B-Trees

You are given a small header file with three function prototypes: *insert*, *remove*, and *query*.

Everything else is up to you.

B-Tree Assignment: The Good News

This assignment is possibly the hardest one you will have in your entire college career. This is a good thing, because it gives you a chance to apply all the skills you've learned until now. It also gives you a chance to apply the most important skill (in my opinion) that you can possibly learn: *how to design*.

Design is not just about solving a problem that somebody else has specified for you. Design is not 'solve for x'. Design is as much about *understanding and framing the problem* as it is about solving the problem.

Congratulations: Every single one of you is now a designer. This is the most critical and widely applicable skill you can acquire in college.

B-Tree Assignment: The Good News

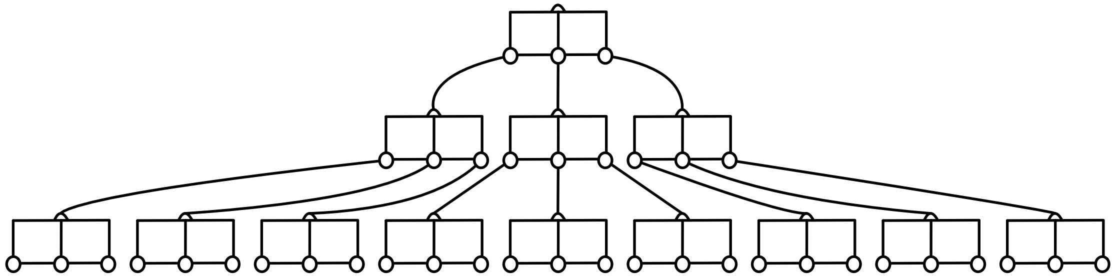
It is also good news because I am going to make this assignment worth 30/15 points. In other words, if you turn it in on time you can get up to 30 points, which is 200%.

In other good news, this particular assignment will be capped at 15 after the due date. So even if you don't get everything done before the due date, you can still receive 100% (15/15) on the assignment up until the end of the semester.

RetroGrade will run several tests for each function you must write. So even if your *insert* function doesn't properly handle all possible configurations you can still get points for those that it does correctly handle.

B-Tree Structure:

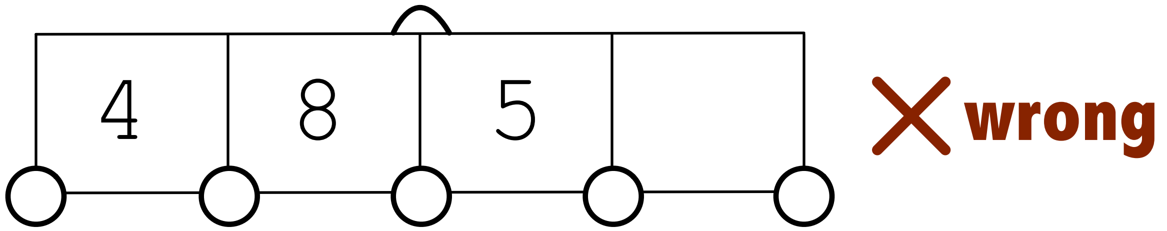
A B-Tree of order **m** has **nodes** that:
- contain up to m-1 **keys**
- may refer to up to m **children**
- may be **root**, an **index**, or a **leaf**
- record their current **key count**

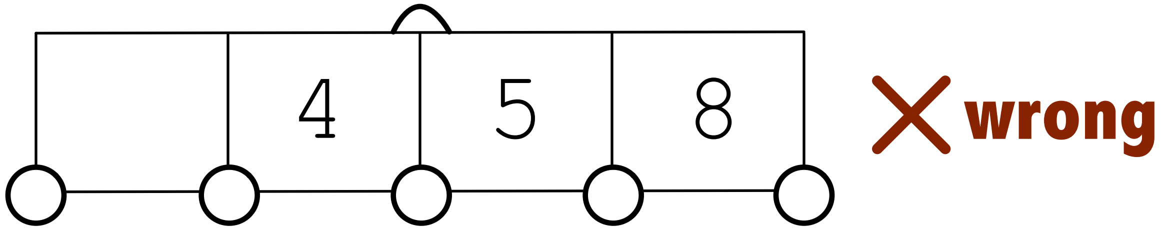

B-Trees look like this.

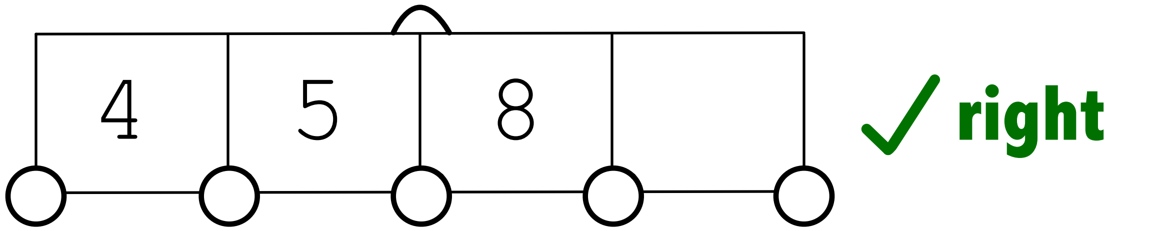
```
struct btree {
  int num_keys;
  int keys[m-1];
  bool is_leaf;
  btree* children[m];
};
```

B-Tree Invariants:

(1) Node keys are kept in ascending order starting at **0**.

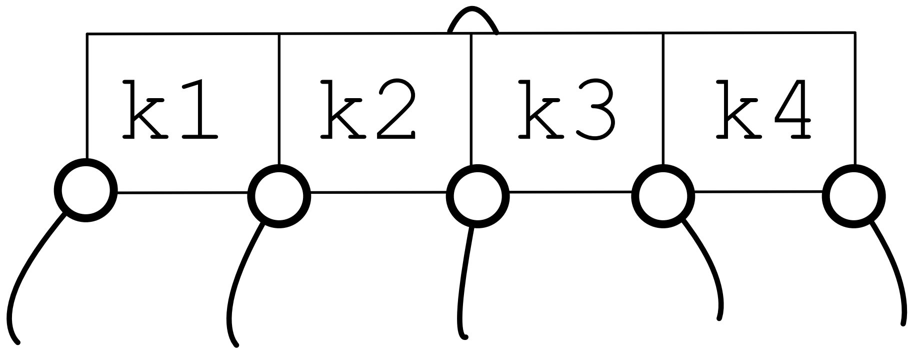






B-Tree Invariants:

(2) Every node has at most *m* children.



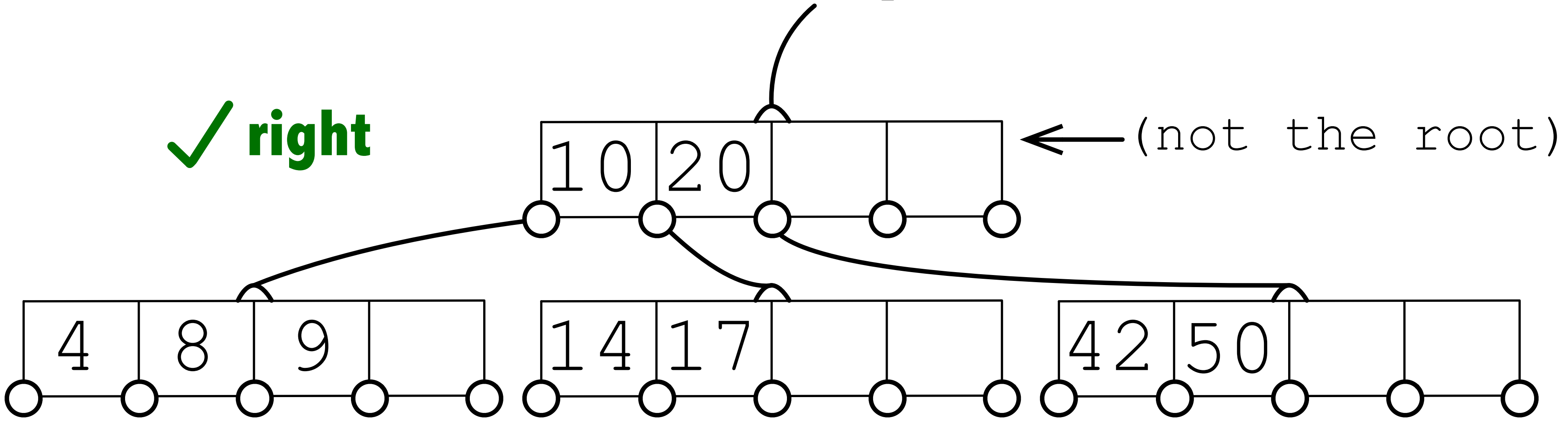
✓ right

In this case, m=5. There are four keys, which mean there are five slots 'between' these keys. After that, there's no more room.

✗ I don't know how to draw it wrong.

