

CS2040S

Recitation 4
AY22/23S2

(a,b) -trees

(a,b) -tree overview

- Is an generalisation of Binary Search Trees ideas
- 2 parameters: a and b where $2 \leq a \leq (b+1)/2$
- 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

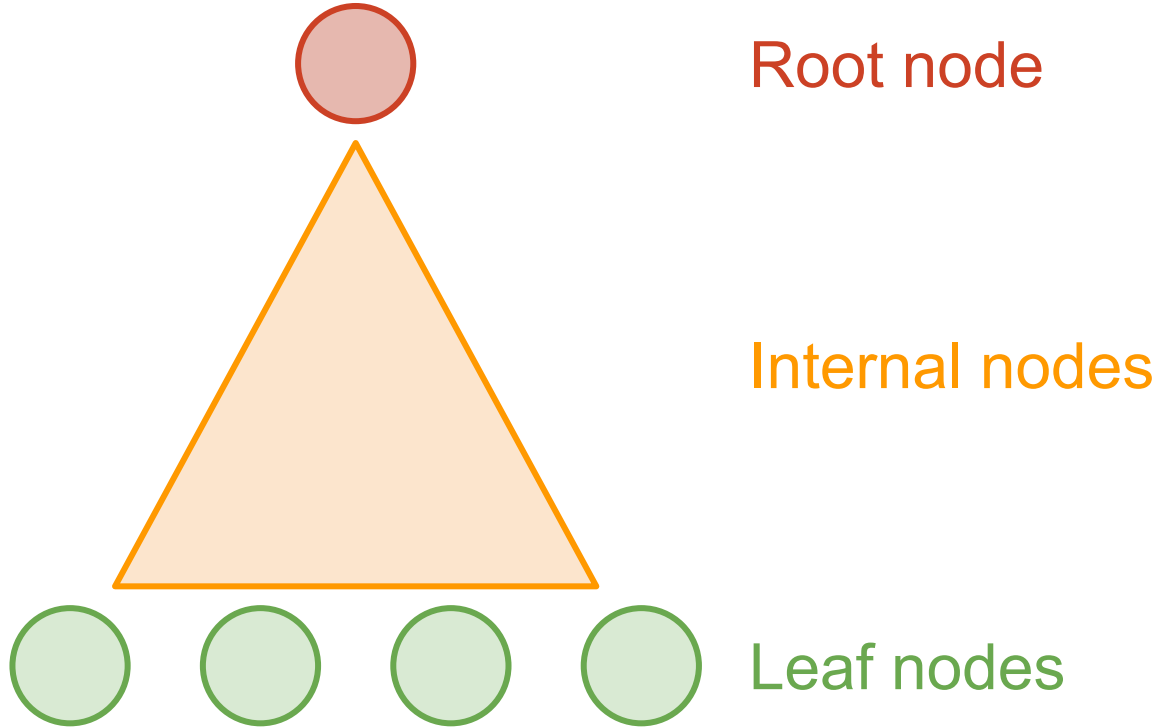
BST	(a,b) -tree
Each node has <i>at most 2</i> children	Each node <i>can have more than 2</i> children
Each node stores <i>exactly 1</i> key	Each node <i>can store multiple</i> keys



(a,b) -tree rules

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

Basic nomenclature



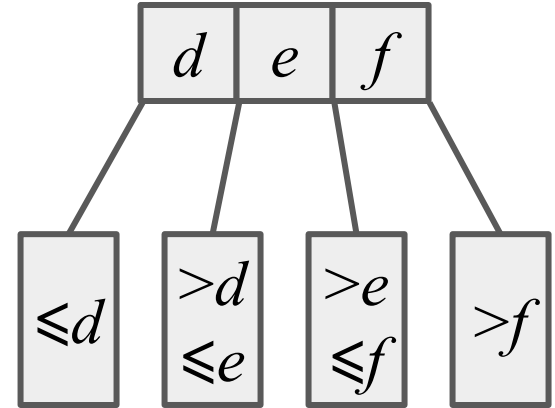
Rule 1 — “ (a,b) -child policy”

- **Root node** has $[2, b]$ number of children
 - Or equivalently $[1, b-1]$ number of keys
- **Internal nodes** has $[a, b]$ number of children
 - Or equivalently $[a-1, b-1]$ number of keys
- **Leaf nodes** has $[a-1, b-1]$ number of keys

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 **sorted** keys $[d, e, f]$ will entail 4 subranges
 1. $\leq d$
 2. $> d$ and $\leq e$
 3. $> e$ and $\leq f$
 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*

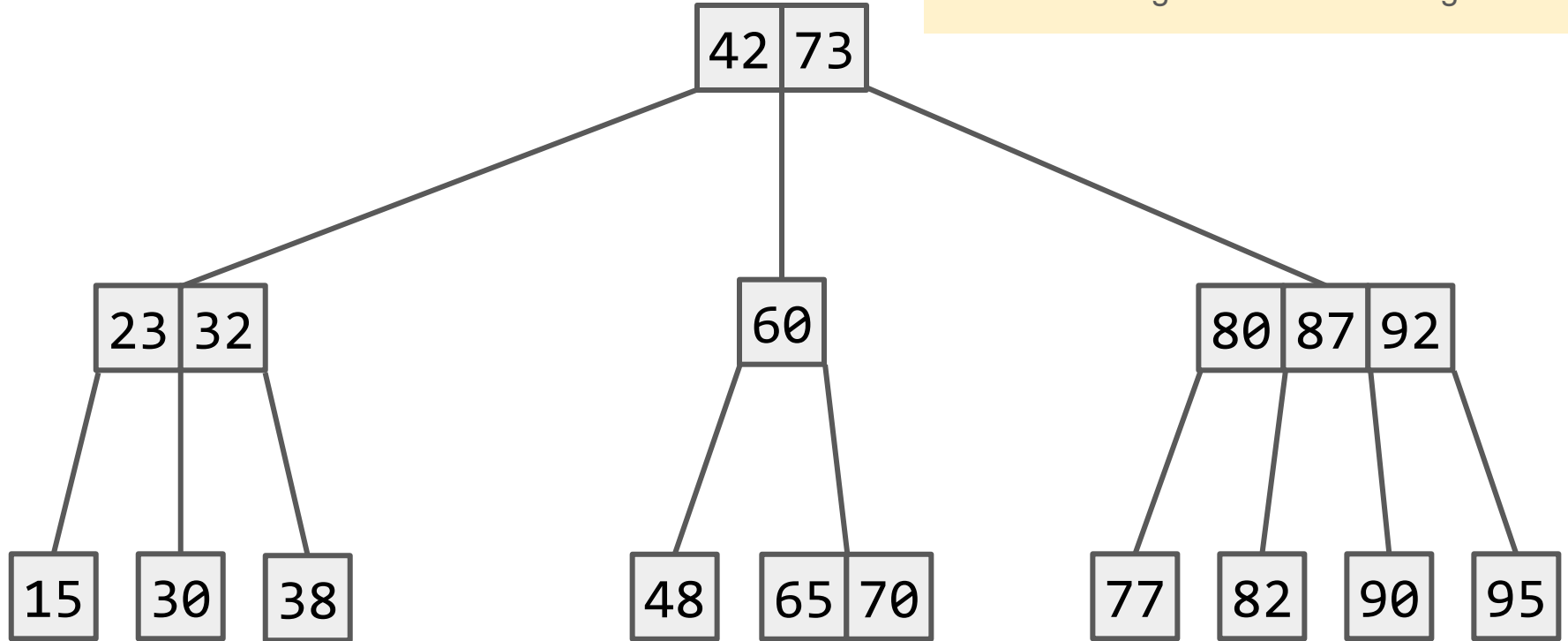
In addition, we shall always treat the keylist in ascending **sorted order**.

Rule 3—“Leaf depth”

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

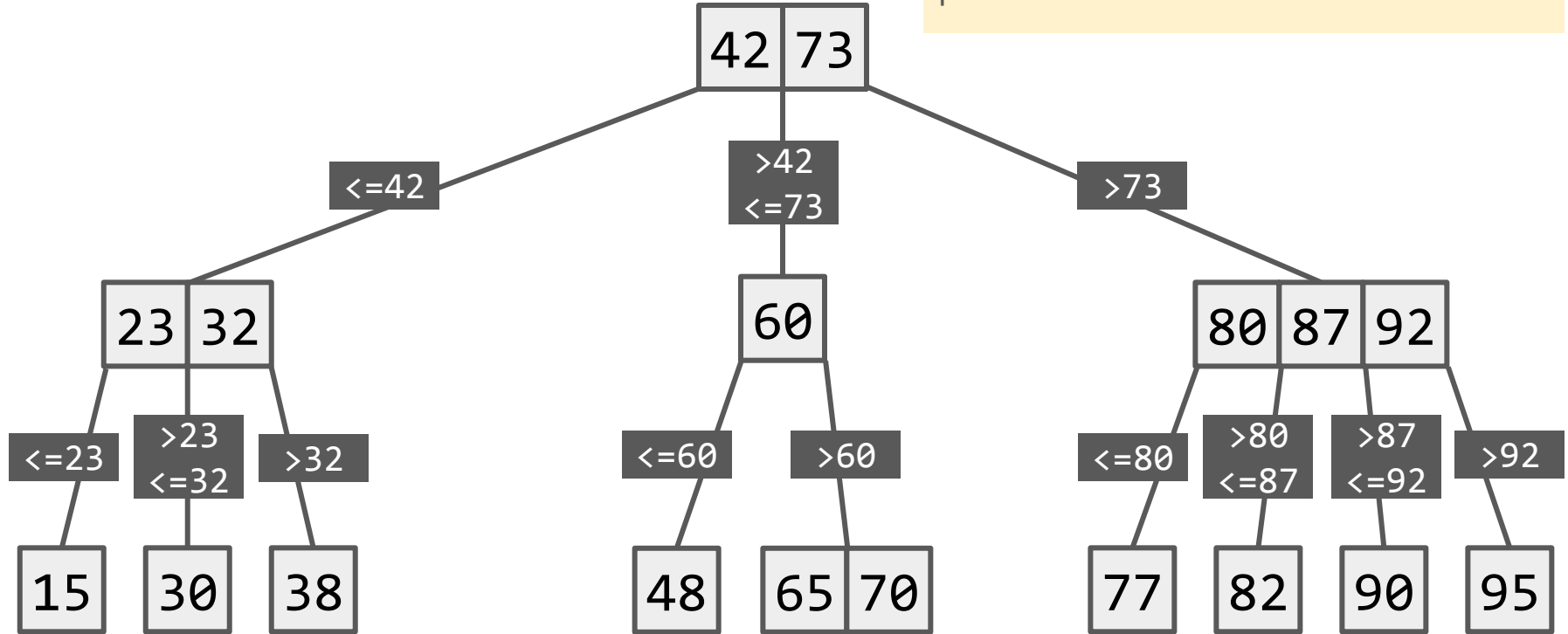
Example: (2,4)-tree

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



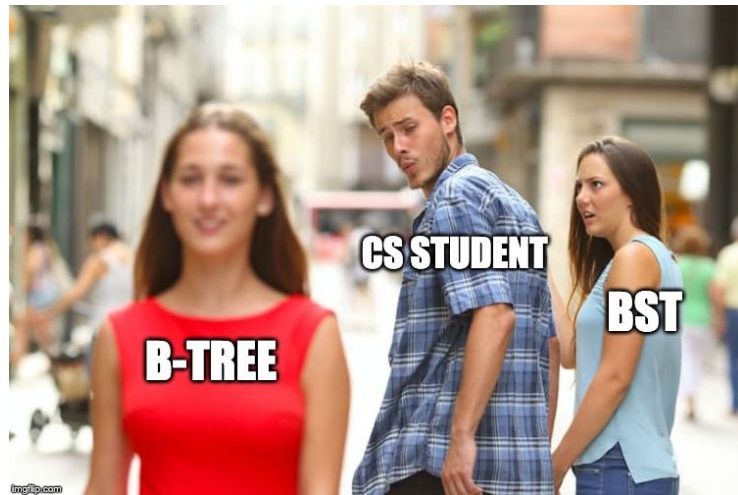
Example: (2,4)-tree

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 $a=B$ and $b=2B$
- B is a number we specify
 - E.g. if $B=2$, 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.b.

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

Problem 1.b.

Maximum height when,

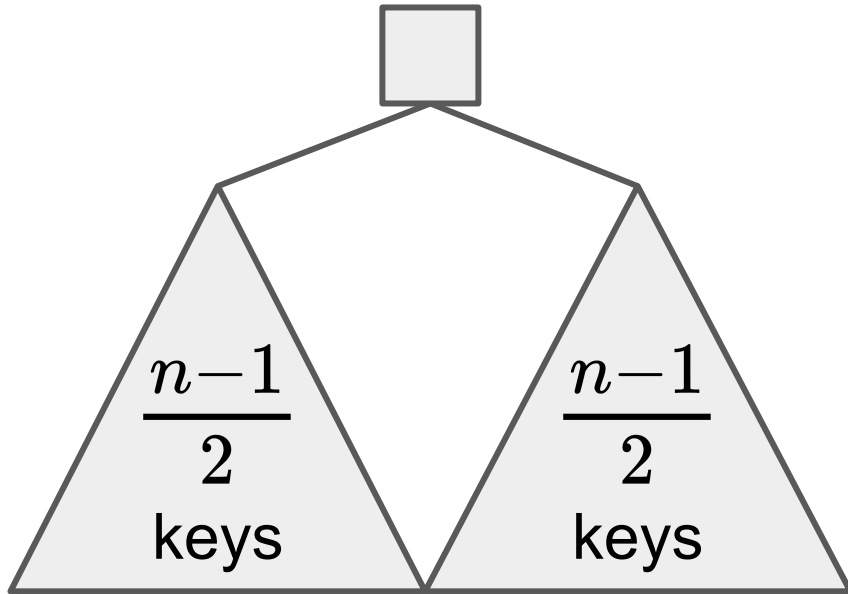
- Minimum branching factor of internal nodes (a children each)
- Equivalently, each non-root node has $a - 1$ keys
- Root node has 1 key and 2 children

Minimum height when,

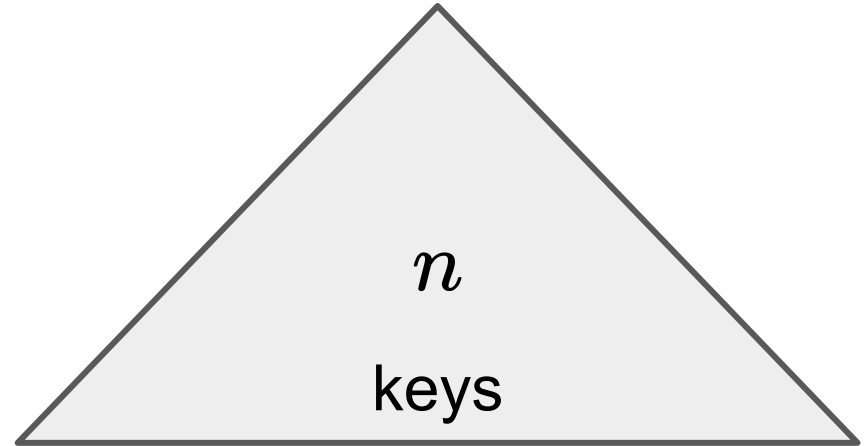
- Maximum branching factor of internal nodes (b children each)
- Equivalently, each non-root node has $b - 1$ keys

Problem 1.b.

Maximum height

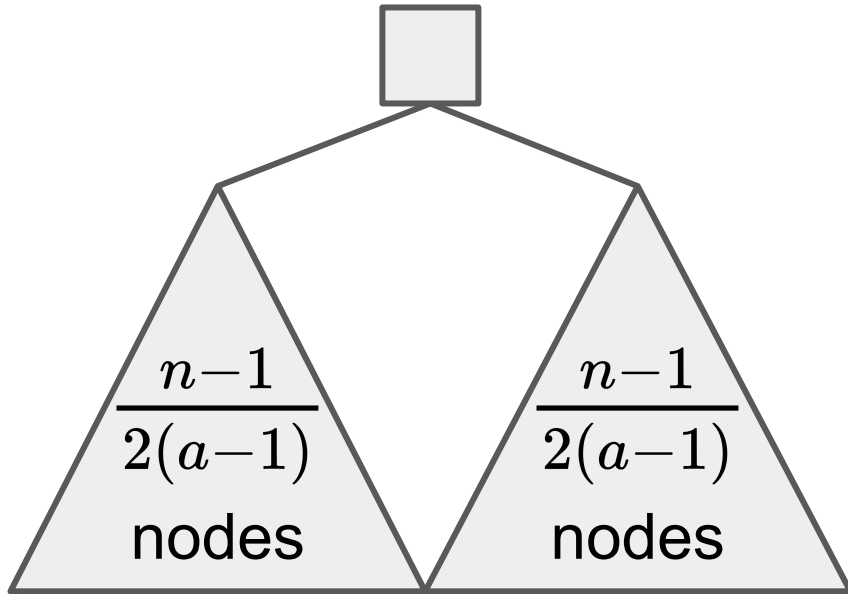


Minimum height

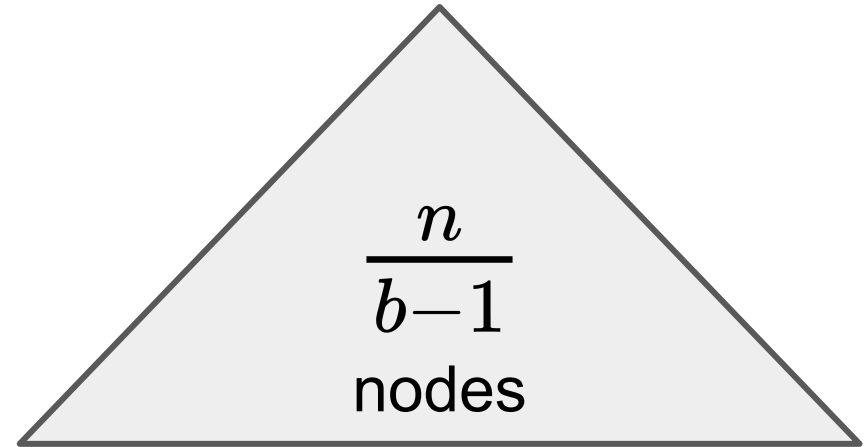


Problem 1.b.

Maximum height



Minimum height



Problem 1.b.

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

Maximum height: $O(\log_a n) + 1$

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?

