-Trees

CS2040S, AY19/20 Sem 1

Eldon Chung | eldon.chung@u.nus.edu
Wang Zhi Jian | wzhijian@u.nus.edu

Where we are so far

So now you have seen some balanced trees, one more practical than the other.

Where we are so far

So now you have seen some balanced trees, one more practical than the other.

But both these trees still share a common factor that can actually still be somewhat “optimised”.

Where we are so far

So now you have seen some balanced trees, one more practical than the other.

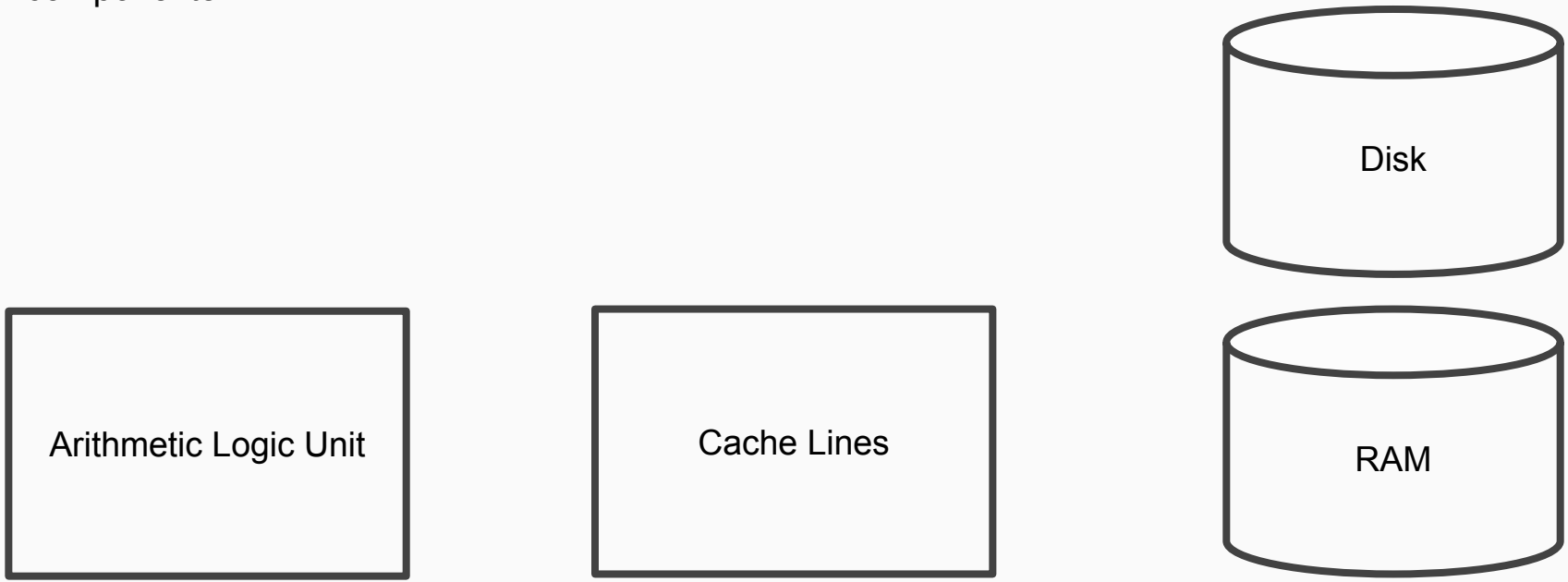
But both these trees still share a common factor that can actually still be somewhat “optimised”.

Let us temporarily step outside of our world of asymptotics for a while and take a look at real world CPUs.

A smol peek into CPUs

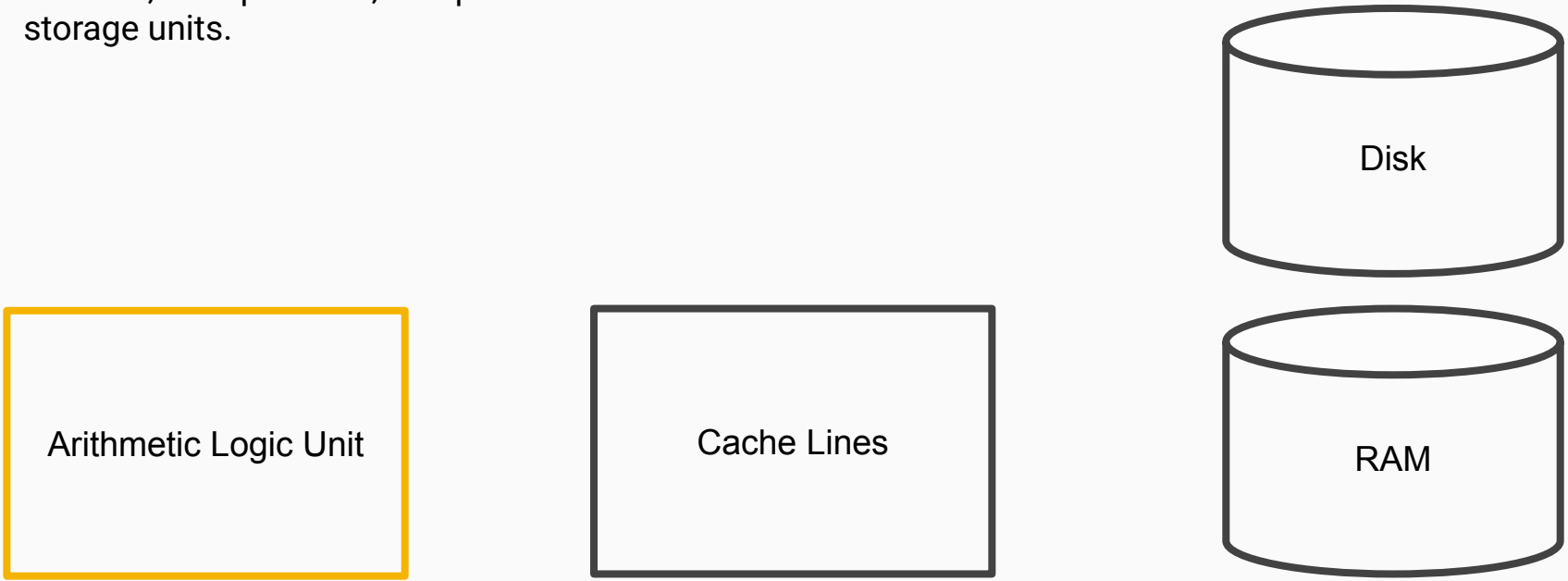
The anatomy of a CPU (abridged)

Let’s take a look at a very simple view of the CPU, along with other relevant components.



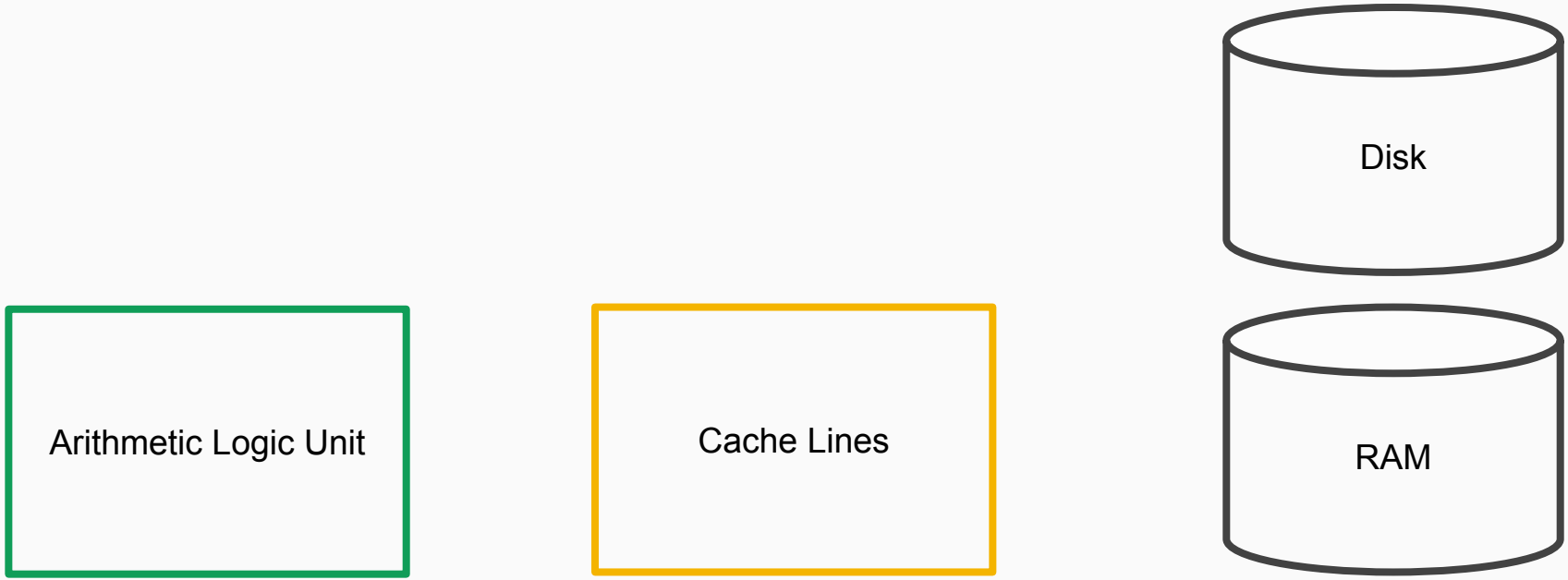
The anatomy of a CPU (abridged)

Firstly we have the arithmetic logic unit (ALU). It basically does all the operations, like addition, multiplication, comparisons etc. It needs to fetch information from the storage units.



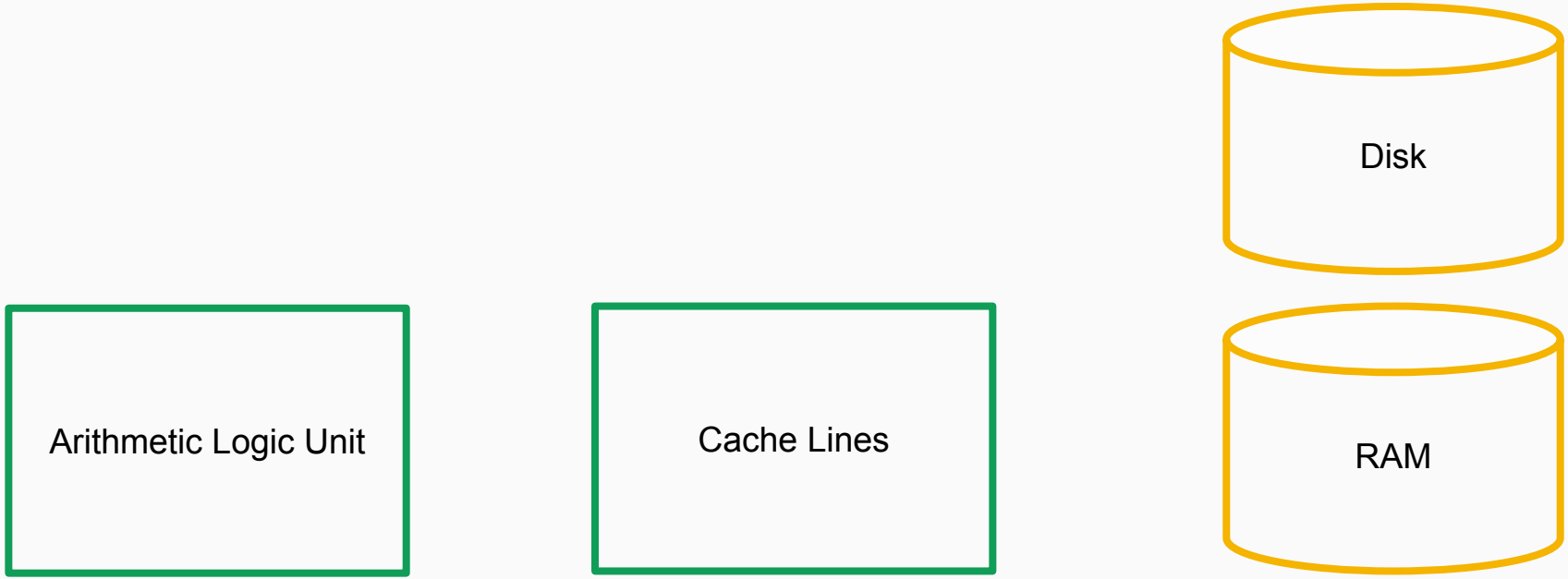
The anatomy of a CPU (abridged)

Next, we have the cache lines. Think of them as a staging area for where the CPU can store some information.



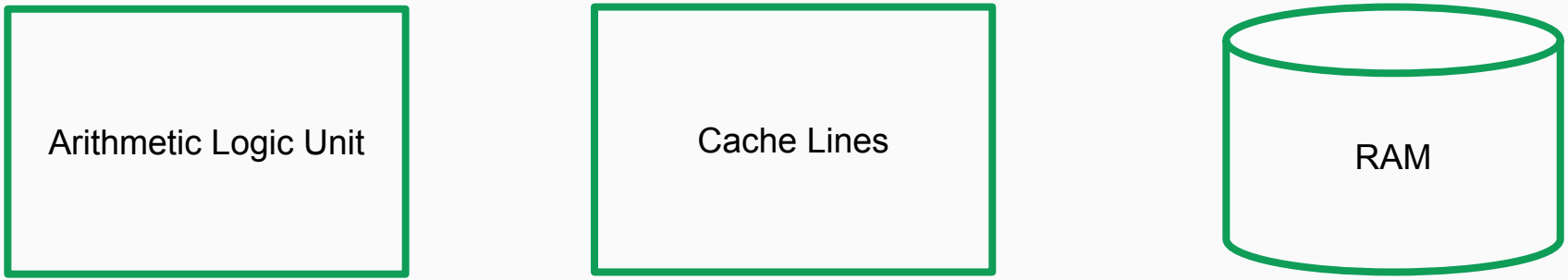
The anatomy of a CPU (abridged)

Lastly, we have RAM and Disk, which basically for our intents and purposes, store all the relevant data.



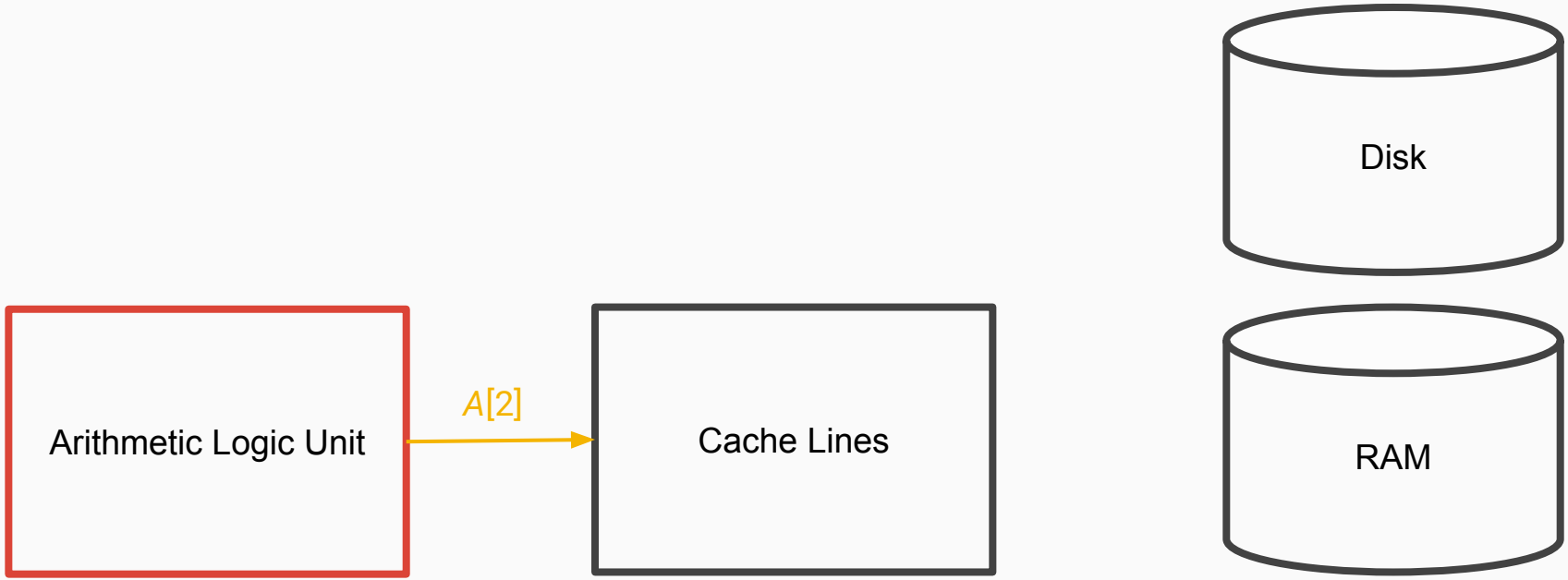
The anatomy of a CPU (abridged)

Okay! So what happens when a CPU wants some information to work on? For example say the CPU now needed to add a value to some element in an Array? E.g. the third element of some array *A*.



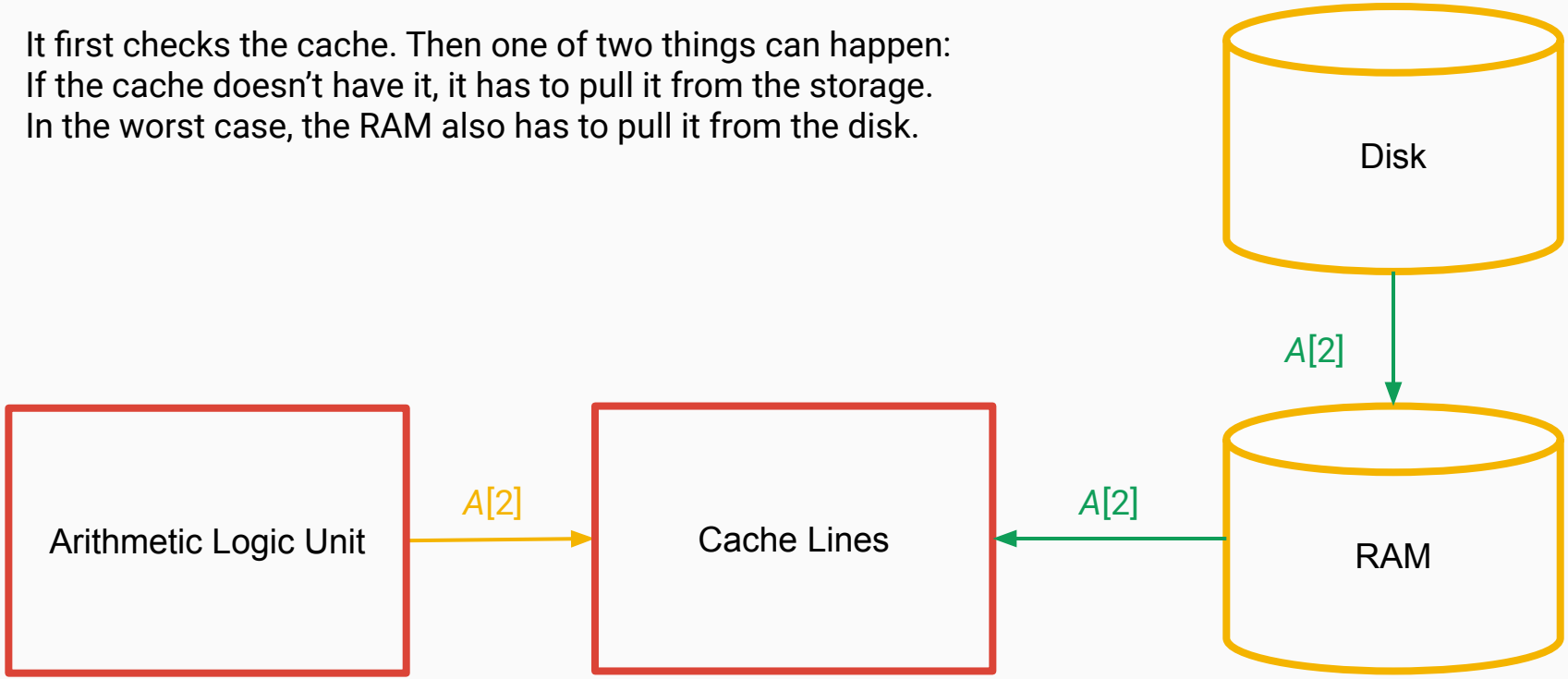
The anatomy of a CPU (abridged)

It first checks the cache. Then one of two things can happen:



The anatomy of a CPU (abridged)

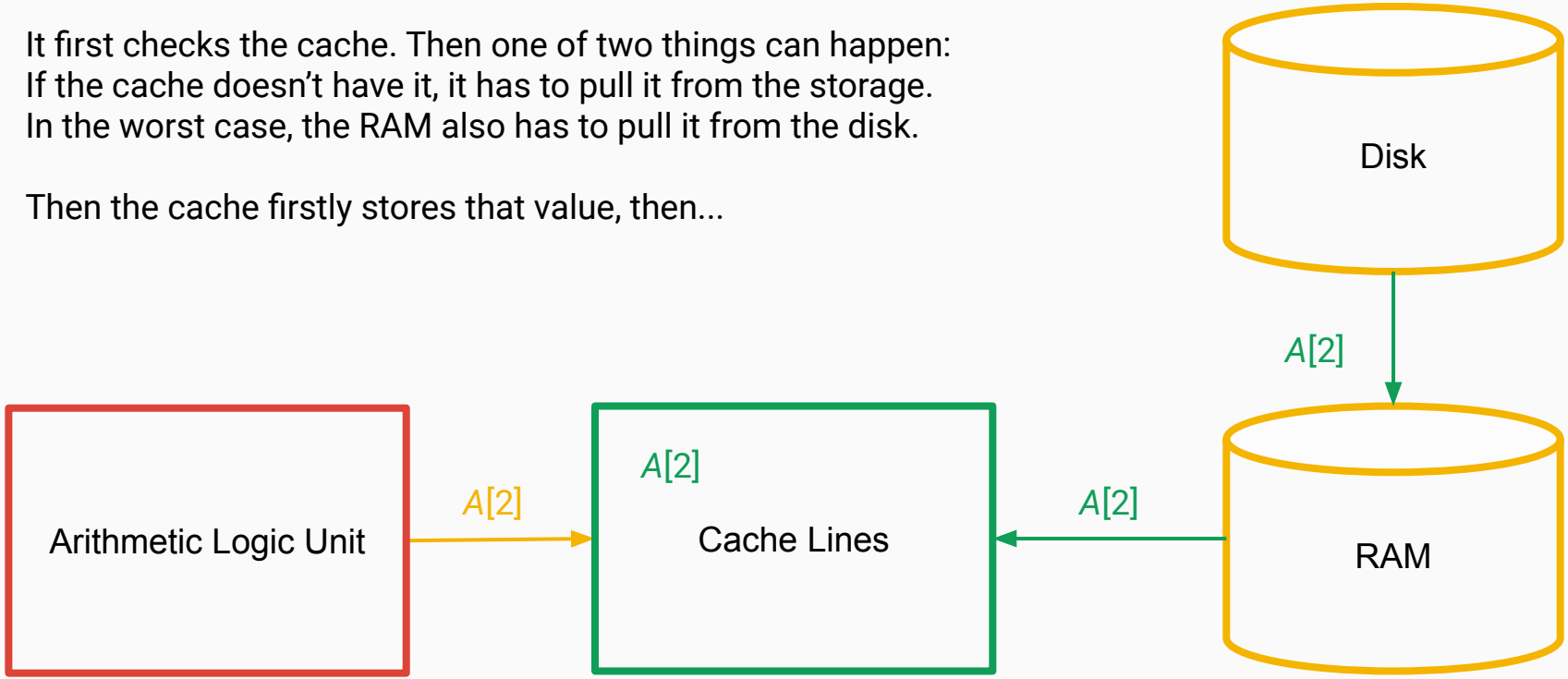
It first checks the cache. Then one of two things can happen:
If the cache doesn't have it, it has to pull it from the storage.
In the worst case, the RAM also has to pull it from the disk.



The anatomy of a CPU (abridged)

It first checks the cache. Then one of two things can happen:
If the cache doesn't have it, it has to pull it from the storage.
In the worst case, the RAM also has to pull it from the disk.