B-Tree Invariants:

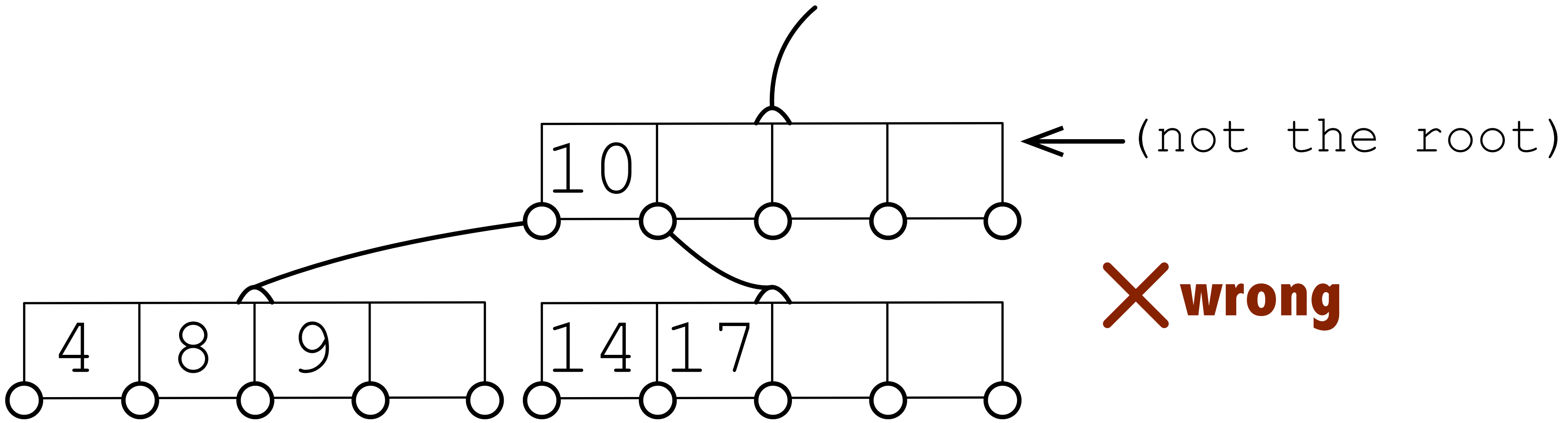
(3) Index nodes have at least **ceil(m/2)** children. (m/2 rounded up to nearest int)



m is the number of allowed children. In this case that is 5. 5/2 is 2.5. Round up to get the minimum number of children = 3.

B-Tree Invariants:

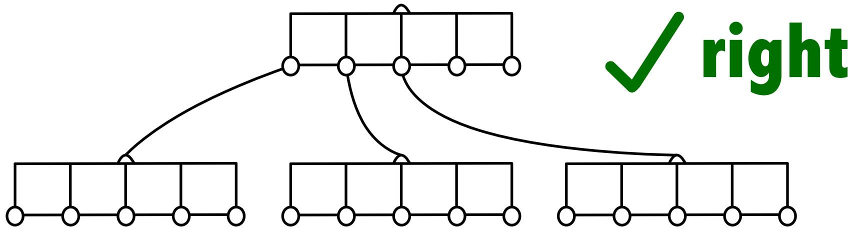
(3) Index nodes have at least **ceil(m/2)** children. (m/2 rounded up to nearest int)

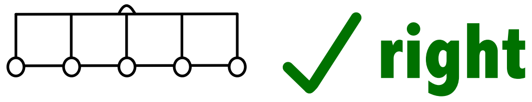


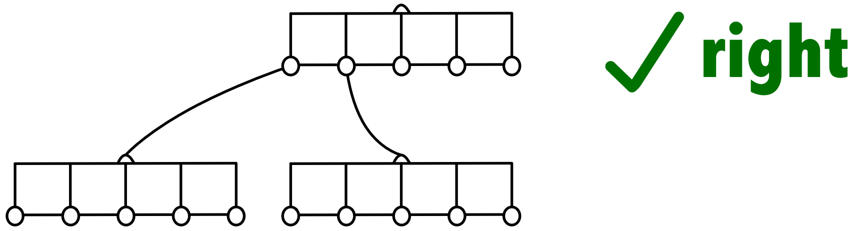
Together with invariant #6, this means that index nodes necessarily have at least **ceil(m/2) - 1** keys. So for an order 5 B-Tree, index nodes must have at least 2 keys.

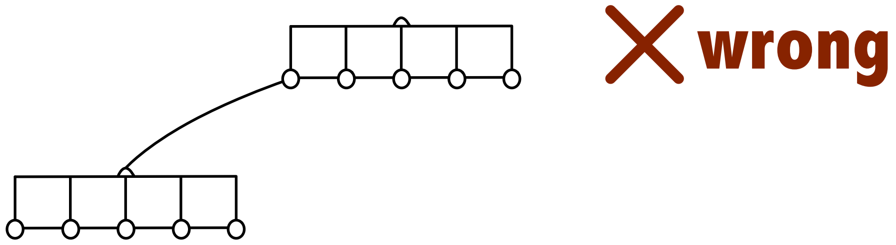
B-Tree Invariants:

(4) If the root is not a leaf, it has at least two children.



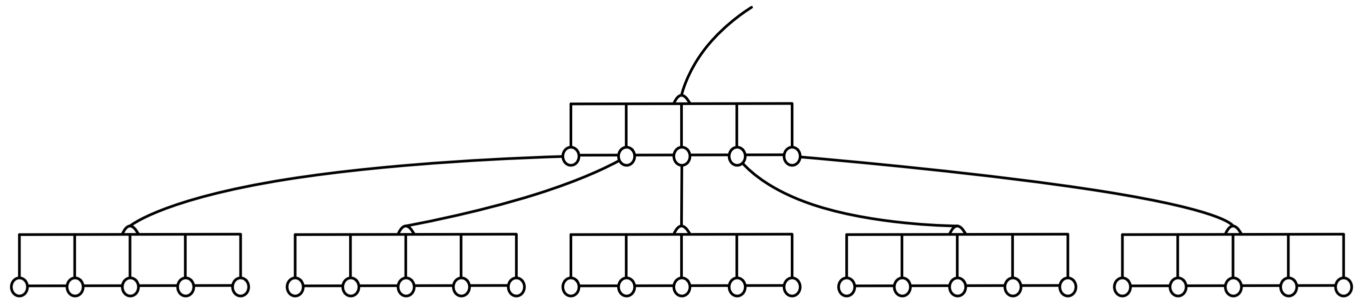






B-Tree Invariants:

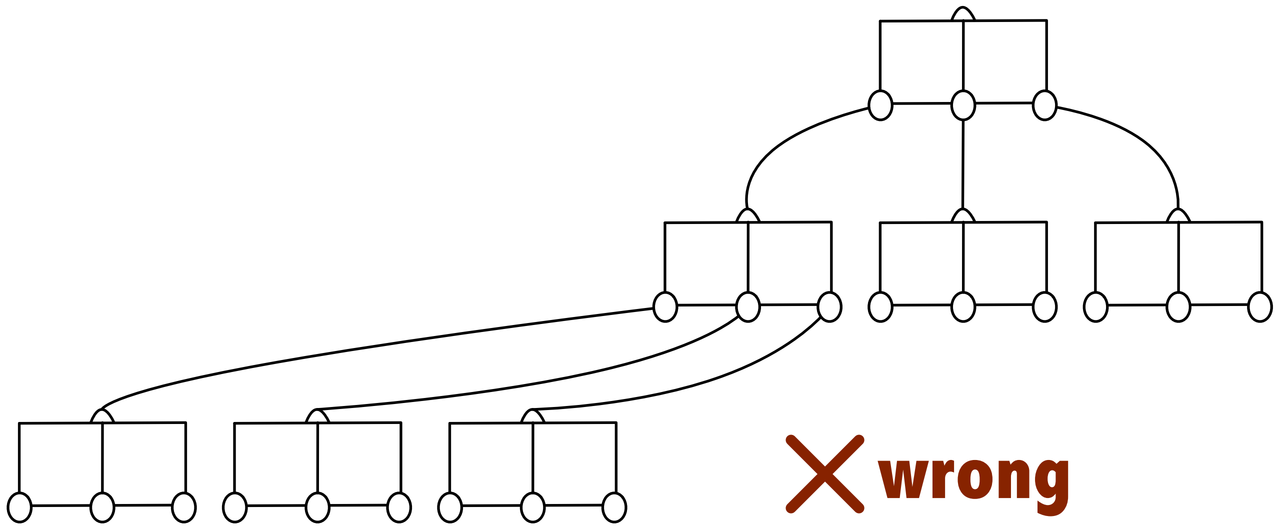
(5) All leaves are at the same level (and for *real* B-trees, they hold information (the payload)).



This means the tree is perfectly balanced from a node perspective. Keys may not be evenly distributed (meaning there is wasted space), so long as the nodes support the B-Tree invariant.

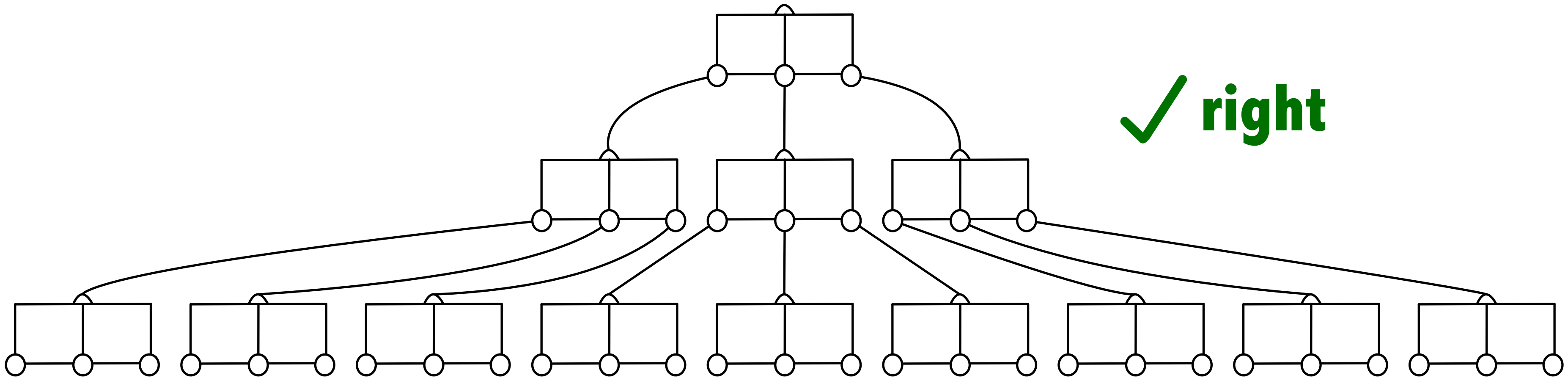
B-Tree Invariants:

(5) All leaves are at the same level.