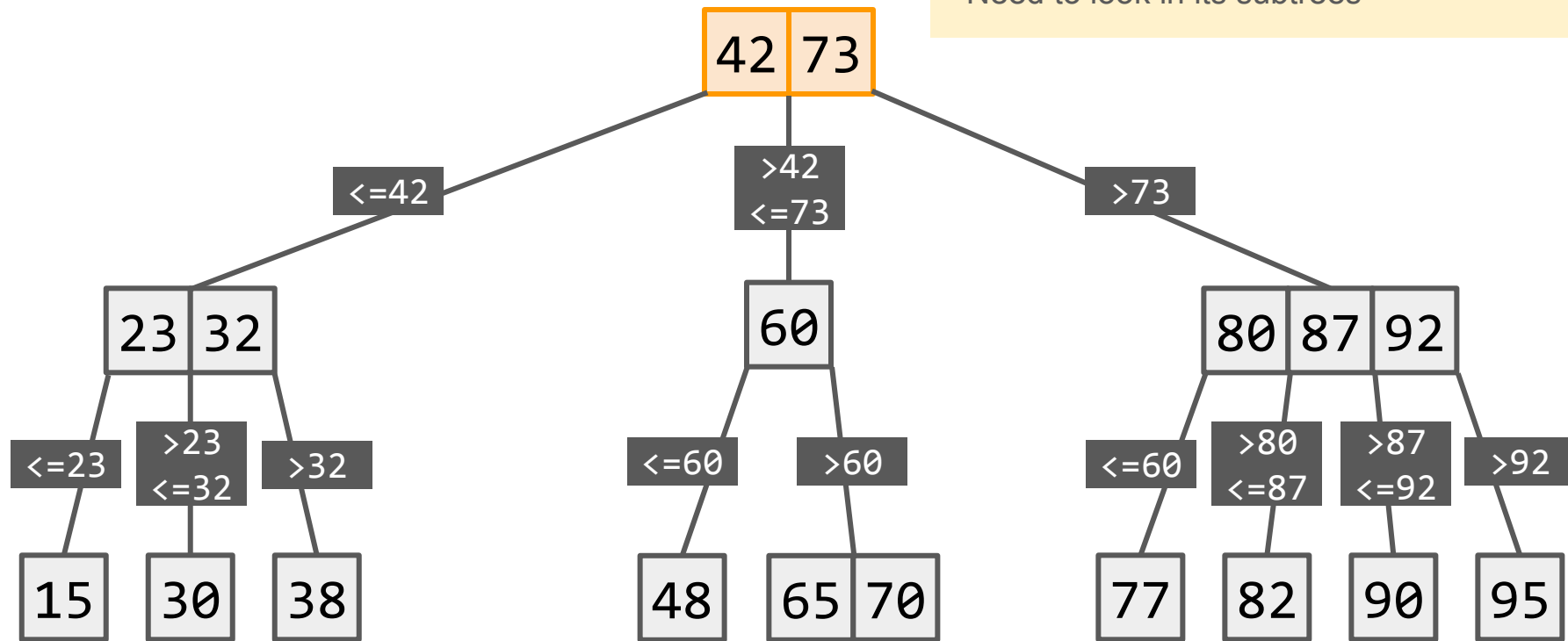
Problem 1.c.

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

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

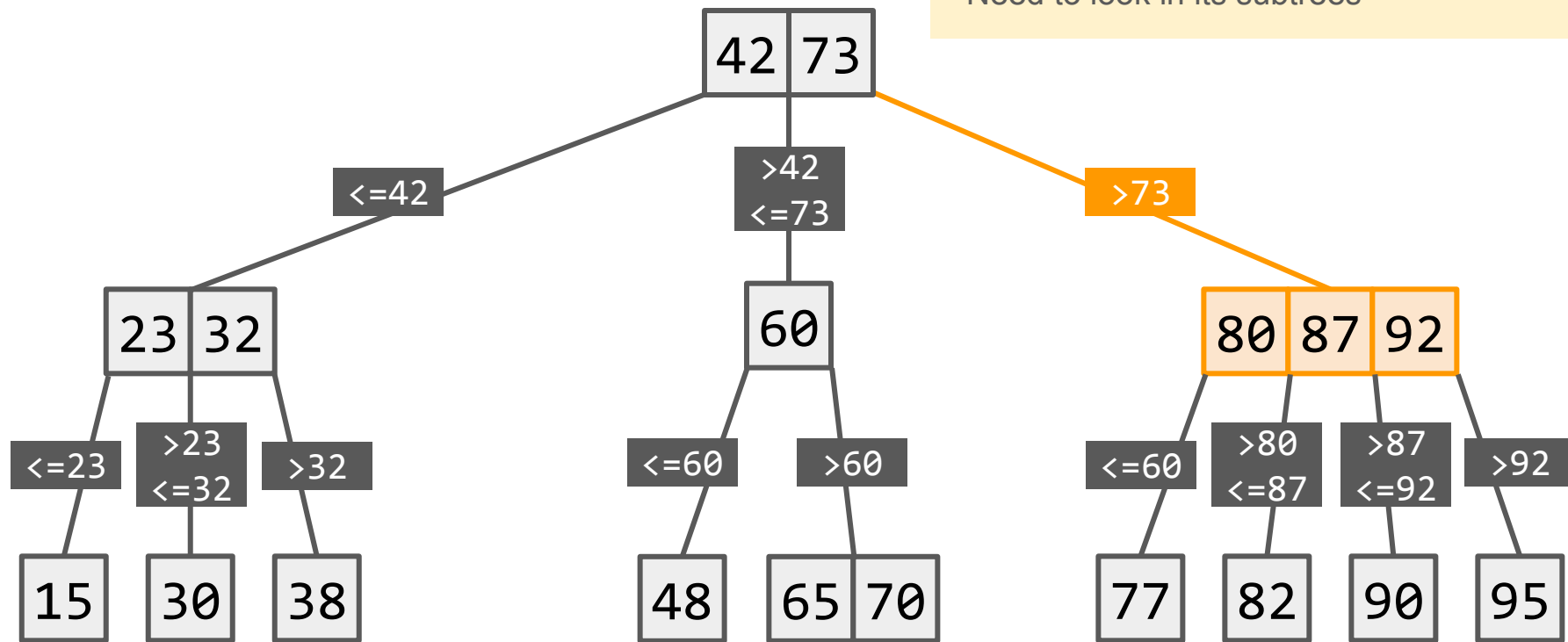
(2,4)-tree — search(82)

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



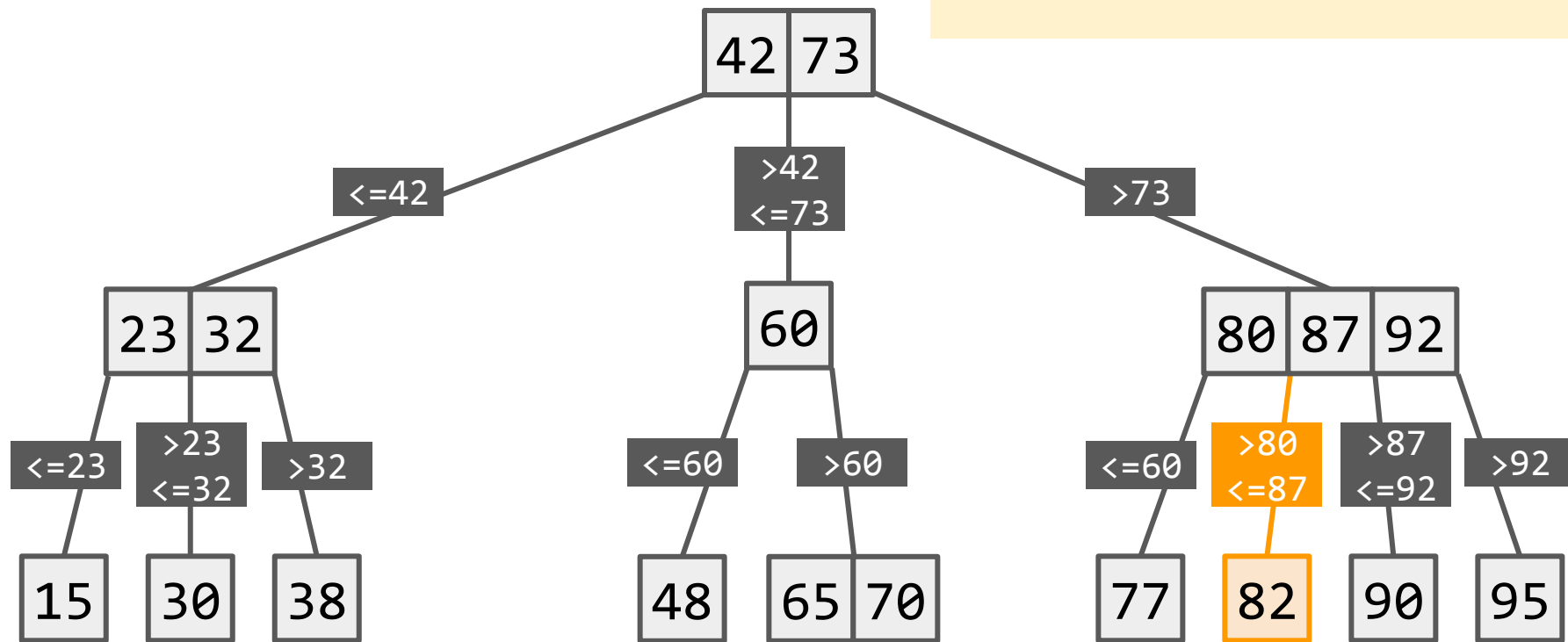
(2,4)-tree — search(82)

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



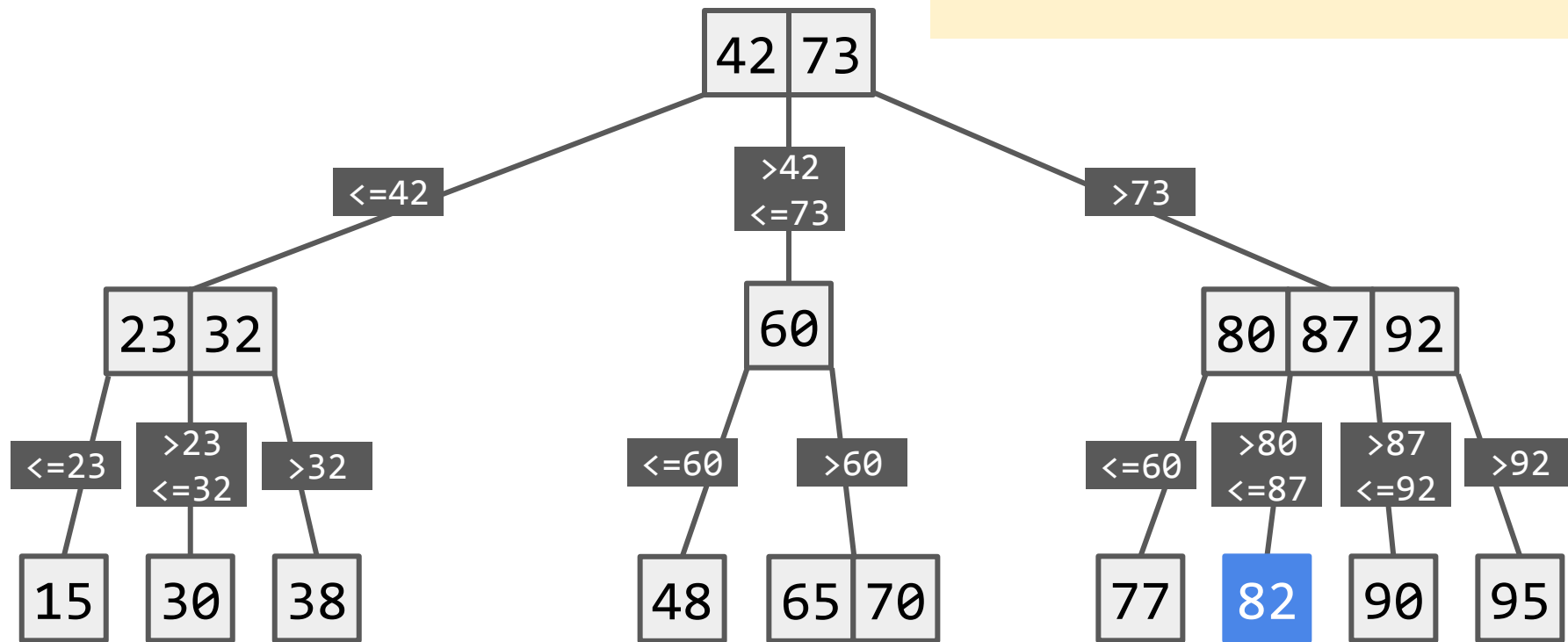
(2,4)-tree — search(82)

- $87 \geq 82 > 80$, go to second left child



(2,4)-tree — search(82)

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



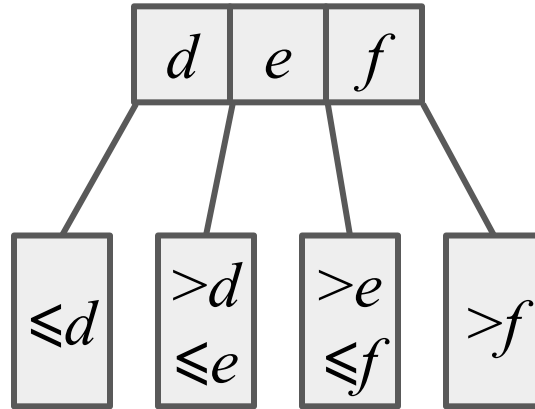
Problem 1.c.

The pseudocode for searching is left for you as an exercise.

Just follow the trace :)

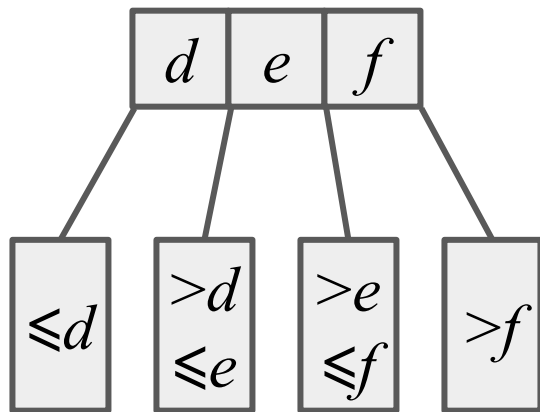
Problem 1.c.

What data structure should we use for storing the keys and subtree links 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)
------------	------------	------------	---------------------

$$t_1 > -\infty$$

$$t_1 \leq d$$

$$t_2 > d$$

$$t_2 \leq e$$

$$t_3 > e$$

$$t_3 \leq f$$

$$t_4 > f$$

$$t_4 \leq +\infty$$

Notice that for the key range of a subtree, the right-hand-bound (i.e. \leq) is **explicit** in the pairing, but the left-hand-bound (i.e. $>$) is **implicit** because you just need to look to the left neighbour 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 $O(\log_a n)$ height
- Binary search for a key at every node takes $O(\log_2 b)$ time
- Hence total search cost:

$$O(\log_2 b \cdot \log_a n)$$

$$= O(\log_a n) \quad \text{since } \log_2 b \text{ can be treated as a constant}$$