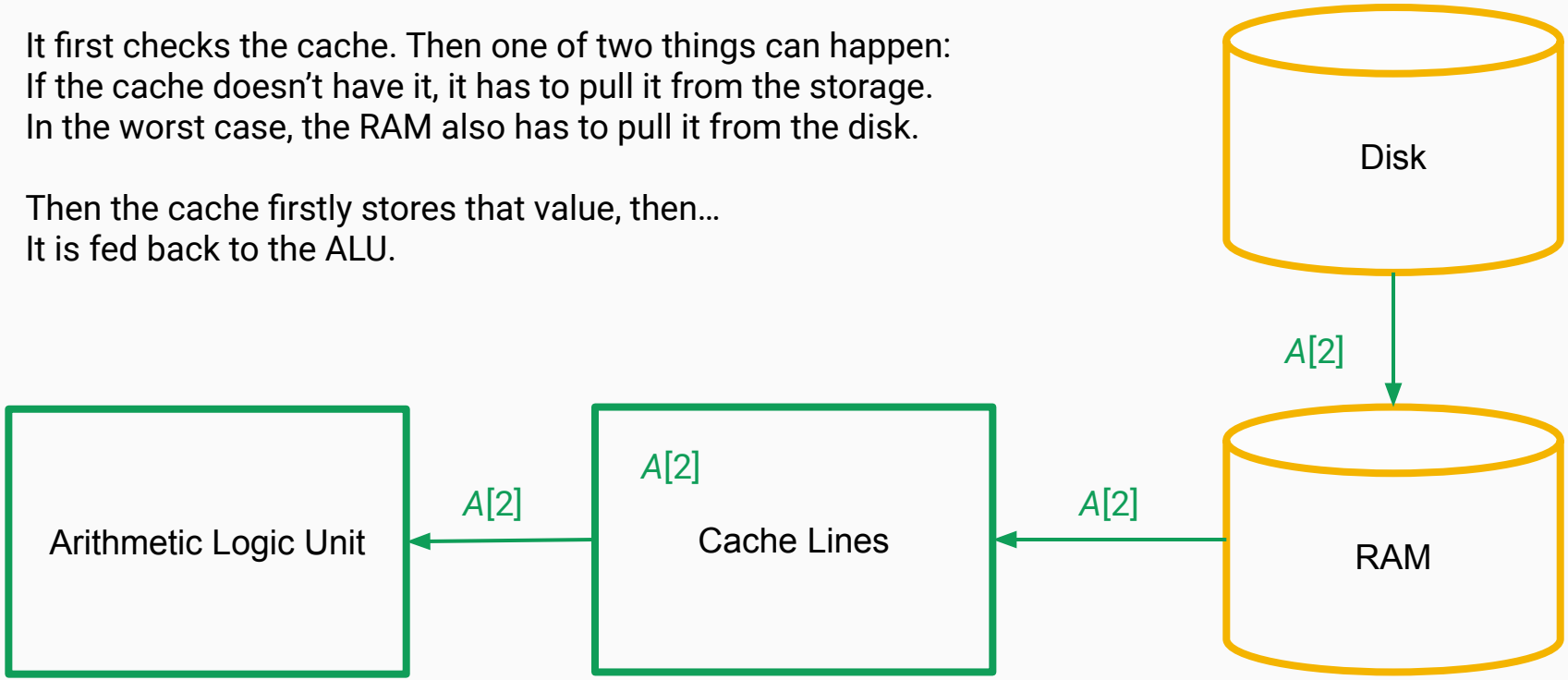
Then the cache firstly stores that value, then...



The anatomy of a CPU (abridged)

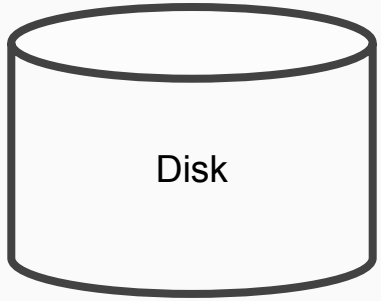
It first checks the cache. Then one of two things can happen:
If the cache doesn't have it, it has to pull it from the storage.
In the worst case, the RAM also has to pull it from the disk.

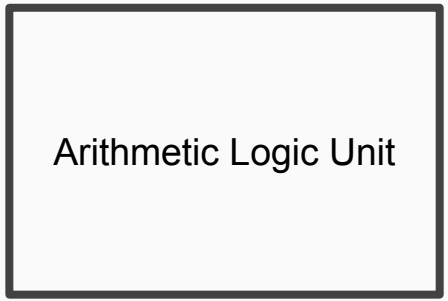
Then the cache firstly stores that value, then…
It is fed back to the ALU.

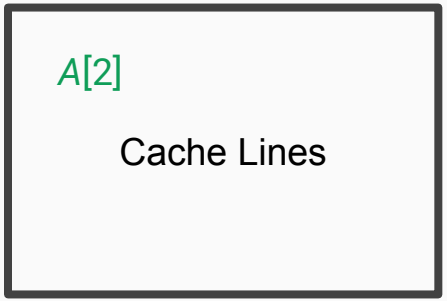


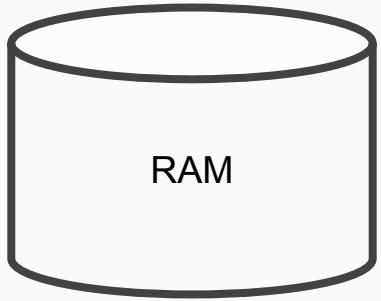
The anatomy of a CPU (abridged)

Okay! What happens now if the ALU wishes to ask for *A*[2] again?



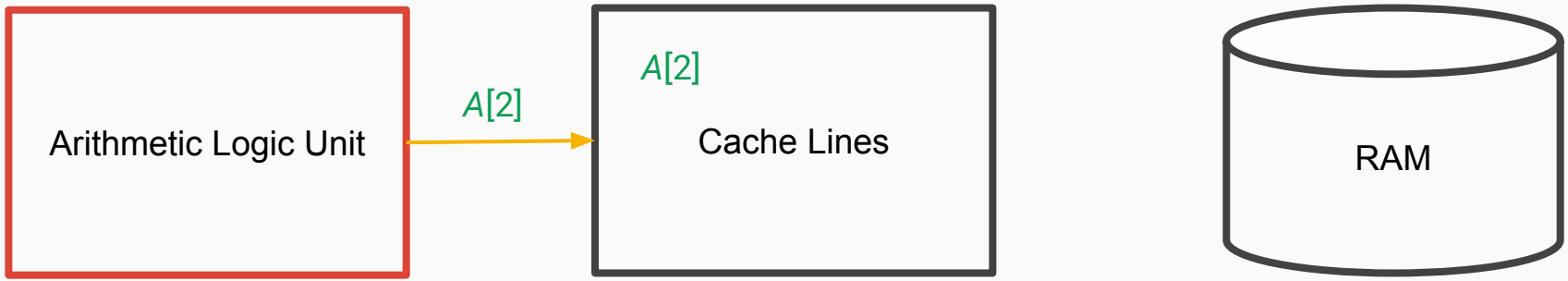






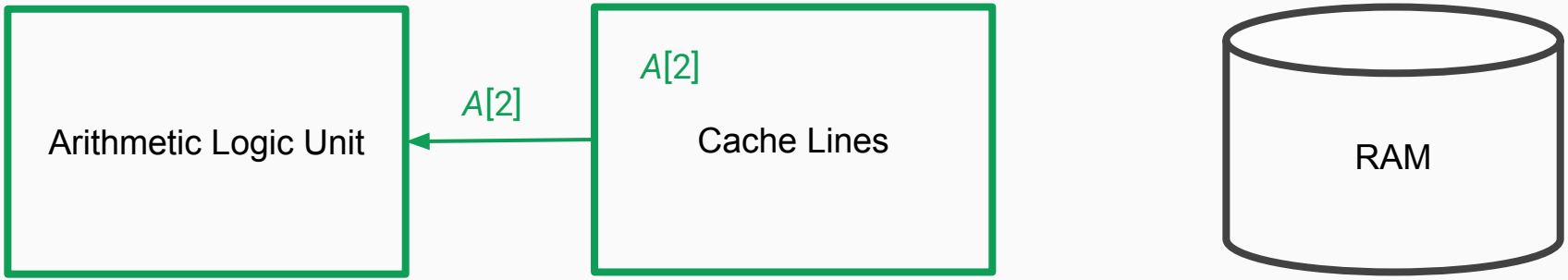
The anatomy of a CPU (abridged)

Okay! What happens now if the ALU wishes to ask for *A*[2] again?
It checks the cache aaaand?



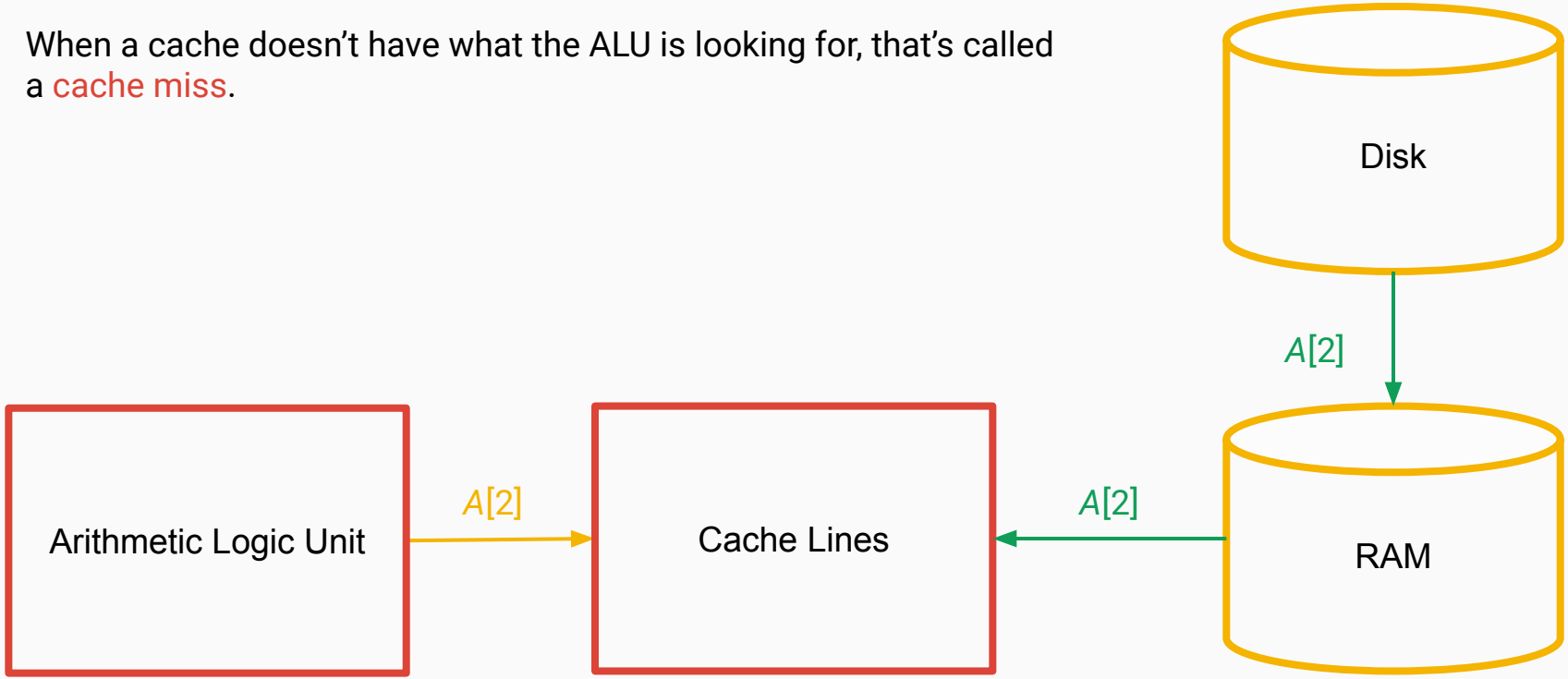
The anatomy of a CPU (abridged)

Okay! What happens now if the ALU wishes to ask for *A*[2] again?
It checks the cache aaaand?
Bingo! The cache does not need to go back to the disk or RAM for it.



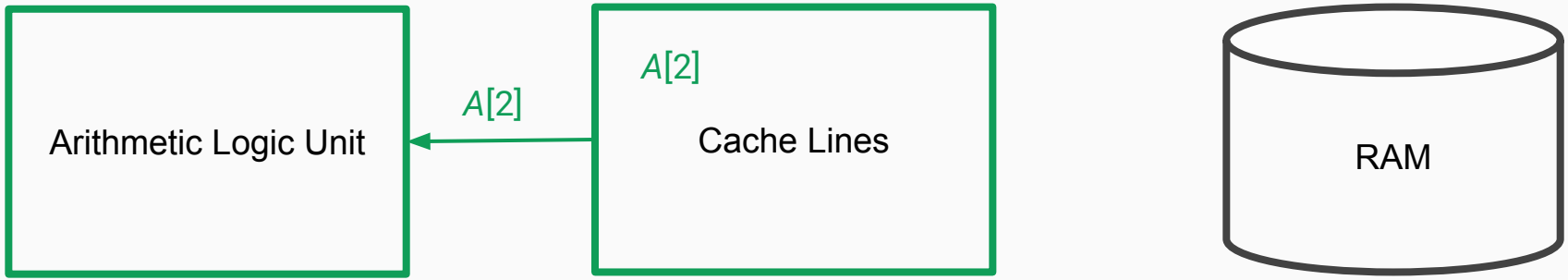
The anatomy of a CPU (abridged)

When a cache doesn't have what the ALU is looking for, that's called a cache miss.



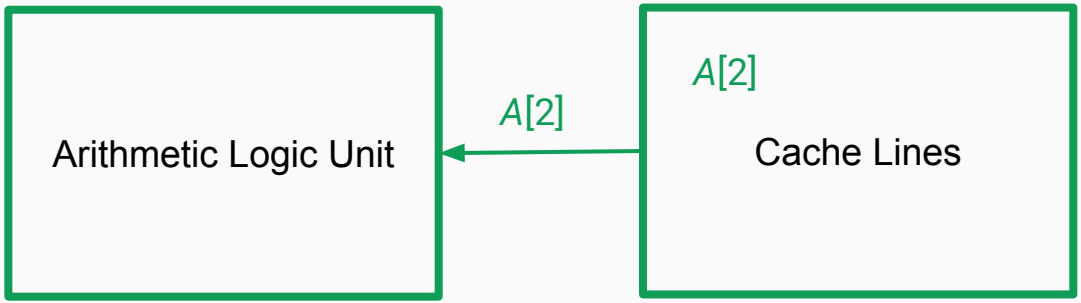
The anatomy of a CPU (abridged)

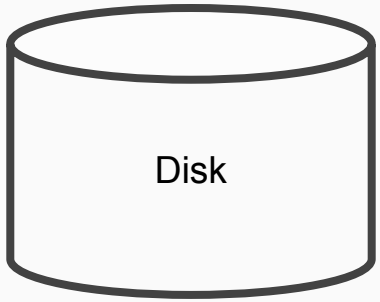
When a cache does have what the ALU is looking for, that's called a cache hit.

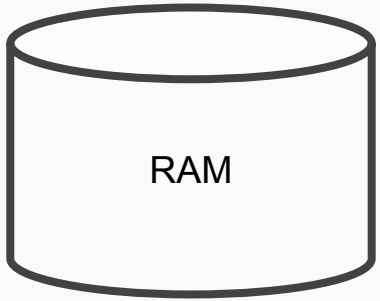


The anatomy of a CPU (abridged)

When a cache does have what the ALU is looking for, that's called a cache hit.
In general, a cache can store a few lines, so it can try to maximise the number of cache hits. In practice, there are even different levels of caches!

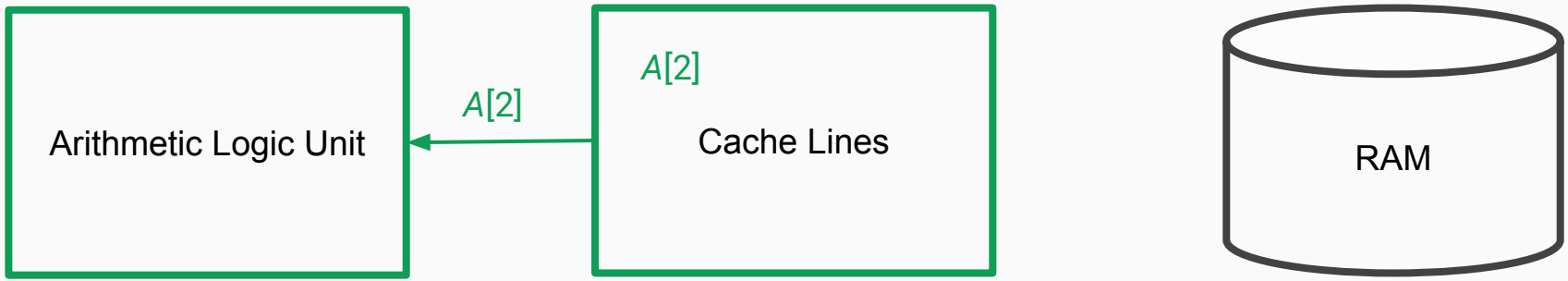






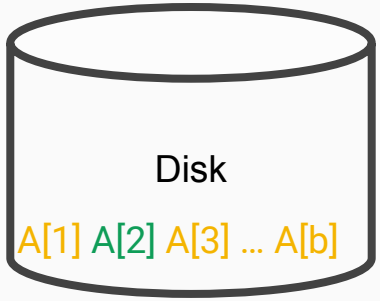
The anatomy of a CPU (abridged)

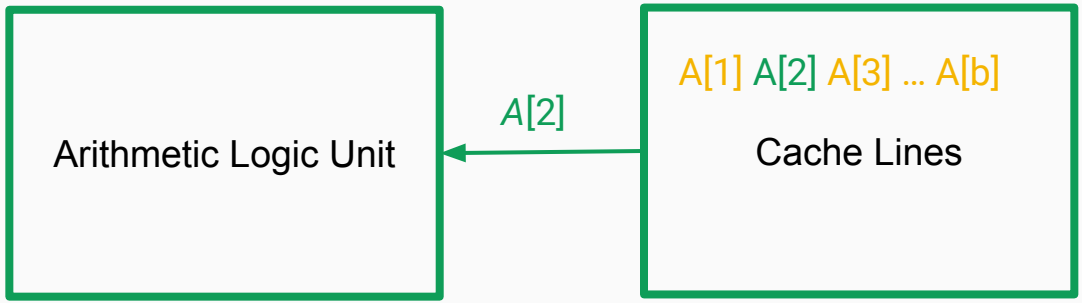
Last thing to note, when cache lines are loaded, it isn't done so for just singular values...

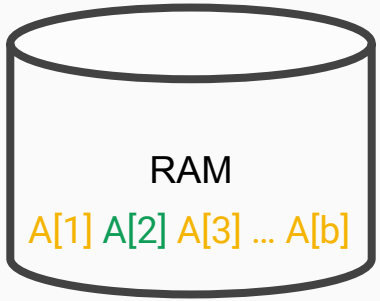


The anatomy of a CPU (abridged)

Last thing to note, when cache lines are loaded, it isn't done so for just singular values…
The cache will hold the **entire** contiguous chunk that the value is in!



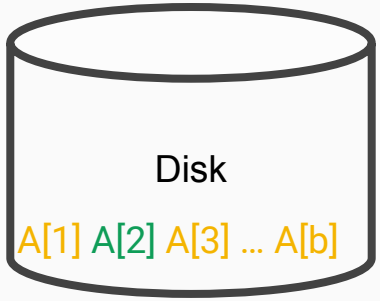


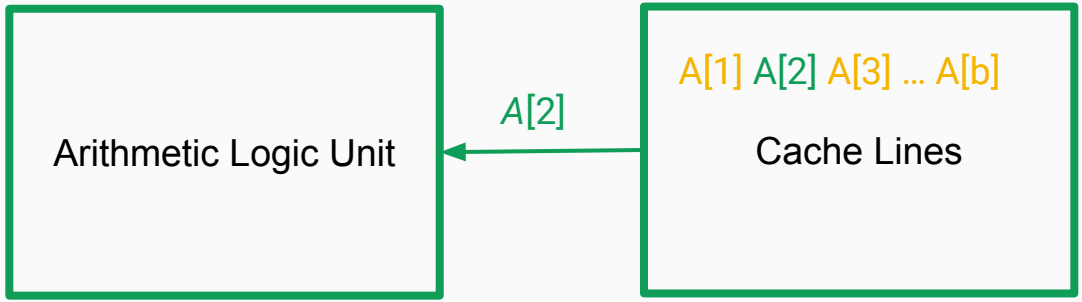


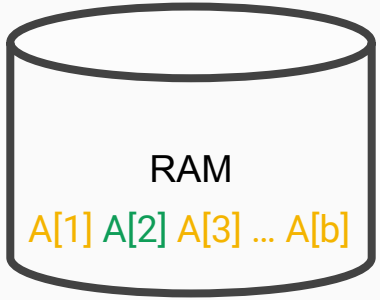
The anatomy of a CPU (abridged)

Last thing to note, when cache lines are loaded, it isn't done so for just singular values…
The cache will hold the **entire** contiguous chunk that the value is in!

In our example, A[1] is next to A[2], which is next to A[3] in the memory, so they all get loaded into the cache together, even though the algorithm only needs A[2].

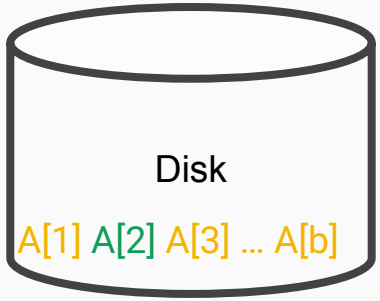


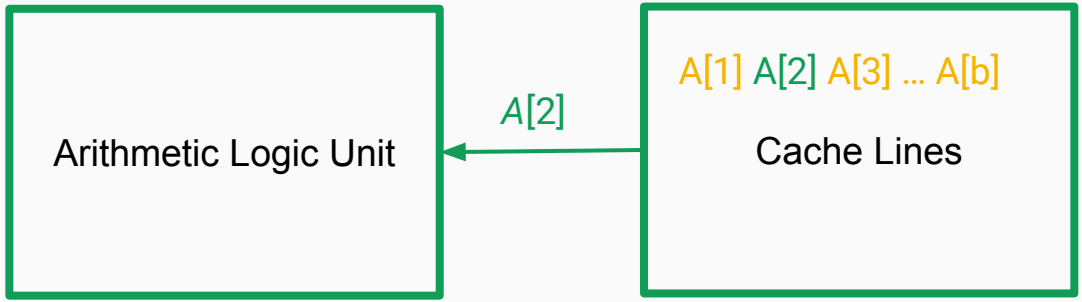


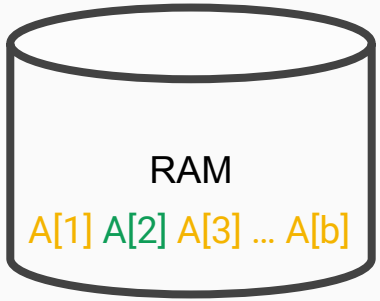


The anatomy of a CPU (abridged)

So how important are cache hits?







How good are cache hits?

Well let’s say that a (for simplicity) accessing cache only took 1 second. Then:

How good are cache hits?

Well let’s say that a (for simplicity) accessing cache only took 1 second. Then:

1. Accessing your RAM would take about 28 minutes.

How good are cache hits?

Well let’s say that a (for simplicity) accessing cache only took 1 second. Then:

1. Accessing your RAM would take about 28 minutes.
2. Accessing your SSD would take about 5 days.

How good are cache hits?

Well let’s say that a (for simplicity) accessing cache only took 1 second. Then:

1. Accessing your RAM would take about 28 minutes.
2. Accessing your SSD would take about 5 days.
3. Accessing your hard disk drive would take somewhere in between 1 to 5 years!

How good are cache hits?

Well let's say that a (for simplicity) accessing cache only took 1 second. Then:

1. Accessing your RAM would take about 28 minutes.
2. Accessing your SSD would take about 5 days.
3. Accessing your hard disk drive would take somewhere in between 1 to 5 years!

Imagine having to wait a few years to traverse down a tree!

How good are cache hits?

Well let's say that a (for simplicity) accessing cache only took 1 second. Then:

1. Accessing your RAM would take about 28 minutes.
2. Accessing your SSD would take about 5 days.
3. Accessing your hard disk drive would take somewhere in between 1 to 5 years!

Imagine having to wait a few years to traverse down a tree!

Every time you have to load a new node, there is no guarantee it is in the same cache line, you may have lots of cache misses.

How good are cache hits?