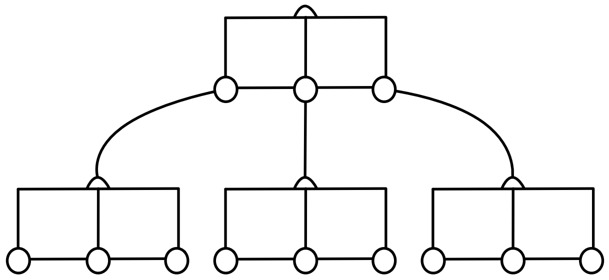


B-Tree Invariants:

(5) All leaves are at the same level.



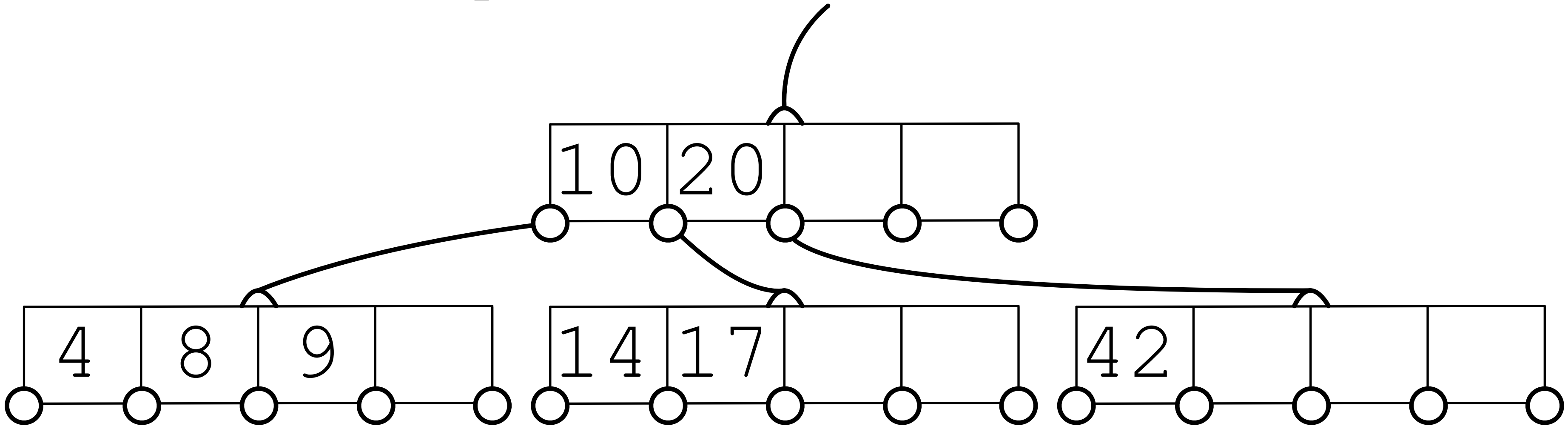
✓ right



✓ right

B-Tree Invariants:

(6) A non-leaf node with **k** children contains **k-1** keys.

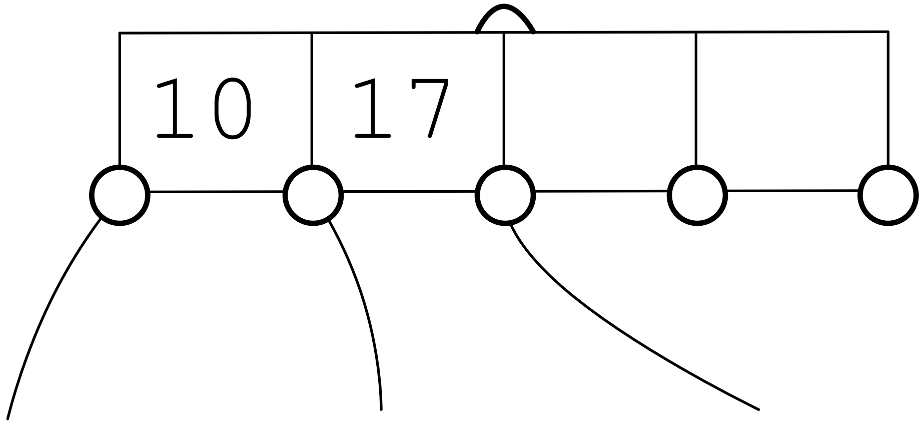


Here you see the [10, 20] node has three child nodes and two keys. If it had four child nodes it would need to have three keys. Five child nodes would require four keys.

B-Tree Nodes (recap):

Nodes have the following fields:

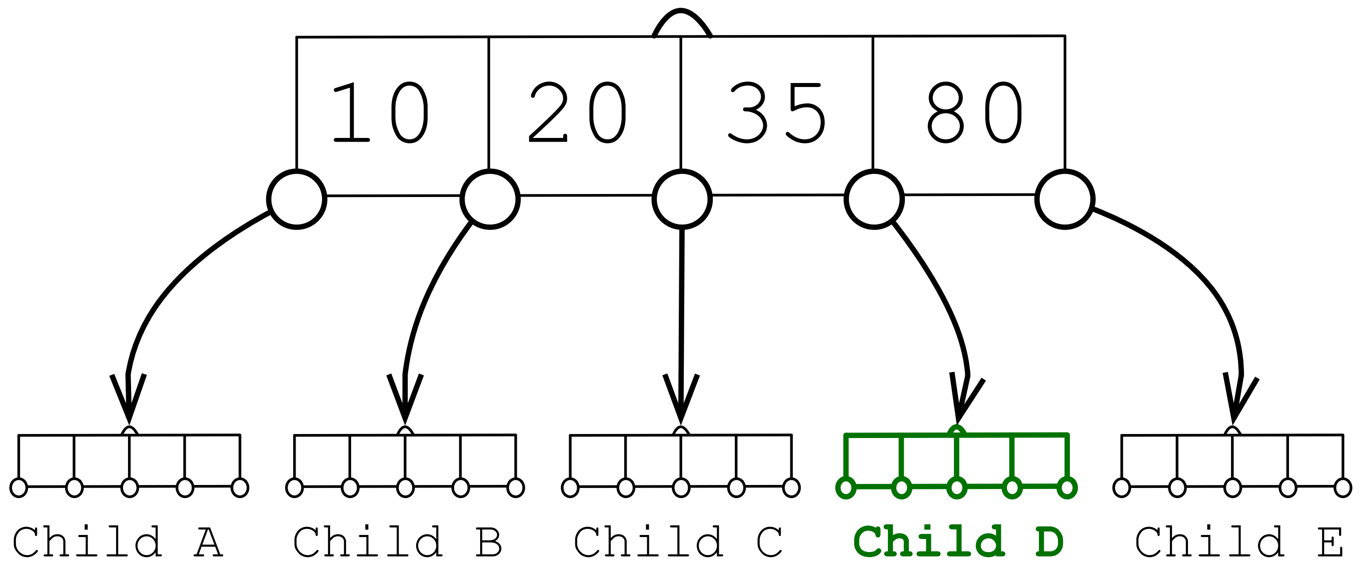
-> **num_keys**: number of keys the node is currently using.
-> **keys**: array of actual key values.
-> **is_leaf**: is the node a leaf? true/false.
-> **children**: array of child pointers. should be (num_keys + 1) of them.



```
num_keys = 2
keys     = [10, 17]
is_leaf  = false
children = <3-array>
```

B-Tree Nodes (recap):

Does the B-tree contain the key 35?
How about the key 63? Where would we look?
How would we use the B-tree node structure
to determine which child to look at?



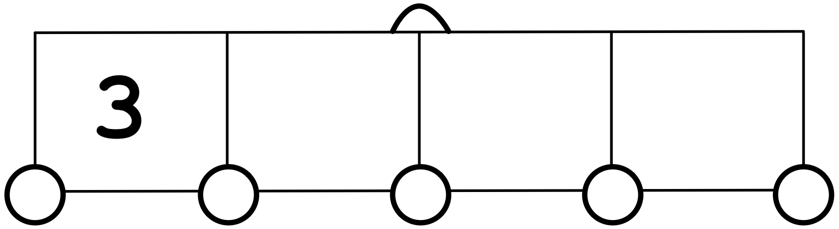
Note: the insert and remove examples were adapted from the excellent guide at this URL:

cis.stvincent.edu/html/tutorials/swd/btree/btree.html

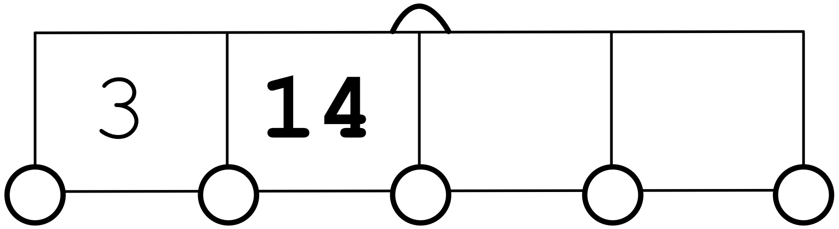
You can get this on Google by searching for "Saint Vincent btree". It is also linked from the homework README.

