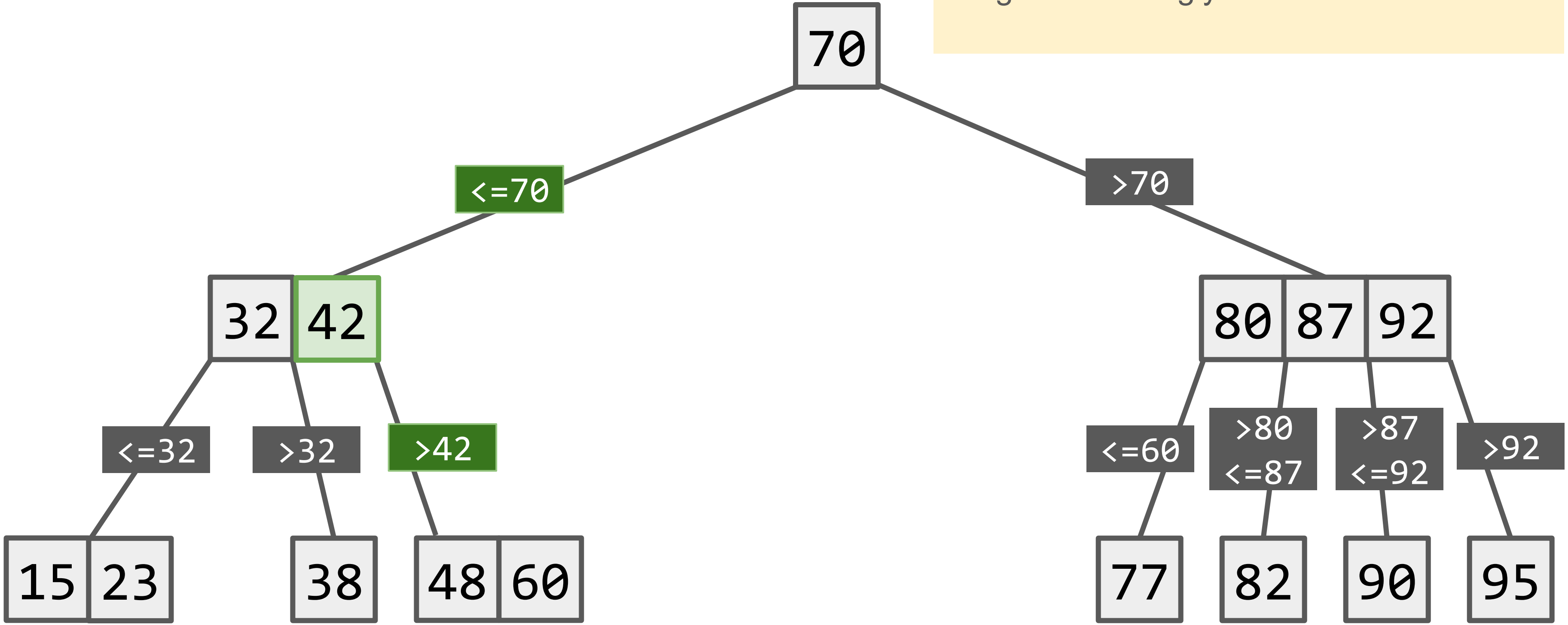
(2,4)-tree — delete(65)

Choose the separating key from the nodes' parent to demote



(2,4)-tree — delete(65)

Demote a key from the offending node's parent to make a new key and update ranges accordingly



Realize something

Intuitively `merge` operation is the reverse of `split` operation (this shouldn't come as any surprise!)

- In `split`, we *promote* the median key to parent to form LHS and RHS nodes
- In `merge`, we *demote* the key in parent separating LHS and RHS to join them together

Guiding question

Why must `merge` bring down one key from the parent? Why can’t we just join the 2 children directly?

Guiding question

Why must `merge` bring down one key from the parent? Why can't we just join the 2 children directly?