Well let's say that a (for simplicity) accessing cache only took 1 second. Then:

1. Accessing your RAM would take about 28 minutes.
2. Accessing your SSD would take about 5 days.
3. Accessing your hard disk drive would take somewhere in between 1 to 5 years!

Imagine having to wait a few years to traverse down a tree!

What should we do instead?

How good are cache hits?

Well let's say that a (for simplicity) accessing cache only took 1 second. Then:

1. Accessing your RAM would take about 28 minutes.
2. Accessing your SSD would take about 5 days.
3. Accessing your hard disk drive would take somewhere in between 1 to 5 years!

Imagine having to wait a few years to traverse down a tree!

What should we do instead? Make use of the cache! Get the cache to hold as many values as you need to check before having to load more values.

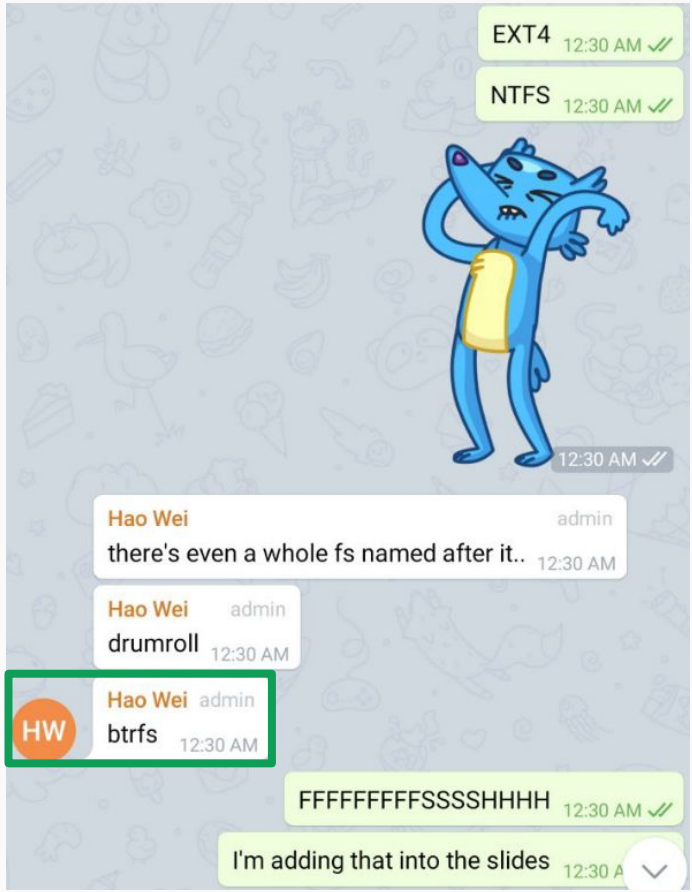
Motivations

- B-trees see heavy usage in database servers, where you might often need to find data from lots and lots and lots of hard drives. Imagine Facebook or Google having to retrieve your profile or your email from billions of other accounts.

- Variants of B-trees also see usage in popular OS filesystems, such as ext4 (Linux) or NTFS, HPFS (Windows), or HFS+ (Mac).

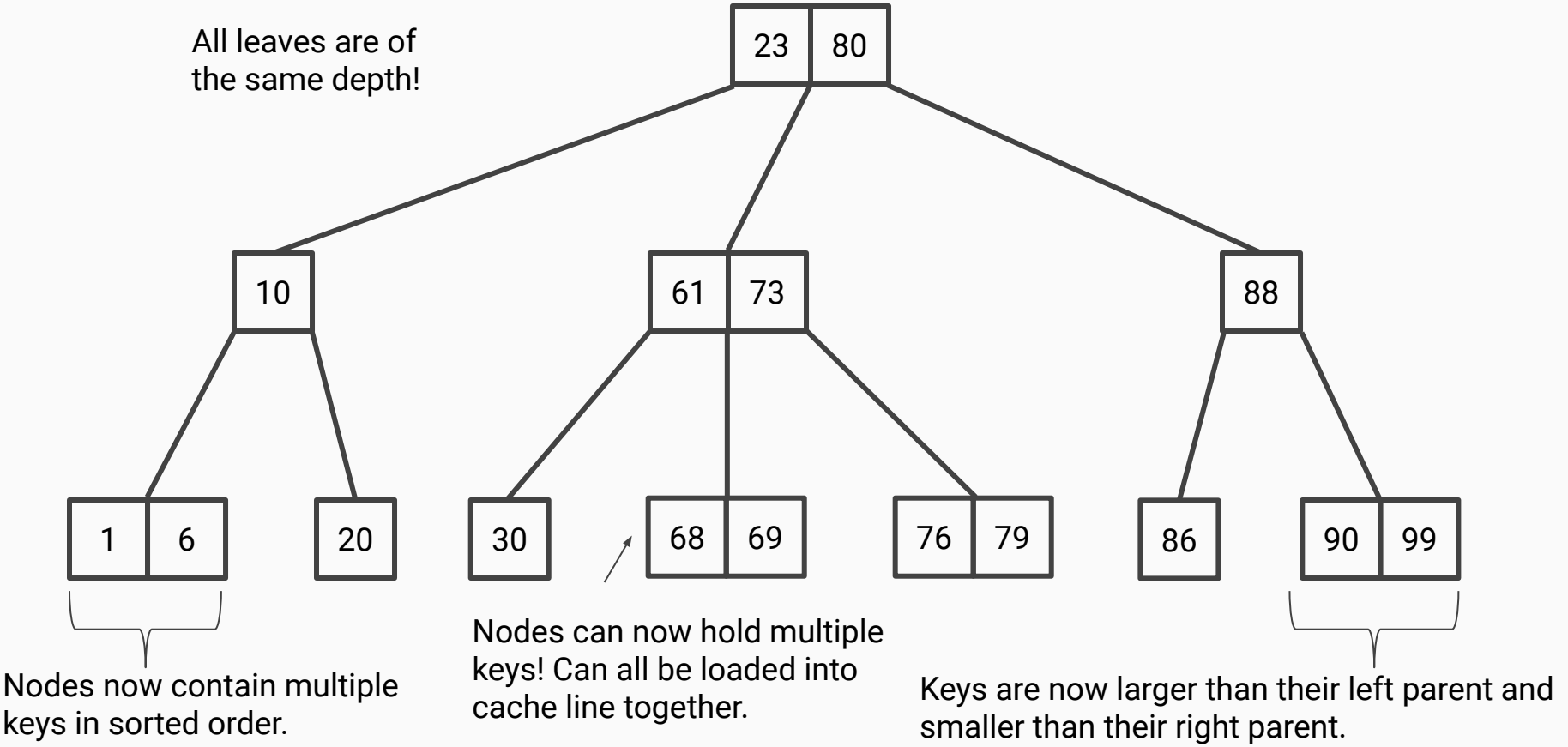
- Have a very nice notion of what is considered balanced!
- Our usual operations still run in log *n* time, with far better constants.

Here are some testimonials



Introducing 🅱️-Trees

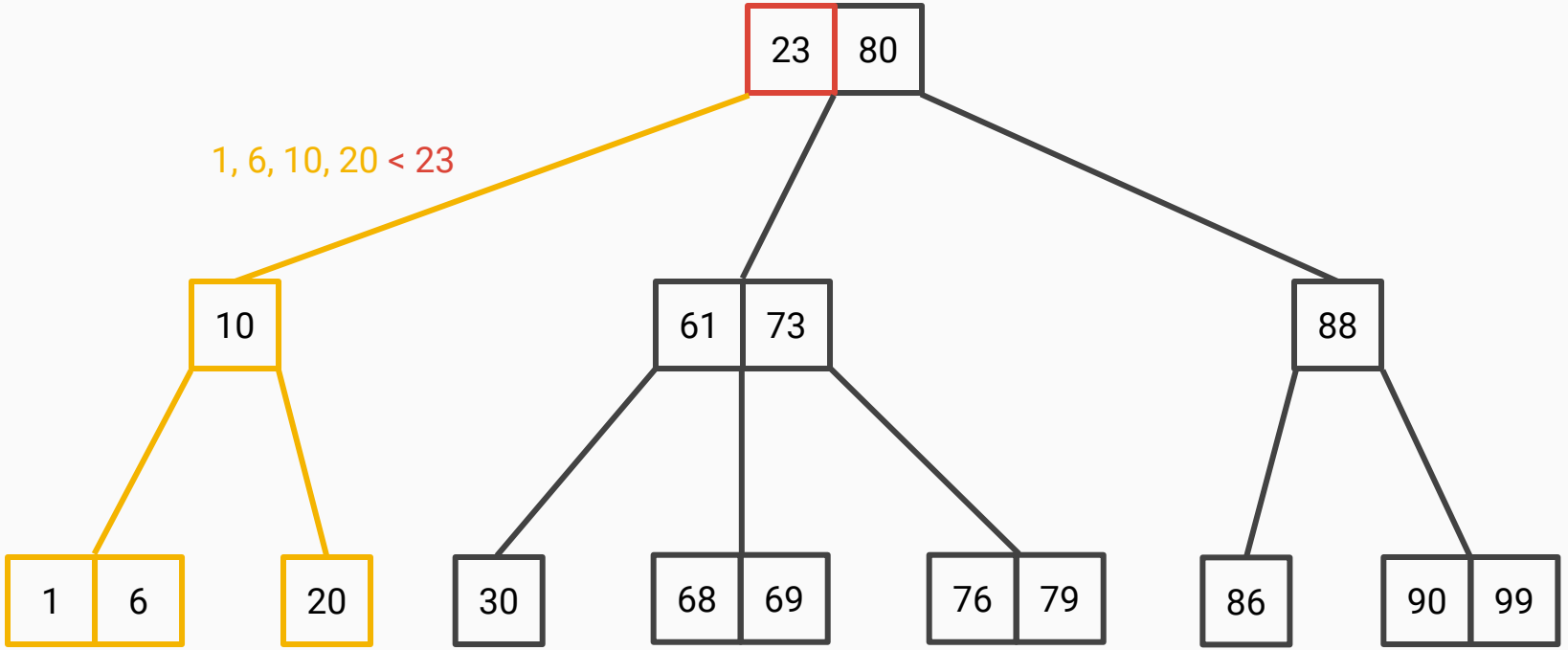
Structure of a B-tree



Properties of a B-Tree

Property 0

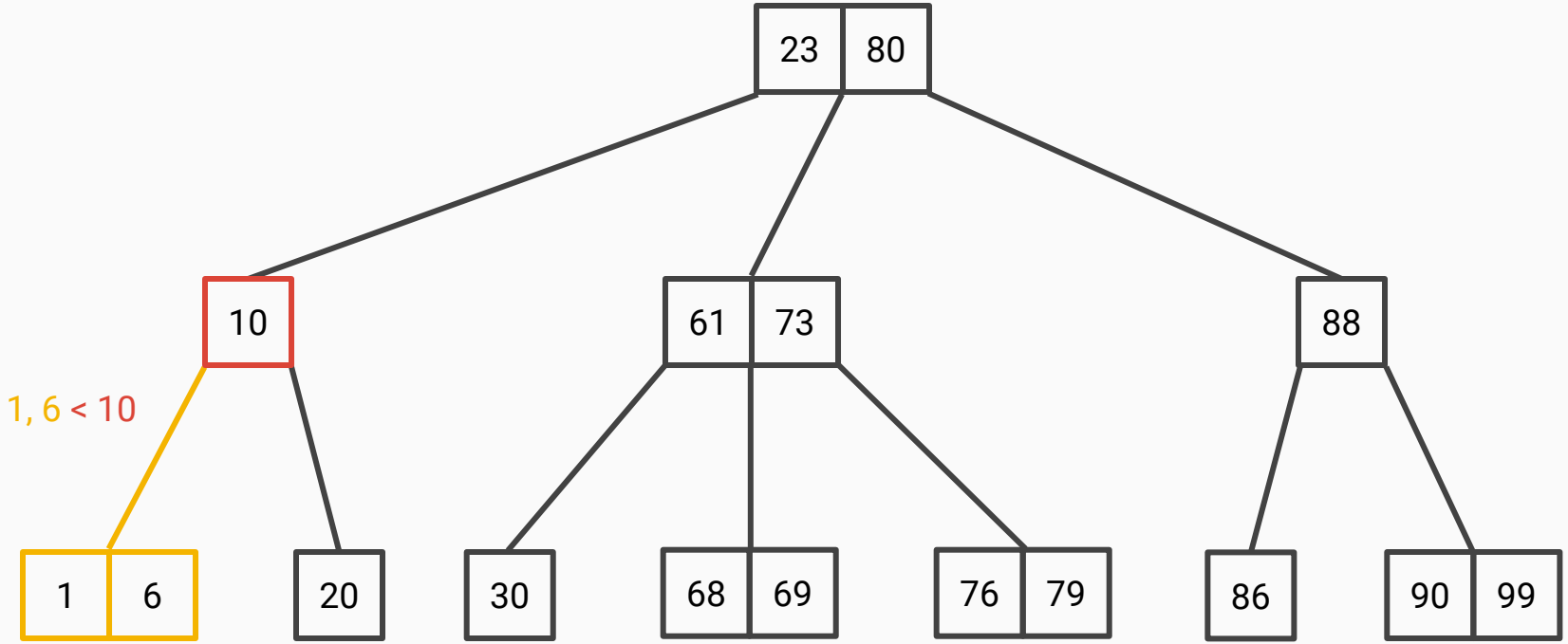
The BST property is maintained for all nodes in the B-Tree.



Properties of a B-Tree

Property 0

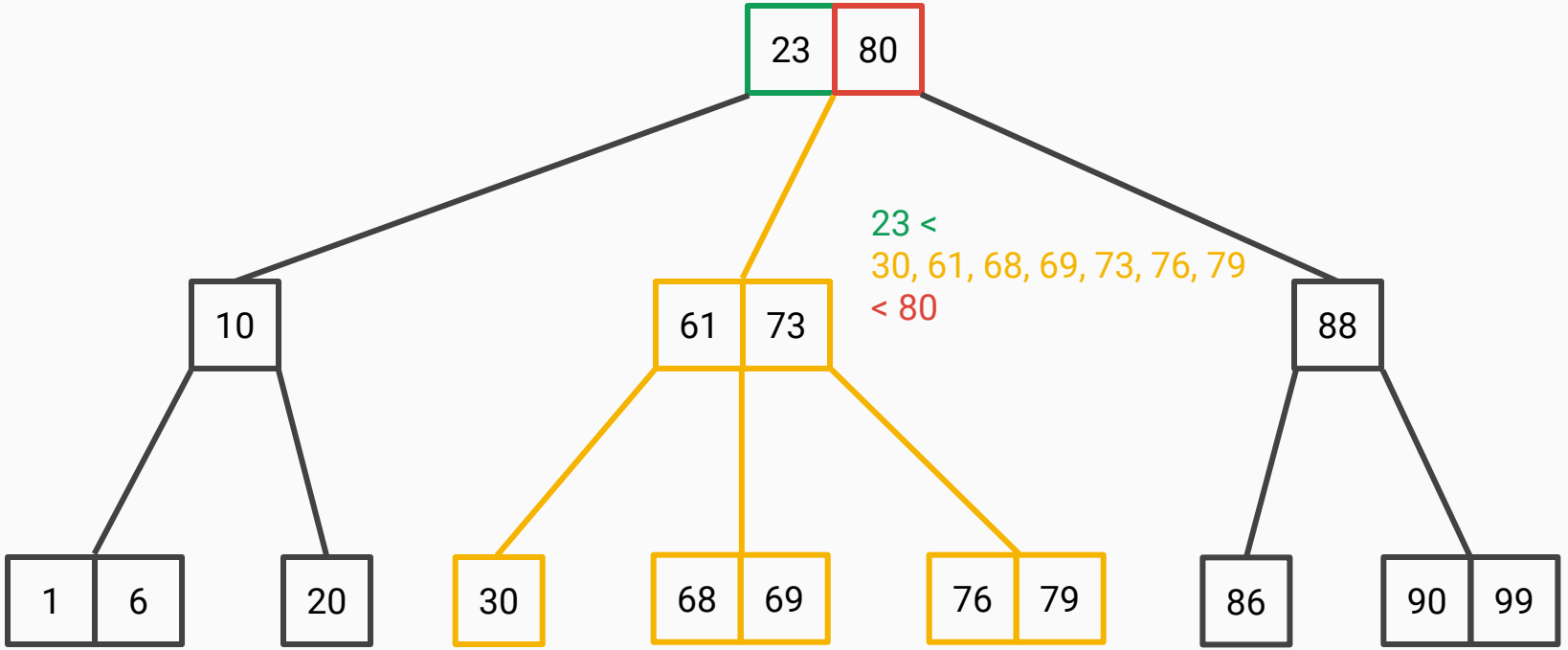
The BST property is maintained for all nodes in the B-Tree.



Properties of a B-Tree

Property 0

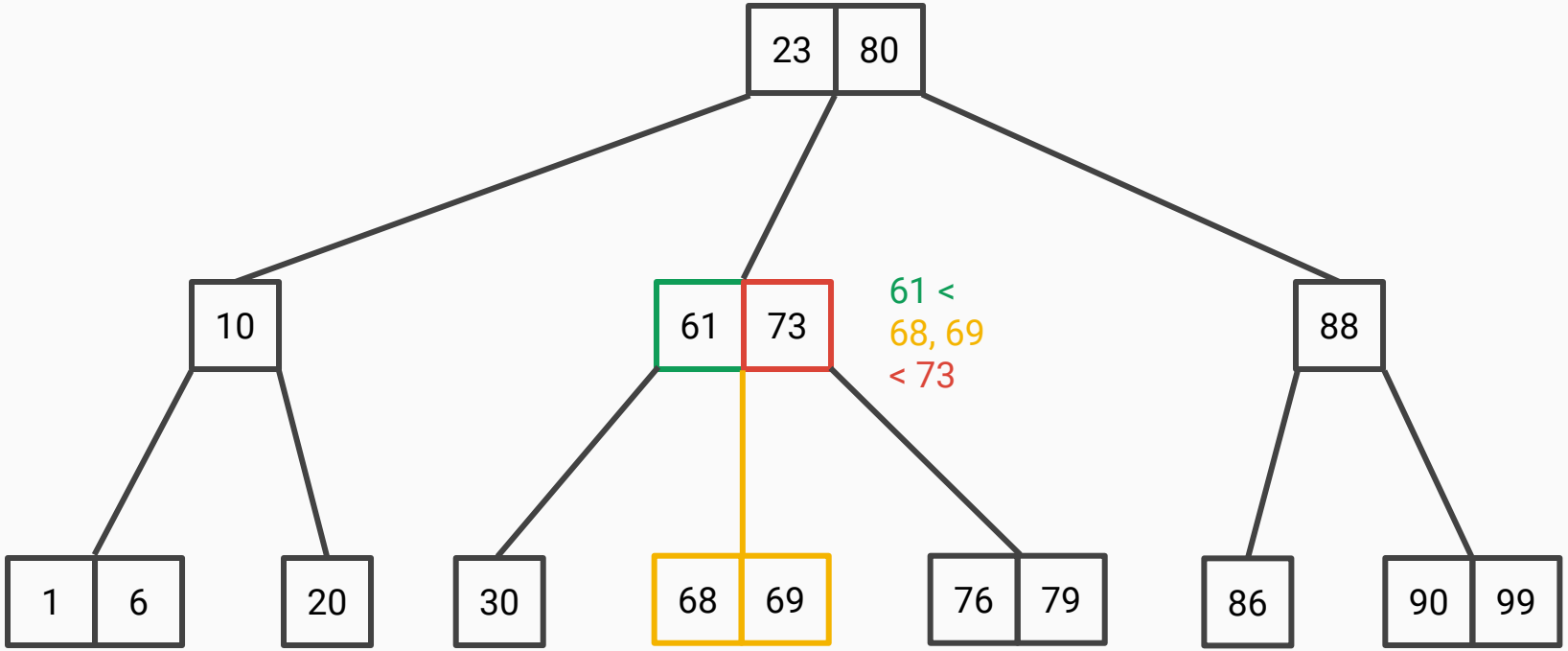
The BST property is maintained for all nodes in the B-Tree.



Properties of a B-Tree

Property 0

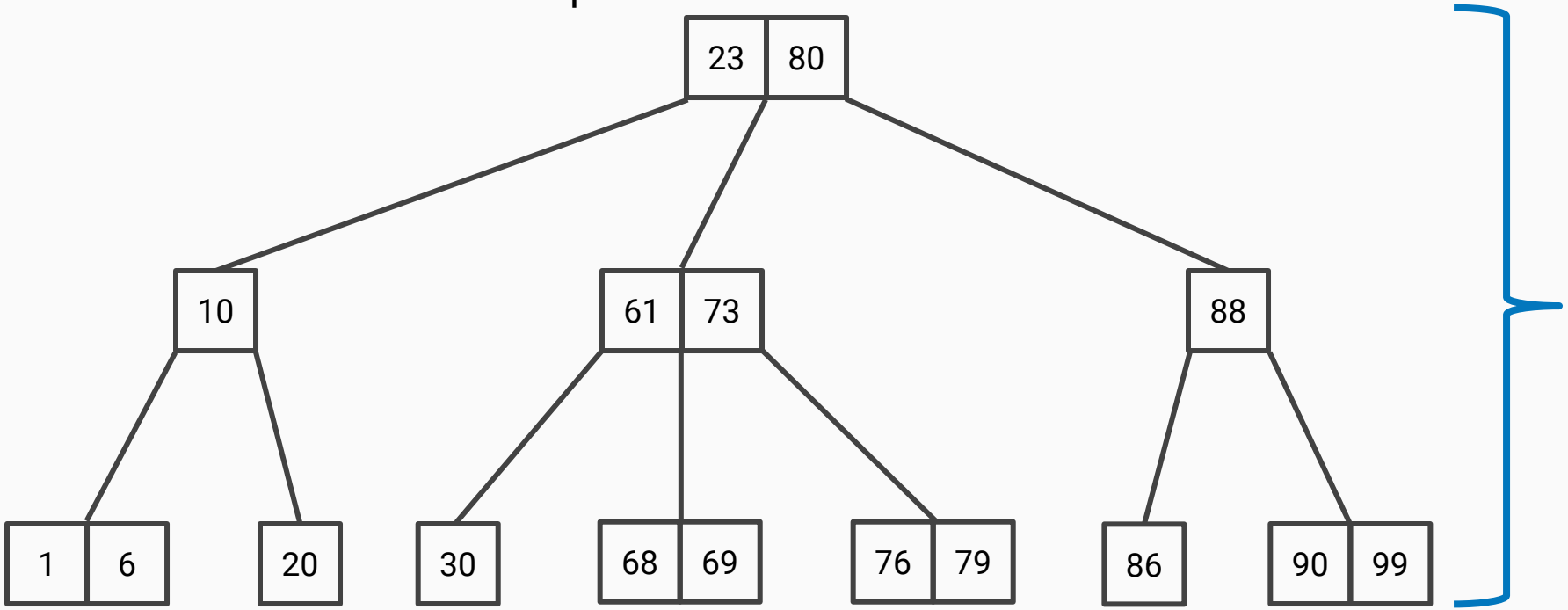
The BST property is maintained for all nodes in the B-Tree.



Properties of a B-Tree

Property 1

All leaf nodes are of the same depth.