B-Tree Operations: Insert

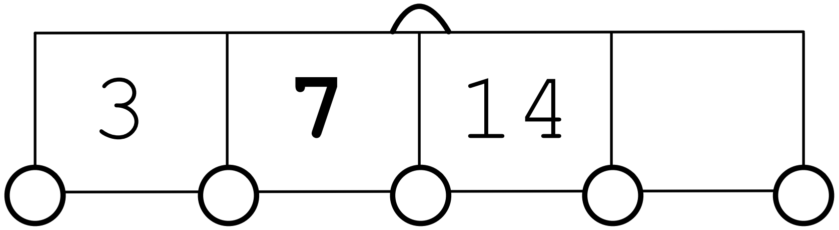
Insert 3, 14, 7, 1 into an initially empty B-Tree.



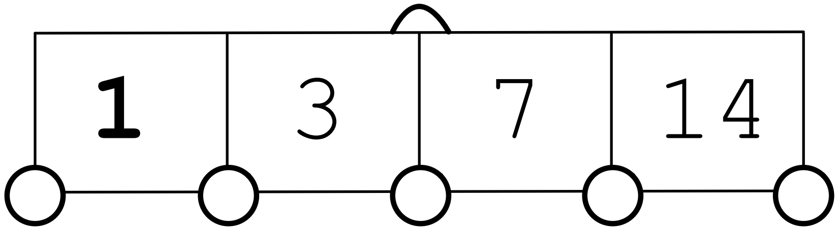
Special case: insert into an empty root node. Easy.



Always grow left to right.



Keep keys in ascending order.

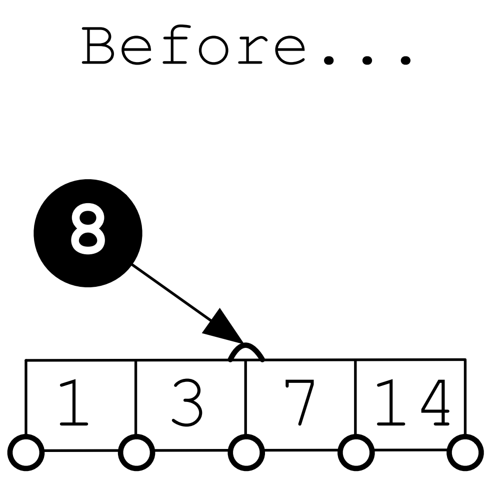


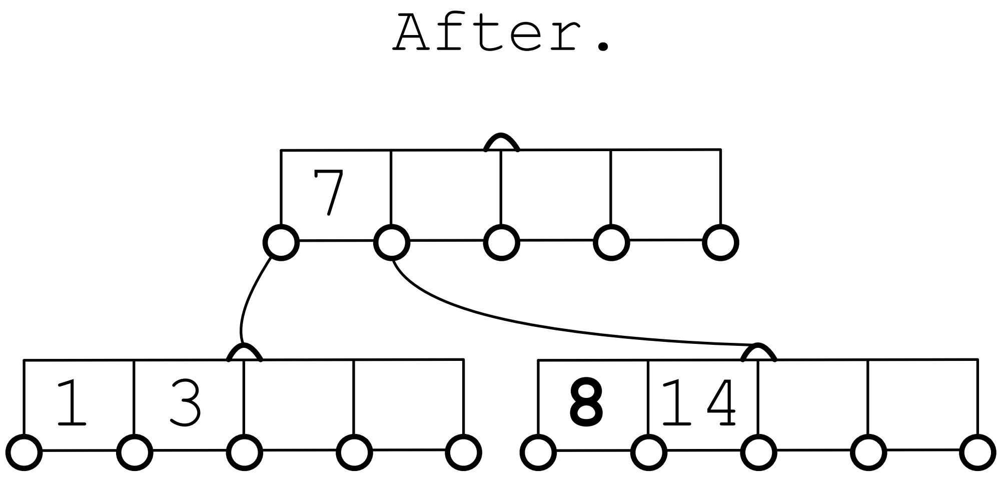
Now the node is *full*.

Note: the insert and remove examples were adapted from the excellent guide at **http://cis.stvincent.edu/html/tutorials/swd/btree/btree.html**

B-Tree Operations: Insert

When we insert 8, we'll overflow the root node.
Solution: split things up!



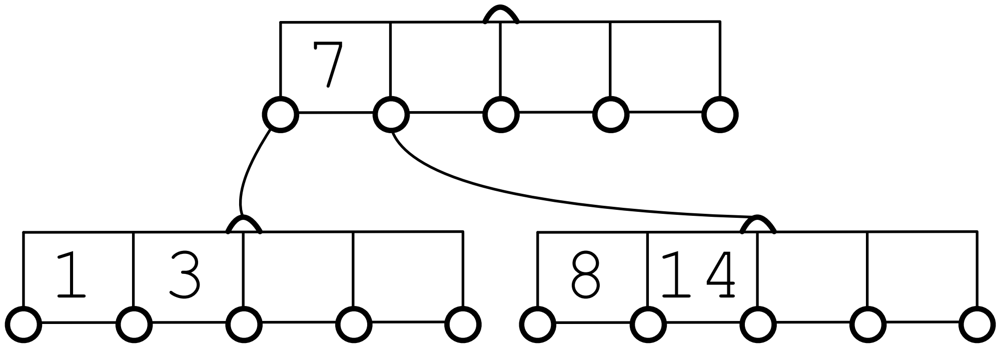


Now our tree has height=2. Remember to ensure your root node pointer is correct.

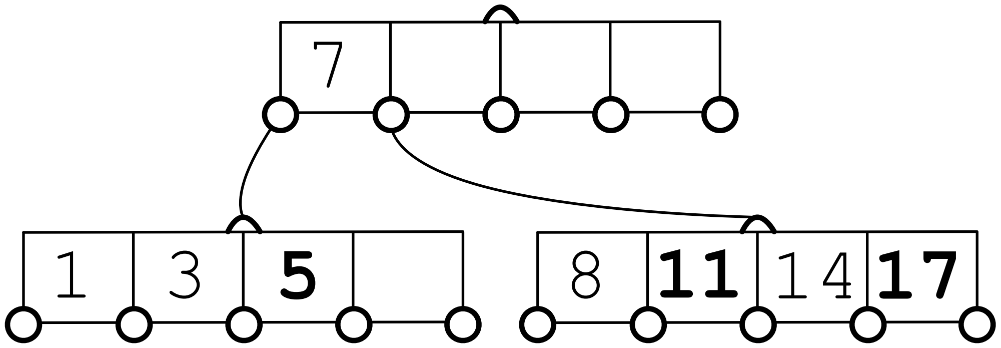
B-Tree Operations: Insert

We can insert 5, 11, and 17 without splitting nodes any more because there is room in the appropriate node when needed.

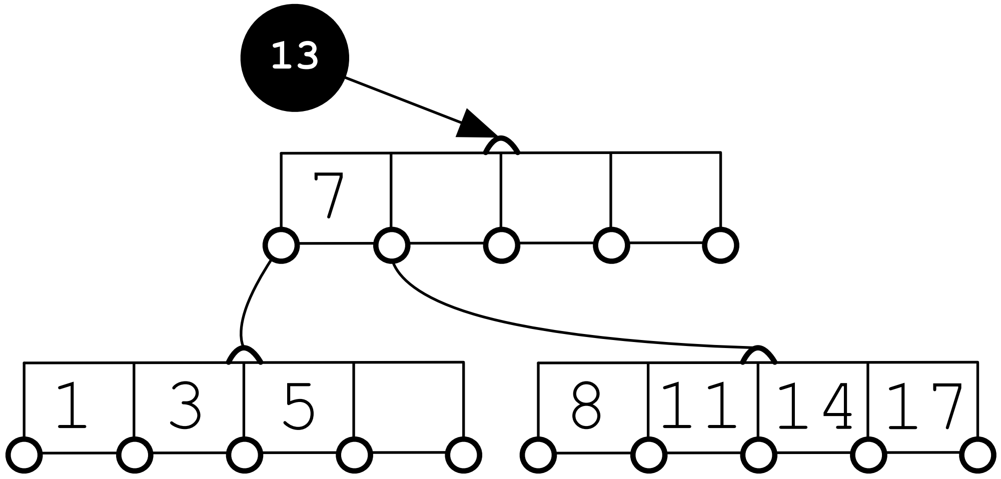
Before:



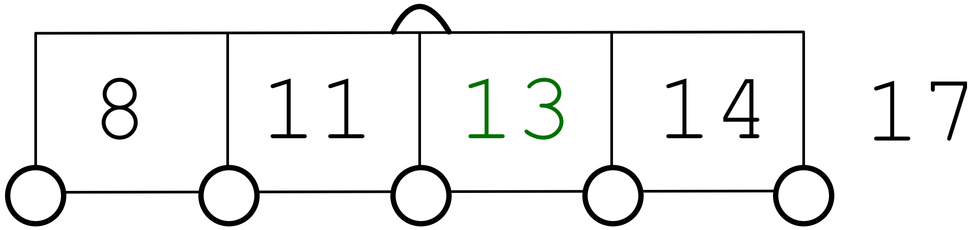
After:



B-Tree Operations: Insert

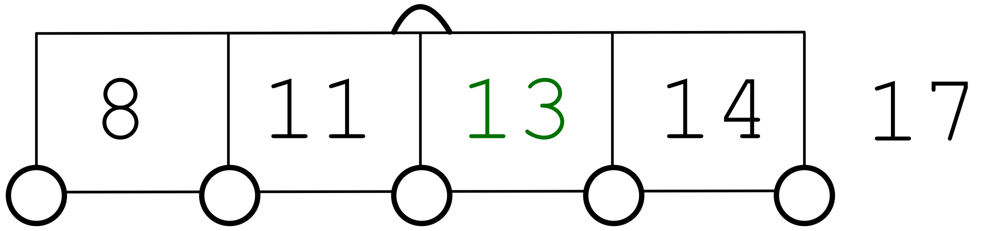


What happens when we insert a number that causes a leaf to overflow? Split!