Answer: After merging, the parent will have one child less than before, therefore it must have one key less after the operation. Moving that key into the newly merged node is convenient.

Guiding question

Are there possible violations that can result from a `merge` operation?

If so, what are they?

Guiding question

Are there possible violations that can result from a `merge` operation?

If so, what are they?

Answer: We just need to consider which nodes are gaining or losing a key. Whenever a node gains a key, we risk it having too many keys, whenever a node loses a key, we risk it having too little keys. So the newly merged node may get too large whilst the parent may get too small.

Guiding question

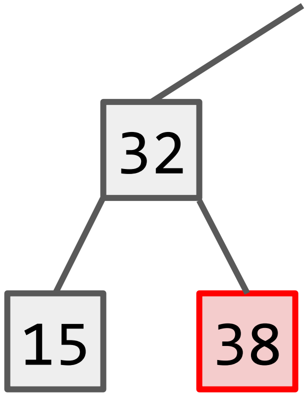
How can a `merge` operation cause a *parent* node to be “too small” (have less than $a - 1$ keys)? How should we resolve it?

Guiding question

How can a `merge` operation cause a *parent* node to be "too small" (have less than a - 1 keys)? How should we resolve it?

Answer: Consider deleting 38 from a (2,4)-tree in the scenario on the right.

We can resolve this by continually merging up the ancestors until we stop encountering the violation.



Challenge yourself!

What happens when a `merge` operation causes the root node to become “too small”?

Hint: What does it mean when the root node is “too small”?

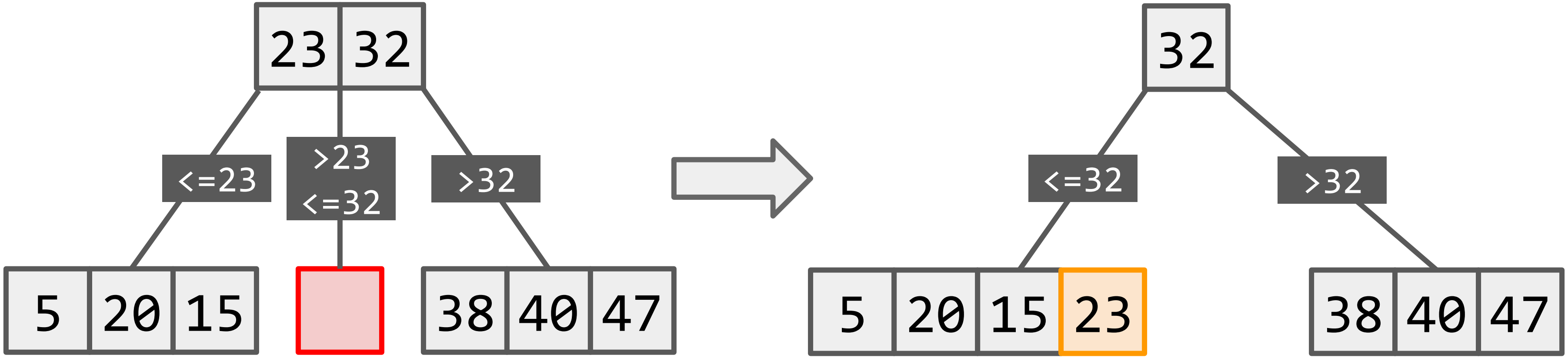
Guiding question

How can a `merge` operation cause the *merged* node to be "too large" (have more than $b - 1$ keys)?

Guiding question

How can a `merge` operation cause the *merged* node to be "too large" (have more than $b - 1$ keys)?

Answer: Yes. Consider this $(2,4)$-tree just after a deletion:



Guiding question

How do we resolve the case for when the merged node has more than b - 1 keys?

Guiding question

How do we resolve the case for when the merged node has more than b - 1 keys?

Answer: `merge` them, then `split` them!