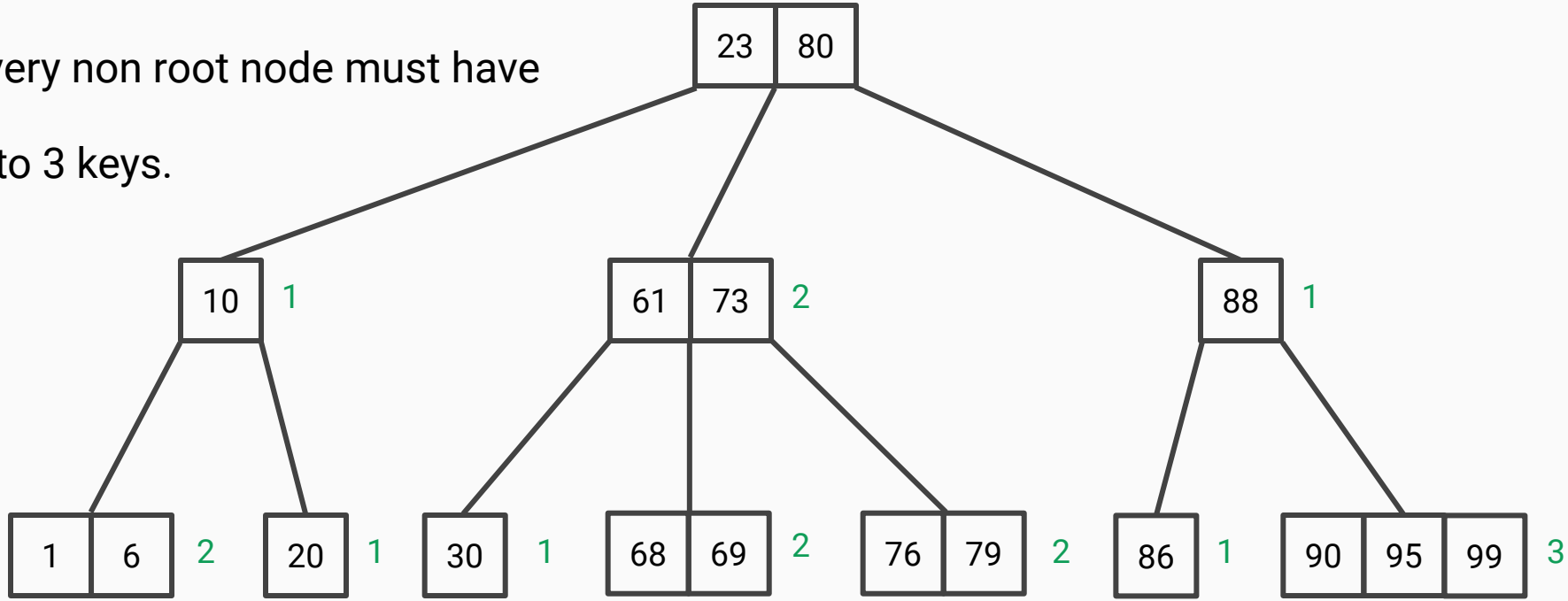
Properties of a B-Tree

Property 2

All non-root nodes have between $b - 1$ to $2b - 1$ keys. E.g. here is one where $b = 2$.

Every non root node must have

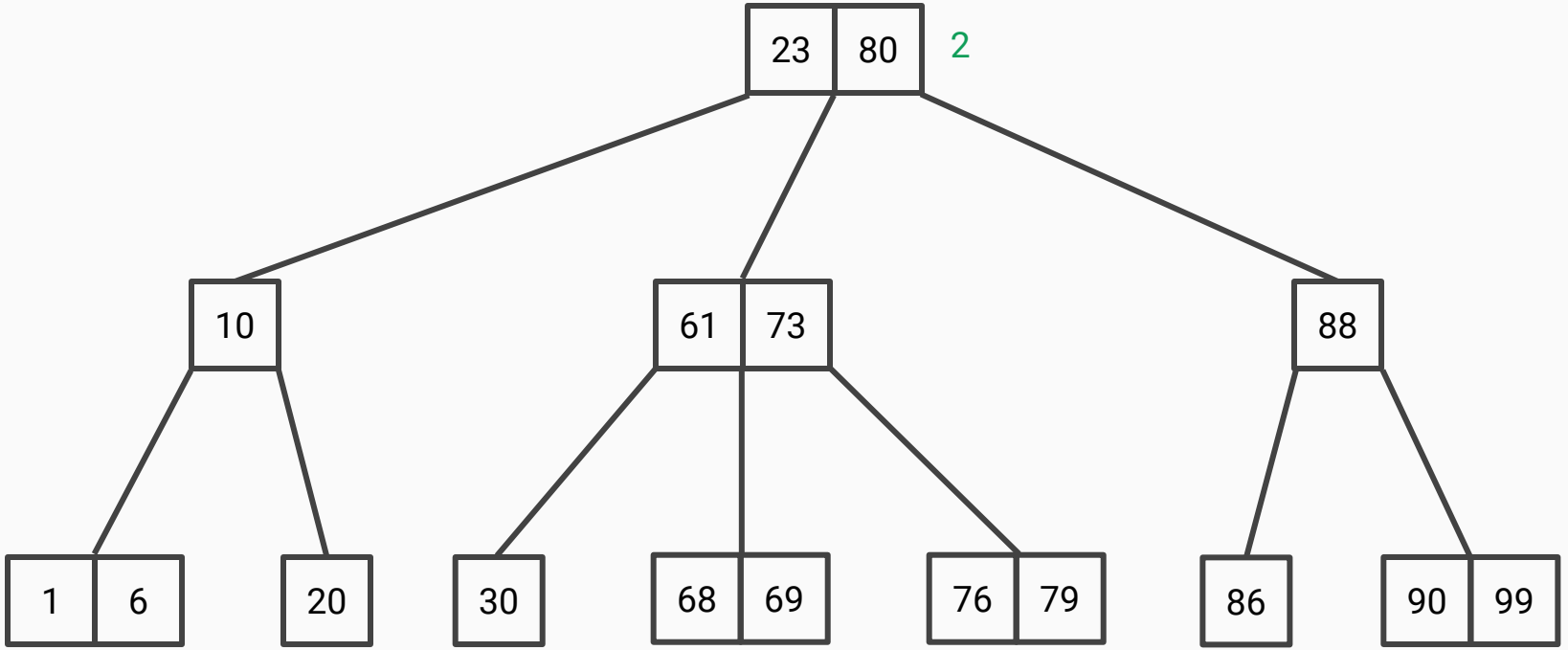
1 to 3 keys.



Properties of a B-Tree

Property 3

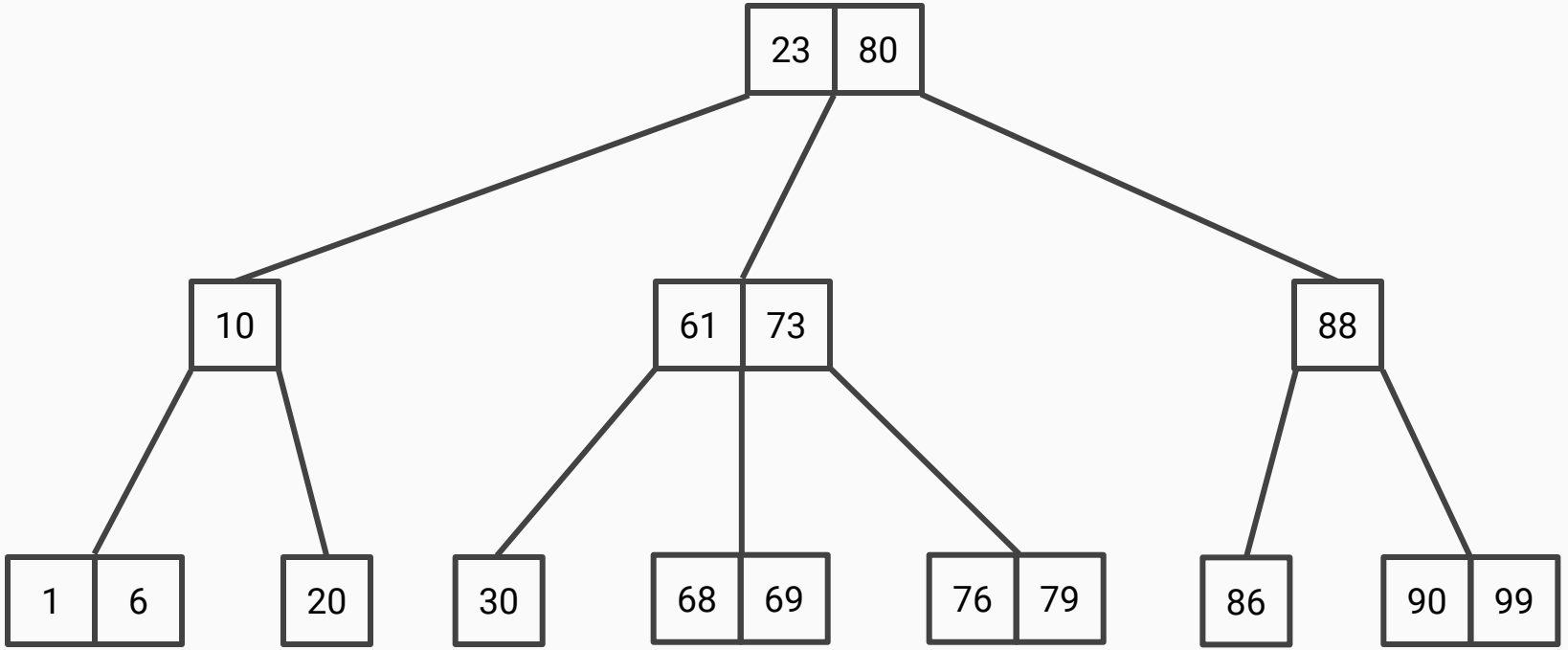
The root has at most $2b - 1$ keys. (E.g. $b = 2$ means the root has at most 3 keys)



Properties of a B-Tree

Property 4

An internal node with *k* keys must have *k* + 1 subtrees.



Properties of a B-Tree: Summary

Properties of a B-Tree

1. All leaf nodes are of the same depth.
2. All non-root nodes have between $b - 1$ to $2b - 1$ keys.
3. The root has at most $2b - 1$ keys.
4. An internal node with k keys must have $k + 1$ subtrees.

Okay, but how does this help us?

Analysis of a B-Tree

Analysis

Here we want to firstly show that the height of our B-tree of n keys, is always still of $O(\log n)$ height.

Gather into your groups!

DG Question 1

Properties of a B-Tree

1. All leaf nodes are of the same depth.
2. All non-root nodes have between $b - 1$ to $2b - 1$ keys.
3. The root has at most $2b - 1$ keys.
4. An internal node with k keys must have $k + 1$ subtrees.

Show that the height of B-tree is still O(log n)!

Solutions to Q1

Proof: Every node besides the root needs to have at least $b - 1$ keys. Each node must also have at least b children as subtrees.

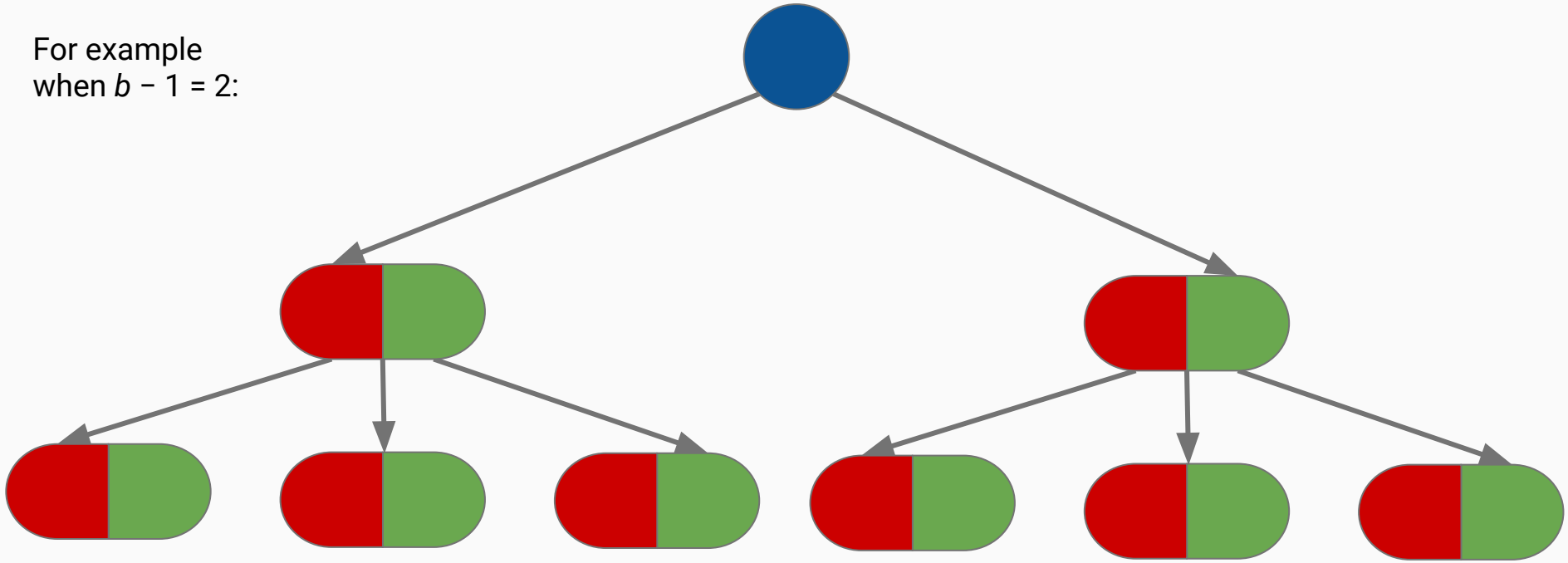
Solutions to Q1

Proof: Every node besides the root needs to have at least $b - 1$ keys. Each node must also have at least b children as subtrees.

This means that we end up getting a tree that “at the very least” looks something like this:

The smallest possible tree of height h (example: b - 1 = 2)

For example
when $b - 1 = 2$:



Every time we go down a level, the number of nodes in that level triples.

In general, if every internal node has at least k children, then every time we go down a level, the number of nodes in that level is multiplied by k.

Thus in general:

Since our B-trees have at least $b - 1$ keys:

1. On the first level, we should have at least 1 key.
2. On the second level, we should have at least $2 \times (b - 1)$ keys, because in the worst case, we only have one key in the root node.
3. On the third level, we should have at least $2 \times b \times (b - 1)$ keys, since each node has b nodes as children.
4. So, in general, on the ith level, we should have at least $2 \times b^{i-2} \times (b - 1)$ keys.

   So if you had a tree of height h, how many keys do you have at least?

Some maths

For a B-tree of height h, it must have at least:

$1 + 2(b - 1) + 2 \times b \times (b - 1) + 2 \times b^2 \times (b - 1) + \ldots + 2 \times b^{h-1}(b - 1)$

$= 1 + 2(b - 1)(1 + b + b^2 + \ldots + b^{h-1})$

$= 1 + 2(b - 1)\,[b^h - 1\,/(b - 1)]$ (geometric progression)

$= 2b^h - 1$ keys

Which means that if a tree has n keys, the height is at most roughly $\log_b n$.

Operations on a B-Tree

Operations on a B-Tree

Again we will want to at the very least support this list of operations:

- Insert
- Delete
- Lookup

In O(log n) time.

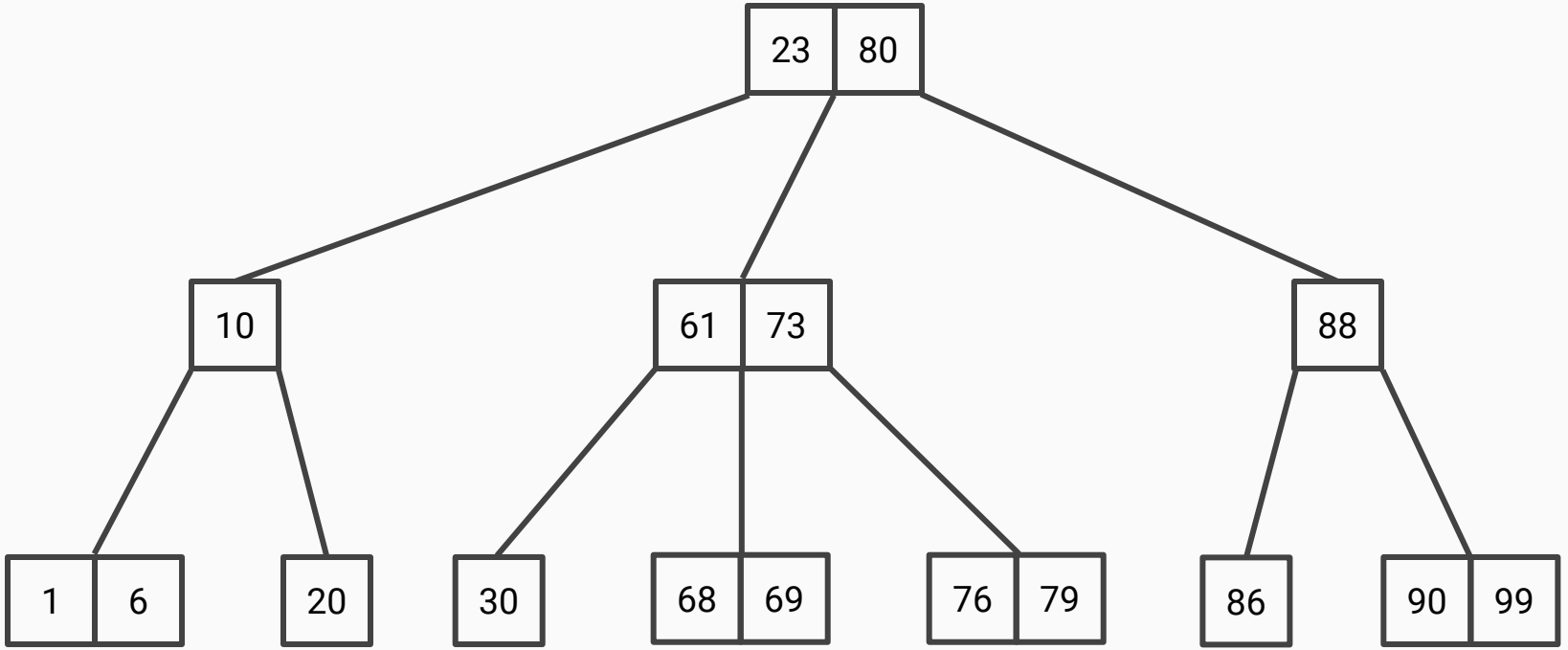
Insertion on a B-Tree

Rebalancing after Inserts

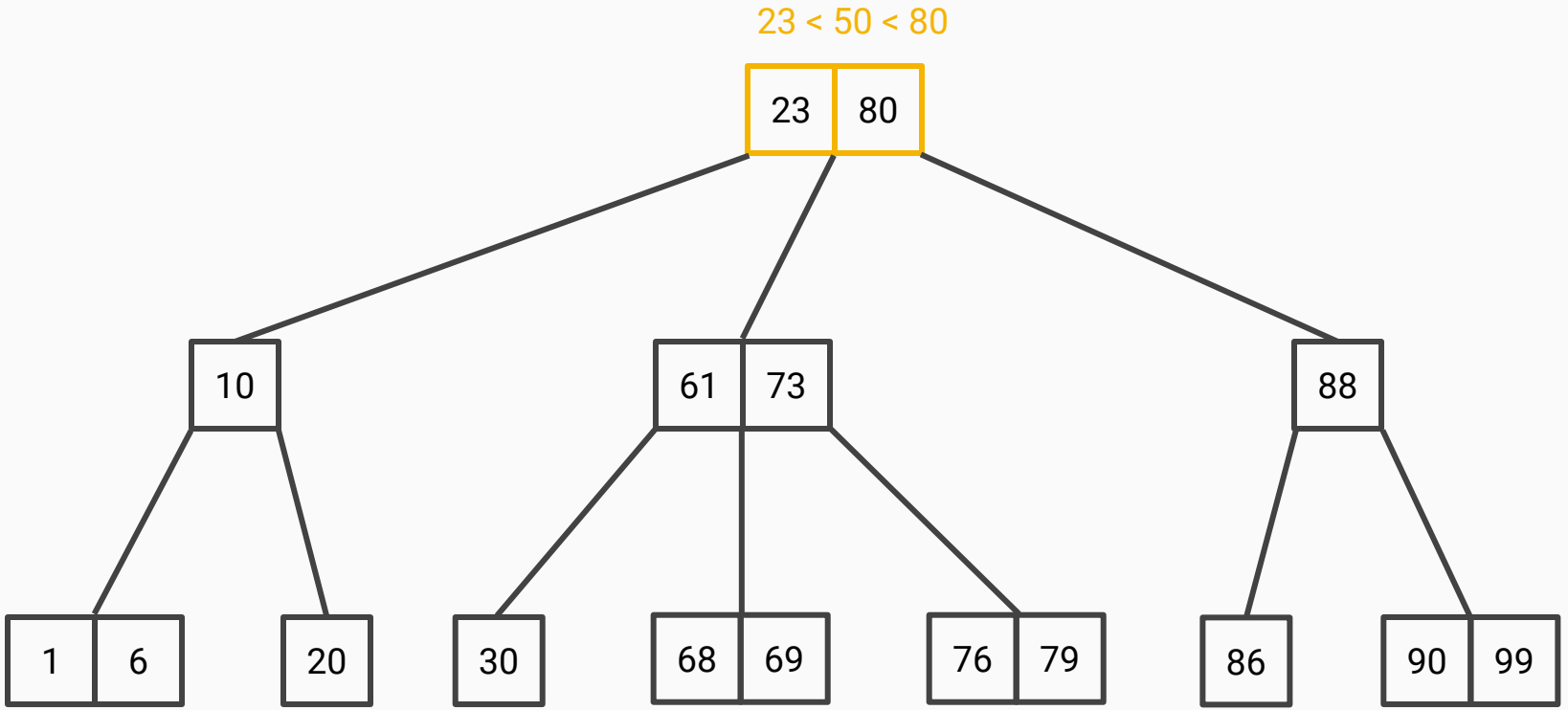
1. If the current node has less than 2*b* keys, we do nothing.
2. Else, we check our siblings, if either of them has less than 2*b* - *1* keys, we rotate a value to them.
3. Else, we do something called a “split”. Which involves breaking the current node into 3 parts: one node containing just the middle key, the other two nodes containing *b* keys each. (See example below) Then we push the middle key up into the parent.
4. We recurse this upwards if we split.

(Note: There is a version where you share the load with your siblings, but we will focus only on the splitting variant here.)

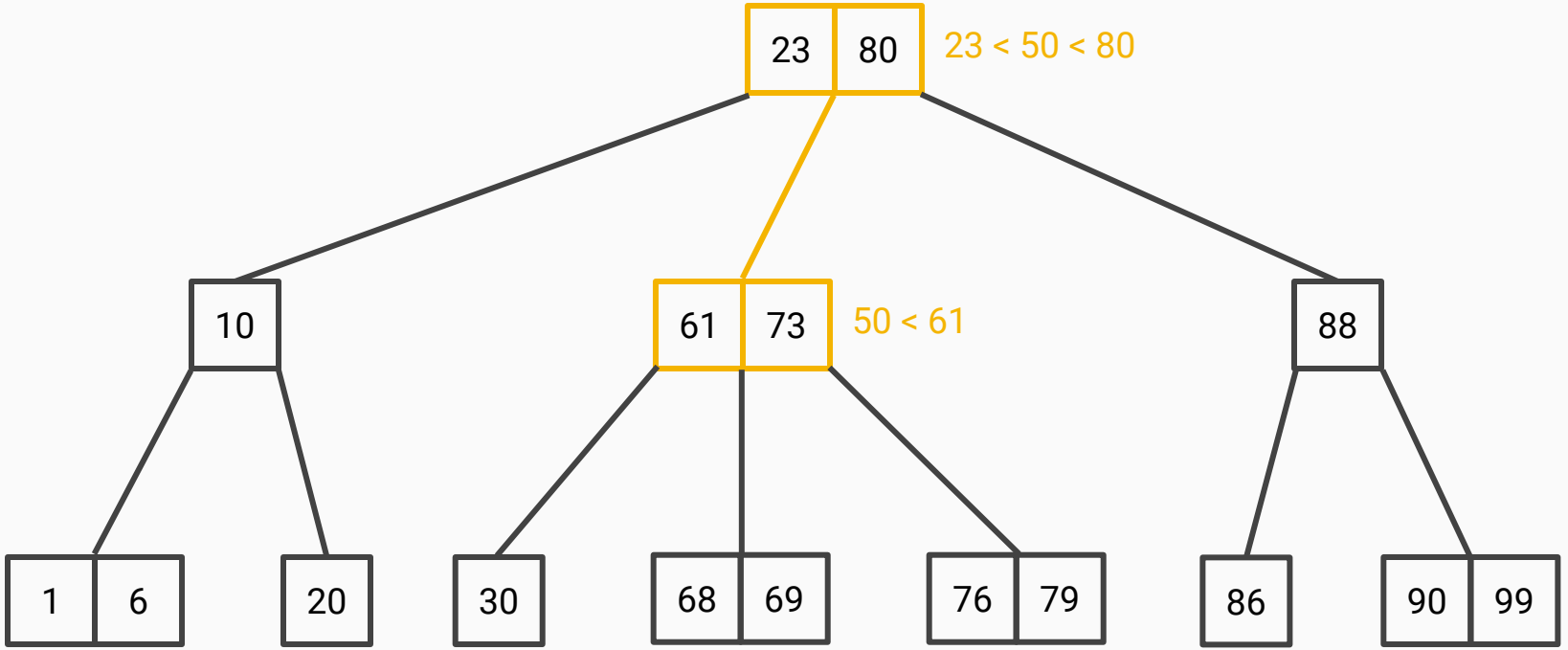
Let’s insert 50!