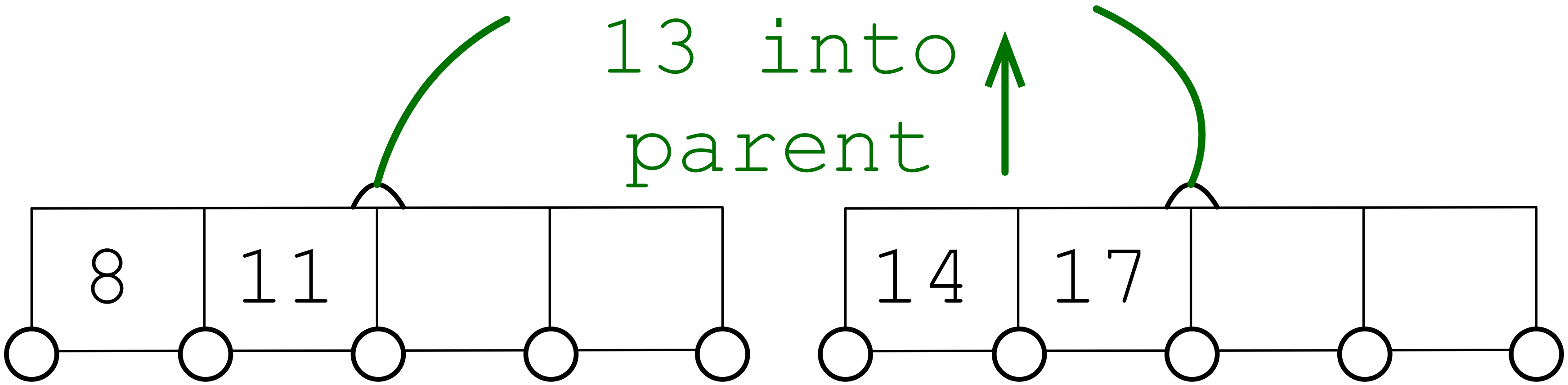


There are a few ways to split a node. They all have the same end effect. Here's how I think of it. Insert your new key into the key array (or a copy of it), then identify the median value (13, in this case).

B-Tree Operations: Insert



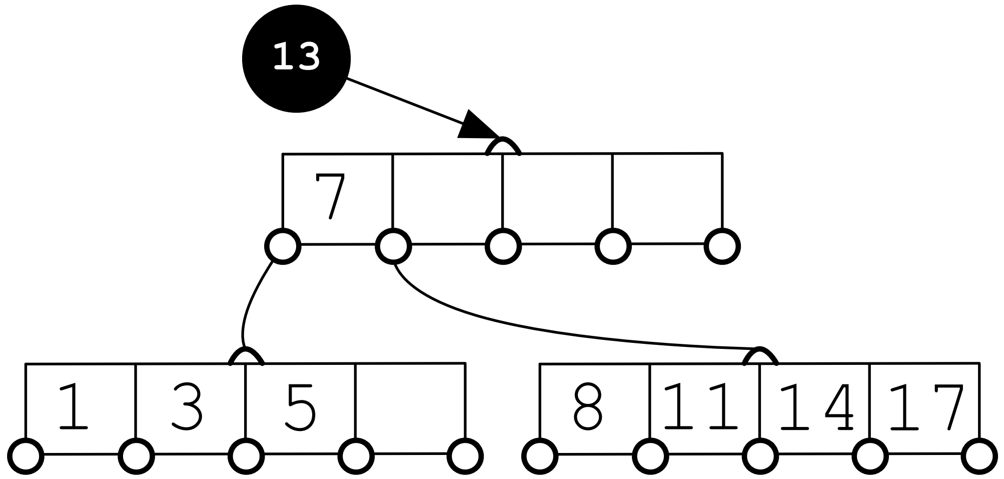
Then split the node into two. You get left side and right sides. You may re-use the original node for one.



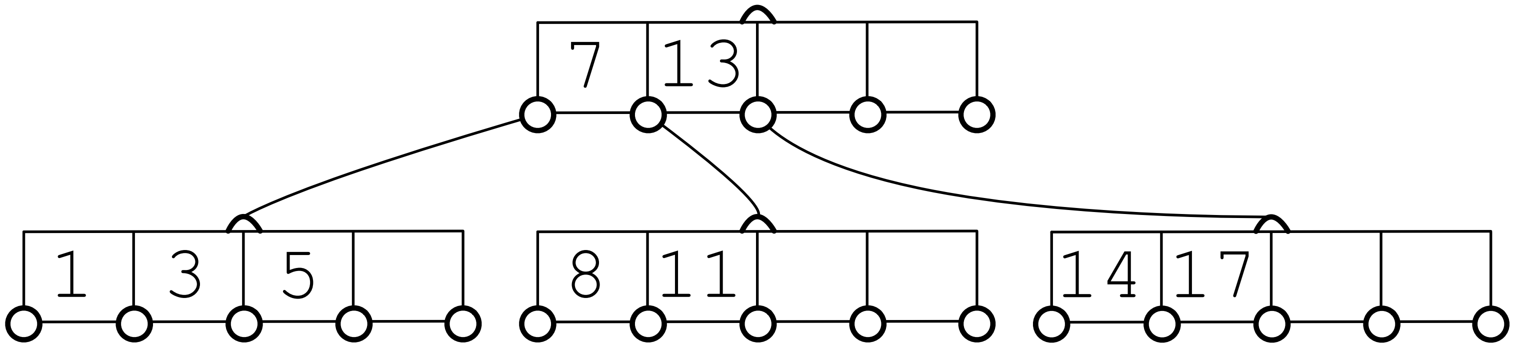
Two remaining problems: the parent is still pointing to the original node. Also: the median value 13 is now homeless. See if the parent has room, and if it does, you can solve both of these problems at one time.

B-Tree Operations: Insert

Before we inserted 13:



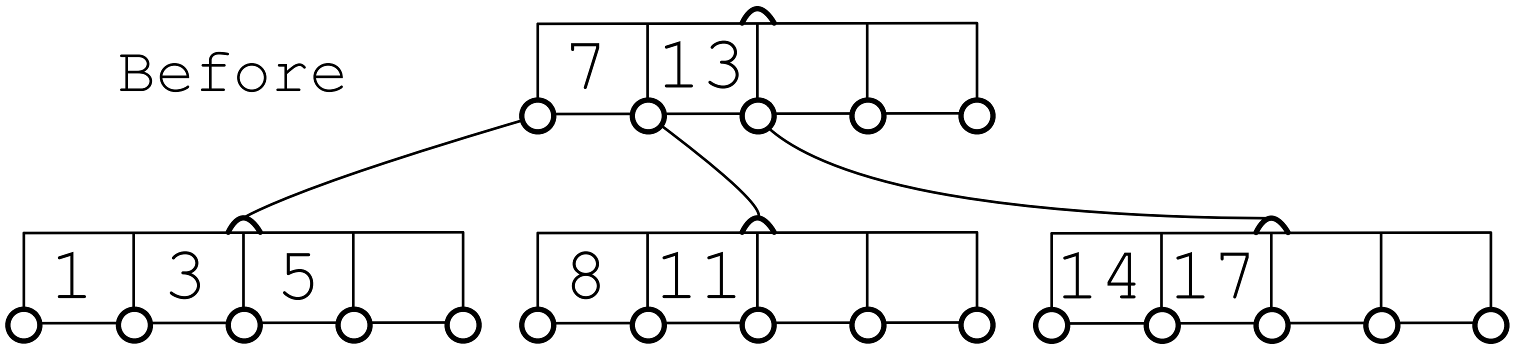
After inserting 13 and splitting the full node:



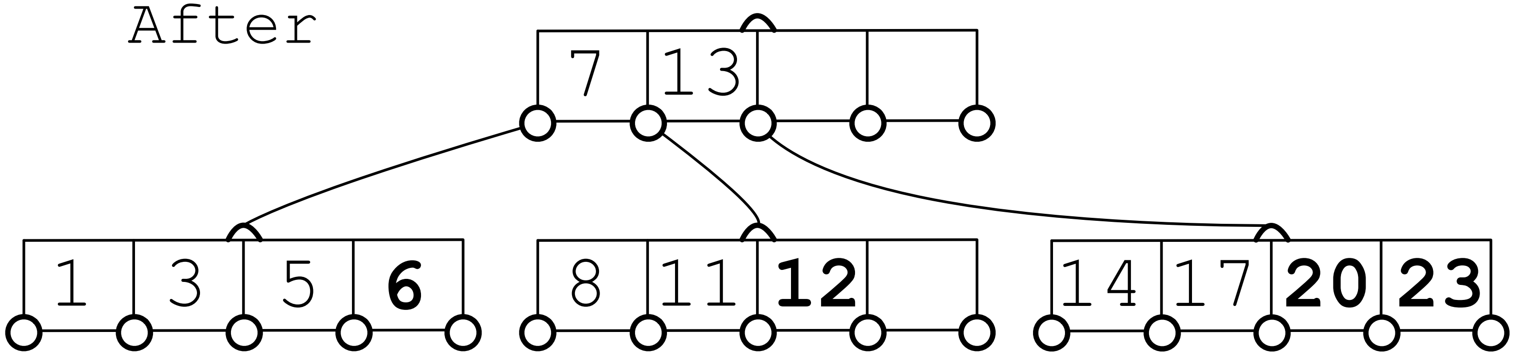
B-Tree Operations: Insert

Can insert 6, 23, 12, and 20 without splitting:

Before



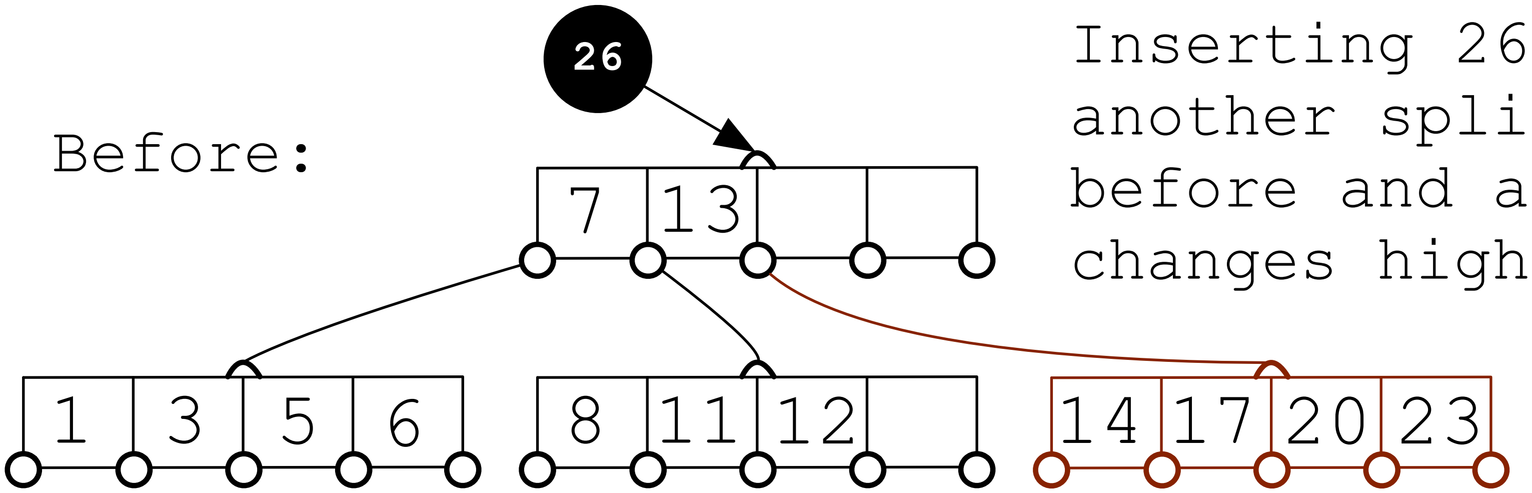
After



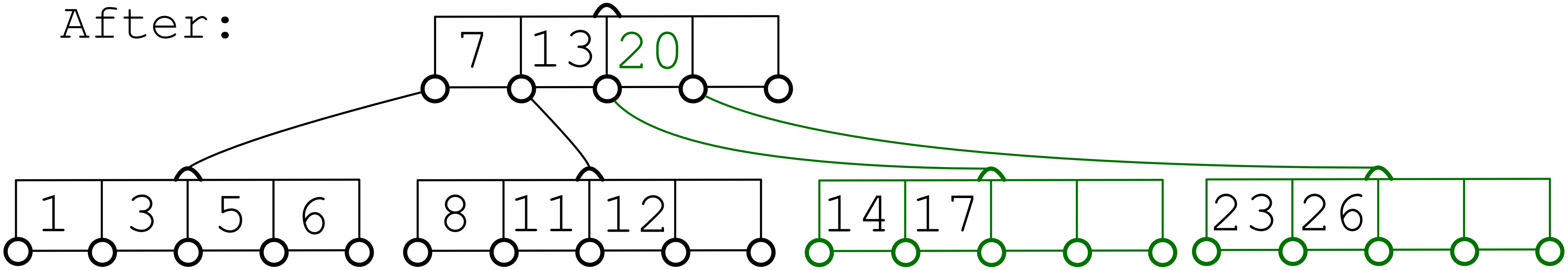
B-Tree Operations: Insert

Inserting 26 leads to another split. Here are before and after with changes highlighted.

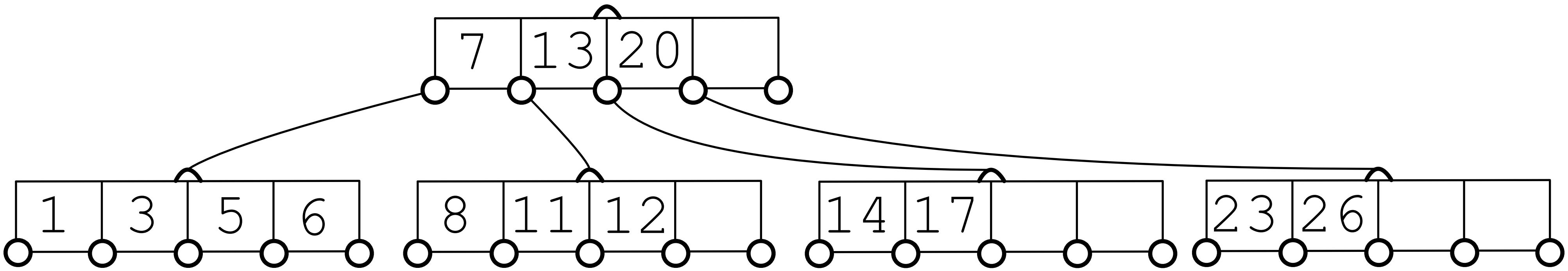
Before:



After:



B-Tree Operations: Insert



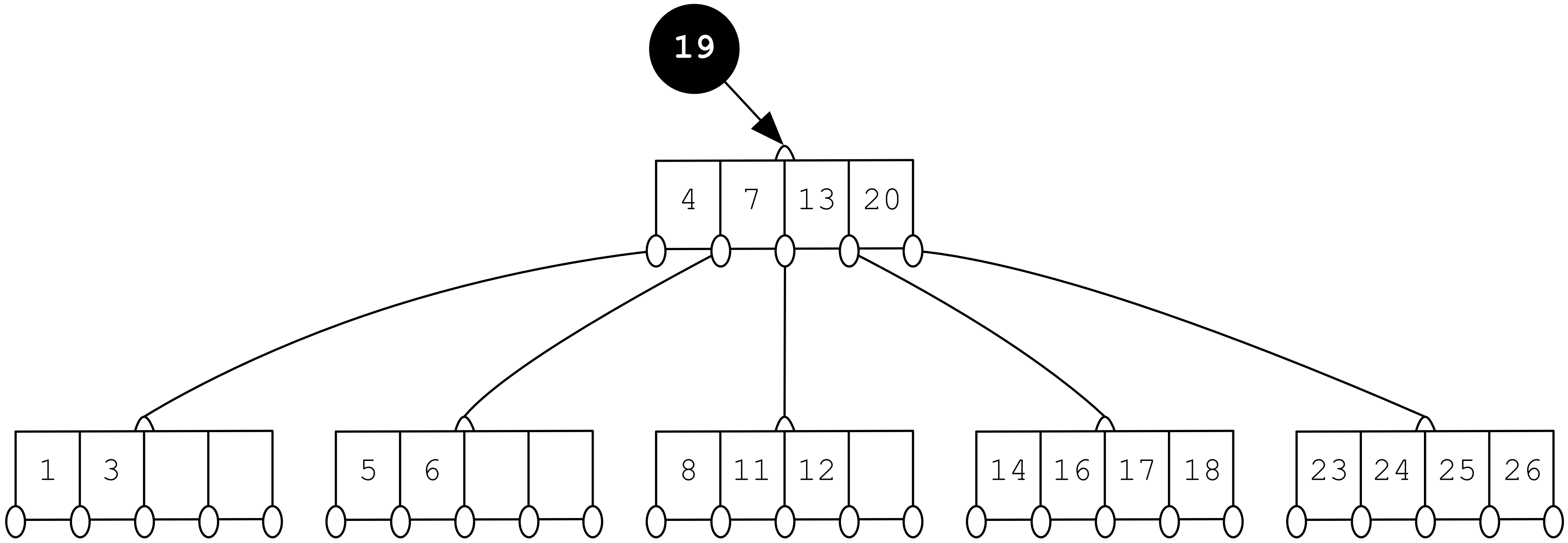
Take a second to look at the tree and think about the B-Tree invariants. Note that the leaf nodes may have **ceil(m/2) - 1** keys, so 2 is the minimum number that we can accept. This value is needed so we can split leaf nodes the way we're doing it.

I'll add the remaining numbers until we have the next (and scariest) insertion situation: when we need to split *and* raise the height of the tree.