This is known as a `share` operation.

Deletion—summary of violation resolutions

Scenario (post-deletion, just before key from parent is demoted)	**Operation**	**Algorithm**
• y and z are siblings • Have $< b-1$ keys together	`merge(y,z)`	1. In parent, delete key v (the key separating the siblings) 2. Add v to the keylist of y 3. In y, merge in z's keylist and treelist 4. Remove z
• y and z are siblings • Have $\geq b-1$ keys together	`share(y,z)`	1. `merge(y,z)` 2. Split newly combined node using the regular split operation

Guiding question

Can a `share` operation ever cause a *sibling* node to have keylist size lower than $a - 1$?

Guiding question

Can a `share` operation ever cause a *sibling* node to have keylist size lower than a - 1?

Answer: No, because

- Sibling nodes have at least b - 1 keys combined to trigger a `share` operation
- So after merging, they have at least a combined b keys together (including one demoted from parent)
- Realize this is also the preconditions of `split`
- We have proven earlier that given the correct preconditions of splitting, its postconditions will not lead to further violations of the split nodes
- Thus this also proves that `share` is always a safe a operation on (a,b)-trees once its preconditions are met

Proactive vs passive strategy

Alike insertion, proactive and passive approaches for deletion:

1. *Proactive*: preemptively `merge`/`share` any node at minimal capacity (i.e. $a - 1$ keys) during search phase
2. *Passive*: lazier strategy whereby deletion is done first and then check parent for violation (potentially `merge`/`share` all the way to the top)

Deletion algorithm outline

```
Delete(BTree, x)
w ← Search(BTree, x)
if w is not leaf then
    p_w, p_x ← GetPredecessor(w, x)        // p_w: predecessor node, p_k: predecessor key
    SwapKeys(w, x, p_w, p_x)
    w ← p_w
DeleteKey(w, x)
while true do
    if w contains < a - 1 keys then        // If rule 1 violated, merge/share
        z ← GetSmallestSibling(w)
        if w and z together contain < b - 1 keys then
            w ← Merge(w, z)                // Set w to be the newly merged node
        else                               // w and z together contain ≥ b - 1 keys
            s_L, s_R ← Share(w, z)
            w ← s_L
    else
        break                              // End routine once no more violations
    if w is not root then
        w ← GetParent(w)                   // Recurse upwards
    else
        break                              // End routine once handled root
```