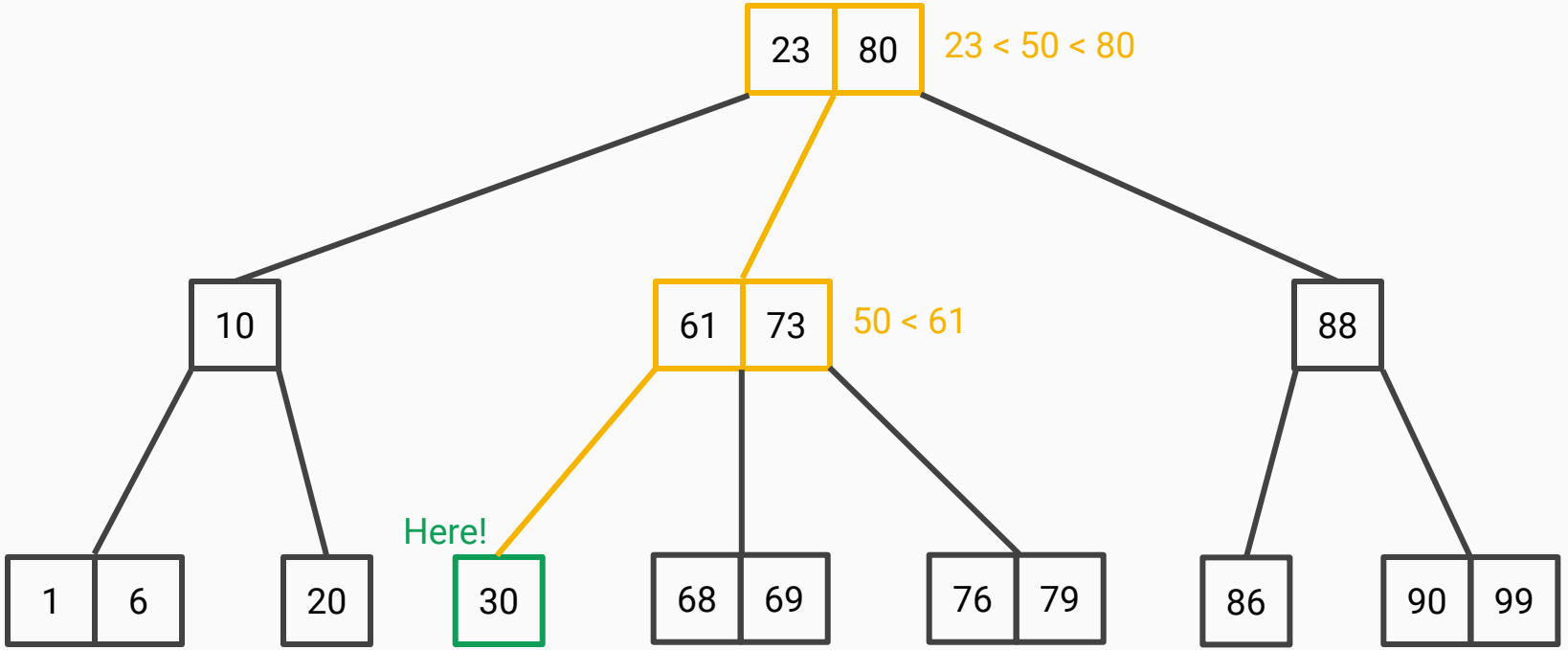
Let’s insert 50!



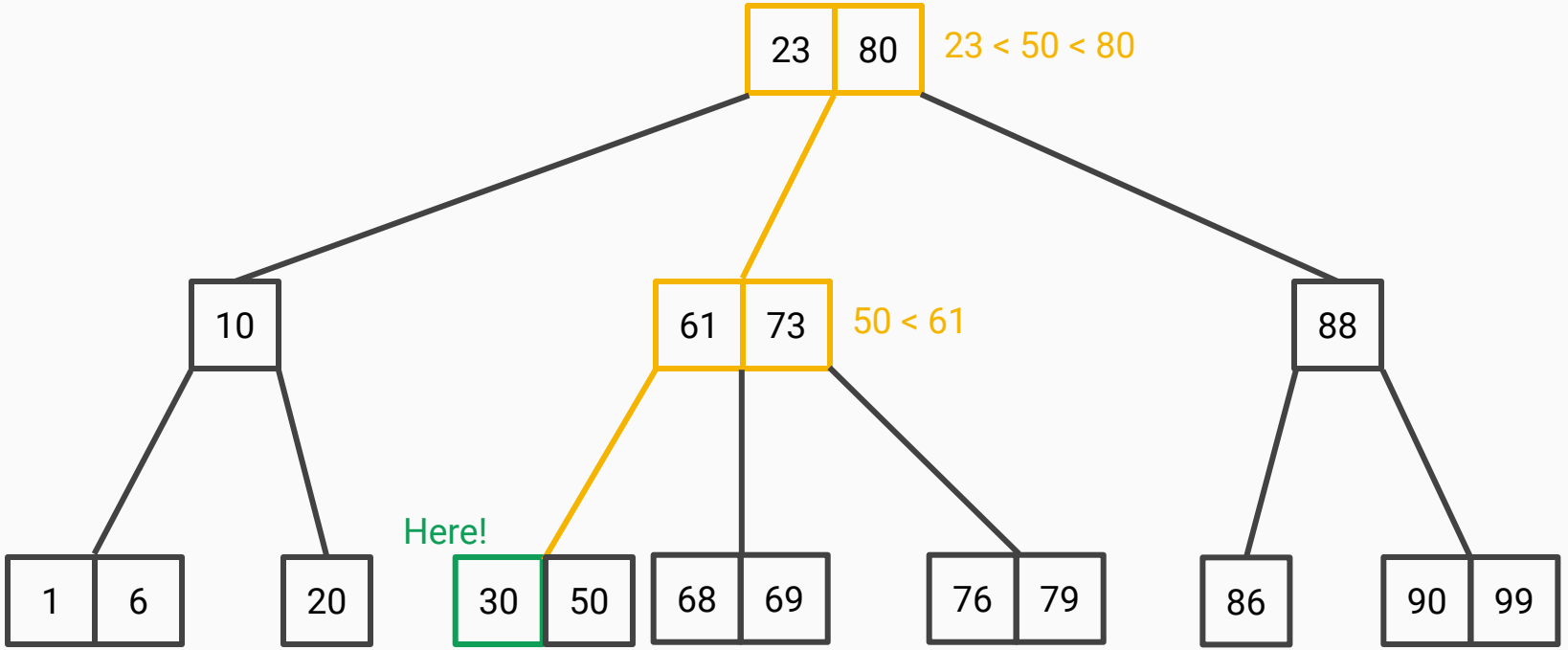
Let’s insert 50!



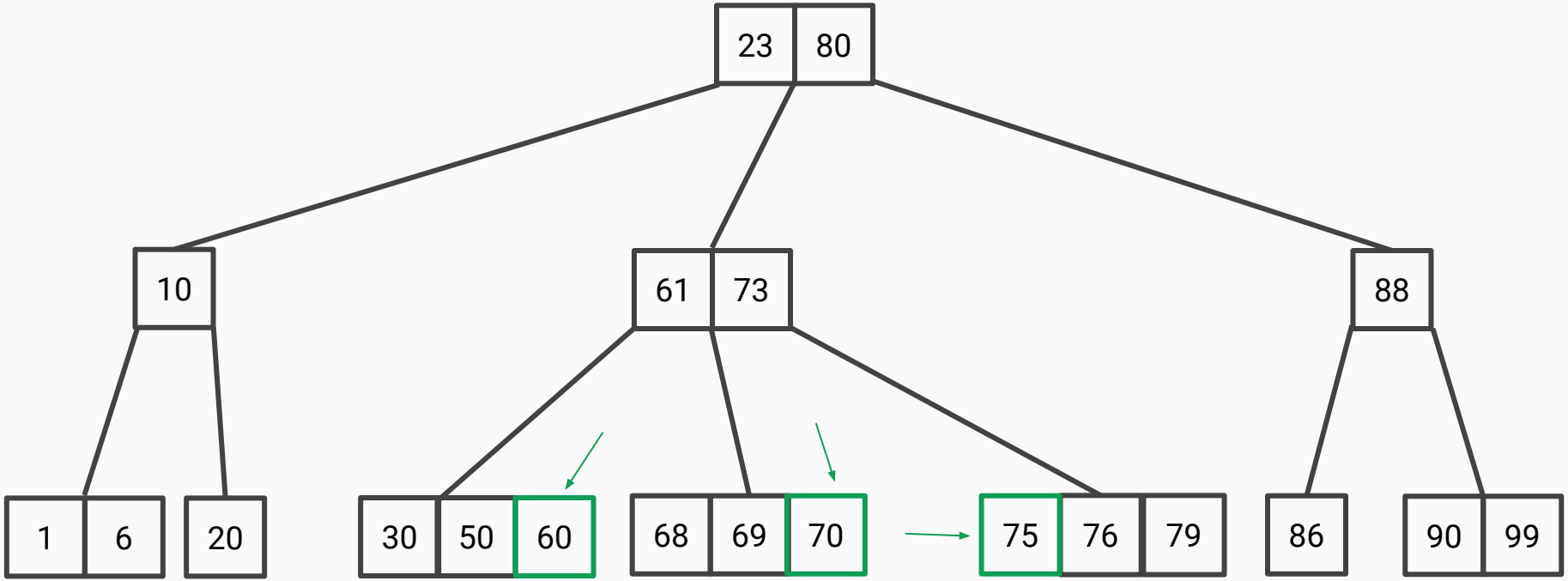
Let’s insert 50!



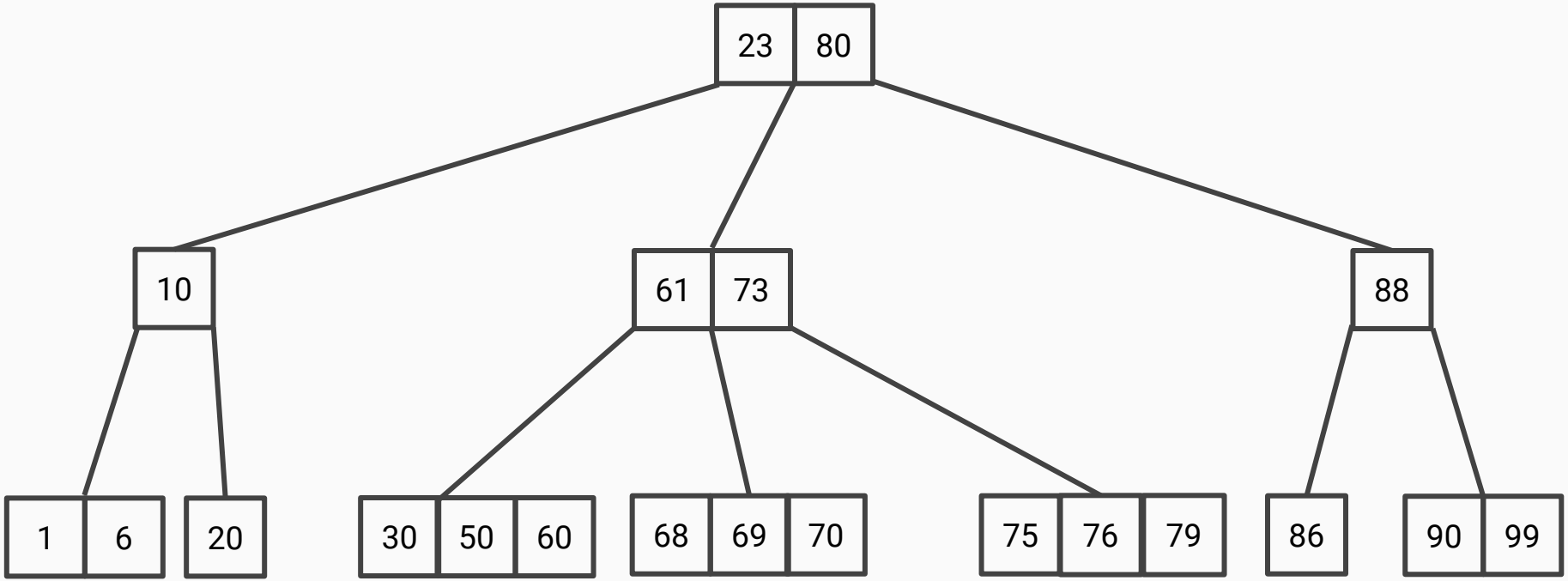
Let’s insert 50! Okay since our node still has less than 2*b* keys, we can stop here.



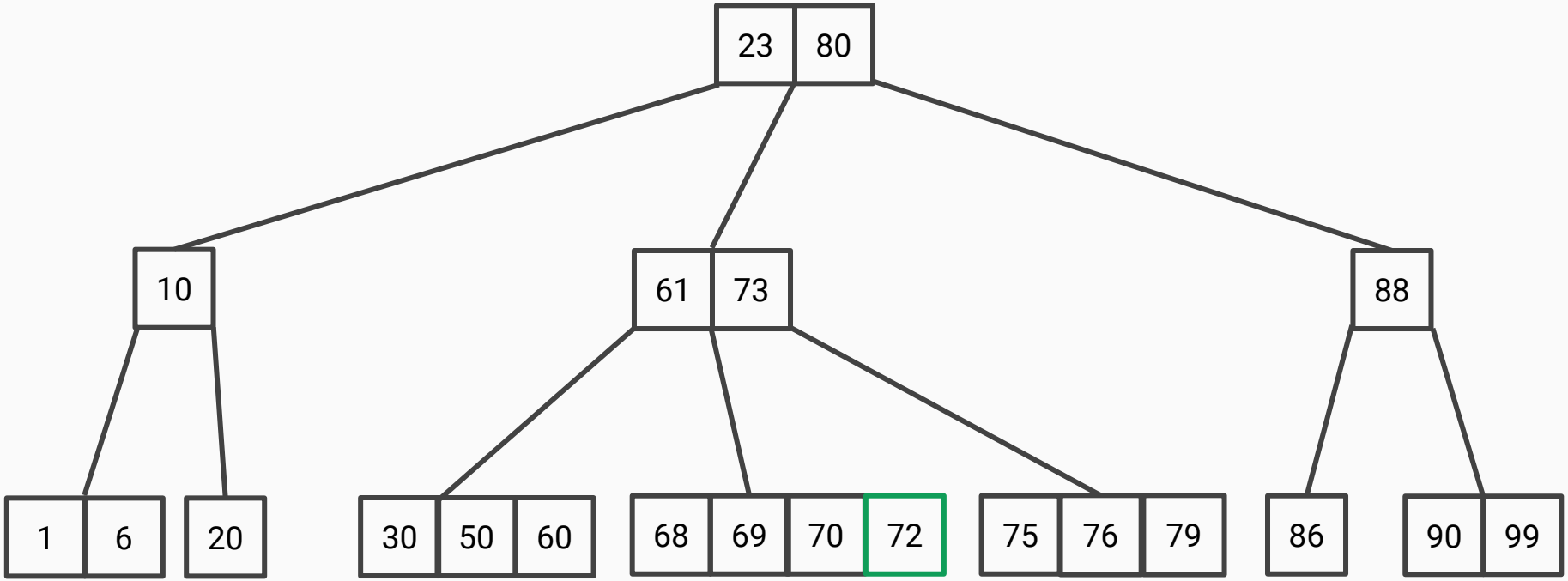
Say now we also inserted 60, 70, and 75. Here is where they end up.



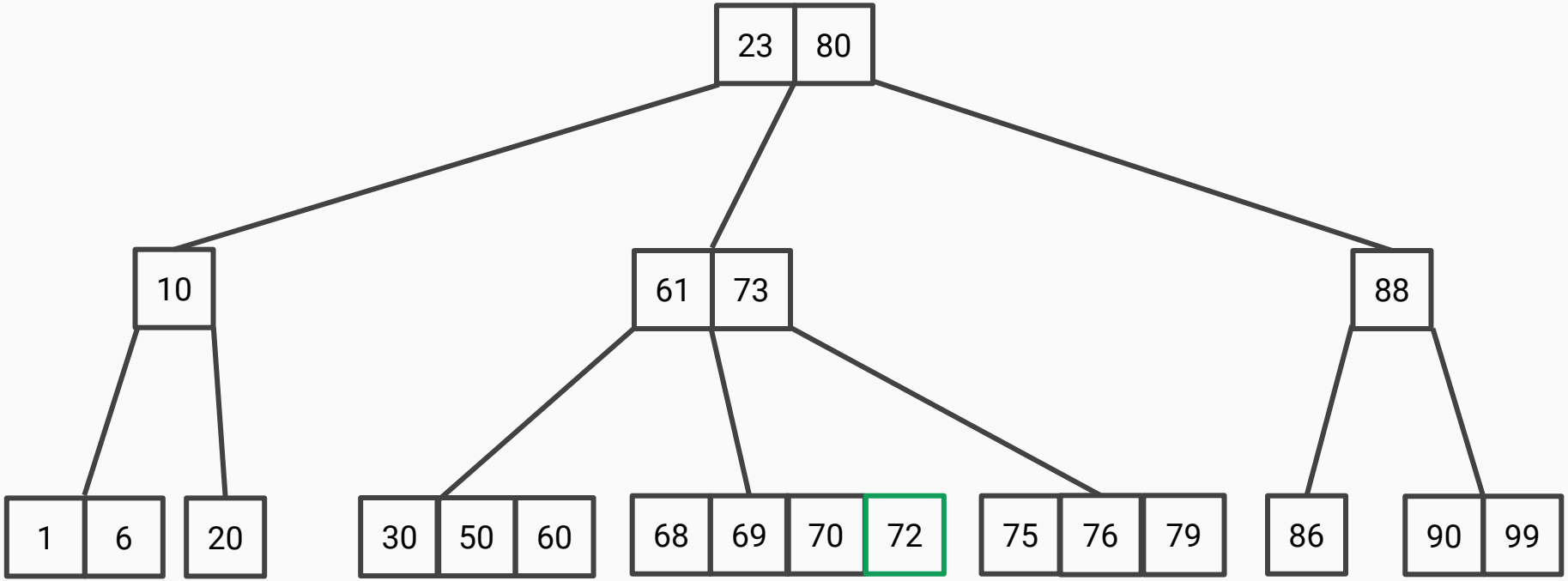
Now let’s try inserting 72!



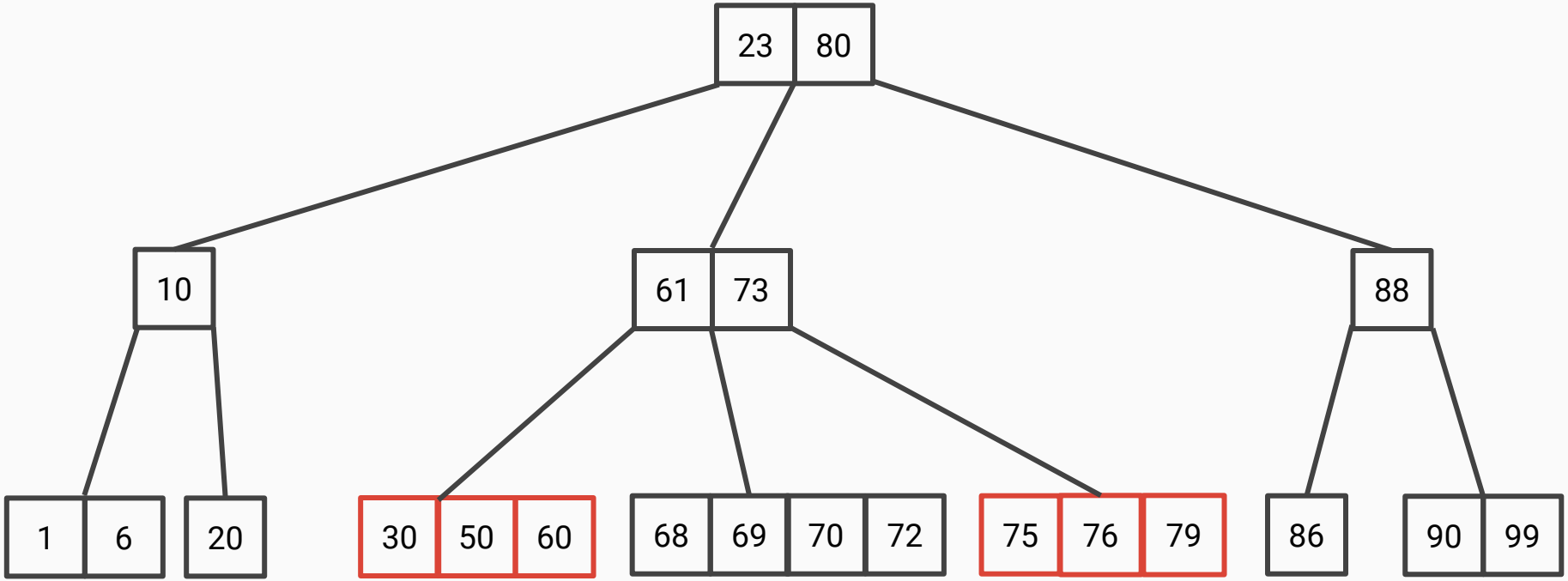
Okay so here's where it lands.



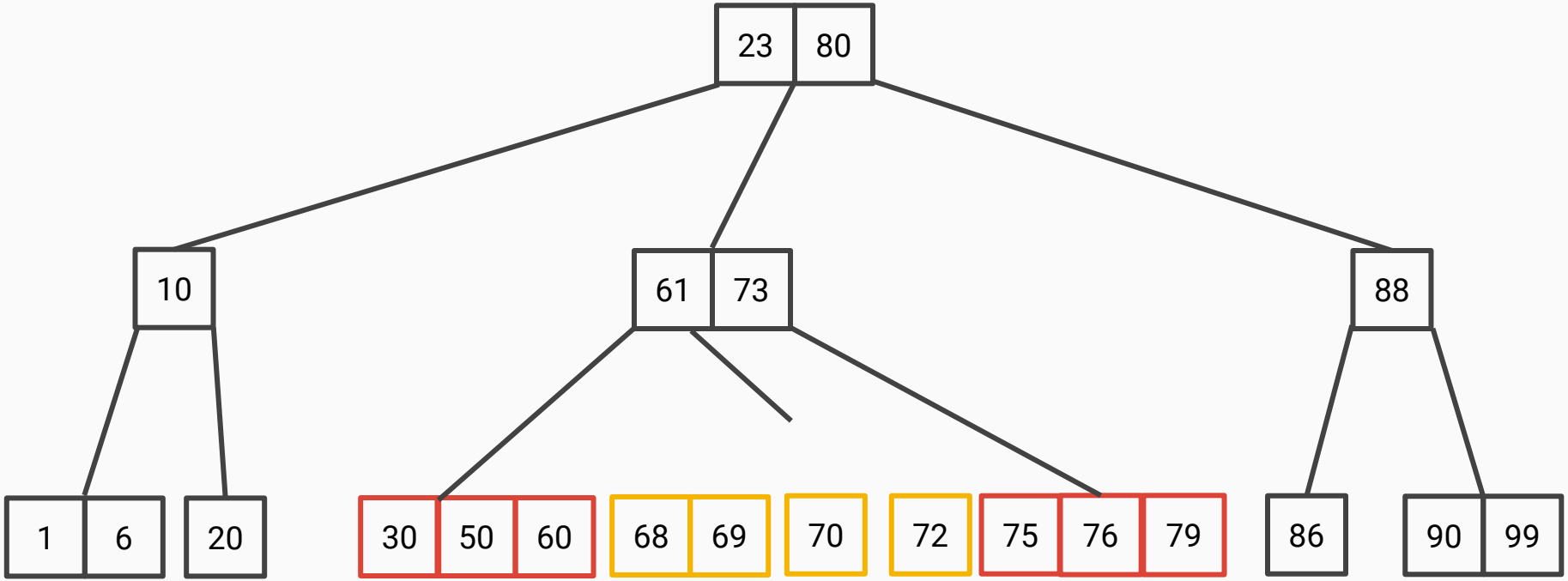
But the current node is now over full (it has 2b keys) ! We need to look at the siblings.