$$= O(\log n)$$



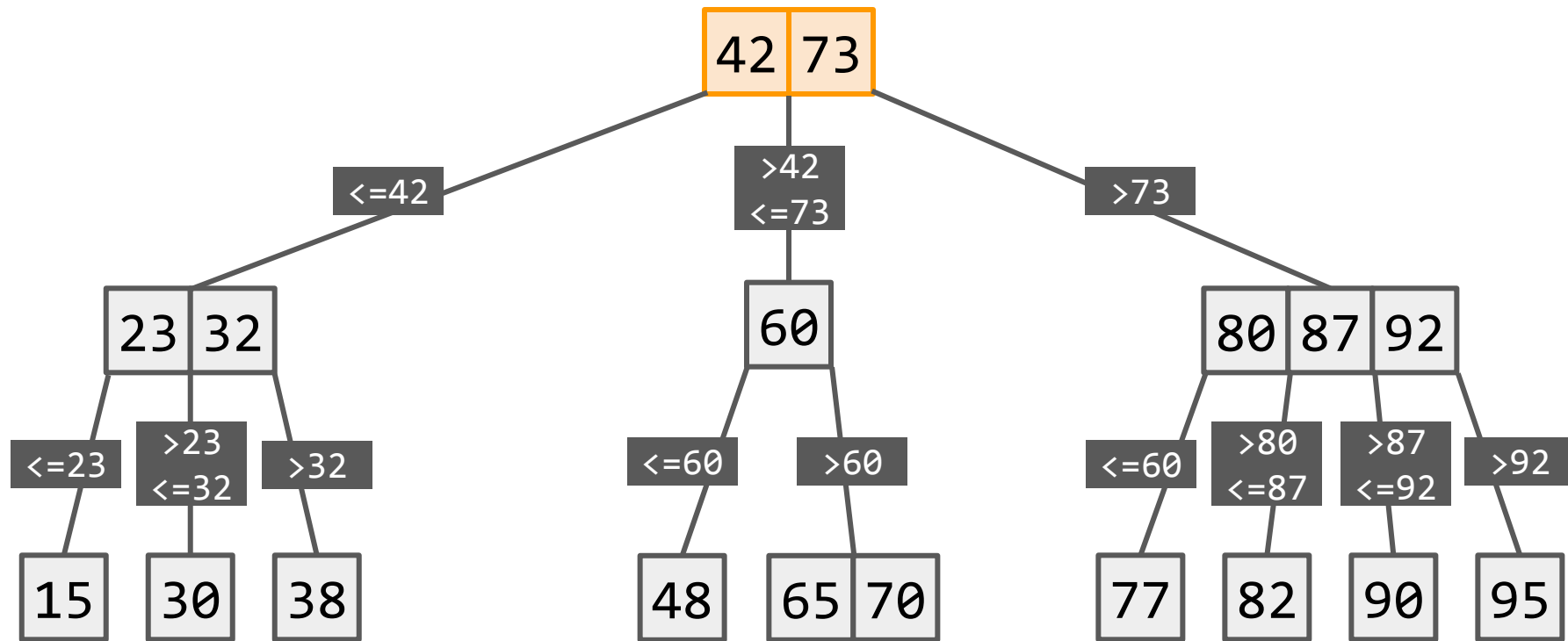
Insertion

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

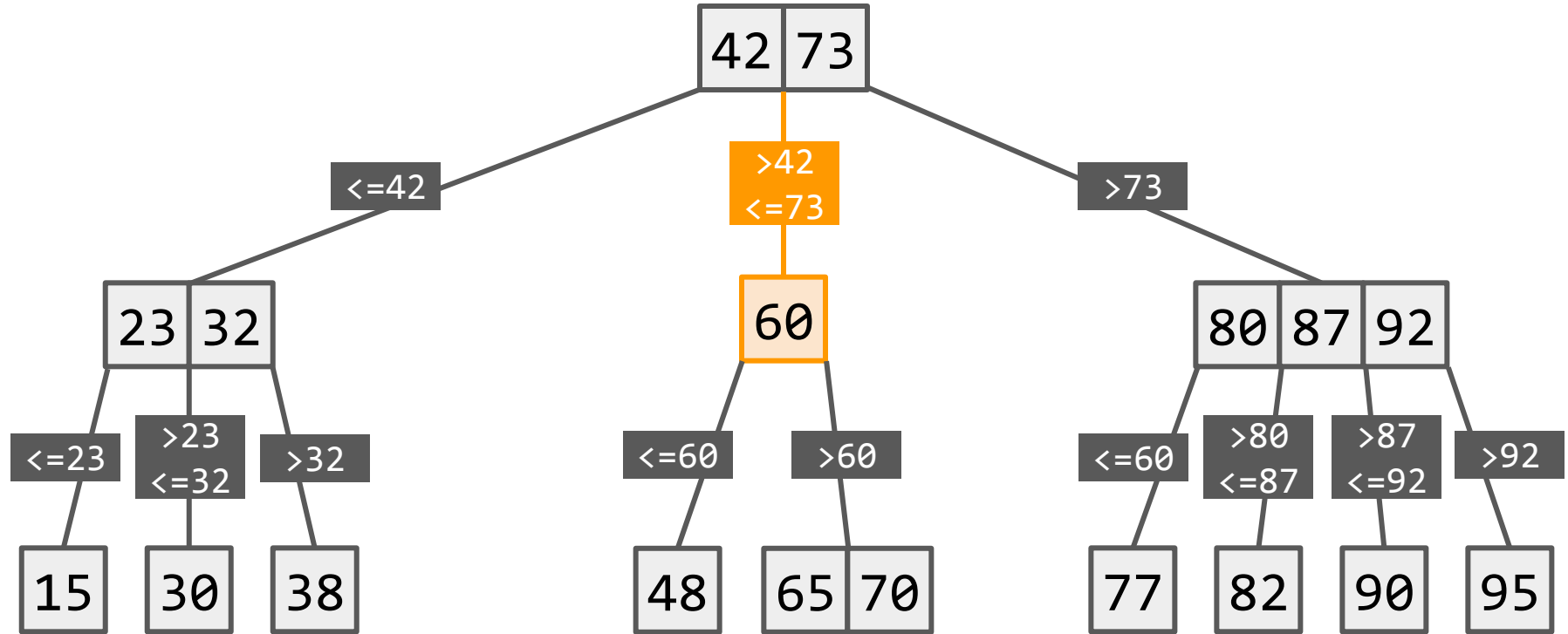
Insertion

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

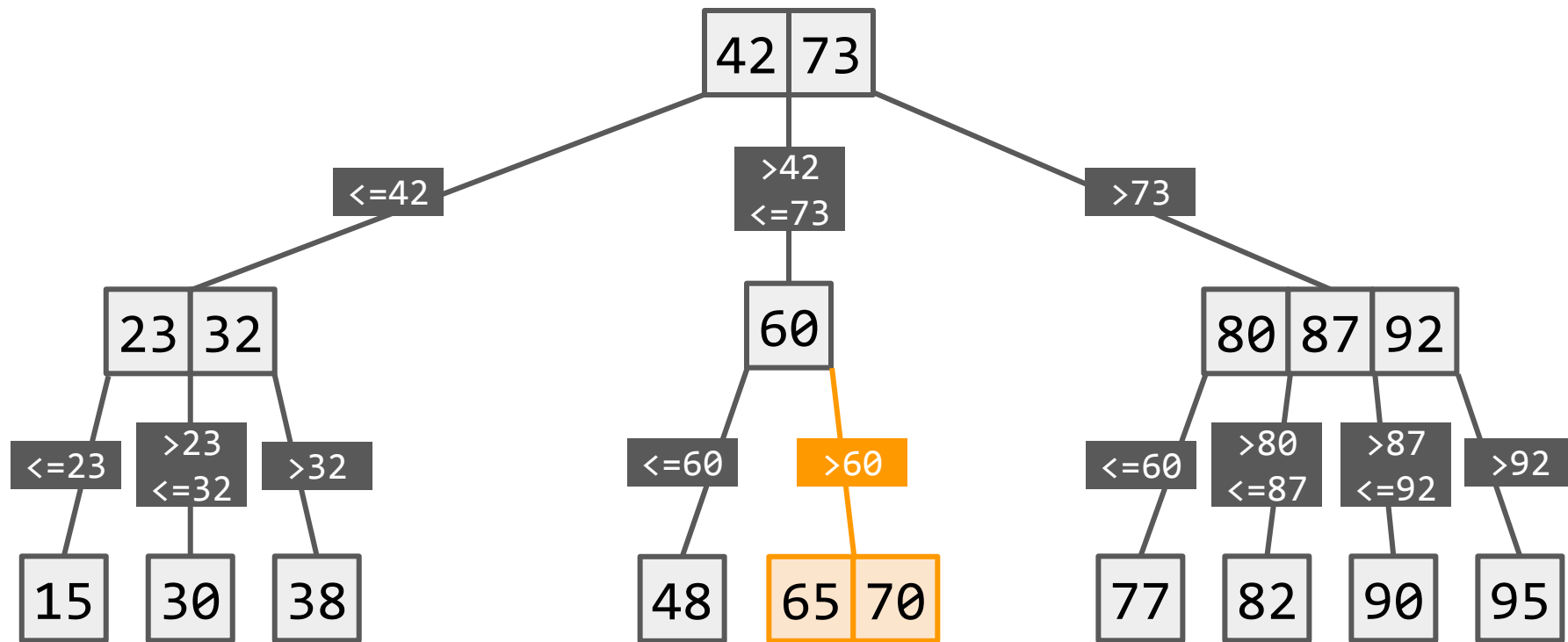
(2,4)-tree — insert(71)



(2,4)-tree — insert(71)

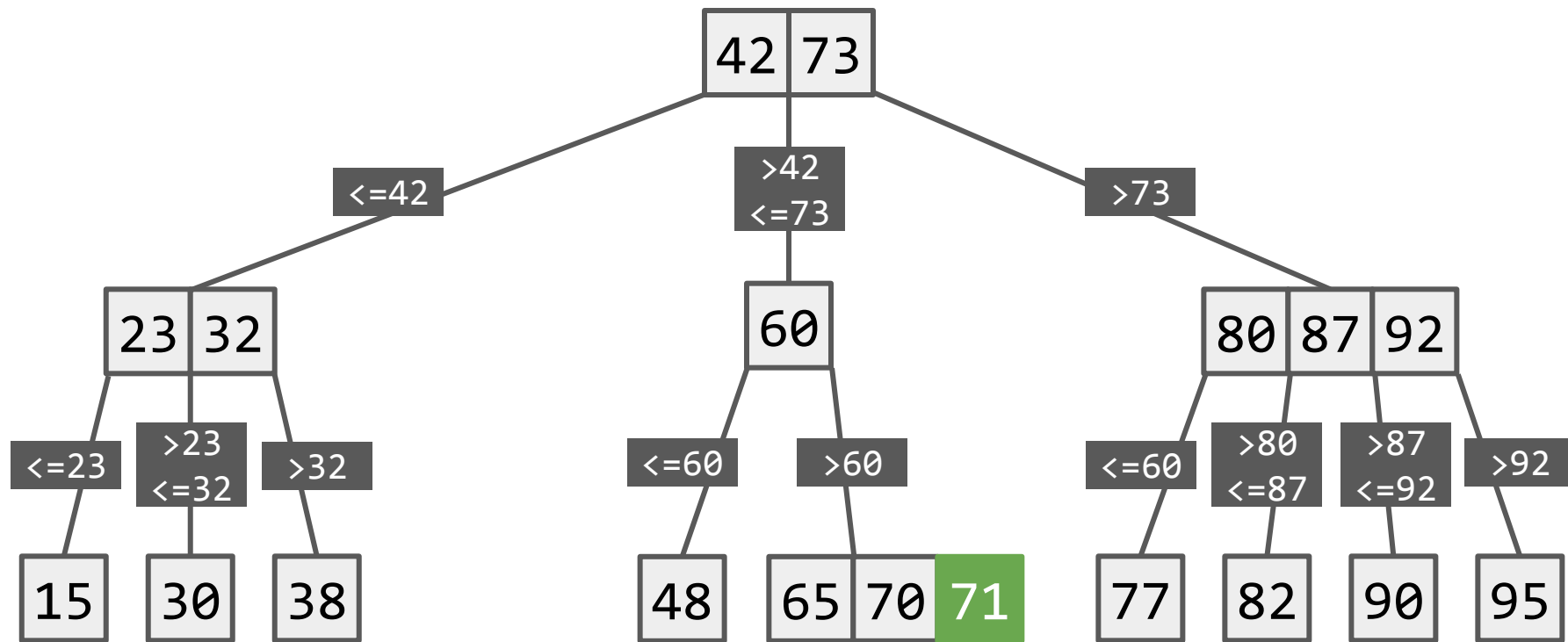


(2,4)-tree — insert(71)



Found appropriate leaf node!

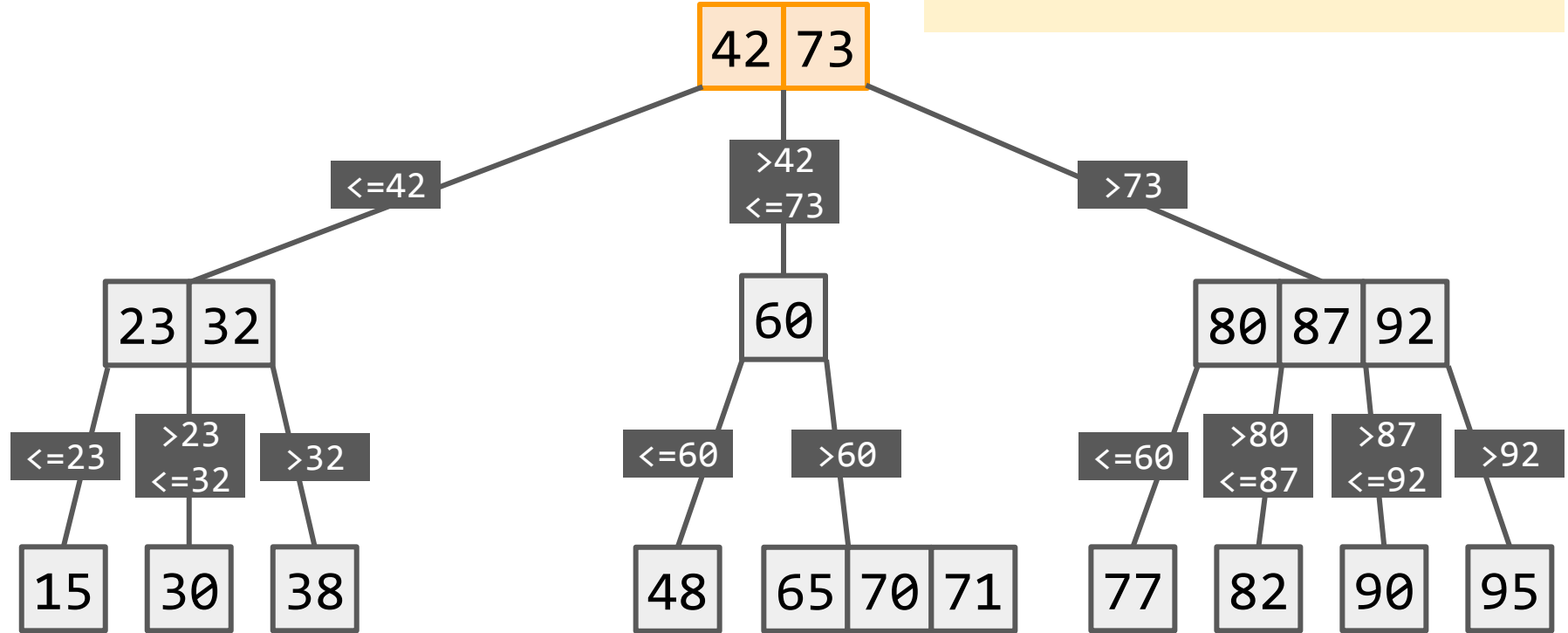
(2,4)-tree — insert(71)



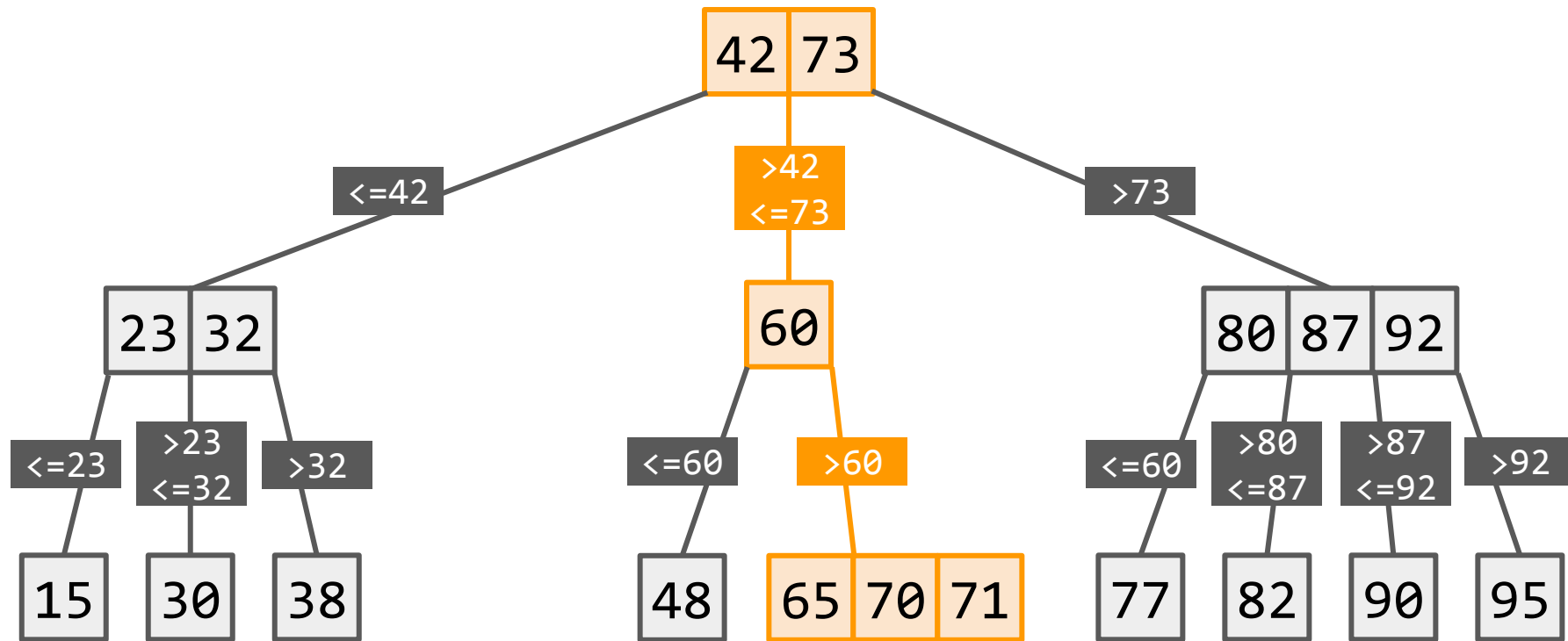
Add 71 to key list

(2,4)-tree — insert(72)

Now let's insert 72

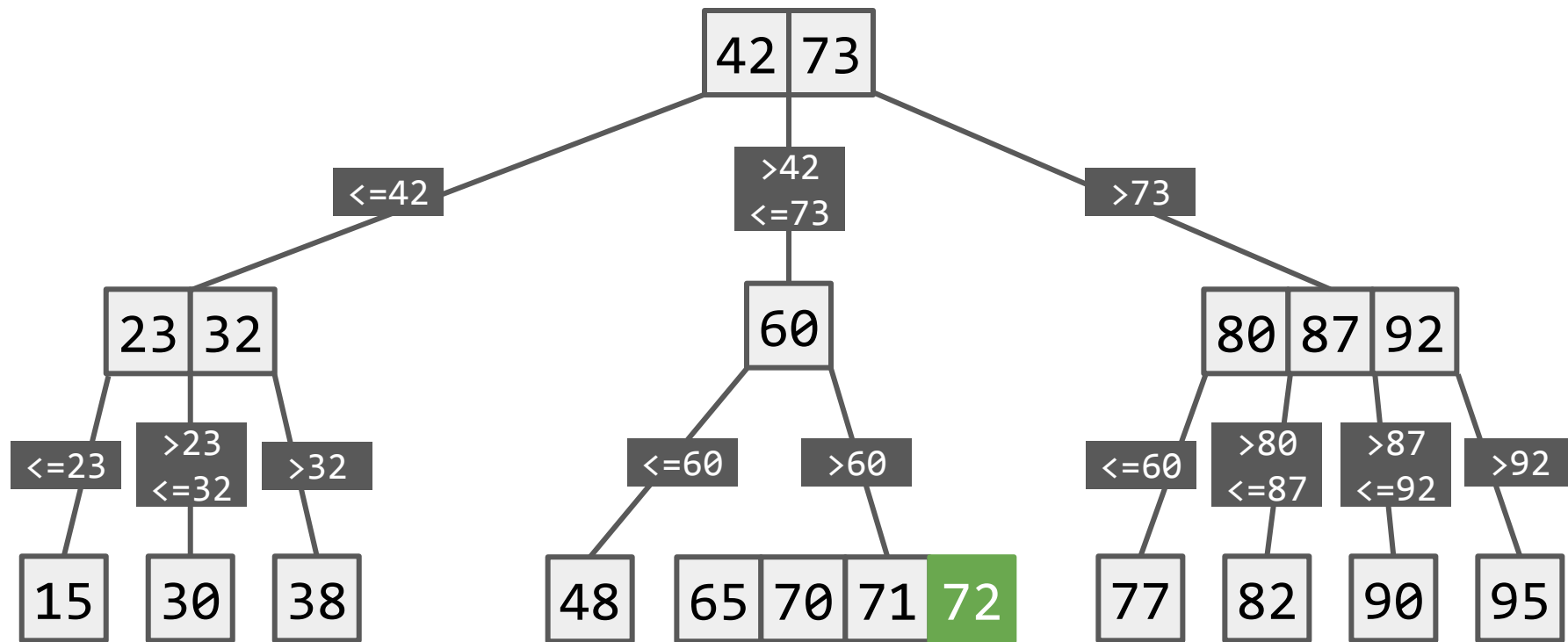


(2,4)-tree — insert(72)