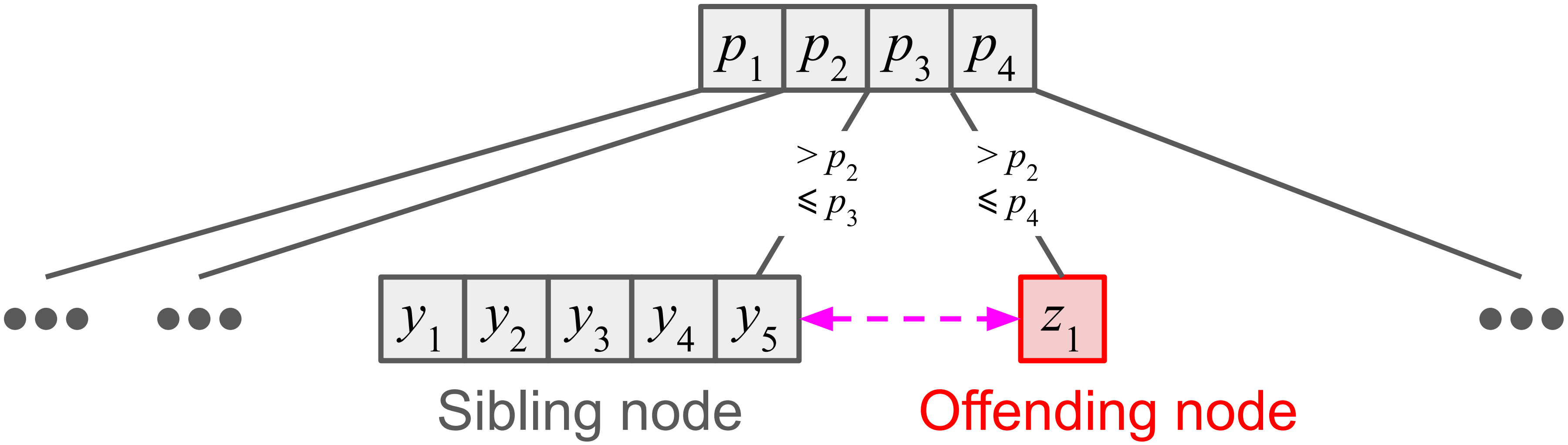
Passive approach

Problem 1.h.

Come up with an example each for `merge` and `share` operations, as well as a key deletion. Prove that for a `merge` or `share`, the necessary preconditions are satisfied before the operation, and that the three rules of an (a,b)-tree are satisfied afterwards. Thereafter, prove that after a delete operation, the three rules of an (a,b)-tree are satisfied.

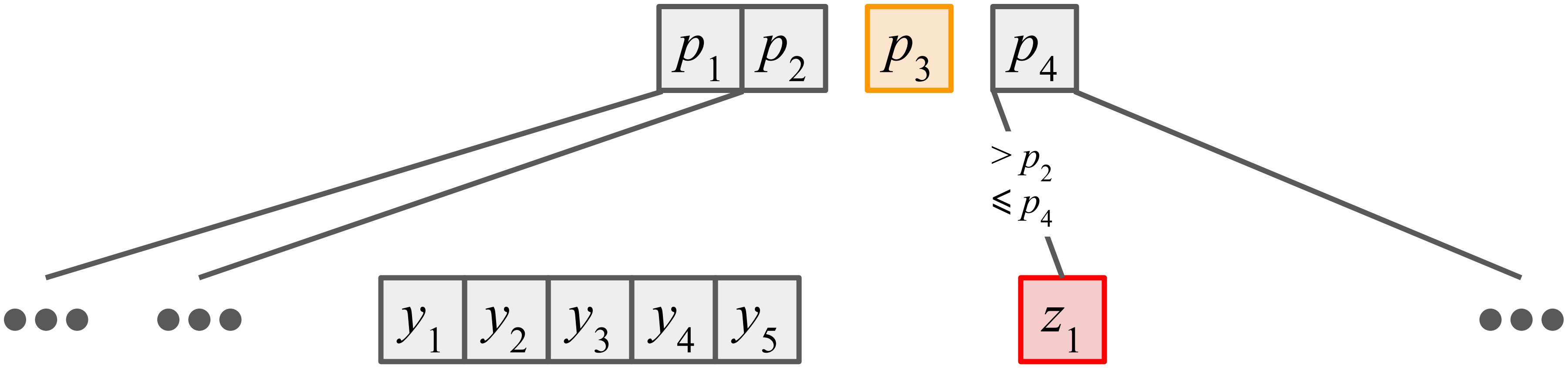
What is the cost of deleting from an (a,b)-tree?