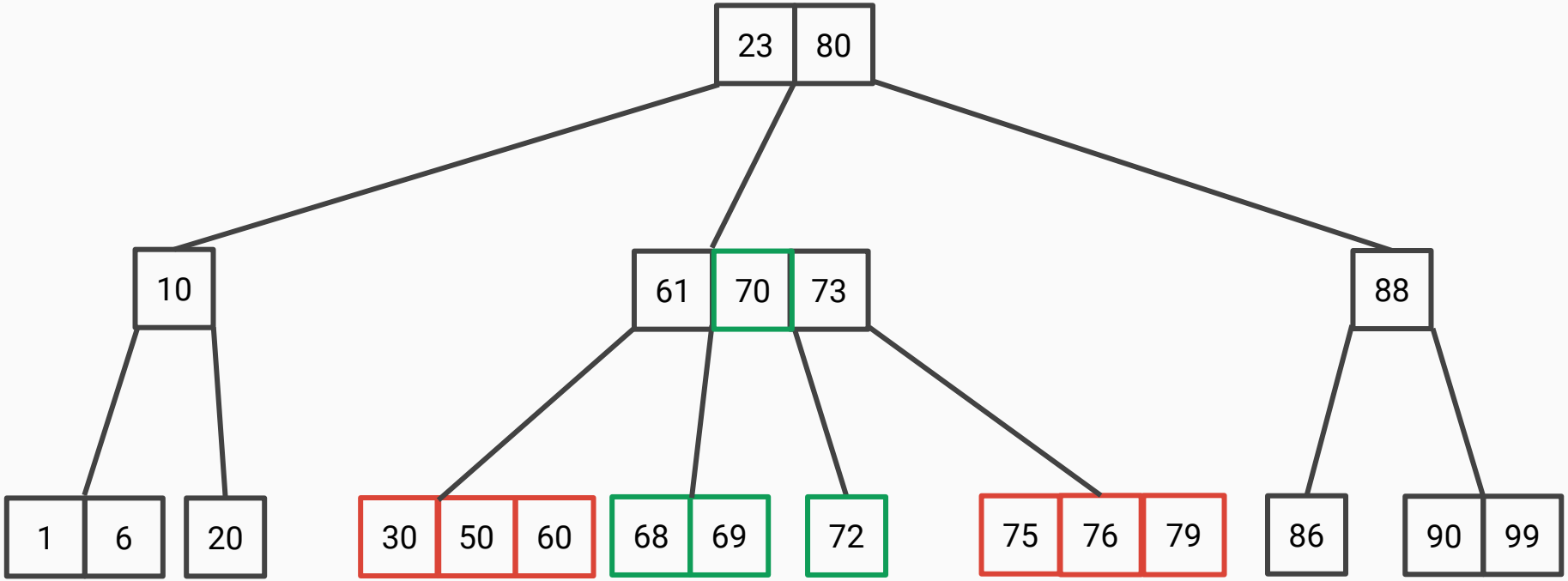
Oh no! They're full too!



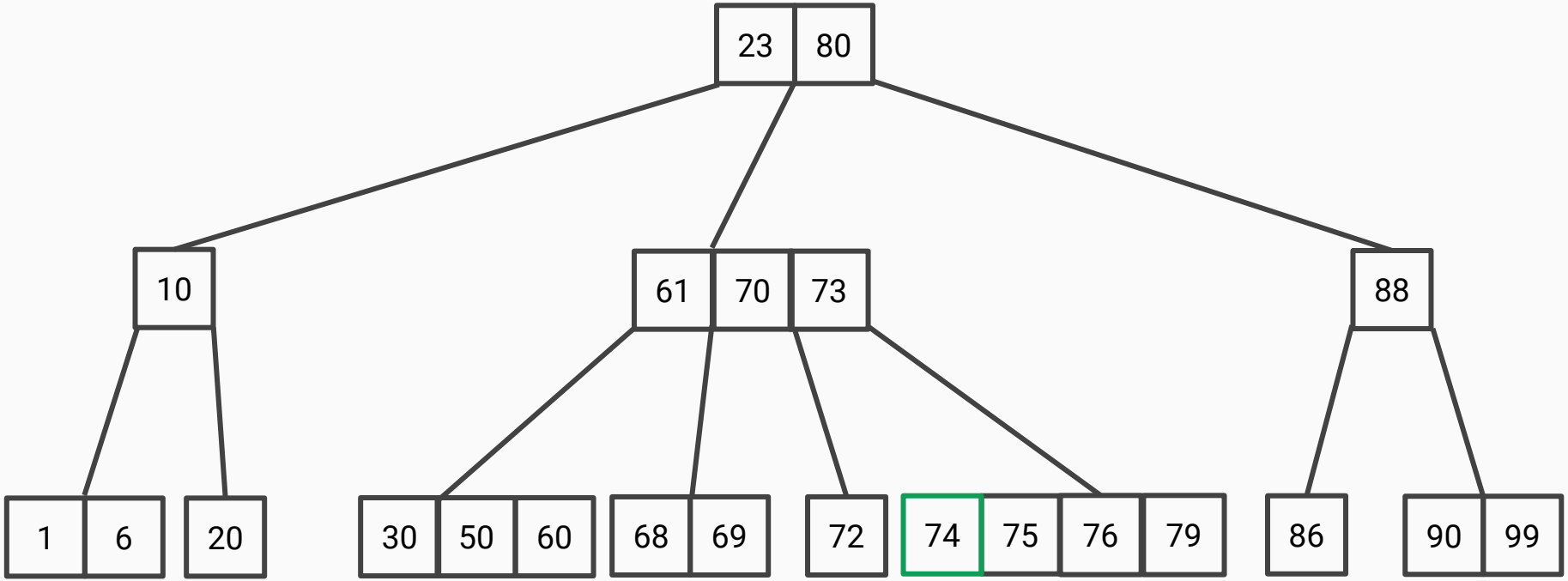
So now what we have to do a split, and push up one of the elements.



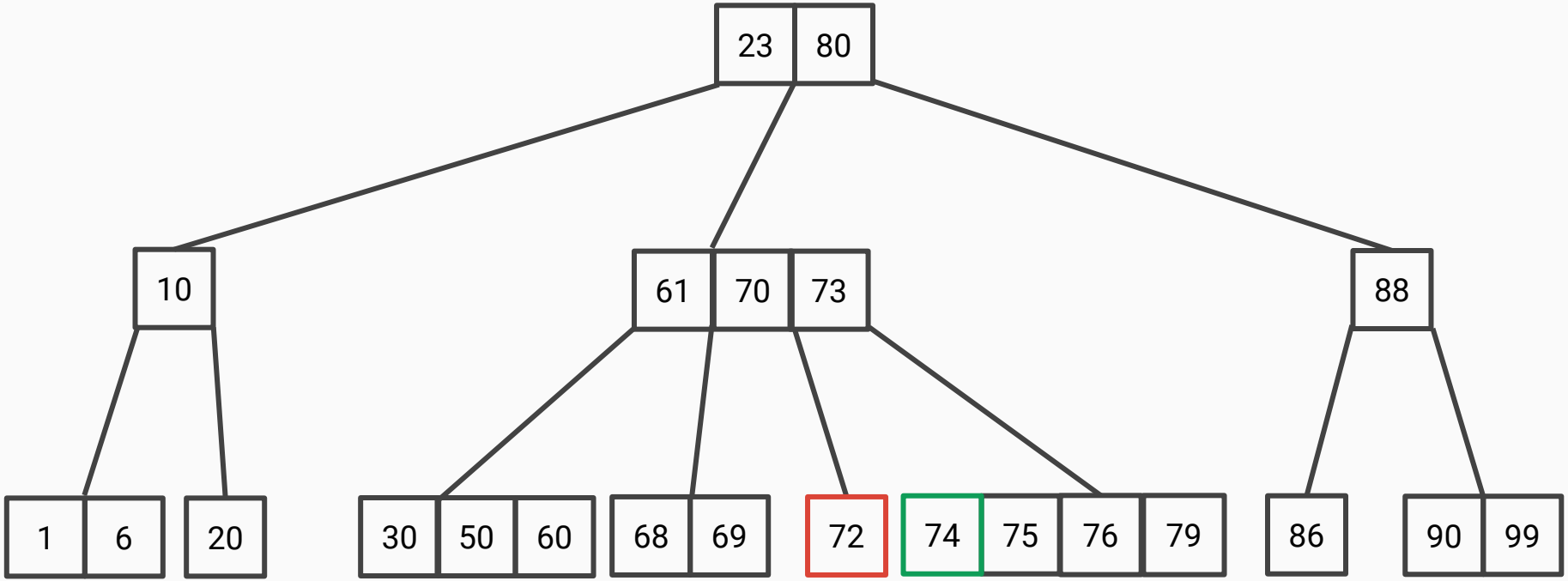
Notice that the parent node has one more key, and has one more child than before. So the key vs child numbers still work out!



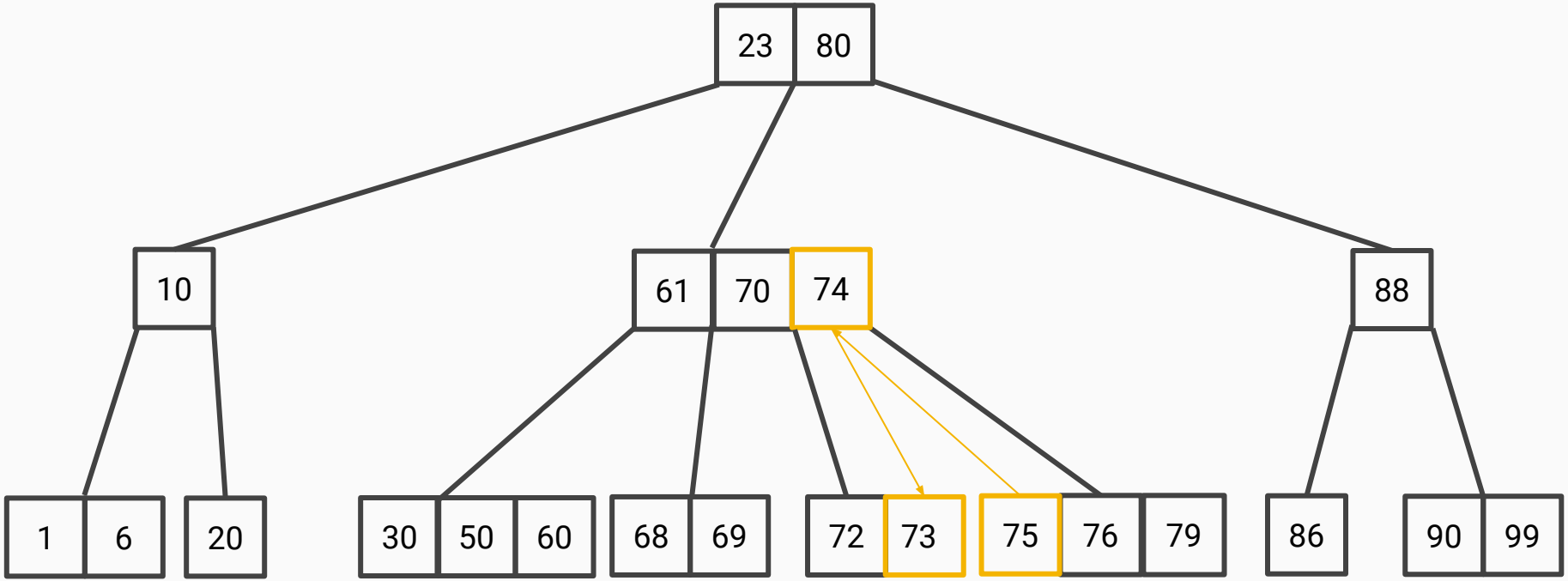
Now let’s say we added 74.



Oh no! It’s over full again! Now we look at our siblings.



Oh good, there's space. Here we do a rotation:



DG Question 2

Properties of a B-Tree

1. All leaf nodes are of the same depth.
2. All non-root nodes have between $b - 1$ to $2b - 1$ keys.
3. The root has at most $2b - 1$ keys.
4. An internal node with k keys must have $k + 1$ subtrees.

Task: Show that after insertion these invariants are maintained!

Solutions to Q2

1. All leaf nodes are of the same depth.

There are 3 possible cases:

(a) No splitting/rotation was done.
(b) There was some splitting, then the keys only propagate upwards.
(c) There was a rotation.

Then in all 3 cases the depth still doesn't change!

Solutions to Q2

2. All non-root nodes have between $b - 1$ to $2b - 1$ keys.

There are 3 possible cases:

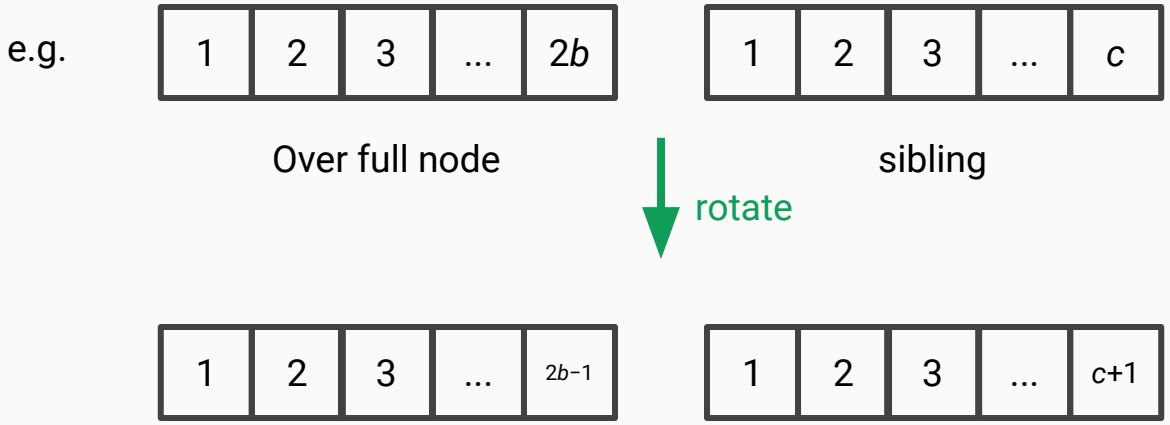
If you split, then it must be that your node had $2b$ keys. After the split, you get 3 new nodes, two of size (roughly) $b - 1$, and one of size 1. Then the size 1 key is pushed up into the nodes above. Since this in the worst case propagates to the root, all the nodes keep this property.

Solutions to Q2

2. All non-root nodes have between $b - 1$ to $2b - 1$ keys.

There are 3 possible cases:

If you rotate, then your sibling had less than $2b - 1$ keys, so now it has at most $2b - 1$ keys. And your current node has $2b - 1$ keys!



Solutions to Q2

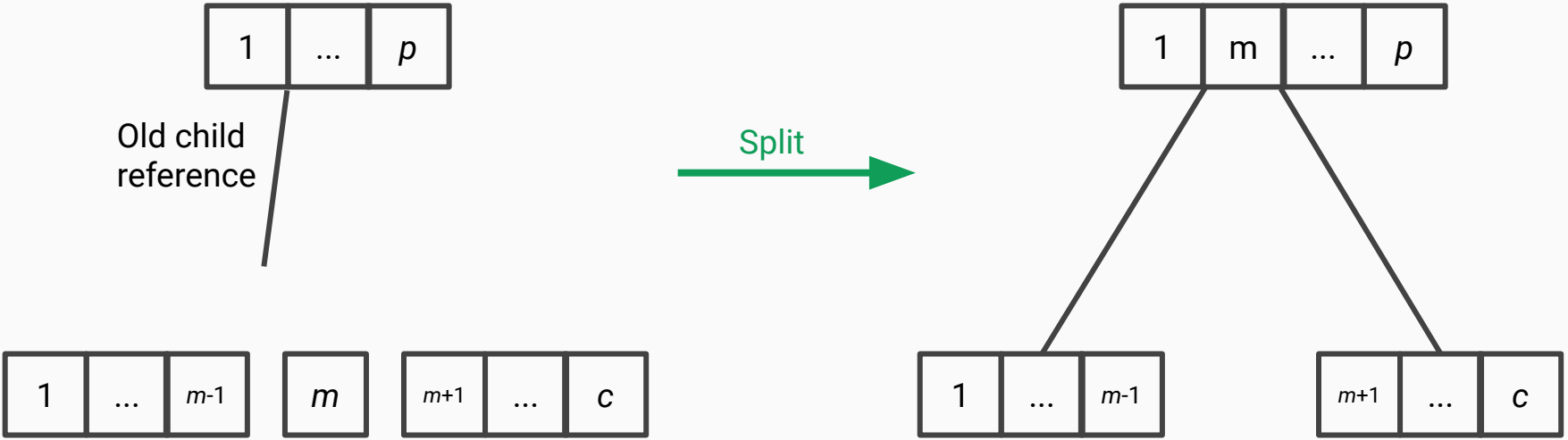
3. The root has at most $2b - 1$ keys.

If the root ever has $2b$ keys, it can just split!

Solutions to Q2

4. An internal node with k keys must have $k + 1$ subtrees.

When we split, we also “bring up” the subtree pointers. Therefore this property is always maintained. Notice that also the BST ordering property is maintained!



Deletion on a B Tree

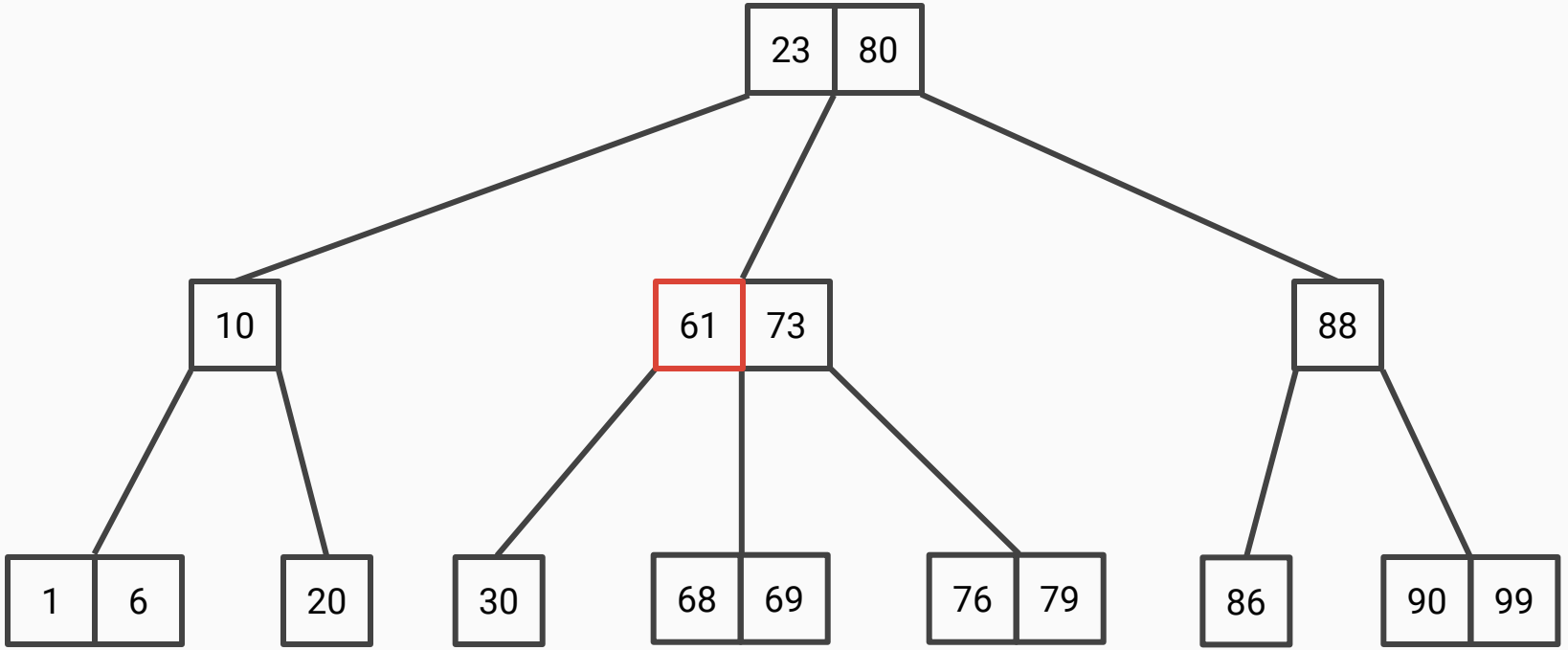
Deletion on a B-Tree

For deletion, if not a leaf, we first need to find the successor, and delete our successor recursively, then set the key that we had wished to be deleted to our successor.

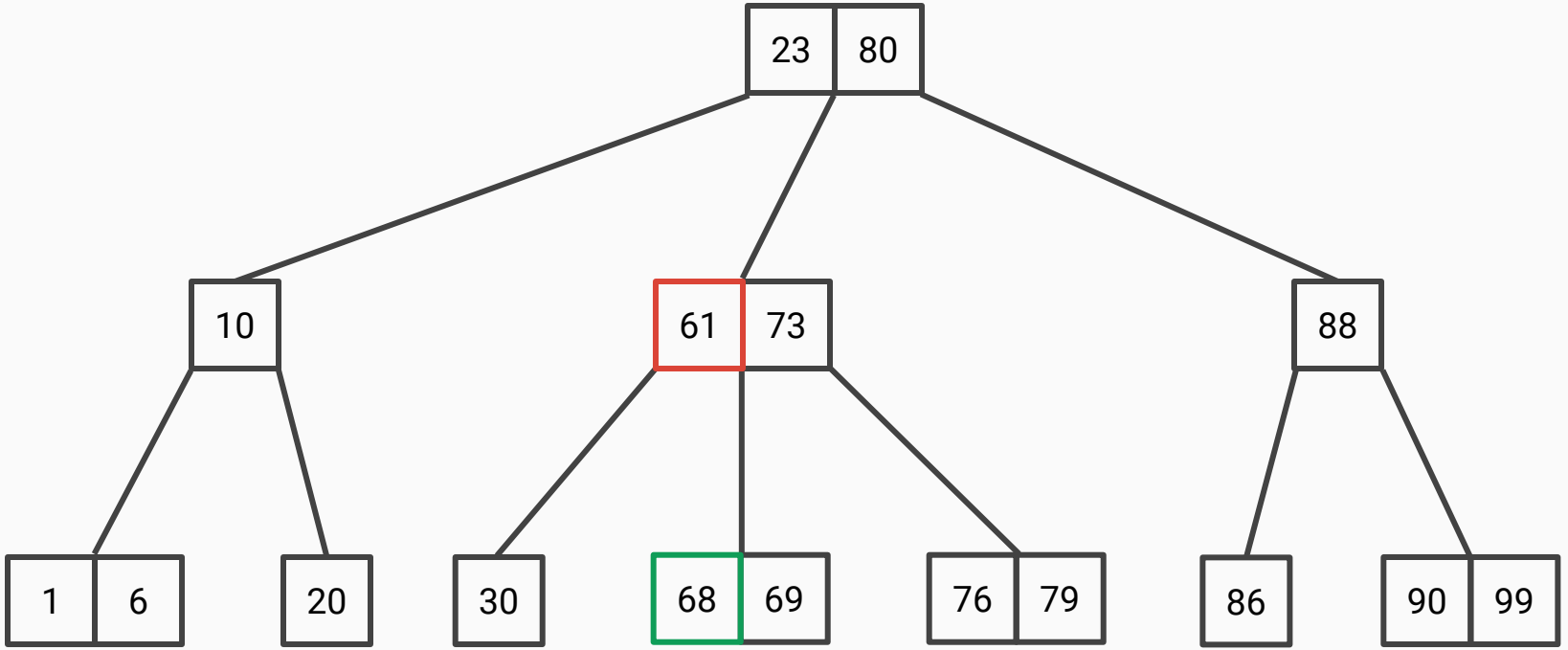
1. If our current node still has at least $b - 1$ keys, we do nothing.
2. Else we look at the siblings, if either of them has greater than $b - 1$ keys, we can take one from either.
3. Else we pick one of the siblings and merge with them using a key from the parent. Then recurse on the parent node to check its size etc.

(Notice the predecessors should always be in a leaf node. In CLRS a more advanced deletion algorithm is presented that hyper-optimises the deletion instead.)