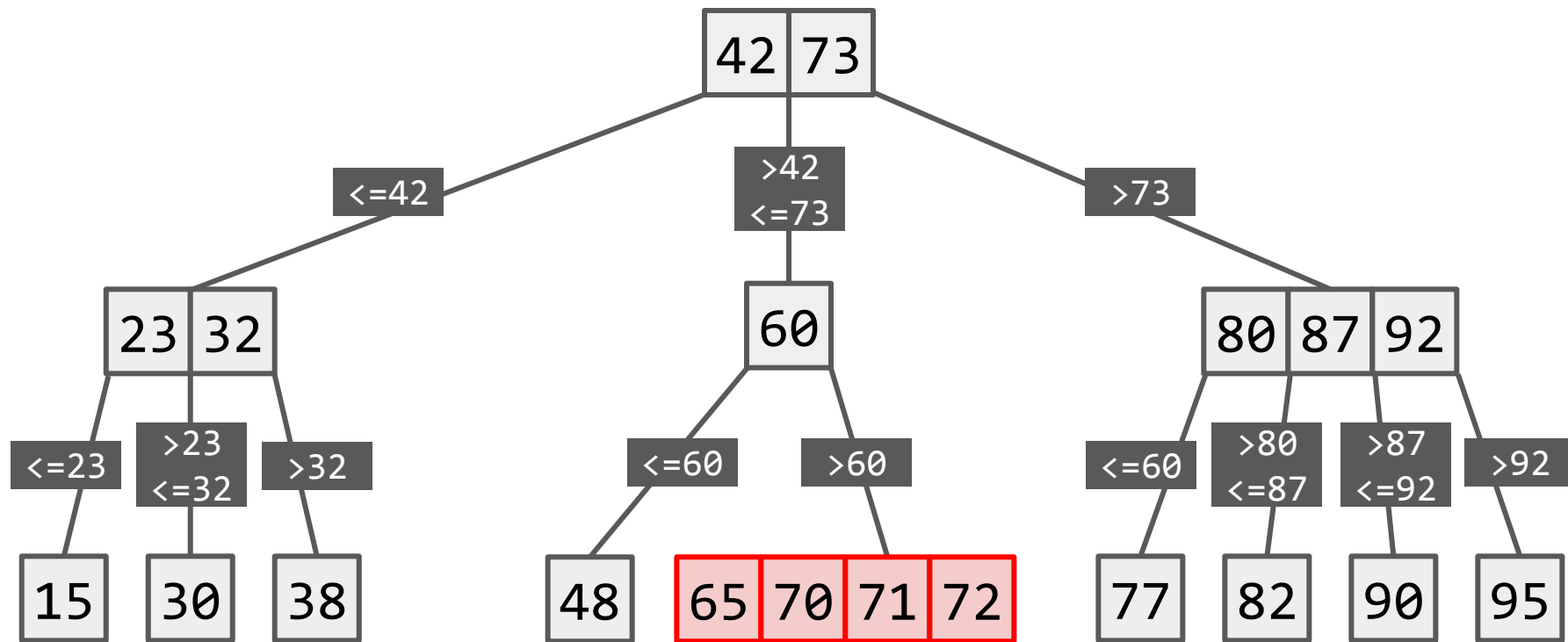
Found appropriate leaf node!

(2,4)-tree — insert(72)



Add 72 to key list

(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?

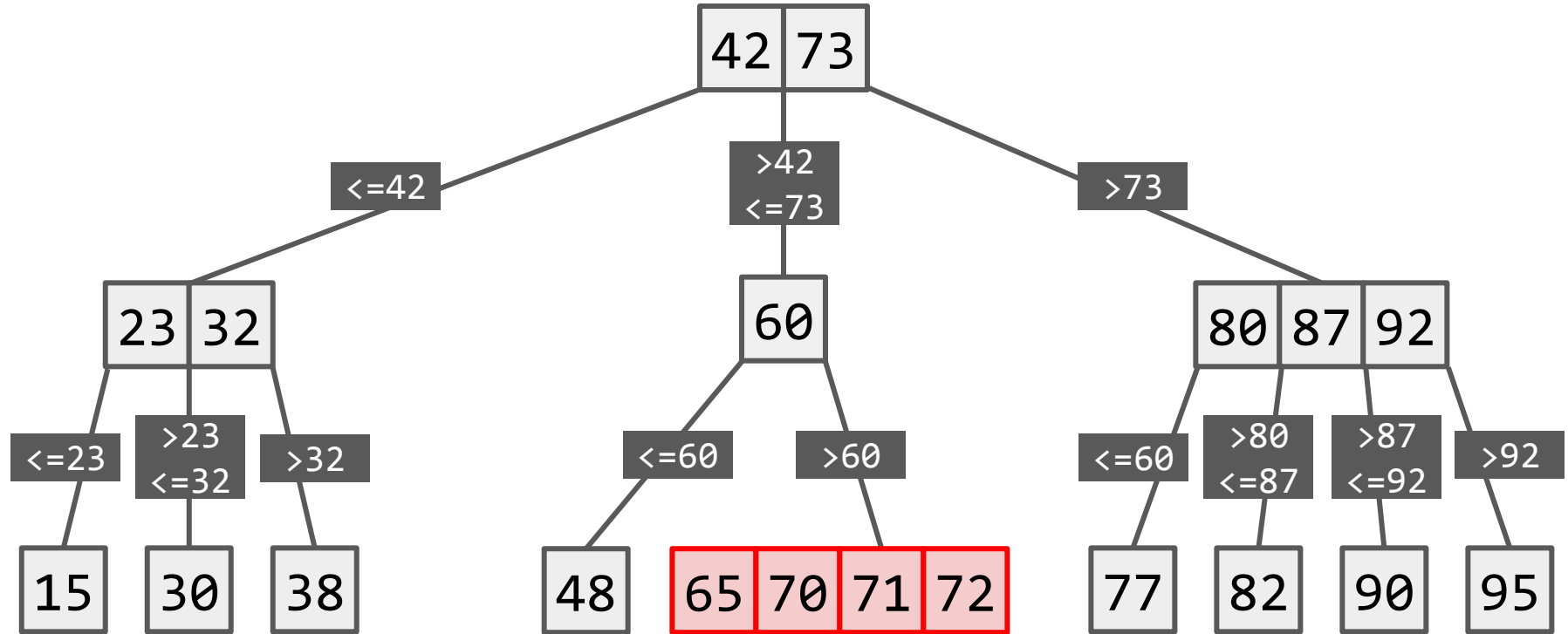
A. Over sibling nodes?

- Cannot guarantee no further violations!

B. Over a new node

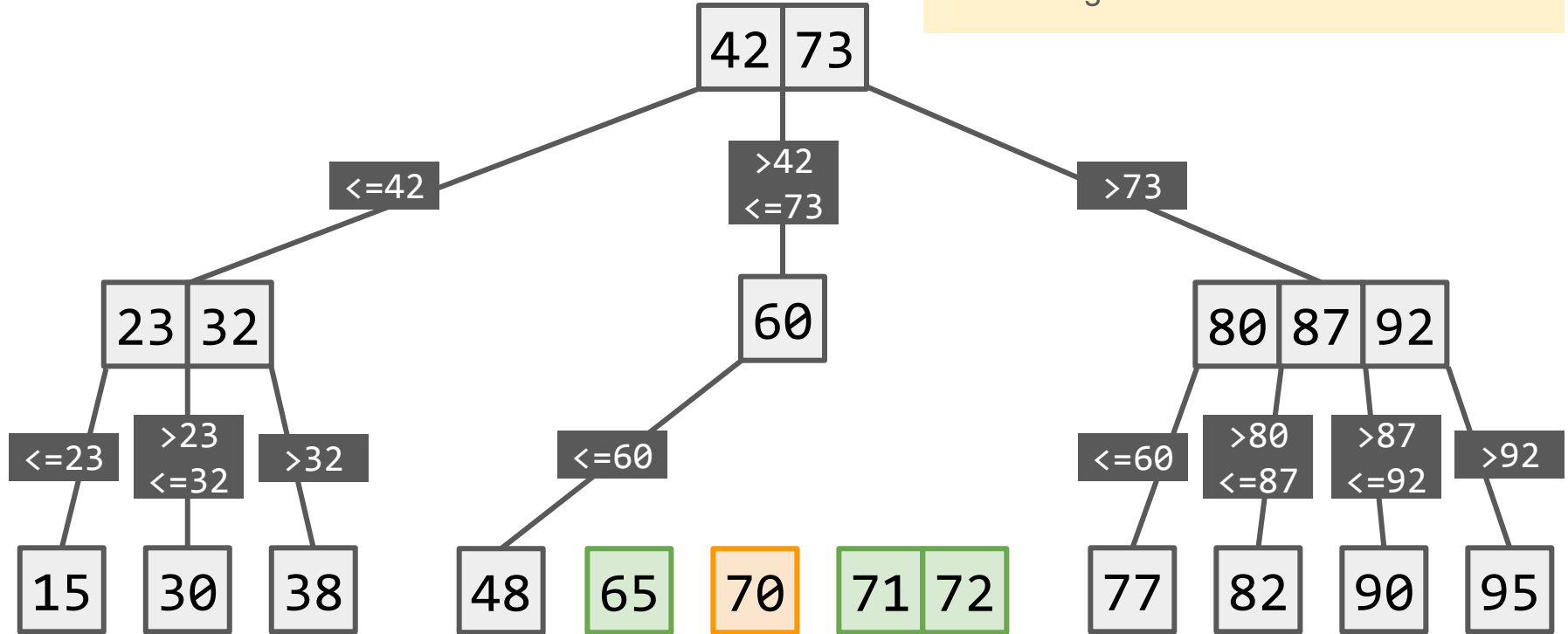
- Can always guarantee postcondition adheres rule 1!