$(3,6)$-tree — `share` operation



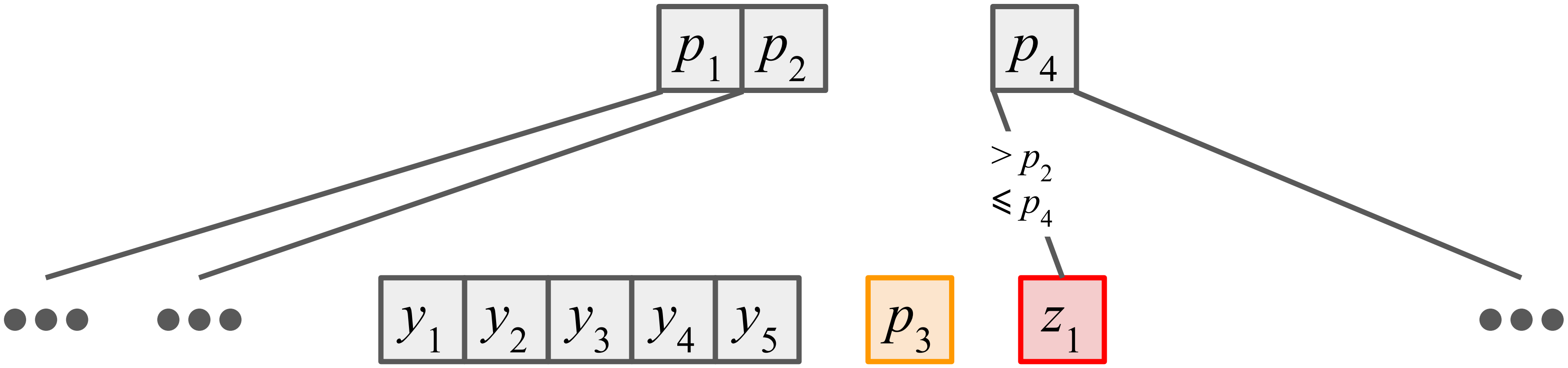
- z is the offending node (must have at least 2 keys in a $(3,6)$-tree)
- We first `merge(`y`,`z`)`

(3,6)-tree — share operation



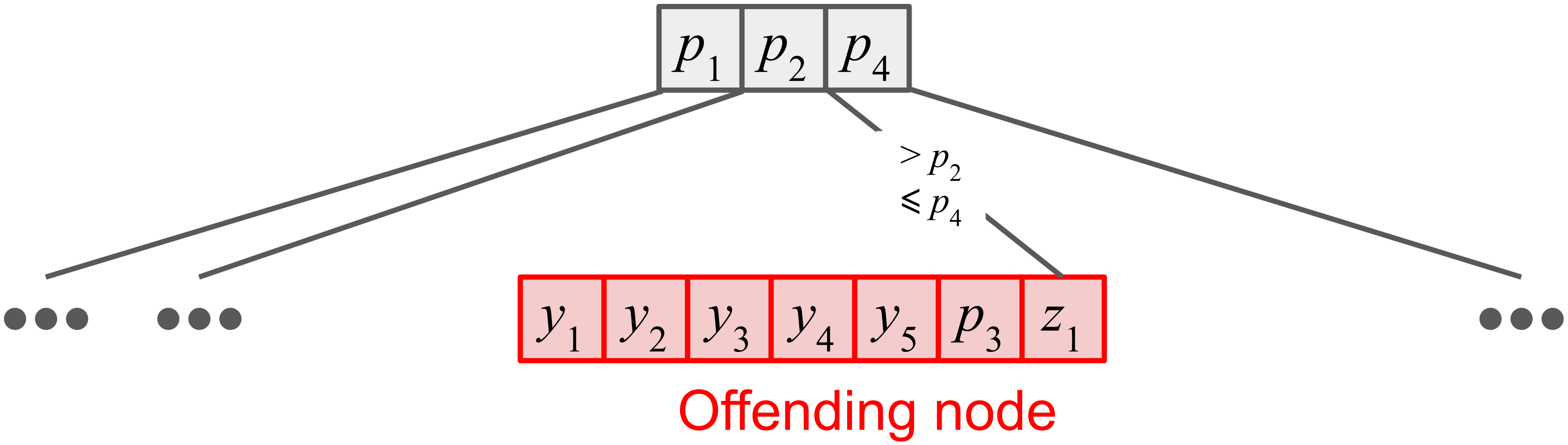
- p_3 is the key in parent node separating nodes y and z