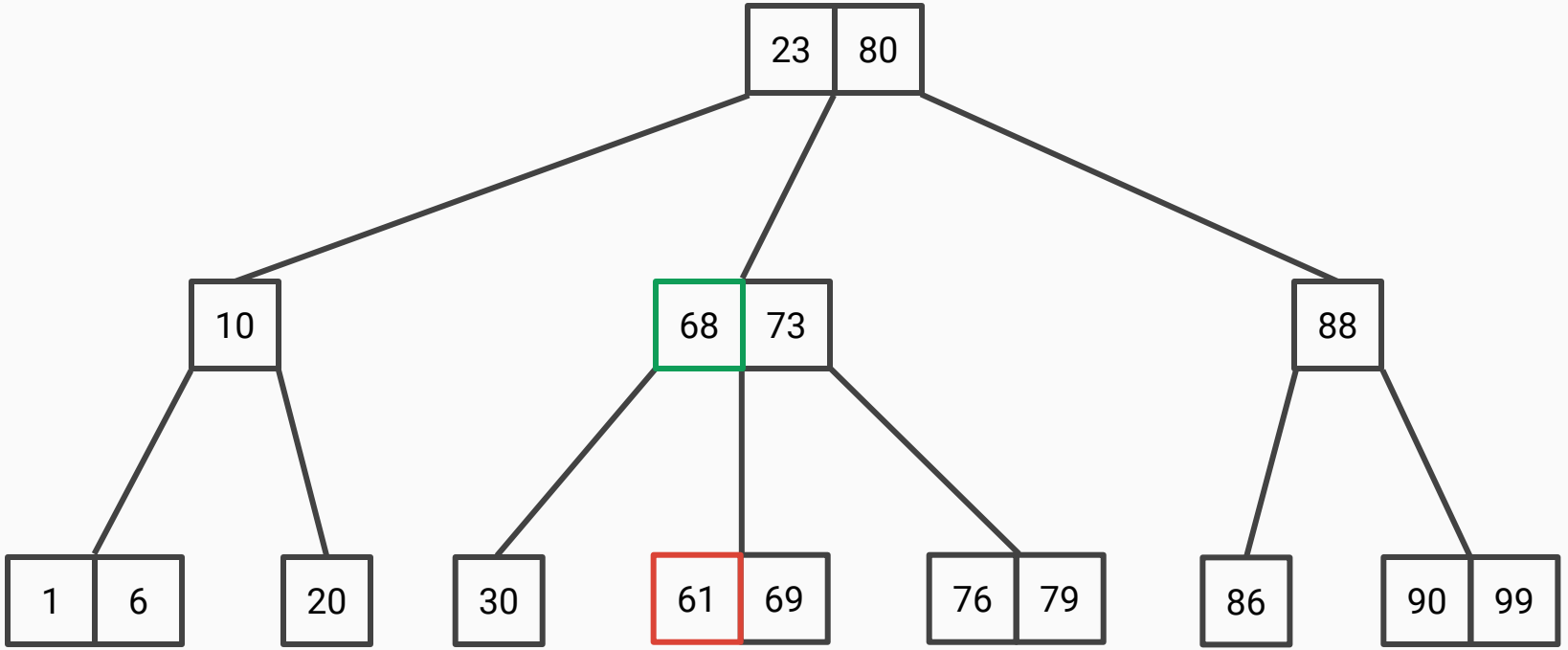
Let’s delete 61! Not a leaf node, so we need to find the successor.



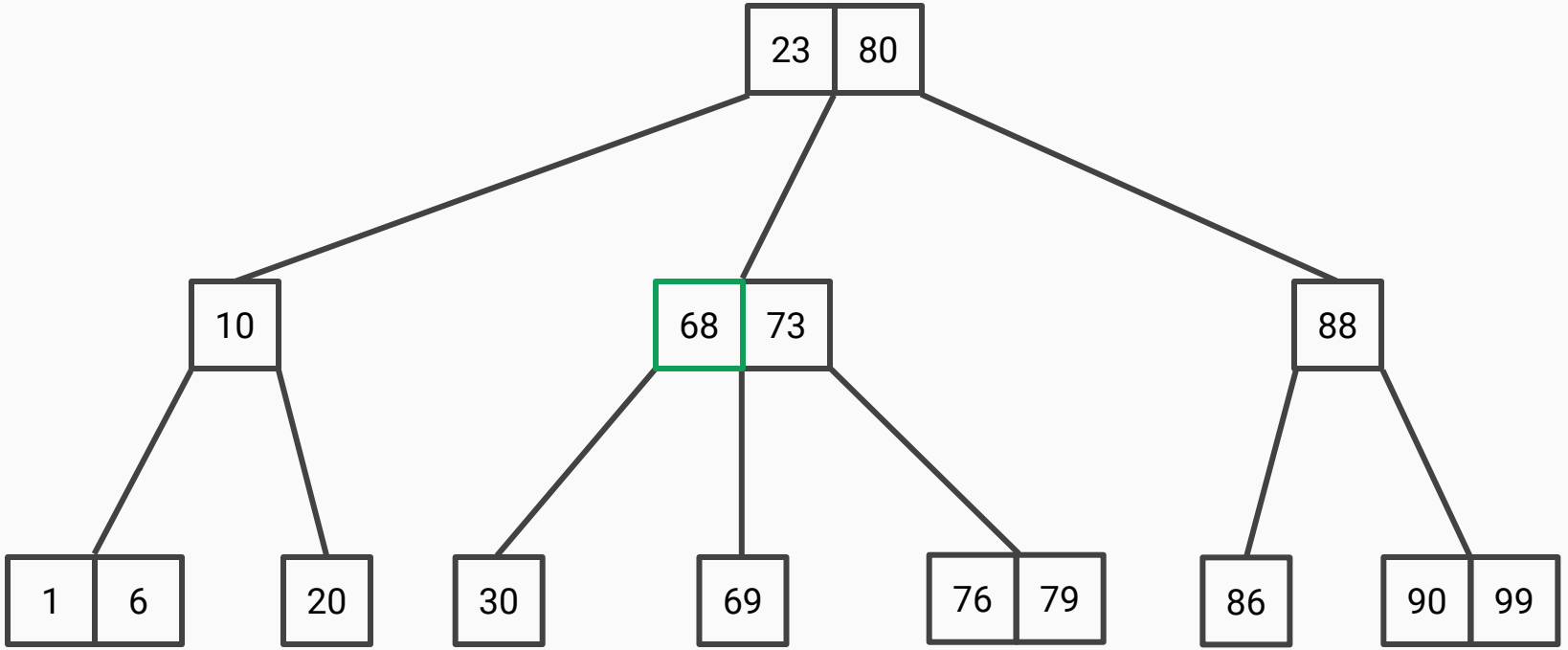
Found the successor!



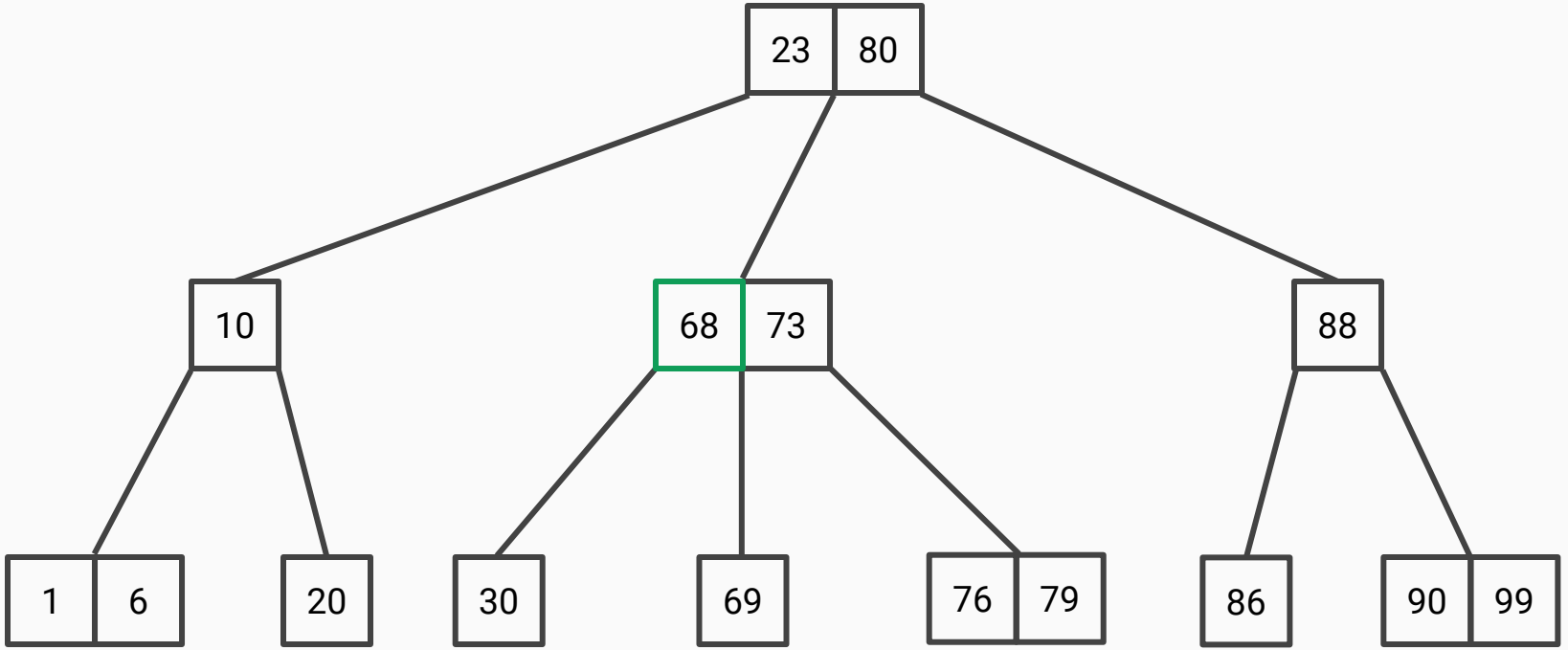
Do the swap, then remove the new key.



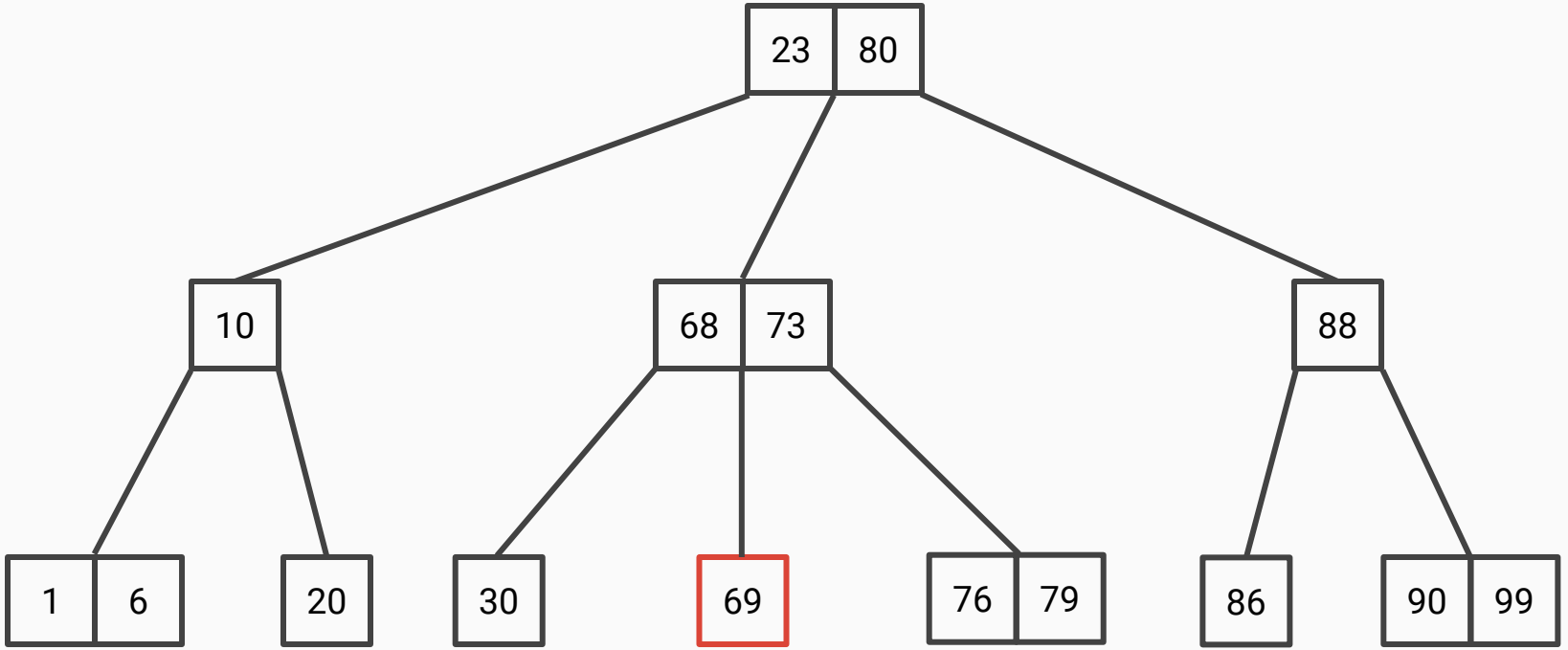
Do the swap, then remove the new key.



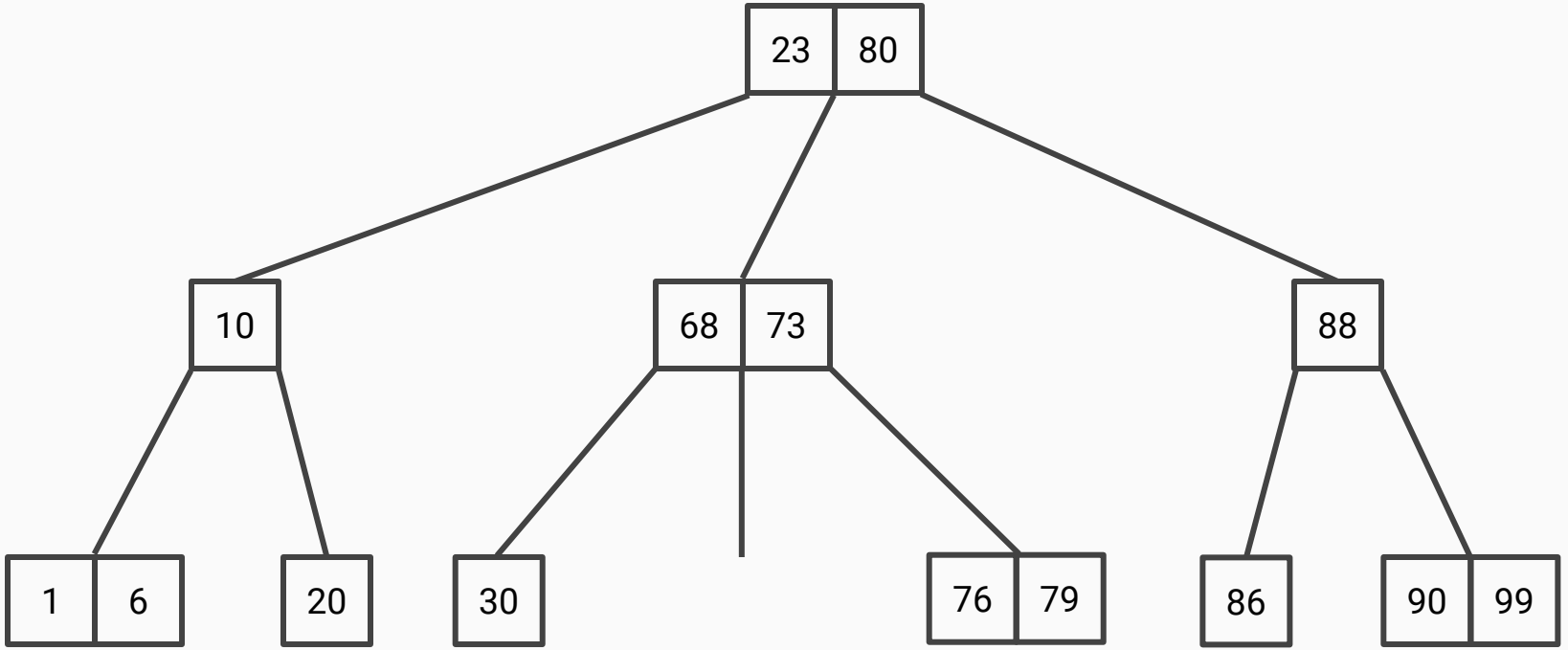
Since we still have at least $b - 1$ keys, we can stop here.



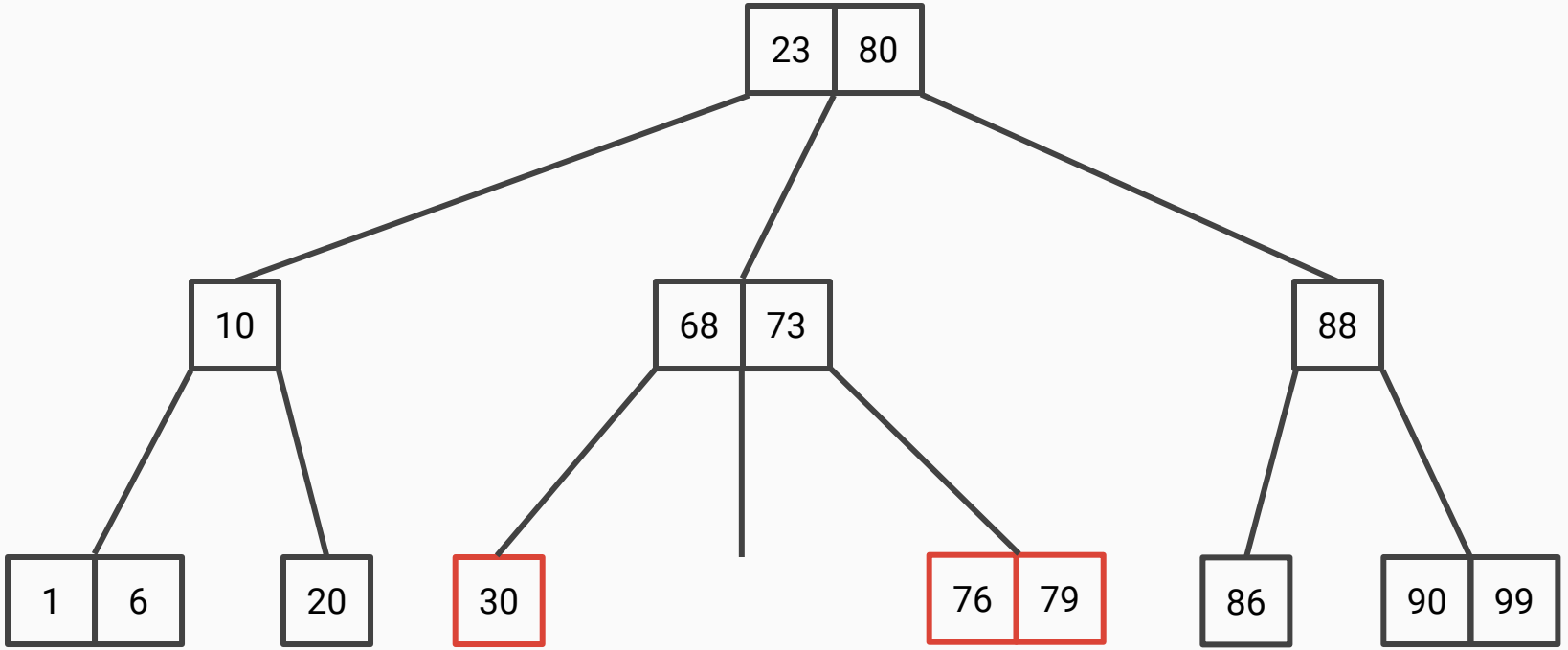
Okay, now lets delete 69!



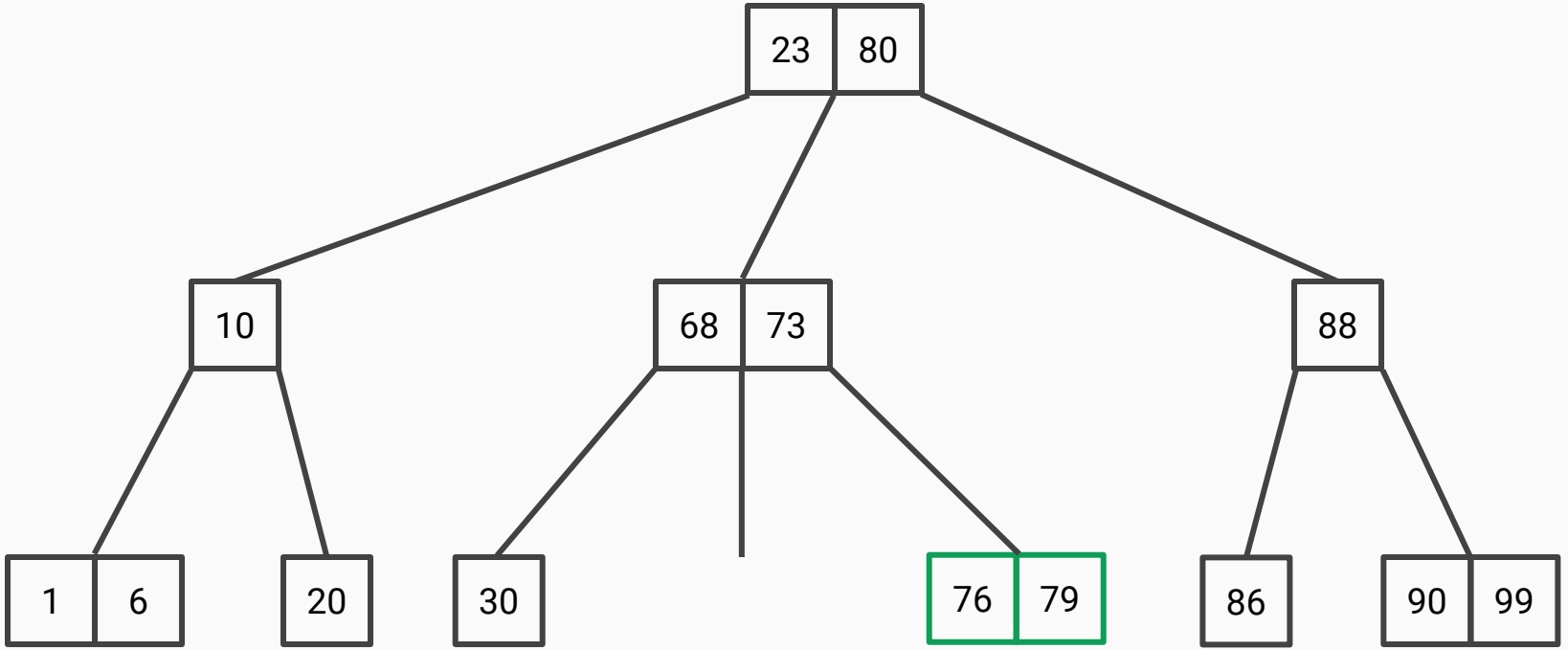
Since it is a leaf, we can just remove it.



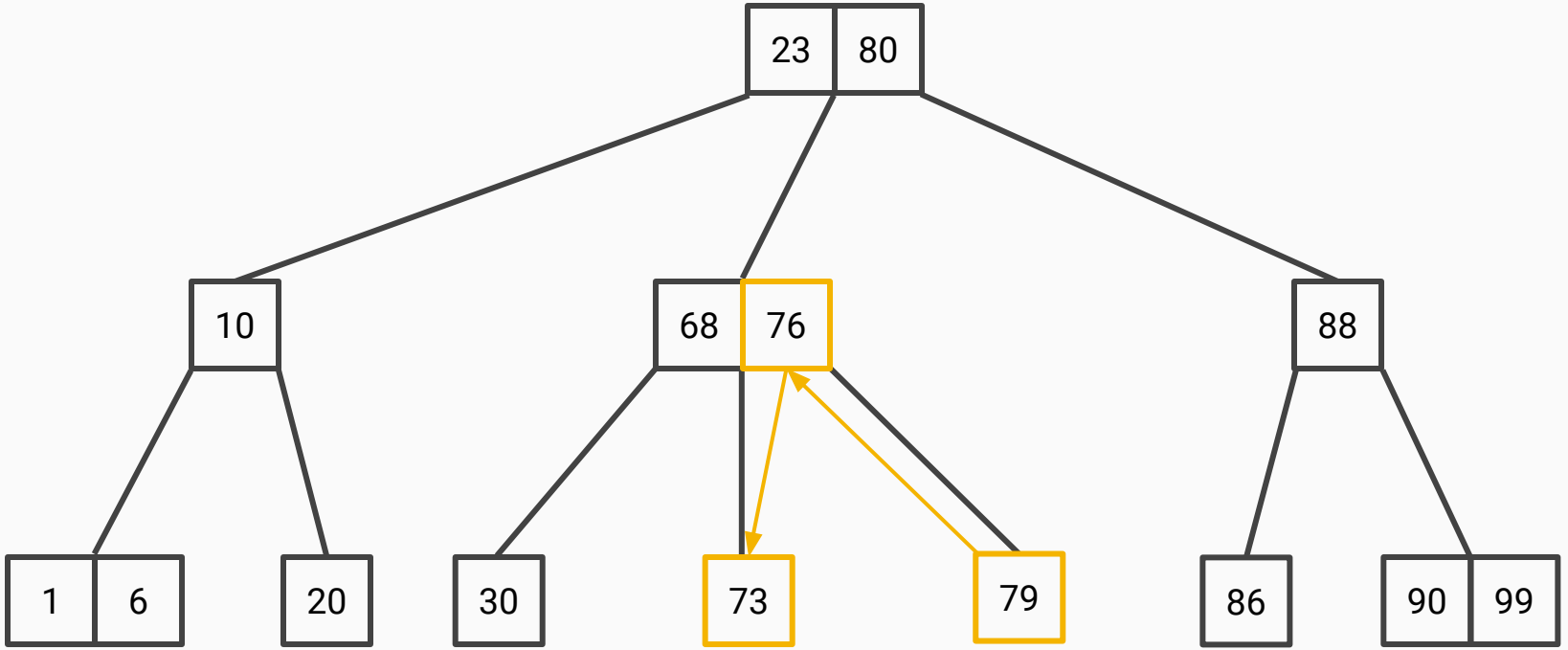
But now it has less than $b - 1$ keys! Again, we look at the siblings.