Key-redistribution



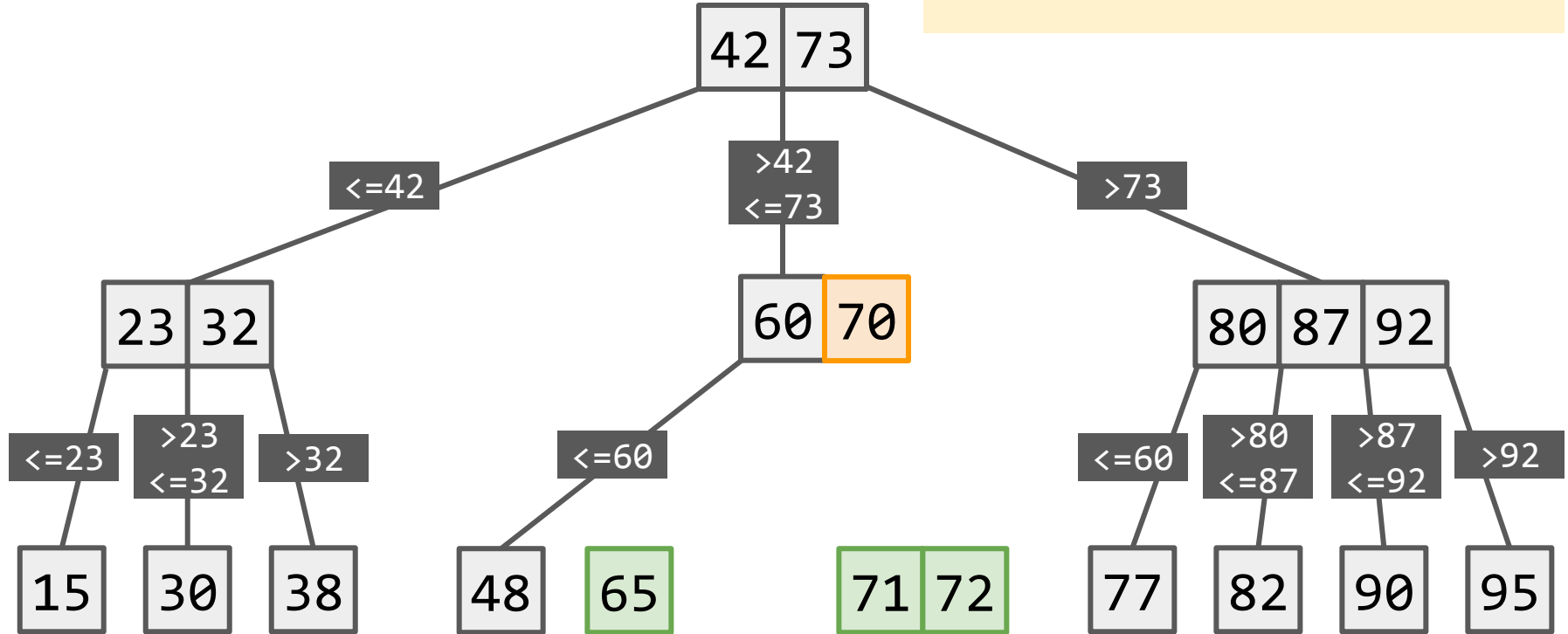
Key-redistribution

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



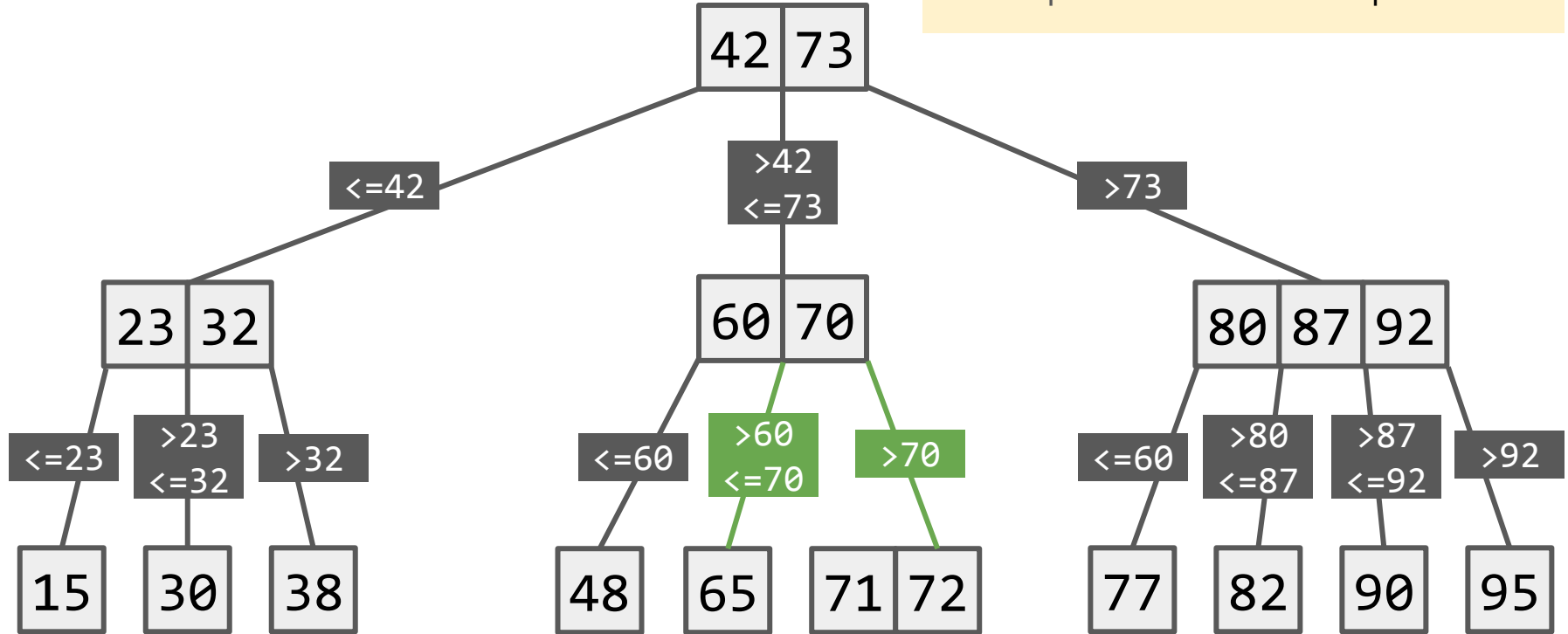
Key-redistribution

- Move median key to parent



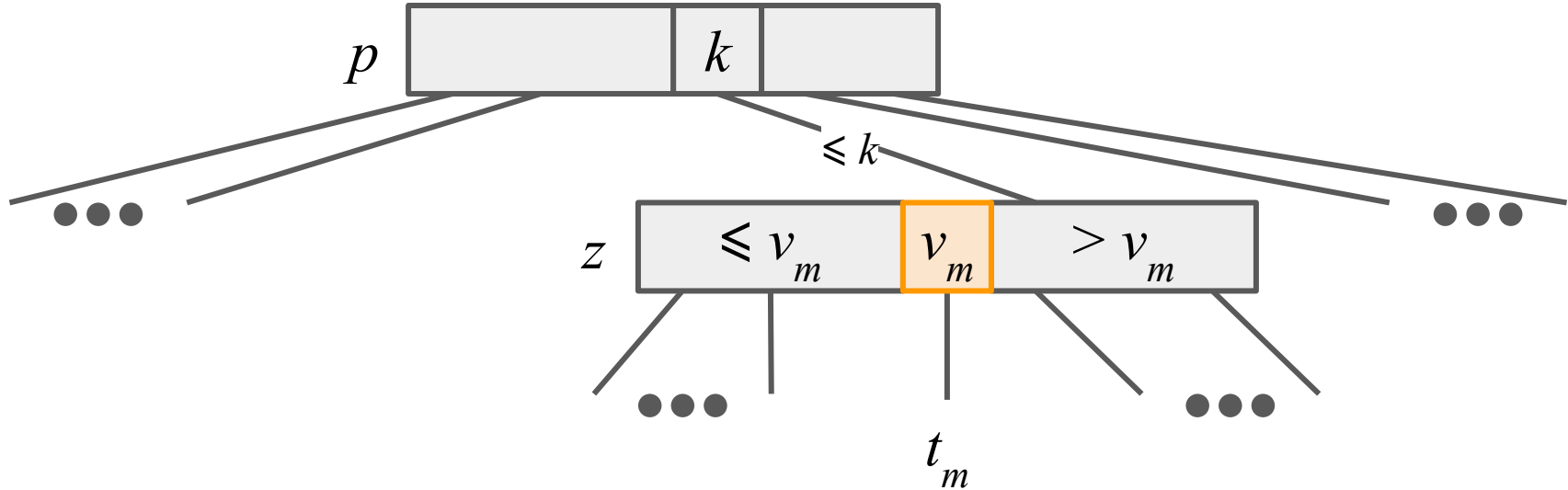
Key-redistribution

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



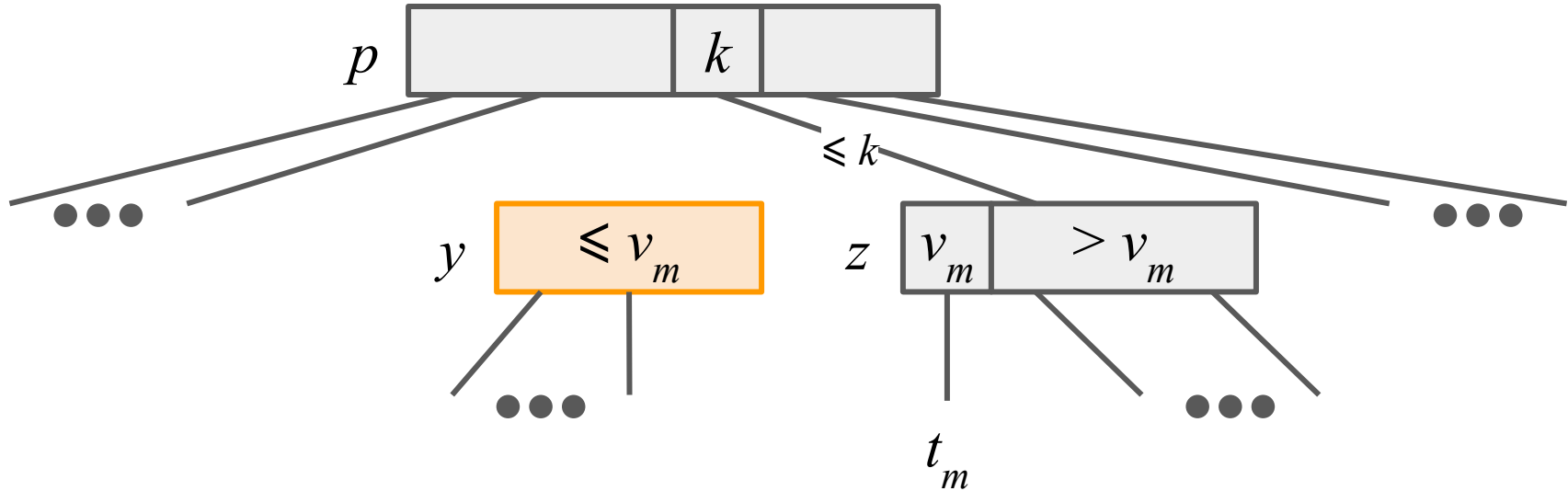
Split operation

Step 1



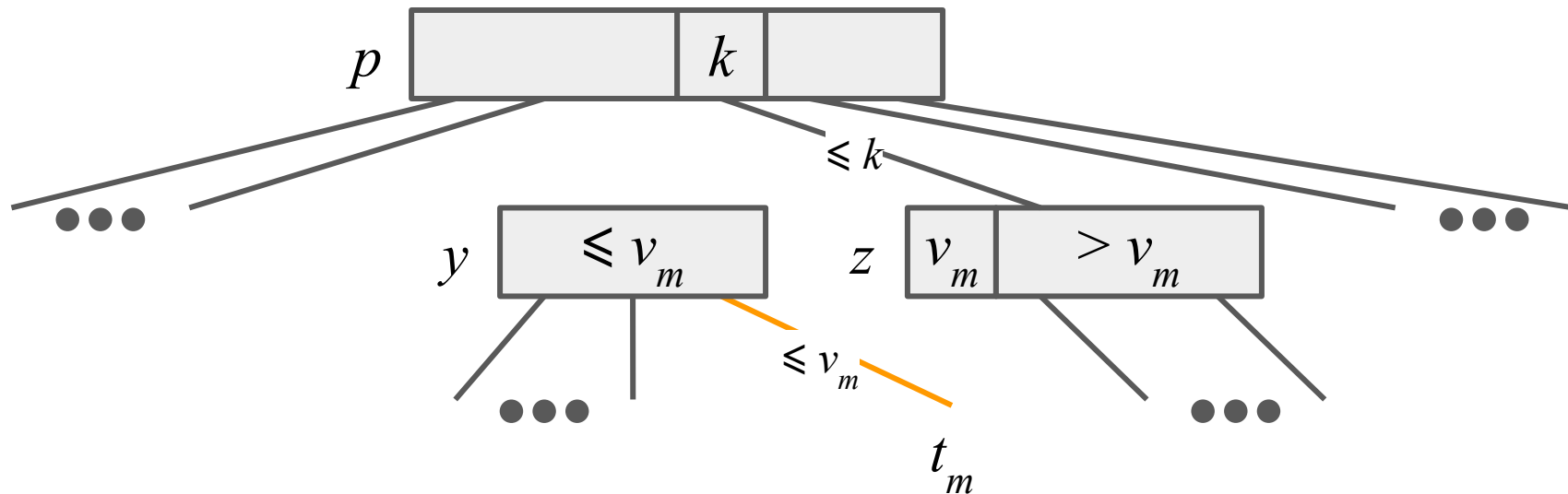
Find the **median key** v_m where the number of z 's keys that are $\leq v_m$ (LHS) is $\lfloor (b-1)/2 \rfloor$ and $> v_m$ (RHS) is $\lceil (b-1)/2 \rceil$.

Step 2



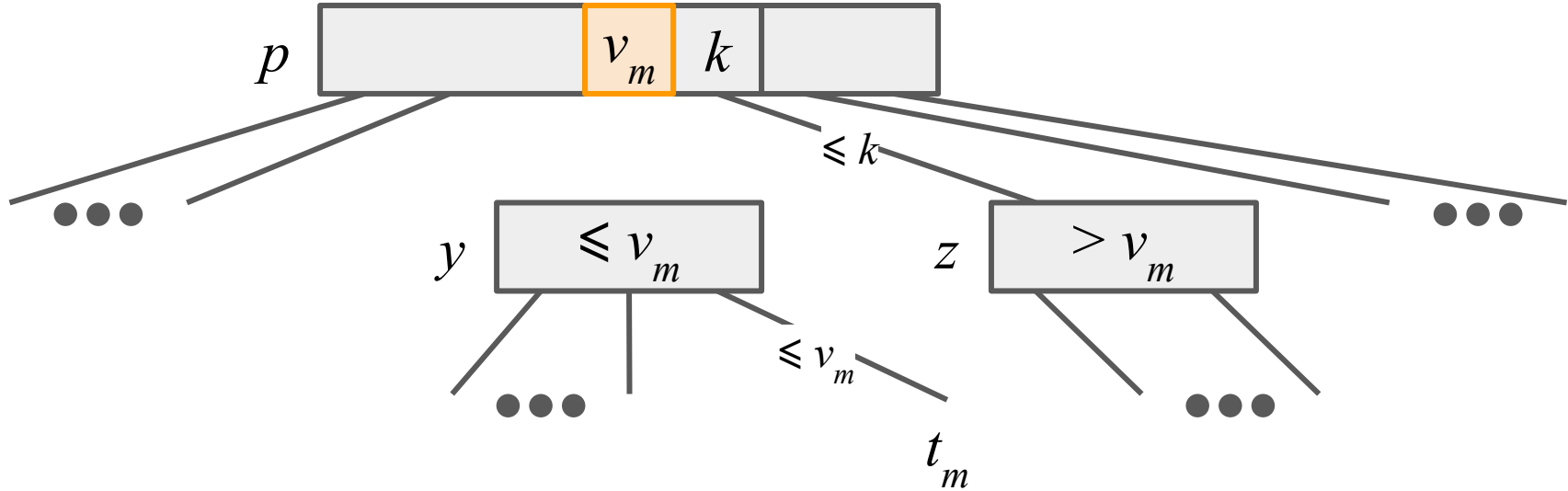
In z , separate all keys $\leq v_m$ as well as their associated subtree links to form a new node y .

Step 3



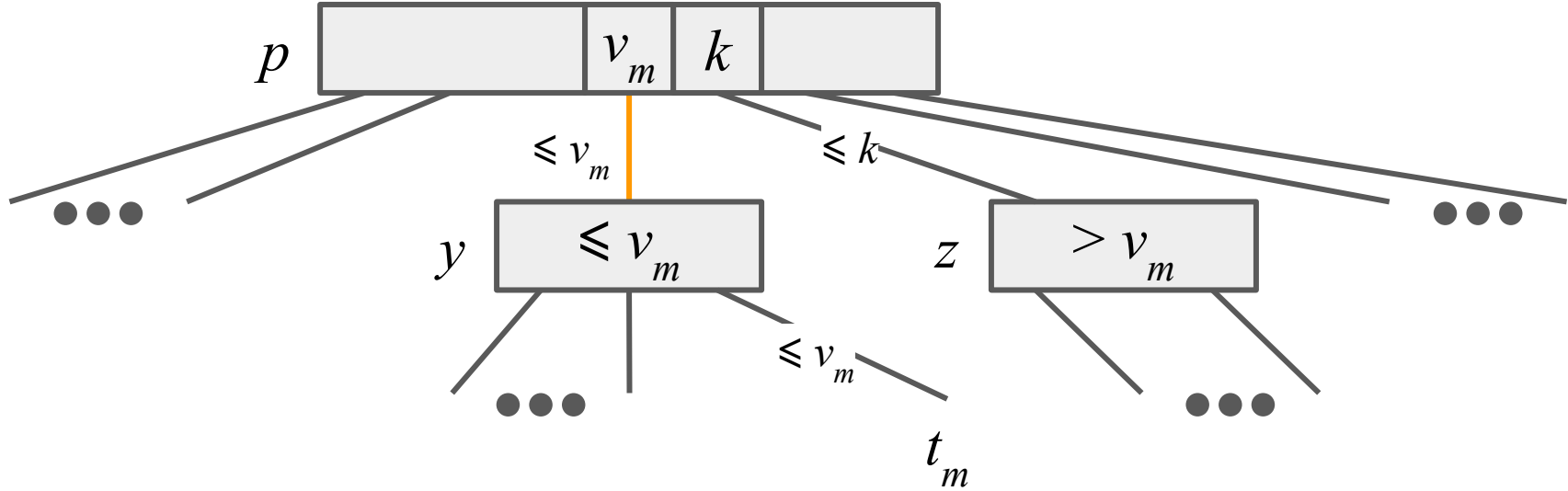
y needs to have a final child, so we assign it to be t_m , (the subtree previously corresponding to v_m). We of course also have to remove z as the parent of t_m .

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)

Test yourself!

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

Test yourself!

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.

Test yourself!

Will the split procedure ever end up with nodes too small and violate rule 1?

Test yourself!

What is one potential problem split can cause?

How can we solve it?

Test yourself!

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.

Test yourself!

- 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

Test yourself!

- 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

Old 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

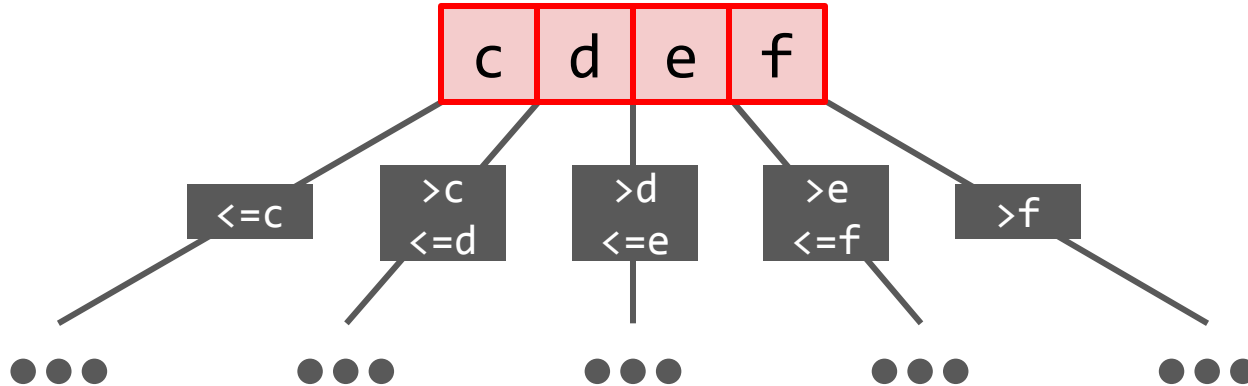
Problem 1.e.

Come up with an example each for

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

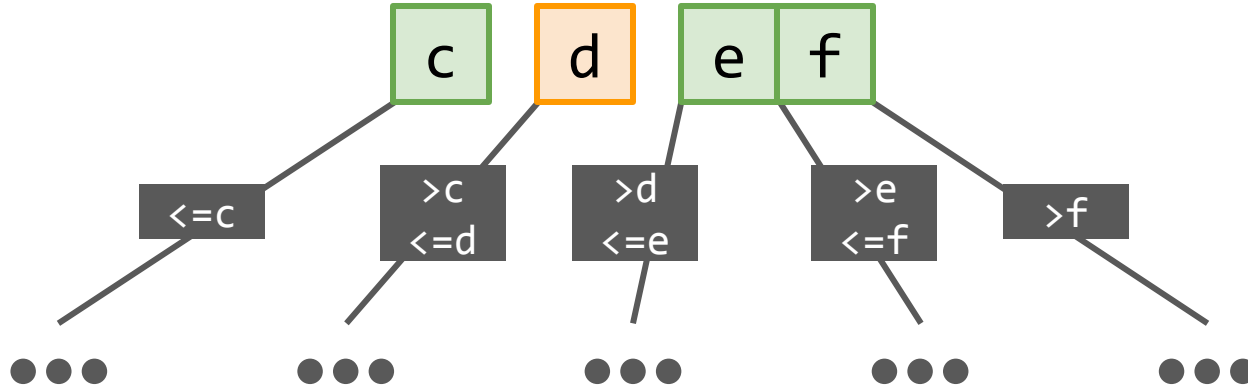
Corner case: splitting the root

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



Corner case: splitting the root

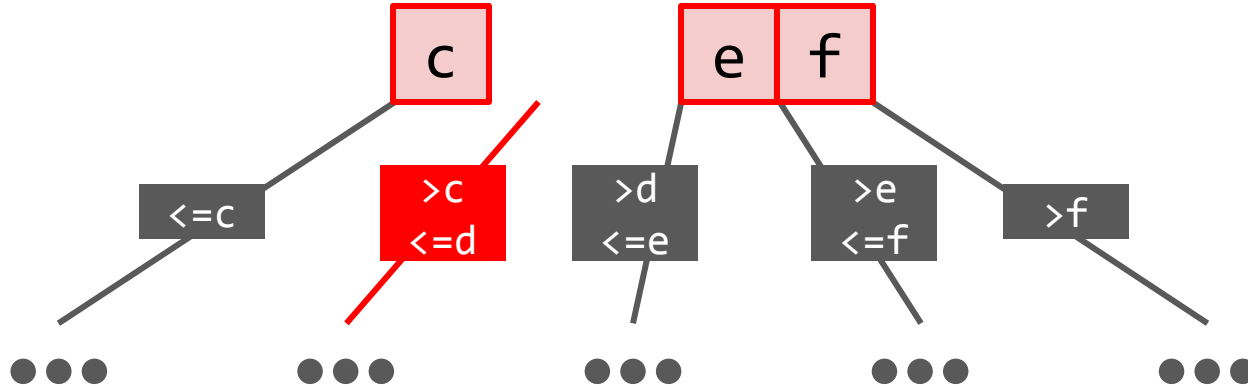
- Split root node keys using median key



Corner case: splitting the root

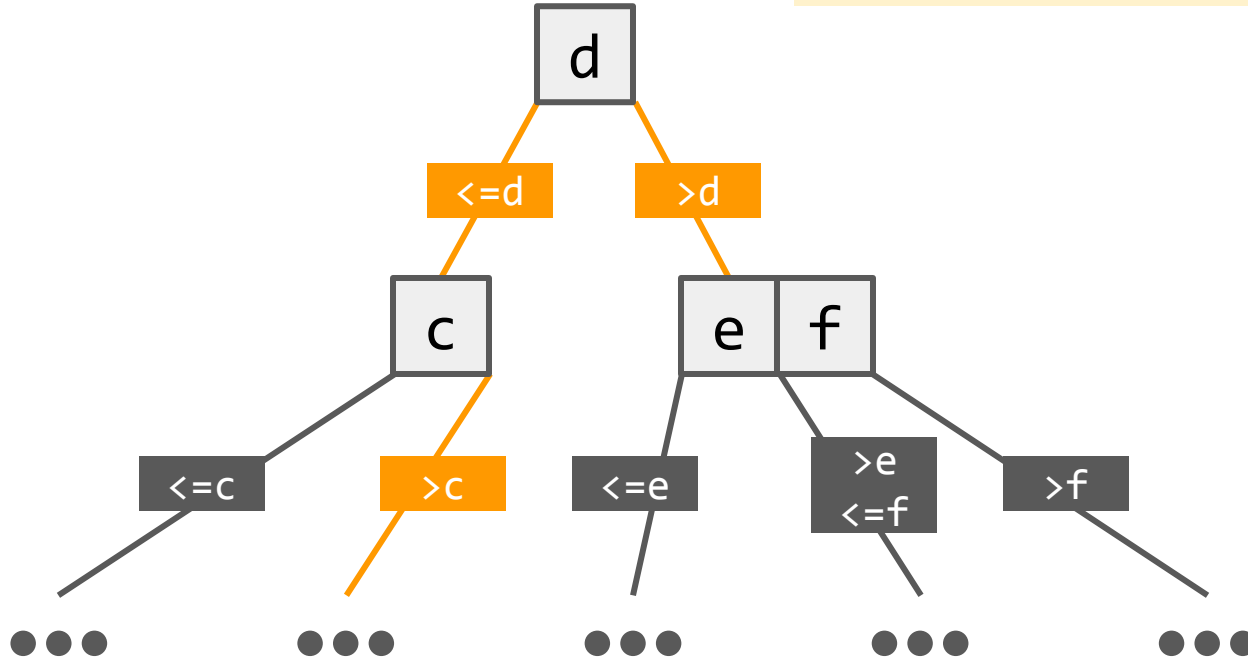
- Promote median key to form new root with just this key
- 3 subtrees will be orphaned

d



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**



Test yourself!

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?

Test yourself!

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.

Test yourself!

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)?

Test yourself!