(3,6)-tree — share operation



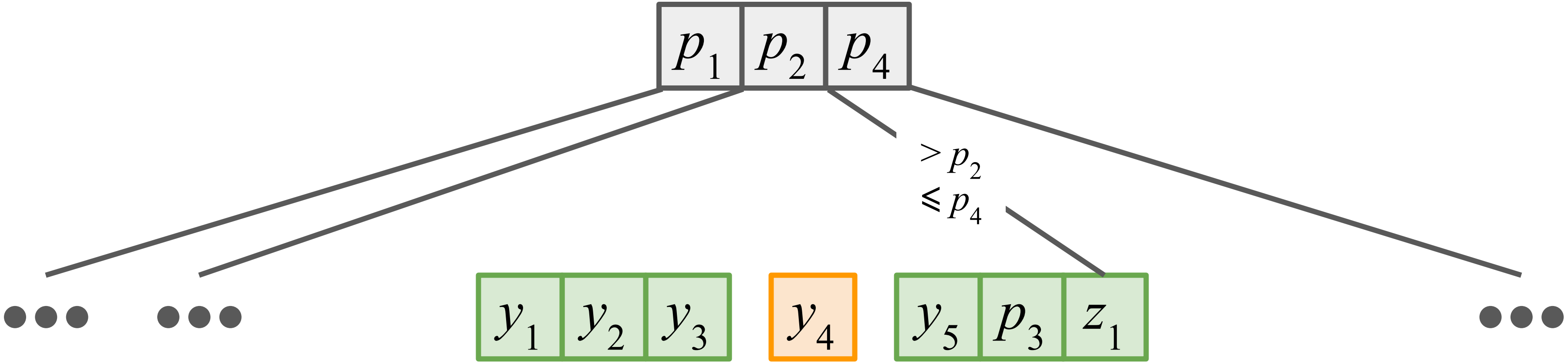
- p_3 shall be now used as the “interfacing” for joining nodes y and z

(3,6)-tree — share operation



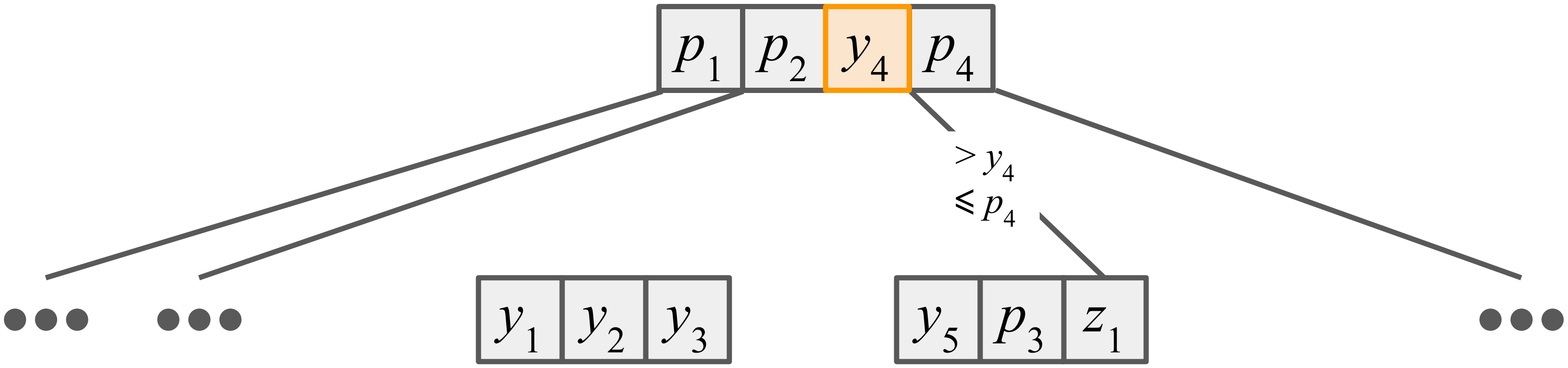
- But alas! After merging, our new node becomes an offending node (can have at most 5 keys in a (3,6)-tree)!