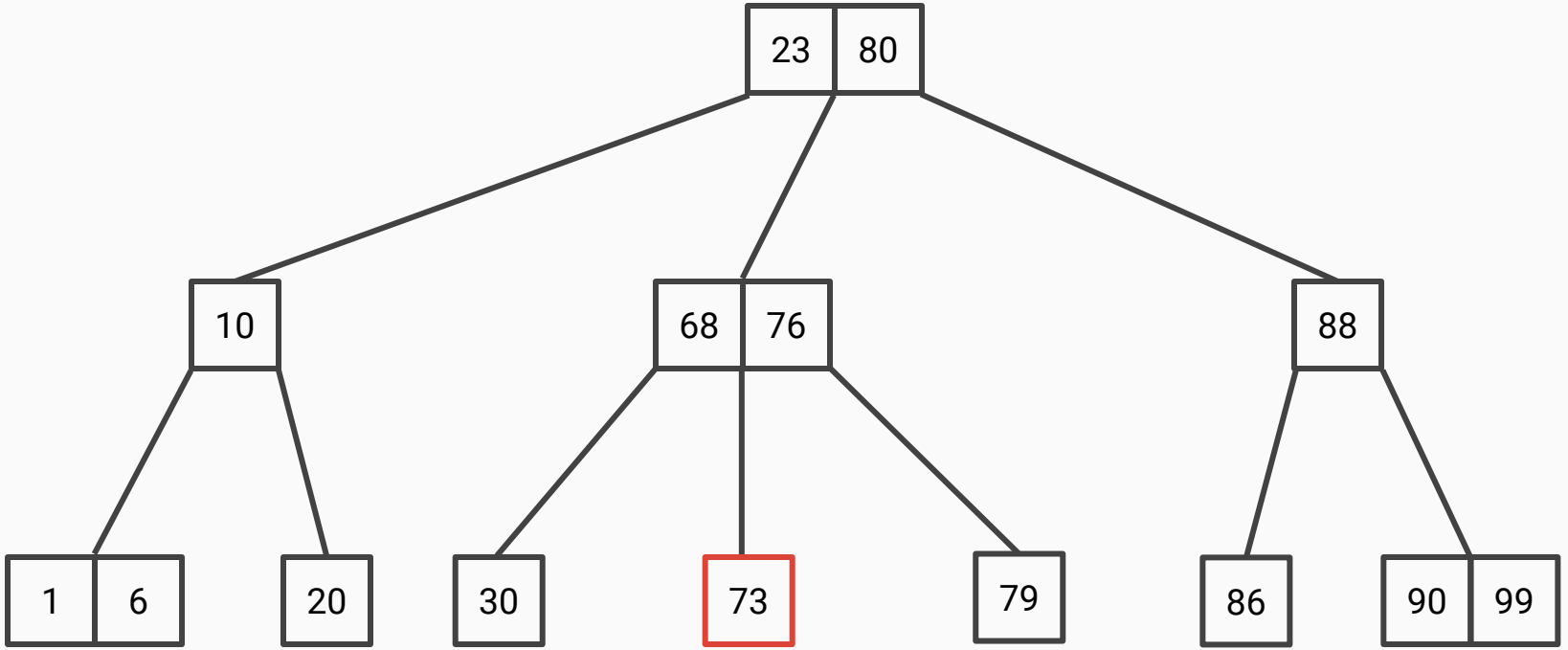
Aha! Our right sibling has one more key than the minimum. Let’s do a rotation.



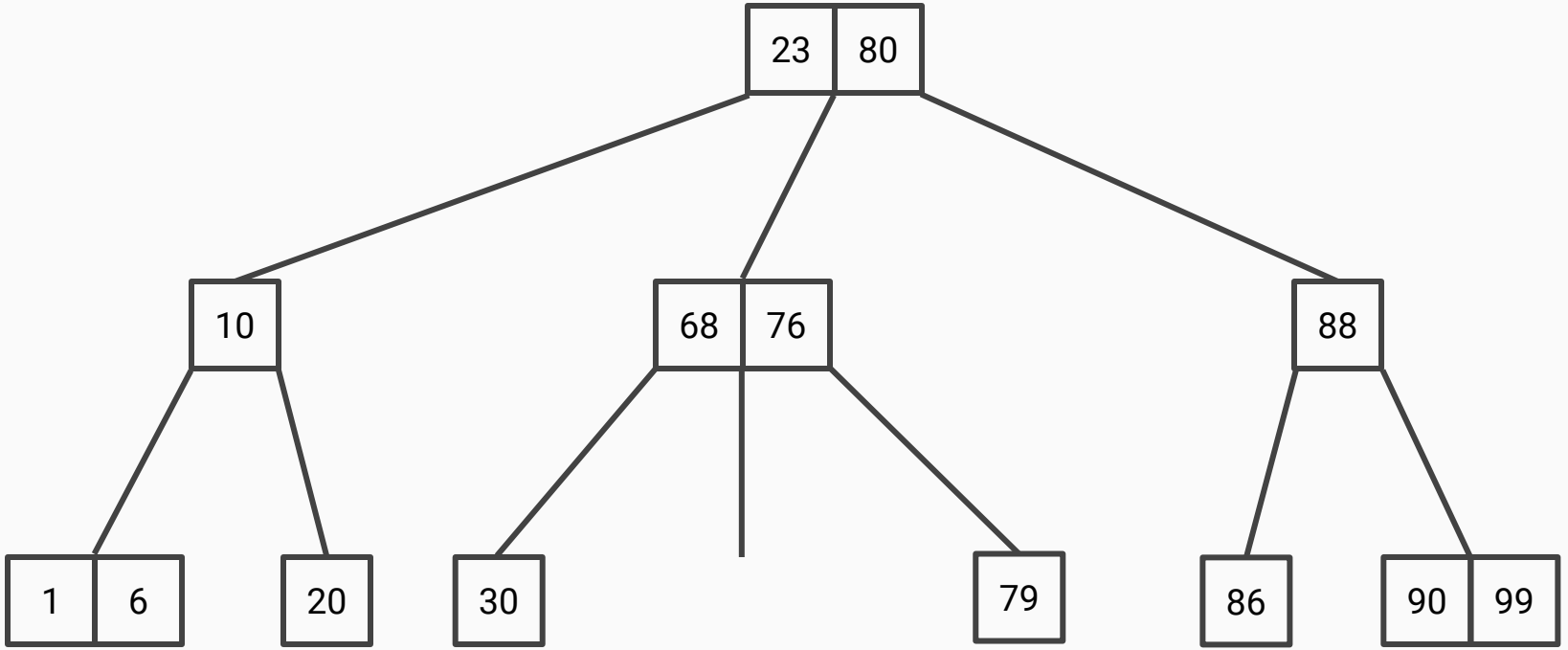
Aha! Our right sibling has one more key than the minimum. Let’s do a rotation.



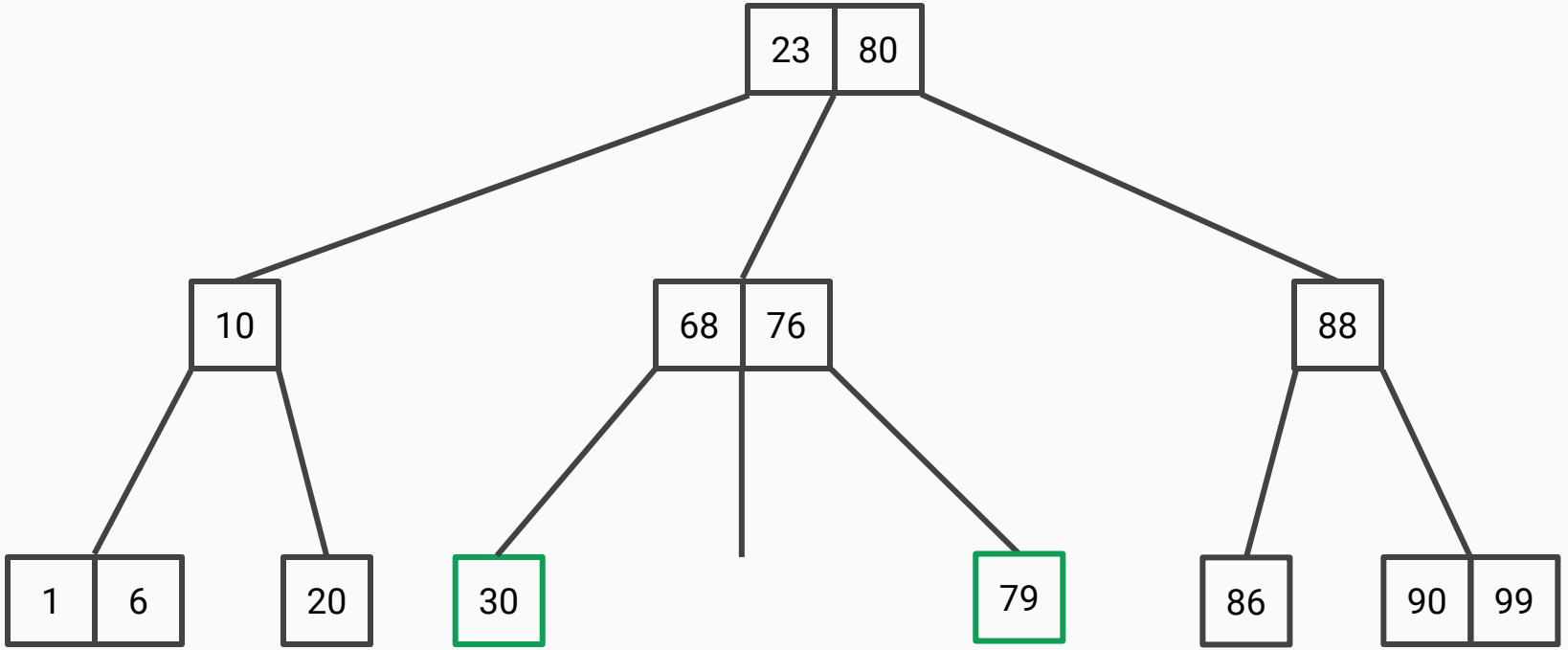
Okay, but what happens now if we remove 73?



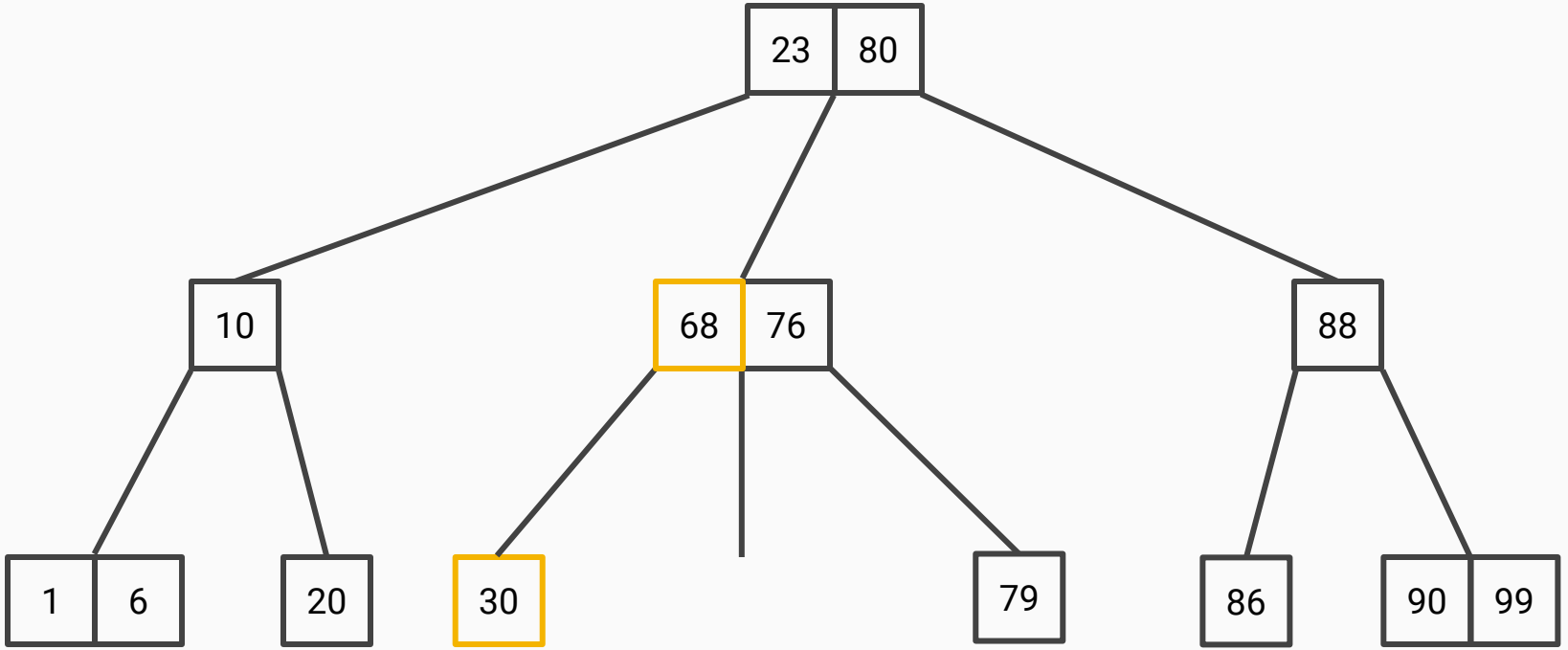
Oh no! Now we have one less than the minimum allowable number of keys in a node!



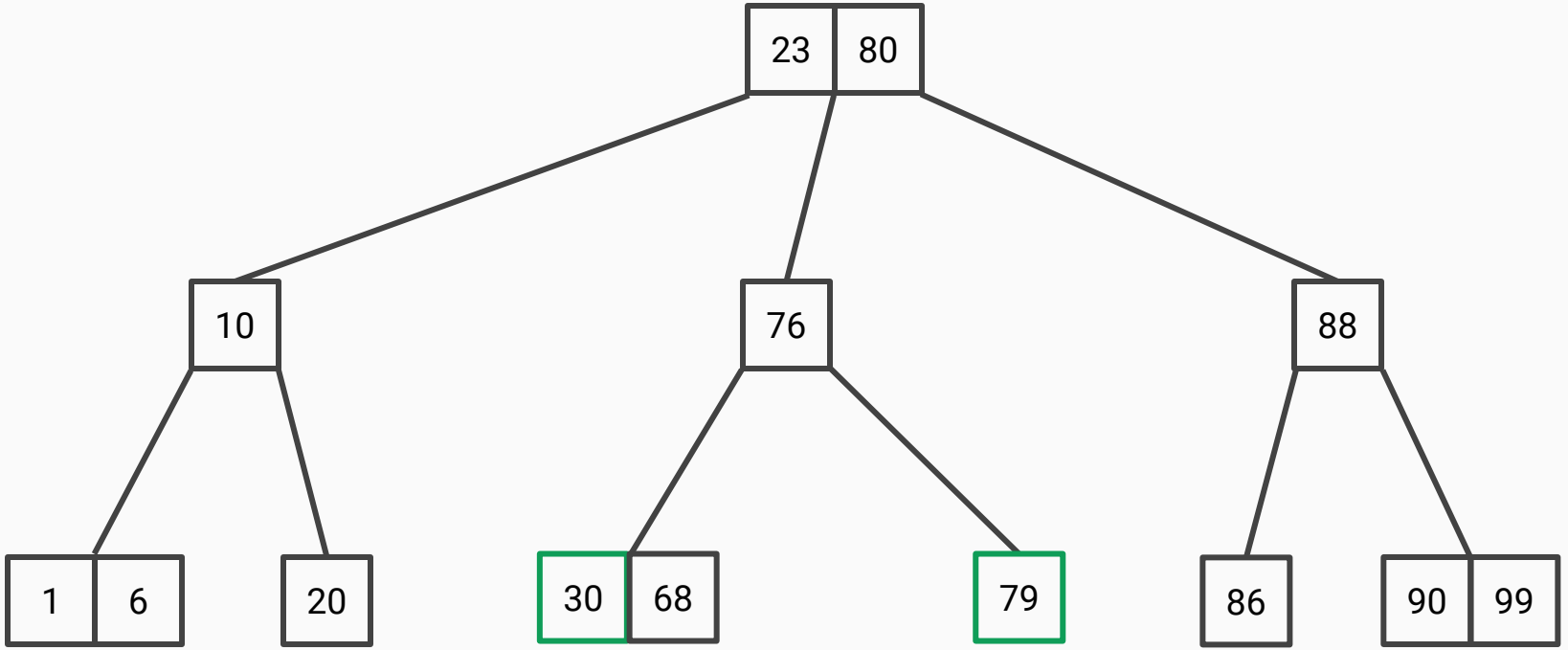
Let’s look at our siblings.



Ok at least one of them has at least the minimum number of elements with it. So let's merge them here with one node from the parent.



And we’re done!



DG Question 3

But wait? Why does this work?

DG Question 3

Properties of a B-Tree

1. All leaf nodes are of the same depth.
2. All non-root nodes have between $b - 1$ to $2b - 1$ keys.
3. The root has at most $2b - 1$ keys.
4. An internal node with k keys must have $k + 1$ subtrees.

Task: Show that after deletion these invariants are maintained! As a hint, think about the splits vs. the merges.

Solutions to Q2

1. All leaf nodes are of the same depth.

There are 3 possible cases:

(a) No merging/rotation was done.
(b) There was some merging, then the keys only propagate downwards.
(c) There was a rotation.

Then in all 3 cases the depth still doesn't change! In the second case, notice that if there was a need to merge, then a key is brought down from the parent recursively. In the worst case, a key is brought down from the root. Notice if the root has only one key left, then after it is used to merge, we would be left with a tree with a single node. So the leaf depth does not change!

Solutions to Q2

2. All non-root nodes have between $b - 1$ to $2b - 1$ keys.

There are 3 possible cases:

If you merge, then it must be that your node had than $b - 2$ keys. Then since you took one key from your parent, and you merged with one of your siblings (they must have had $b - 1$ keys), you get one new node with $b - 2 + b - 1 + 1 = 2(b - 1)$ keys.

Solutions to Q2

2. All non-root nodes have between $b - 1$ to $2b - 1$ keys.

There are 3 possible cases:

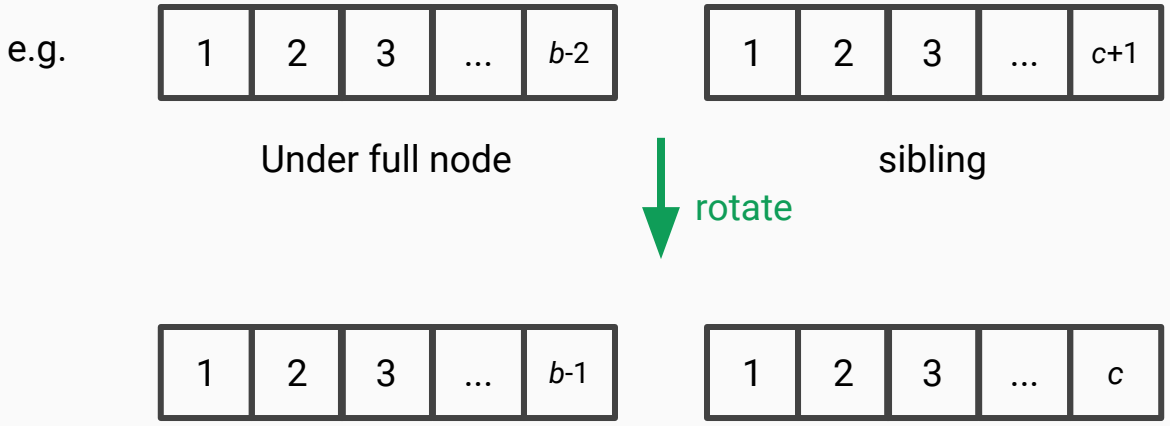
Also notice since you merged with your sibling, your parent has one less child, but that's fine! It also has one less key!

Solutions to Q2

2. All non-root nodes have between $b - 1$ to $2b - 1$ keys.

There are 3 possible cases:

If you rotate, then your sibling had more than than $b - 1$ keys, so now it has at at least $b - 1$ keys. And your current node has at least $b - 1$ keys!



Solutions to Q2

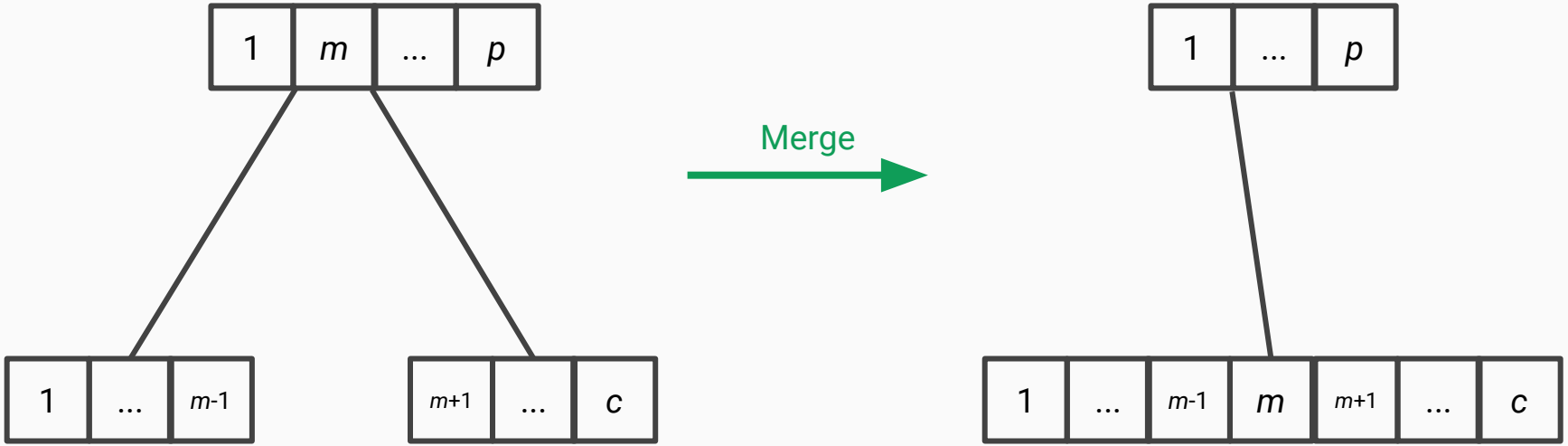
3. The root has at most $2b - 1$ keys.

Well deletion only reduces keys, so there's nothing here to worry about.

Solutions to Q2

4. An internal node with k keys must have $k + 1$ subtrees.

When we merge, we also “bring down” the subtree pointers. Therefore this property is always maintained. Notice that also the BST ordering property is maintained!



Concluding Remarks

B-Trees

Before we end, there are a few things to note:

- B-Trees are heavily modifiable! There are many many variants, which differ in (possible subtle) ways. a-b trees, B+ Trees etc. Go look them up if you wish to know more about this data structure everyone uses!
- Head on over to this B-tree visualisation site if you wish to see them in action!

One last question: If you knew the exact hardware you were implementing a B-Tree on, what should you set the value *b* to be?

For that given value, how many levels must the tree have before it overtakes the number of humans we have on Earth?

Thank you!