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)

Full insertion algorithm

Insertion of x into the tree:

1. Start at node $w = \text{root}$
2. Repeat until w is a leaf:
 - If w contains $b - 1$ keys
 - i. Split w into y (LHS) and z (RHS) using split key v_m
 - ii. Examine split key v_m returned by the split operation
 - If $x \leq v_m$, then set $w = y$
 - Else, set $w = z$
 - Else
 - i. Search for x in the keylist of w
 - If x is larger than all the keys, set w to be the last child
 - Else, look for the first key $v_i \geq x$ and set w to be t_i (subtree with key range $\leq v_i$)
3. Add x to w 's keylist

} Proactive approach

Test yourself!

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

Test yourself!

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
- After split, we have $b-2$ keys across a sibling pair
- Left sibling keys: $\lfloor (b-2)/2 \rfloor = \lfloor b/2 \rfloor - 1$
- Left sibling (least keys) 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.1

If node w is split, then it satisfies the preconditions of the split routine

Preconditions, proactive.

1. z contains $\geq 2a$ keys, because after splitting
 - LHS has $\geq a - 1$ keys
 - RHS has $\geq a$ keys
2. z 's parent contains $\leq b - 2$ keys

Problem 1.f.1

If node w is split, then it satisfies the preconditions of the split routine

Preconditions, proactive.

1. z contains $\geq 2a$ keys, because after splitting

- LHS has $\geq a - 1$ keys
- RHS has $\geq a$ keys

2. z 's parent contains $\leq b - 2$ keys

Split only done when full!

I.e. $B \geq 2a$ keys

Since full nodes are split proactively,
parent guaranteed not to be full.

Problem 1.f.2

Before x is inserted into w (at the last step), there are $< b$ keys in w (so key x can be added)

Problem 1.f.2

Before x is inserted into w (at the last step), there are $< b$ keys in w (so key x can be added)

Full nodes are split proactively

Problem 1.f.3

All three rules of the (a,b) -tree are satisfied after an insertion

1. (a,b) children
2. key ranges
3. all leaf nodes at same height

Test yourself!

Will the split procedure ever end up with nodes too small and violate rule 1?

Answer: No. This is because

- Keylist size of LHS is $\lfloor (b-1)/2 \rfloor = \lfloor b/2 - 1/2 \rfloor = \lfloor a - 1/2 \rfloor$
- Keylist size of RHS is $\lceil (b-1)/2 \rceil = \lceil b/2 - 1/2 \rceil = \lceil a - 1/2 \rceil$
- Since $\lfloor a - 1/2 \rfloor \geq a - 1$, Therefore the keylist size of either LHS and RHS will be at least $a - 1$.

Test yourself!

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 a leaf node at the same level
- If z is the *root* node, everything will be pushed down 1 level

Test yourself!

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

Test yourself!

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?

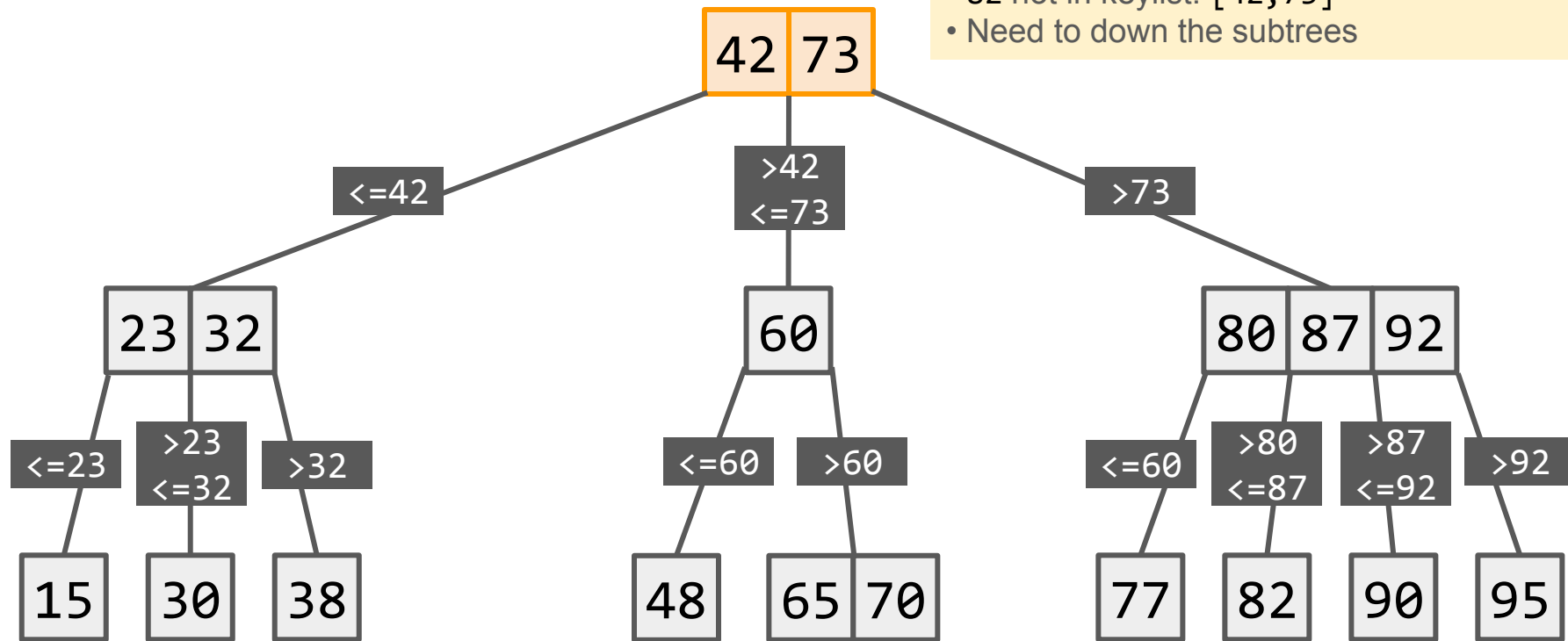


Deletion

Because nothing lasts forever

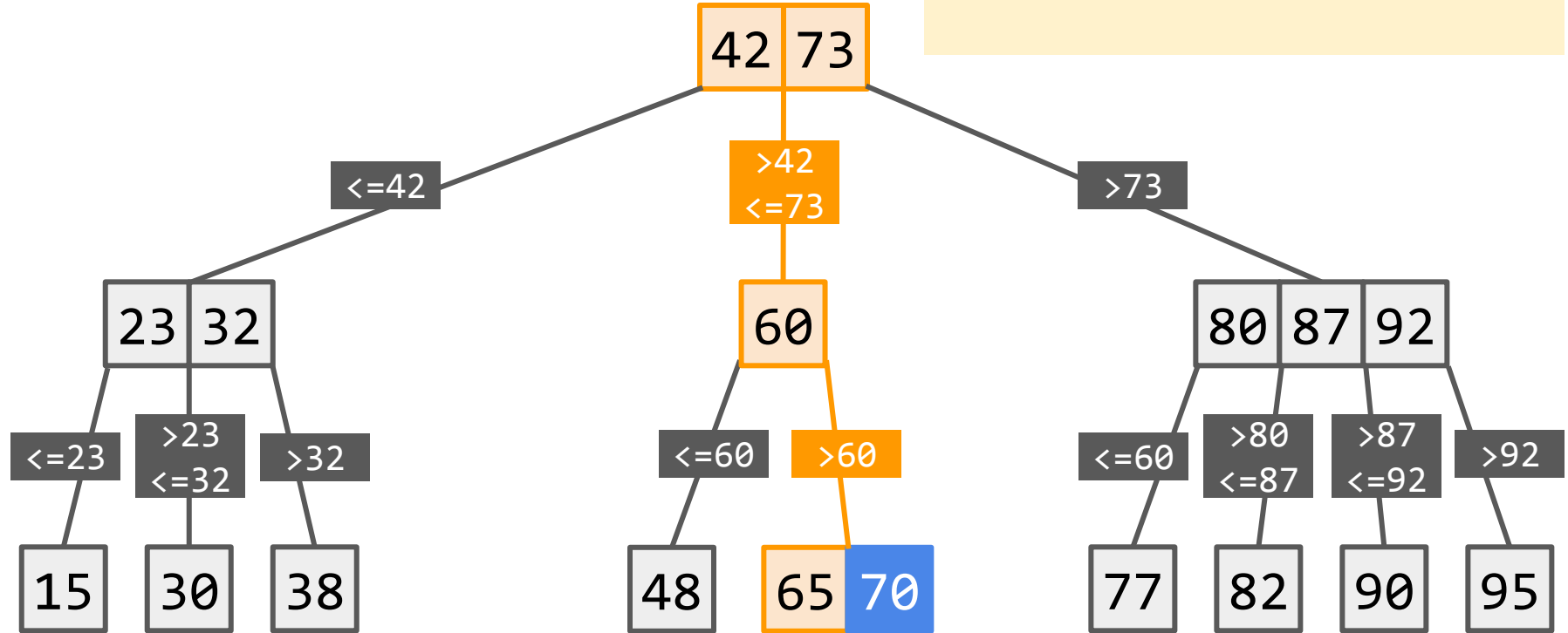
(2,4)-tree — delete(70)

- To delete an item, we first find it
- Starting at the root
- 82 not in keylist: [42, 73]
- Need to down the subtrees



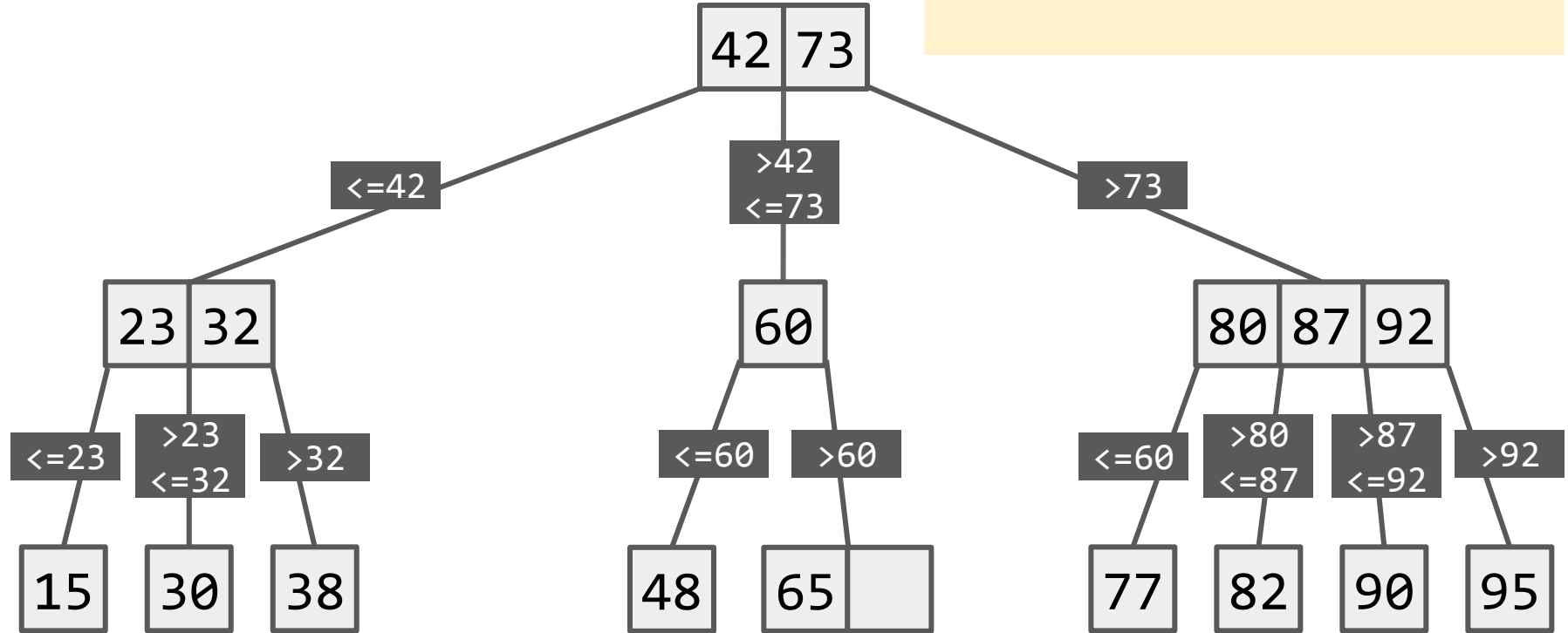
(2,4)-tree — delete(70)

- Found 70



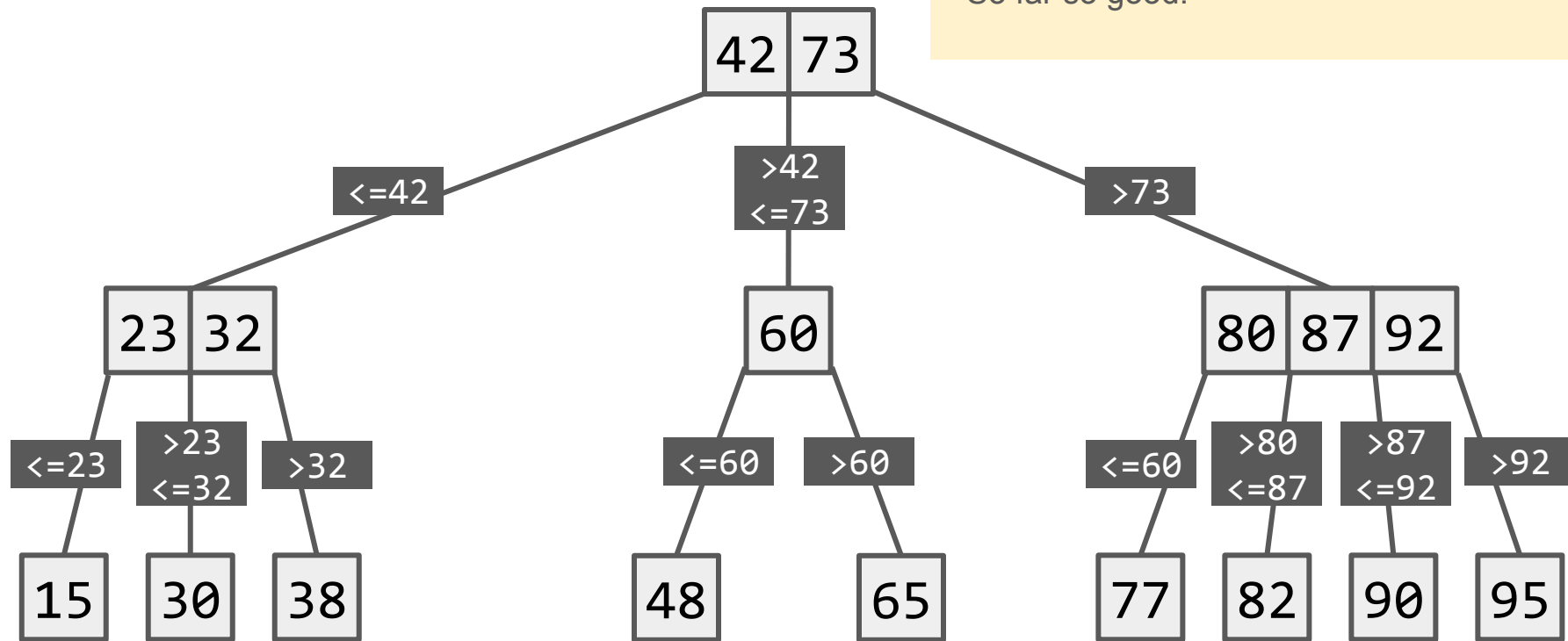
(2,4)-tree — delete(70)

- Delete 70 from keylist



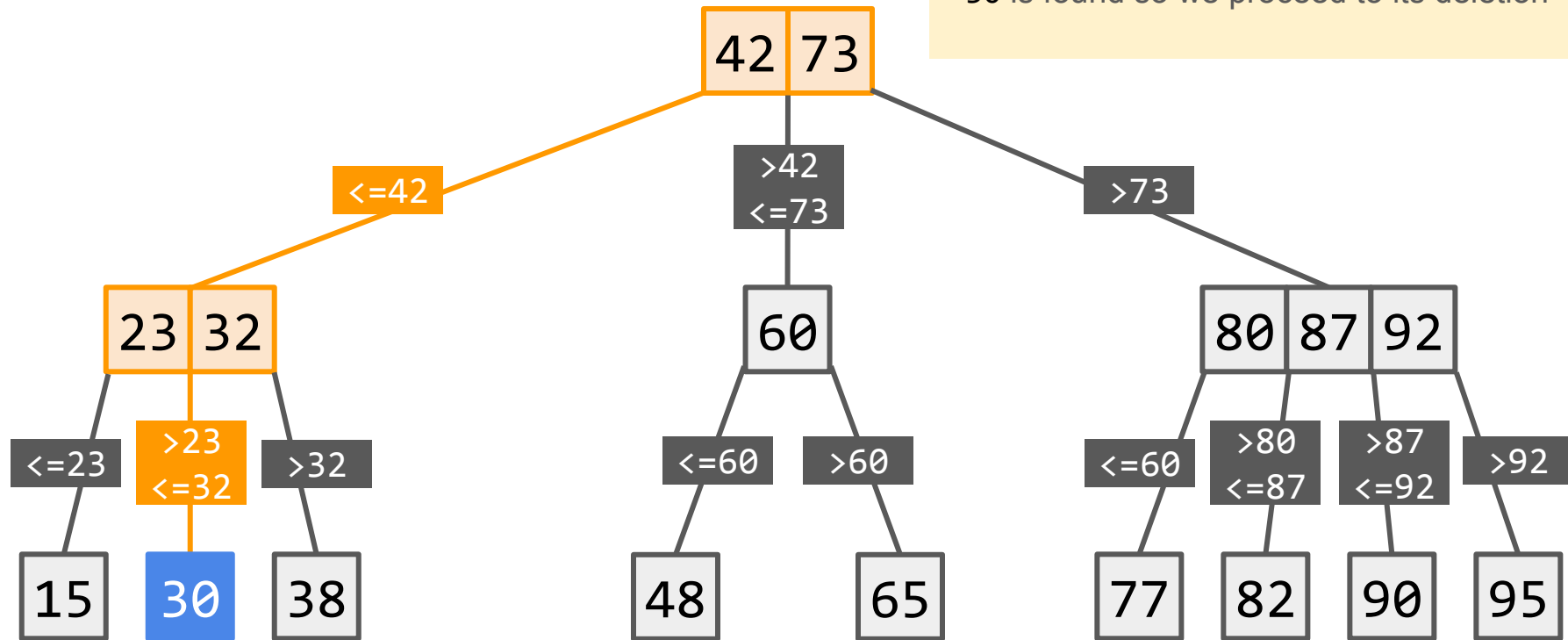
(2,4)-tree — delete(70)