(3,6)-tree — `share` operation



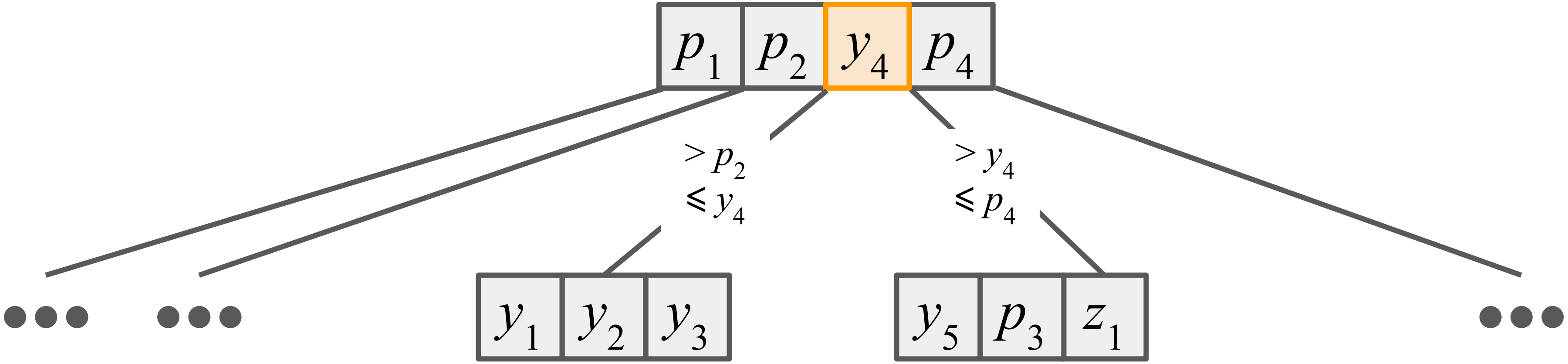
- We resolve that violation using the `split` operation
- The split key is chosen from median y_4

(3,6)-tree—share operation



• Promote y_4 and move it into parent, inserting it after p_2, the key corresponding to the violating node

(3,6)-tree — `share` operation



- Associate y_4 to the orphaned LHS subtree

Guiding question

Our discussions here only concerned unique values (i.e. no duplicates). How can you easily extend the operations to also cater for duplicate values?

Guiding question

Our discussions here only concerned unique values (i.e. no duplicates). How can you easily extend the operations to also cater for duplicate values?

Answer: Instead of just storing keys, store a pair of (key, insertion order) where insertion order refer to the order when said key is inserted (i.e. a timestamp). How should you compare such a pair of values?

Challenge yourself!

Come up with examples for deleting a key from an internal node and the root node respectively. How should the operations be?

Hint: The building blocks for any operations are split and merge.

Wait a minute!

So searching in a B-tree takes $O(\log n)$, but so does searching in a BST!

Why so special about B-trees then?

Problem 2

Description

- In general, data is stored on disk in blocks, e.g., $B_1, B_2, \dots, B_m$
- Each block stores a chunk of memory of size B
- Think of each block as an array of size B
- When accessing a memory location in some block B_j, the entire block is read into memory
- You can assume that your memory can hold some M blocks at a time
- At this level, to measure the time of manipulating data on disk, we need only count the number of blocks we have to move from disk to memory
- We can ignore the cost of accessing a block once it is in memory

Access times

Memory unit	Size	Block size	Access time (clock cycles)
L1 cache	64KB	64B	4
L2 cache	256KB	64B	10
L3 cache	Up to 40MB	64B	40—74
Main memory	128GB	16KB	200—350
(Solid-State) Drive	Arbitrarily big	16KB	20,000
(Magnetic Disk) Drive	Arbitrarily big	16KB	20,000,000

Realize that disk access time is an eternity compared to memory access times! Thus a block transfer would serve as a unit of time at this level of discussion since it dominates. Operations within memory can be treated as instantaneous.