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



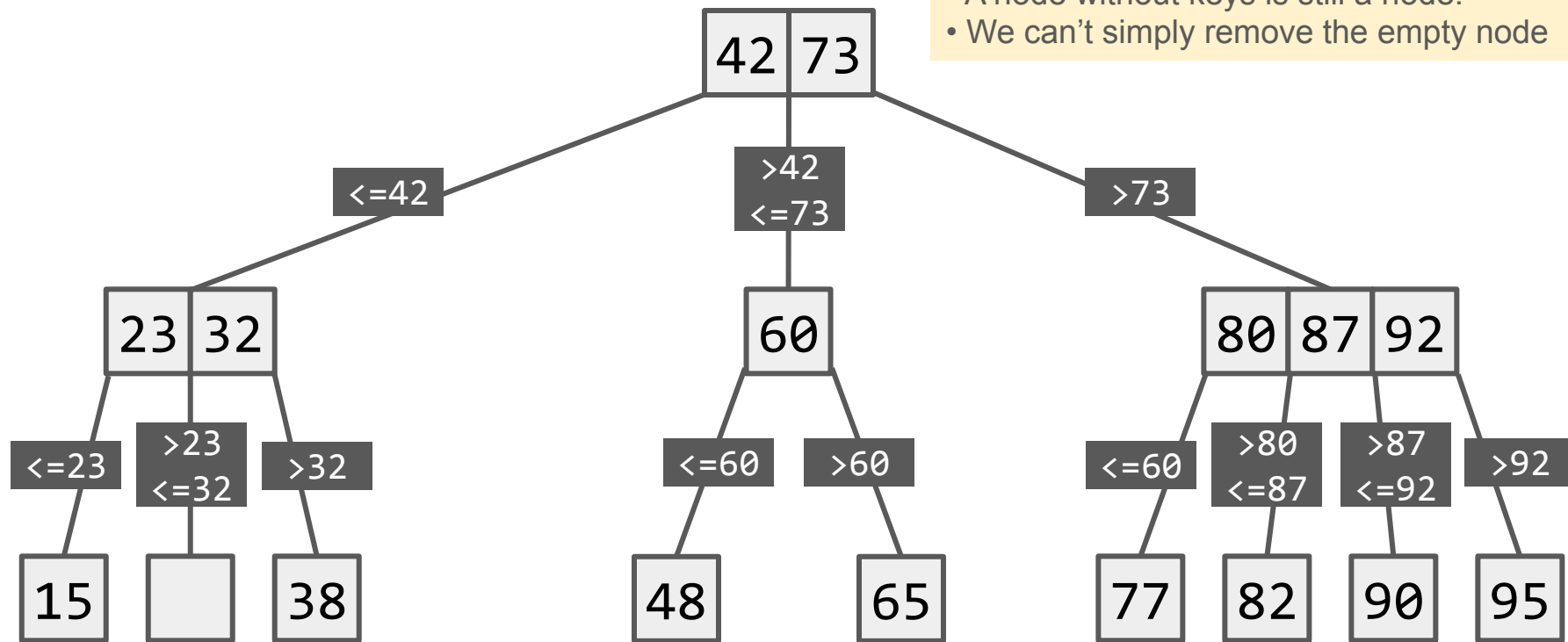
(2,4)-tree — delete(30)

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



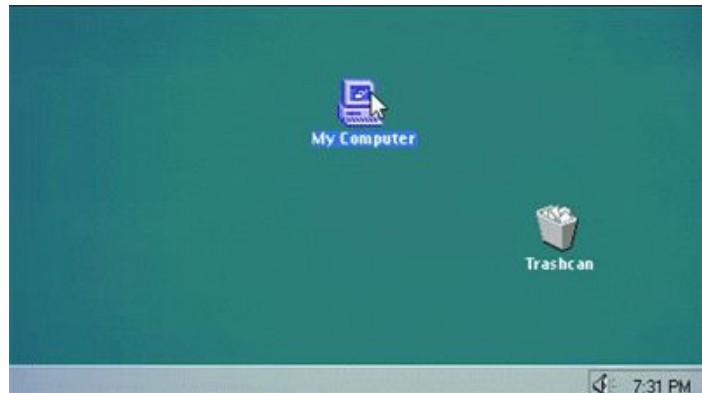
(2,4)-tree — delete(30)

- 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



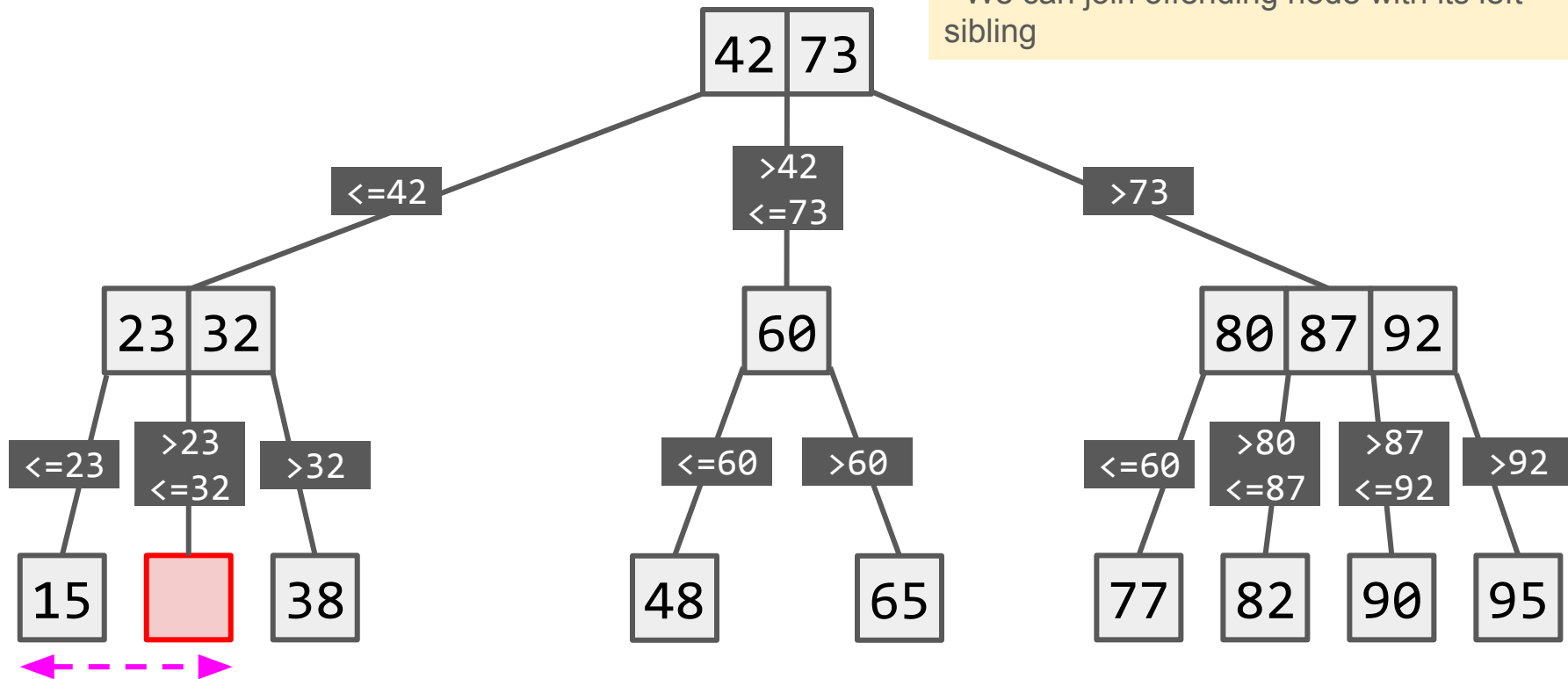
Deletion may violate rule 1!

- 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!



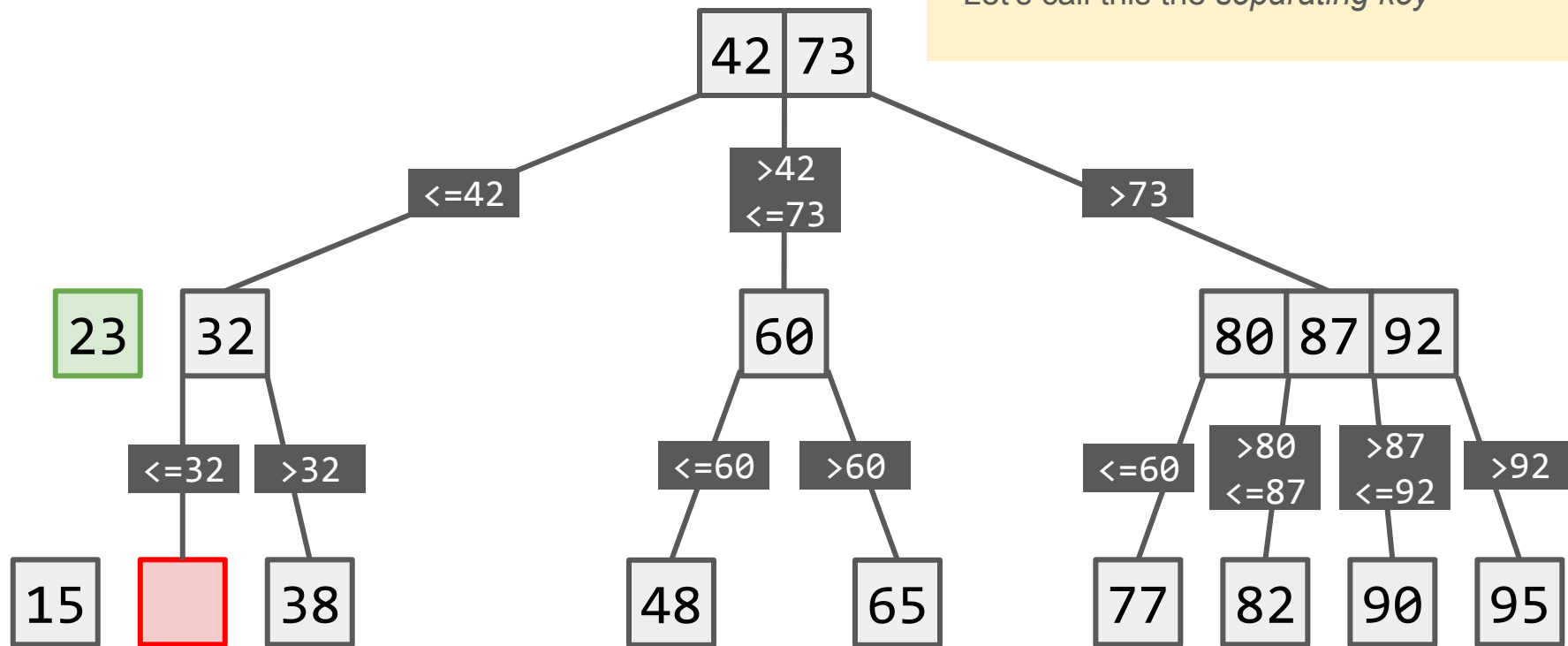
(2,4)-tree — delete(30)

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



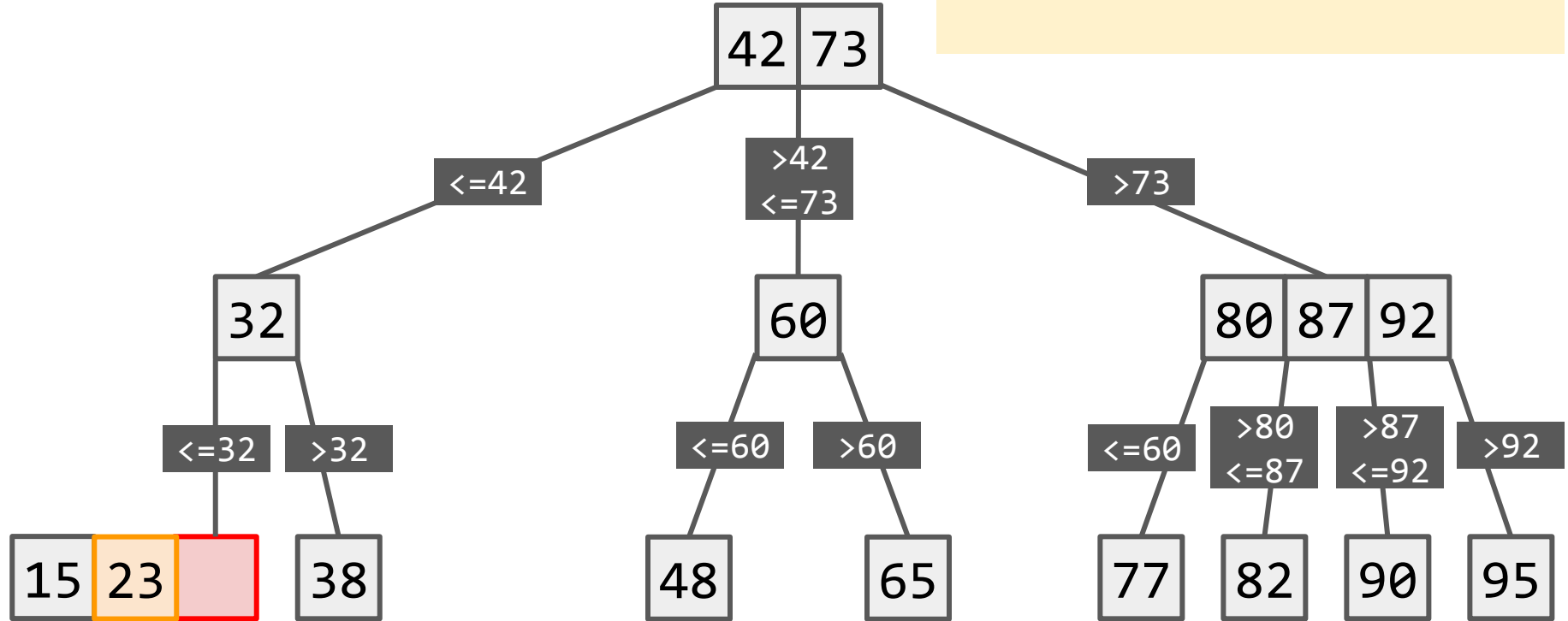
(2,4)-tree — delete(30)

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



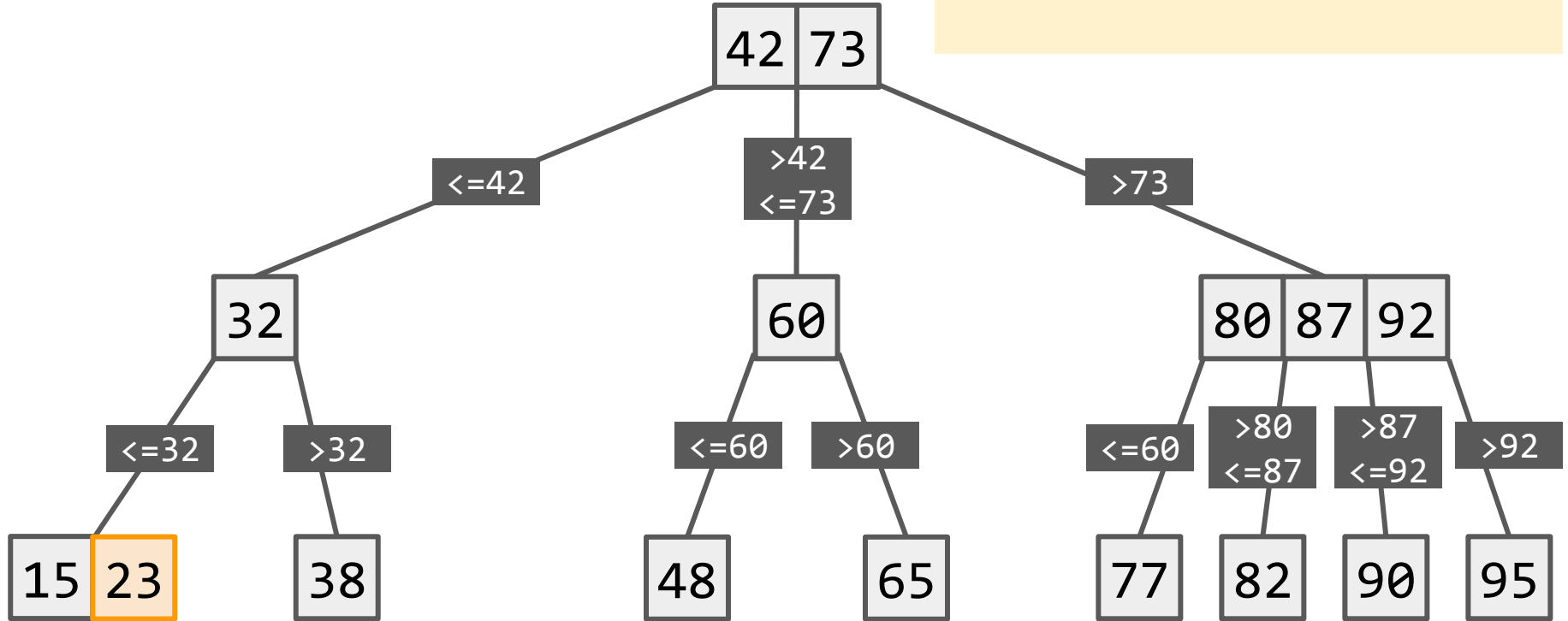
(2,4)-tree — delete(30)

- 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



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

Test yourself!

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

Test yourself!

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.

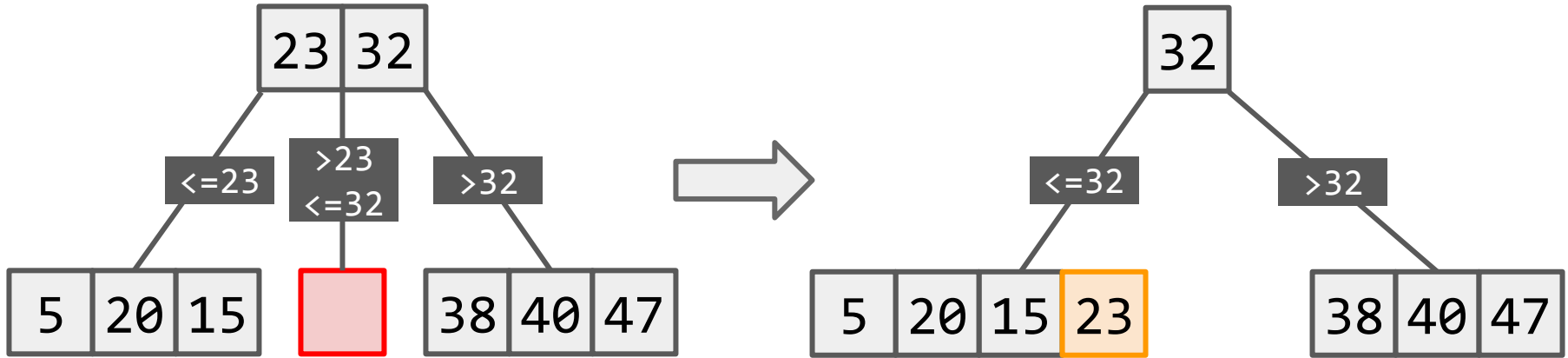
Test yourself!

Can a merged node be “too large”? I.e. violate rule 1

Test yourself!

Can a merged node be “too large”? I.e. violate rule 1

Answer: Yes. Consider this 2-4 tree just after a **deletion**:



Test yourself!

How do we resolve the case for when the combined group has keylist size exceeding b ?

Test yourself!

How do we resolve the case for when the combined group has keylist size exceeding b ?

Answer: merge them, then split them!

This is known as a share operation.

Deletion — summary of violation resolutions

Scenario	Operation	Algorithm
<ul style="list-style-type: none">• y and z are siblings• Have $< b-1$ keys together	$\text{merge}(y, z)$	<ol style="list-style-type: none">1. In parent, delete key v (the key separating the siblings)2. Add v to the keylist of y3. In y, merge in z's keylist and treelist4. Remove z
<ul style="list-style-type: none">• y and z are siblings• Have $\geq b-1$ keys together	$\text{share}(y, z)$	<ol style="list-style-type: none">1. $\text{merge}(y, z)$2. Split newly combined node using the regular split operation

Test yourself!

Can a share operation ever lead to a node with keylist size lower than $a-1$?

Test yourself!

Can a share operation ever lead to a node with keylist size lower than $a-1$?

Answer: No, because

- Offending node and sibling node have at least b keys together
- Recall by definition of (a,b) -trees, $2 \leq a \leq (b+1)/2$
- So after splitting:
 - They have combined $b+1$ keys together (one demoted from parent)
 - LHS with keylist size at least $\lfloor (b+1)/2 \rfloor \geq a$
 - RHS with keylist size at least $\lceil (b+1)/2 \rceil \geq a$

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)

Guiding question

What is another potential problem with deletion of keys from any arbitrary node? How can we circumvent that?

Guiding question

What is another potential problem with deletion of keys from any arbitrary node? How can we circumvent that?

Answer: Deletion of a key from an root/internal node can lead to orphaned children! We could swap out the key to be deleted with a leaf node key before deleting it off.

Guiding question

Suppose we are deleting key k_d . Which leaf node key should we swap out k_d with?

Guiding question

Suppose we are deleting key k_d . Which leaf node key should we swap out k_d with?

Answer: The predecessor or successor key to k_d . This has to be a leaf node! Why? Think about how to obtain the predecessor/successor of a key.


Corollary

For the case without duplicate keys, if a node contains *any* keys in its keylist that is consecutive ordering (i.e. ..., k , $k+1$), then the node must be a leaf node!

Full deletion algorithm

Deletion of x from the tree:

1. Start at node $w = \text{root}$
2. Repeat until w contains x or w is a leaf:
 - a. If w contains $a - 1$ keys and z is a sibling of w
 - If w and z together contain $< b - 1$ keys,
 1. $\text{merge}(w, z)$
 2. Set w to be the newly merged node
 - Else if w and z together contain $\geq b - 1$ keys,
 1. $\text{share}(w, z)$
 2. Set w to be the newly split node whose key range contains x
 - b. Search for x in the keylist of w
 - If x is larger than all the keys, set w to be the last child
 - Else, look for the first key $v_i \geq x$ and set w to be t_i (subtree associated with v_i)
3. Check for x in w 's keylist,
 - a. If it exists, remove it and return success
 - b. Else, return failure



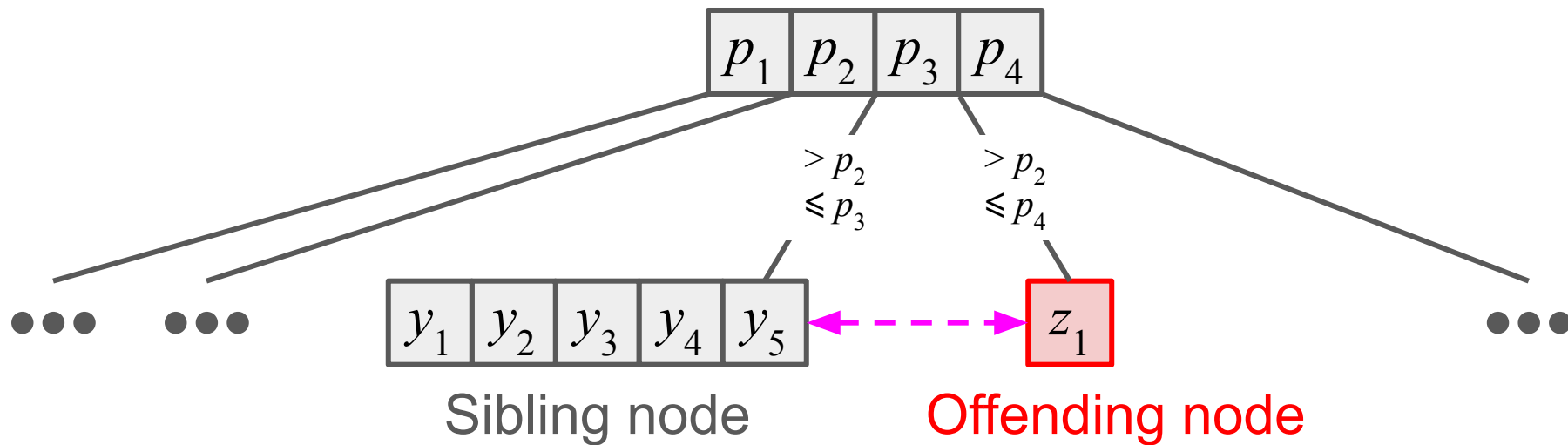
Proactive approach

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.

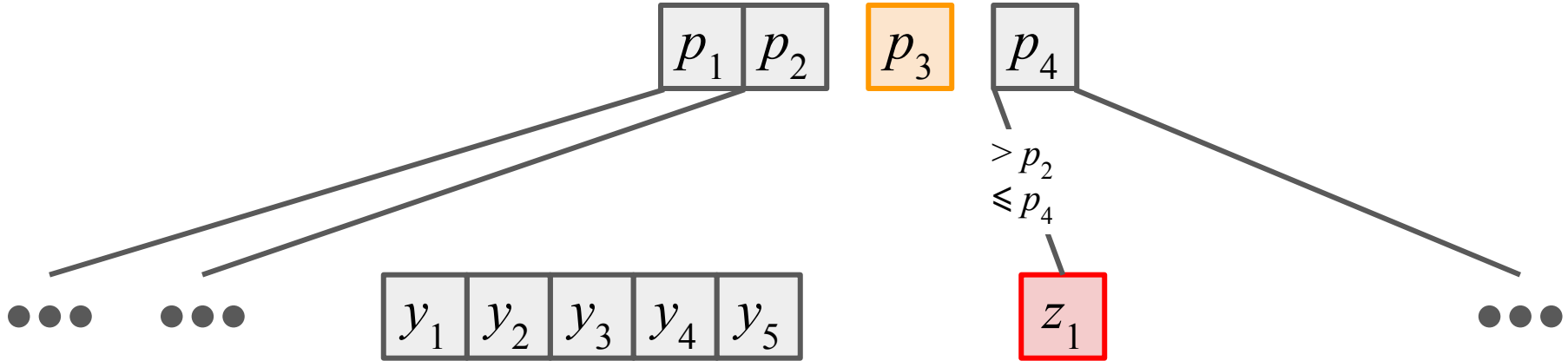
What is the cost of deleting from an (a,b) -tree?

(3,6)-tree — share operation



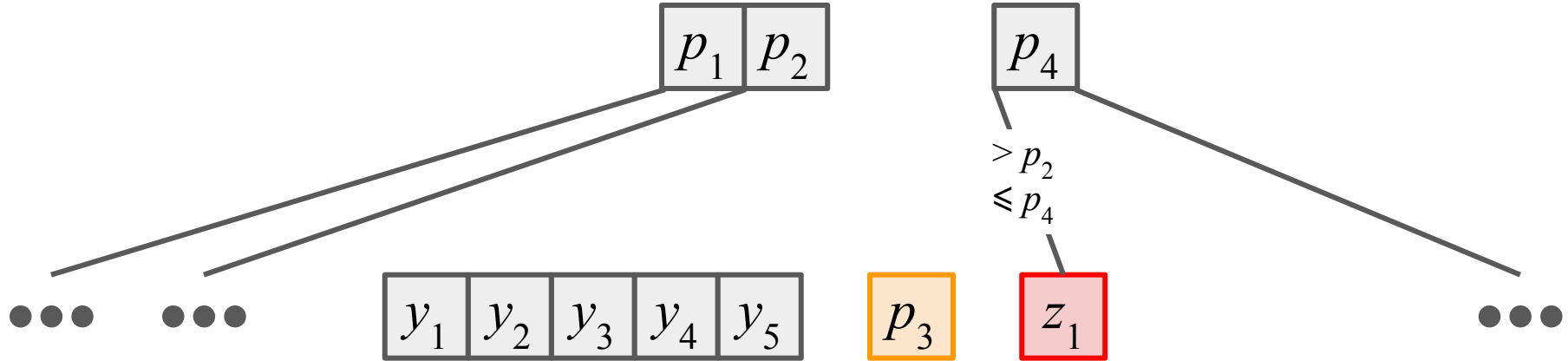
- z is the offending node (must have at least 2 keys in a (3,6)-tree)
- We first $\text{merge}(y, z)$

(3,6)-tree — share operation



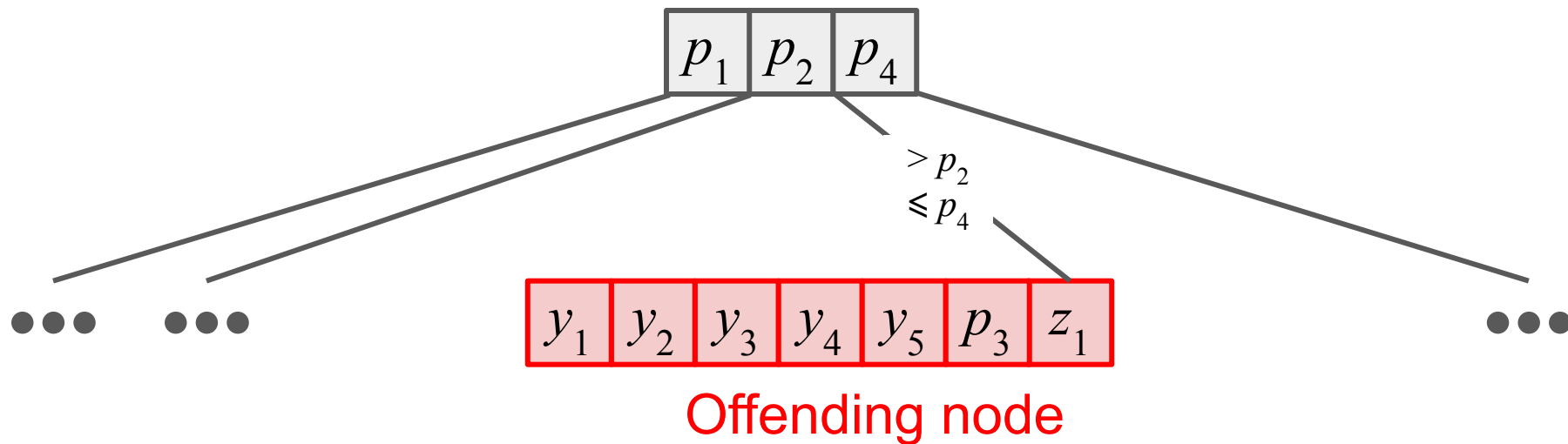
- p_3 is the key in parent node separating nodes y and z

(3,6)-tree — share operation



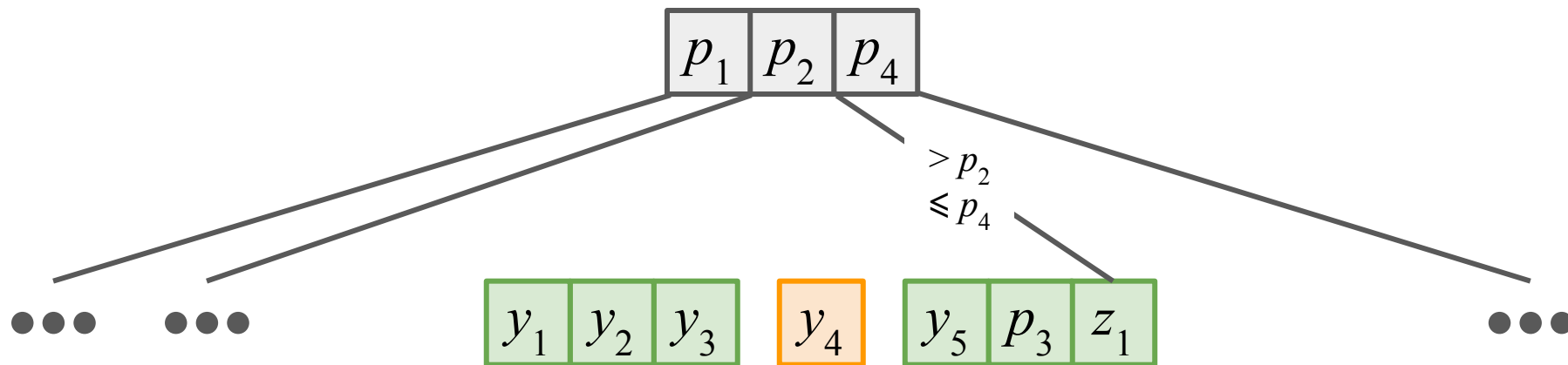
- p_3 shall be now used as the “interfacing” for joining nodes y and z

(3,6)-tree — share operation



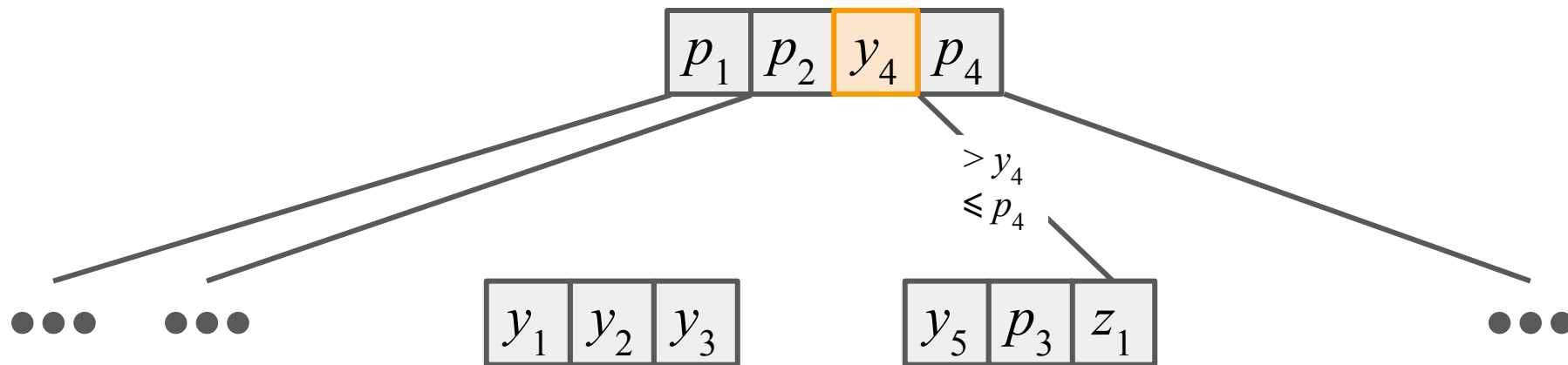
- But alas! After merging, our new node becomes an offending node (can have at most 5 keys in a (3,6)-tree)!

(3,6)-tree — share operation



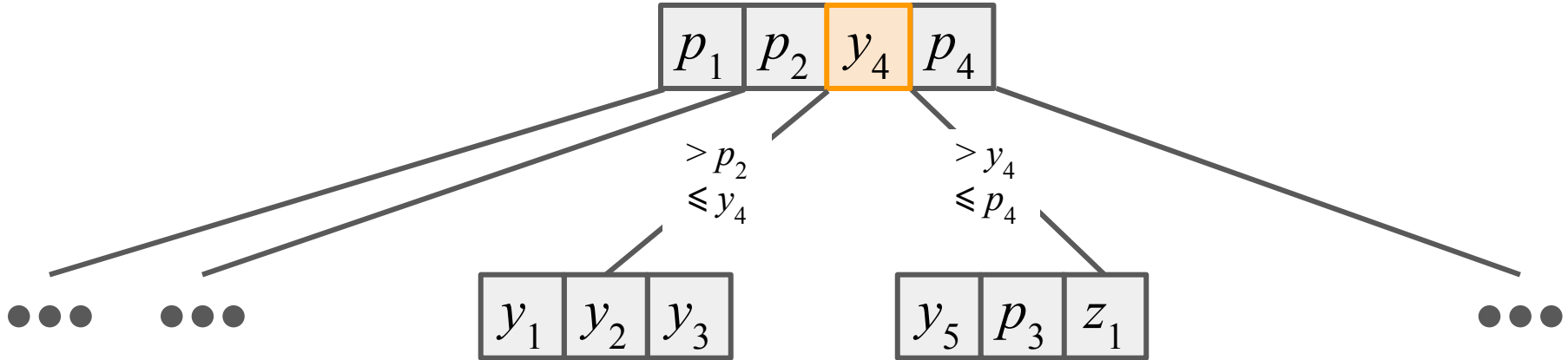
- We resolve that violation using the **split** operation
- The split key is chosen from median y_4

(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

Test yourself!

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

Test yourself!

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 be 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

For your own reading :)

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.
- To estimate the cost of searching data on disk, let us only count the number of blocks we have to move from disk to memory

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.

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

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 $a=B/2$ and $b=B$, or $a=B/4$ and $b=B/2$, etc., depending on how you want to optimize.)

Notice that each node in your B-tree can now be stored in $O(1)$ 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

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 $O(1)$

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.

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 $O(\log_B n)$. I.e., the cost only depends on the height of the tree, since each of the operations to access a single node is only cost $O(1)$.

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 $O(\log_B n)$. In practice, B is a pretty large number.

For instance. If your disk has 16KB blocks (which is reasonably normal) and you set $B=16K$, 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 when you encounter B-trees online