Problem 2.a.

Assume your data is stored on disk.

Your data is a sorted array of size n, and spans many blocks.

What is the number of block transfers needed (i.e. cost) of doing a *linear search* for an item?

Leave your answer in terms of n and B.

Problem 2.a. — Solution

Assume your data is stored on disk.

Your data is a sorted array of size n, and spans many blocks.

What is the number of block transfers needed (i.e. cost) of doing a *linear search* for an item?

Leave your answer in terms of n and B.

Answer: n/B

Problem 2.b

Assume your data is stored on disk.

Your data is a sorted array of size n, and spans many blocks.

What is the number of block transfers needed (i.e. cost) of doing a *binary search* for an item?

Leave your answer in terms of n and B.

Problem 2.b—Solution

Assume your data is stored on disk.

Your data is a sorted array of size n, and spans many blocks.

What is the number of block transfers needed (i.e. cost) of doing a *binary search* for an item?

Leave your answer in terms of n and B.

Answer: $\log \lceil n/B \rceil$ block transfers. You can think of this as doing a binary search on the n/B blocks. Once your search finds the right block, it is loaded into memory and the rest of the search is free.

Description

Now, imagine you are storing your data in a B-tree. (Notice that you might choose $a=\mathrm{B}/2$ and $b=\mathrm{B}$, or $a=\mathrm{B}/4$ and $b=\mathrm{B}/2$, etc., depending on how you want to optimize.)

Notice that each node in your B-tree can now be stored in some constant number of blocks, for example, one block stores the key list, one block stores the subtree links, and one block stores the other information (e.g., the parent pointer, pointers to the two other blocks, and any other auxiliary information).

Problem 2.c

What is the cost of searching a keylist in a B-tree node?

What is the cost of splitting a B-tree node?

What is the cost of merging or sharing B-tree nodes?

Problem 2.c—Solution

What is the cost of searching a keylist in a B-tree node?

What is the cost of splitting a B-tree node?

What is the cost of merging or sharing B-tree nodes?

Answer: They are now all $\mathrm{O}(1)$ since they require some constant number of block transfers followed by operation on the block in-memory.

Problem 2.d.

What is the cost of searching a B-tree?

What is the cost of inserting or deleting in a B-tree?

Problem 2.d. — Solution

What is the cost of searching a B-tree?

What is the cost of inserting or deleting in a B-tree?

Answer: These are all now $O(\log_B n)$ because

1. Each node is contained in a block. I.e., the cost only depends on the height of the tree
2. Cost of the in-memory operations on a single node is only $O(1)$.

Best-tree? Based-tree?

The important thing here is in the value of B. Searching in a BST is $O(\log_2 n)$ whilst searching in a B-tree is $O(\log_\mathrm{B} n)$. In practice, B is a pretty large number.

For instance. If your disk has 16KB blocks (which is reasonably normal) and you set $\mathrm{B{=}16K}$, then given a 10TB database, your B-tree requires 3 just levels! The root of your B-tree will always stay in memory. For typical memory sizes (e.g., 256MB disk cache), the first level of your B-tree will also always be in memory. Thus the cost of searching your 10TB database is typically one block transfer. For a 1000TB disk, you have four levels and two block transfers.

That's why it is really, really hard to beat a well-implemented B-tree!

Alternative design

- In our design, all nodes in (a,b)-trees stores keys
- In alternative designs, people use non-leaf nodes to stores *pivots* for navigating the search and store keys only at the leaves
- For such designs, the operations are mostly identical with the exception of split
- Just be mindful of this alternate scheme (also known as $\mathrm{B}+$-trees) when you encounter B-trees out in the wild

Further reading

(2,4)-trees map to Red-Black trees very nicely.

You may read up more on Red-Black trees here: https://brilliant.org/wiki/red-black-tree/