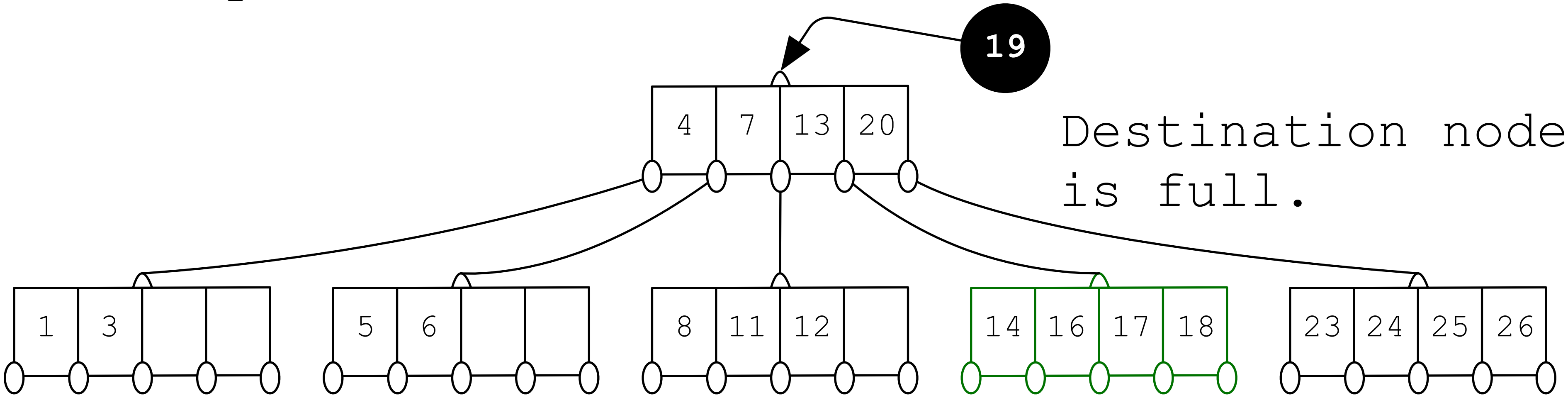
Numbers we add in the meantime: 4, 16, 18, 24, 25.

B-Tree Operations: Insert

When we insert 19, craziness ensues. The 14-18 node gets split, sending 17 up to the parent. But the parent is full too! So, we have to split the parent as well. But that's the root node. We did this one time before. Only difference is we have to babysit the child pointers. This ends up with a B-Tree with 3 layers of nodes.

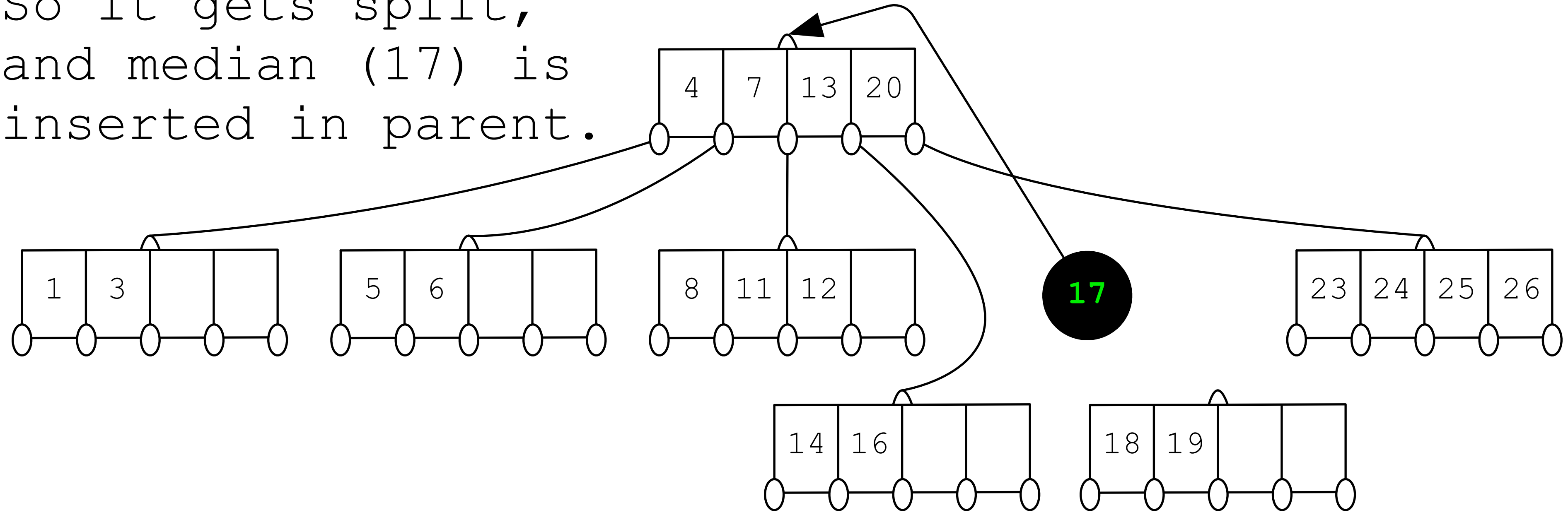


B-Tree Operations: Insert



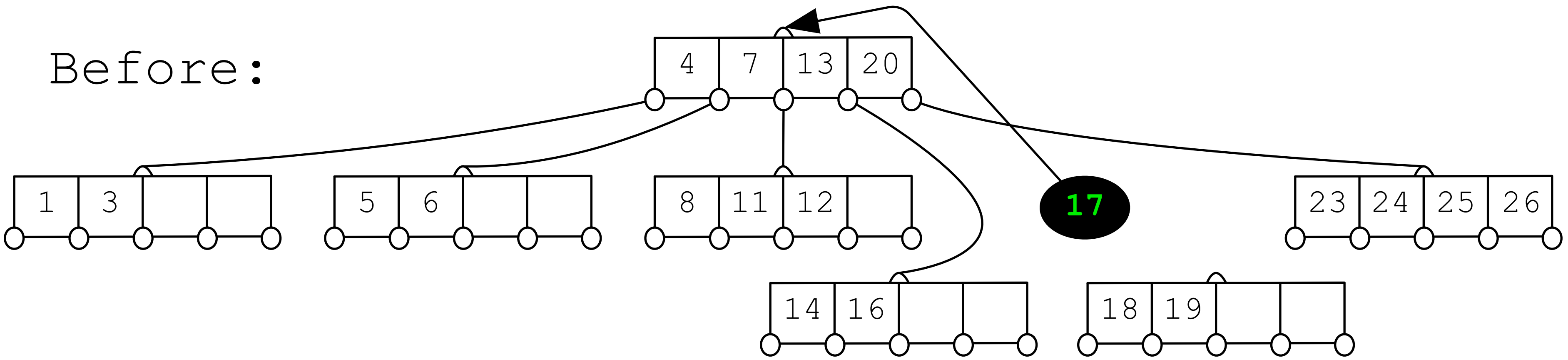
Destination node is full.

So it gets split, and median (17) is inserted in parent.

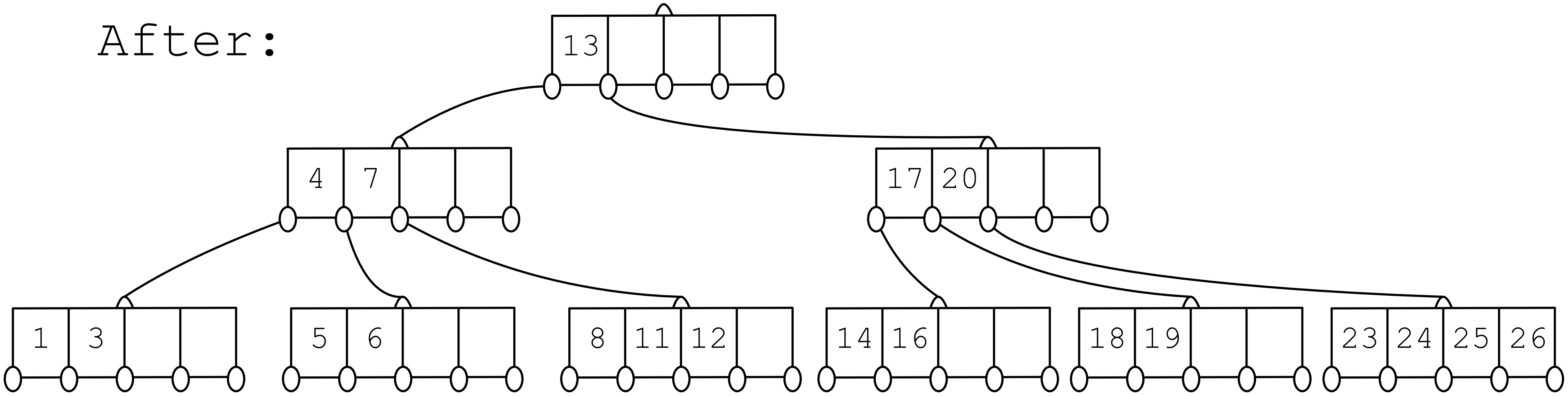


B-Tree Operations: Insert

Before:

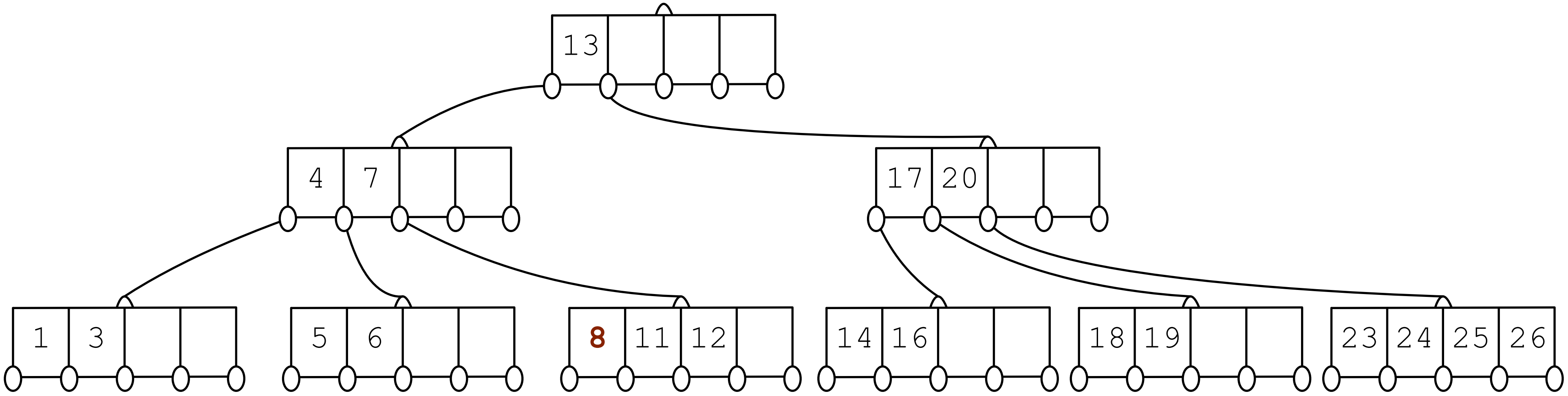


After:

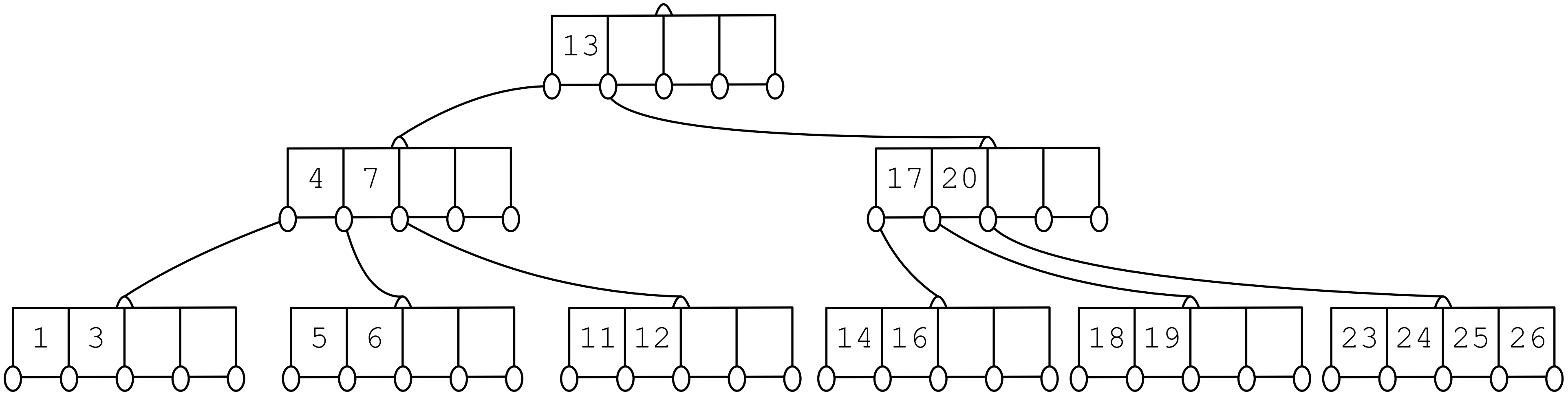


B-Tree Operations: Remove

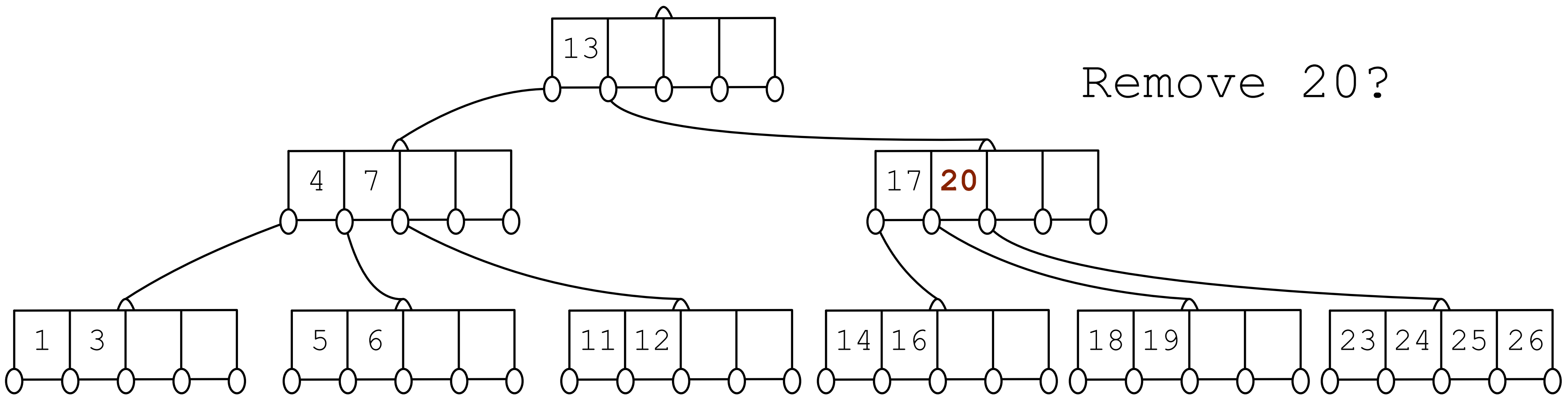
Deleting items can be straightforward. Delete 8:



Easy. Removing '8' breaks no invariants.



B-Tree Operations: Remove



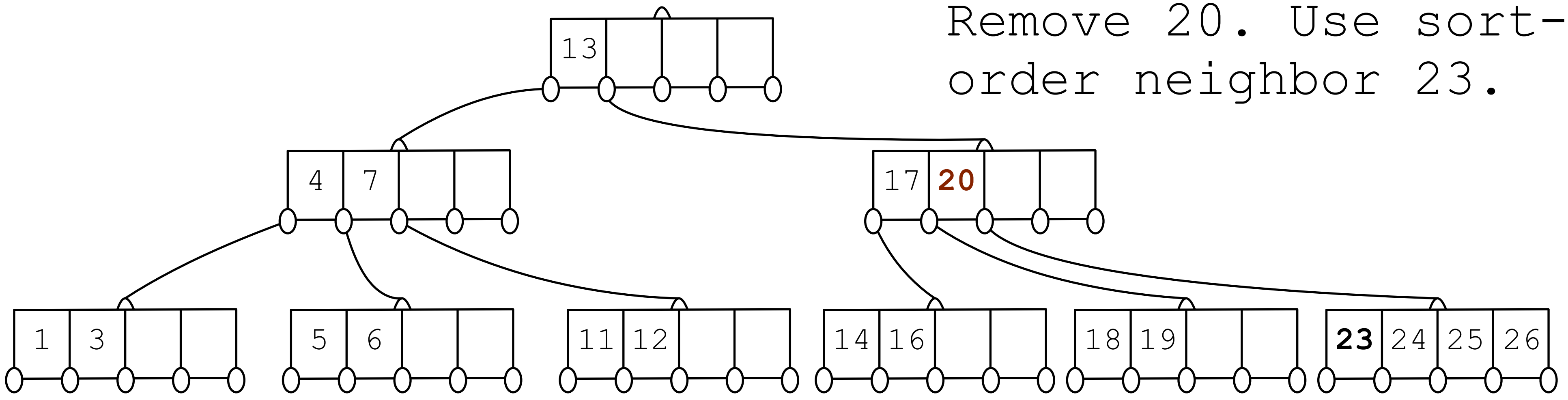
Remove 20?

It gets tricky when we delete from a non-leaf node.

We can use the same idea from the binary search tree's *remove* operation: using doomed key **D**, find a sort-order neighbor **N**. Replace **D** with **N's value**, then remove **N**.

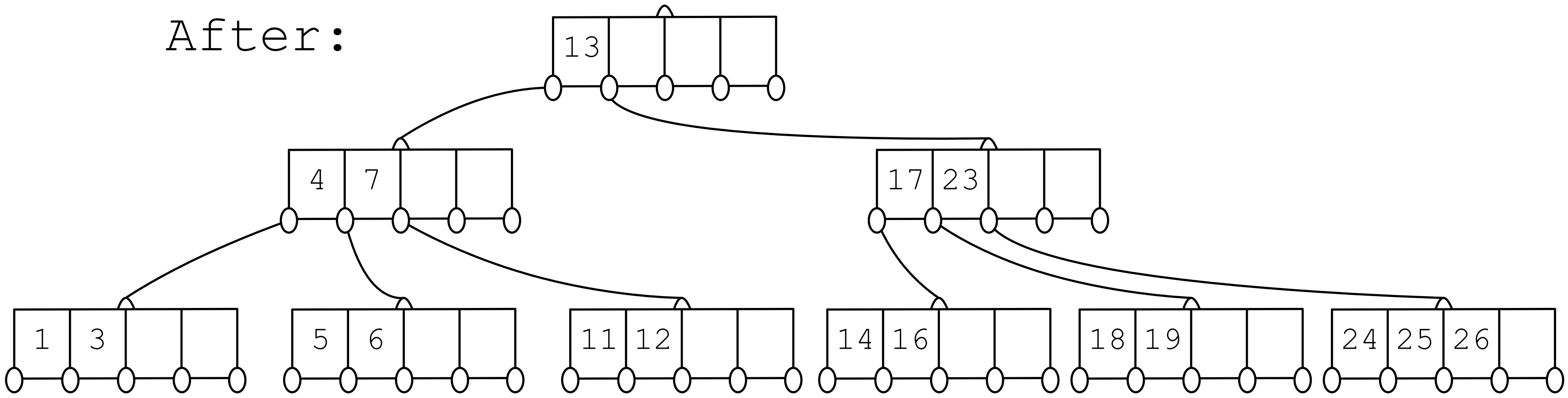
We may need to clean up a mess after that happens. But the idea is that we *always* end up reducing this problem to deleting from the leaf level.

B-Tree Operations: Remove

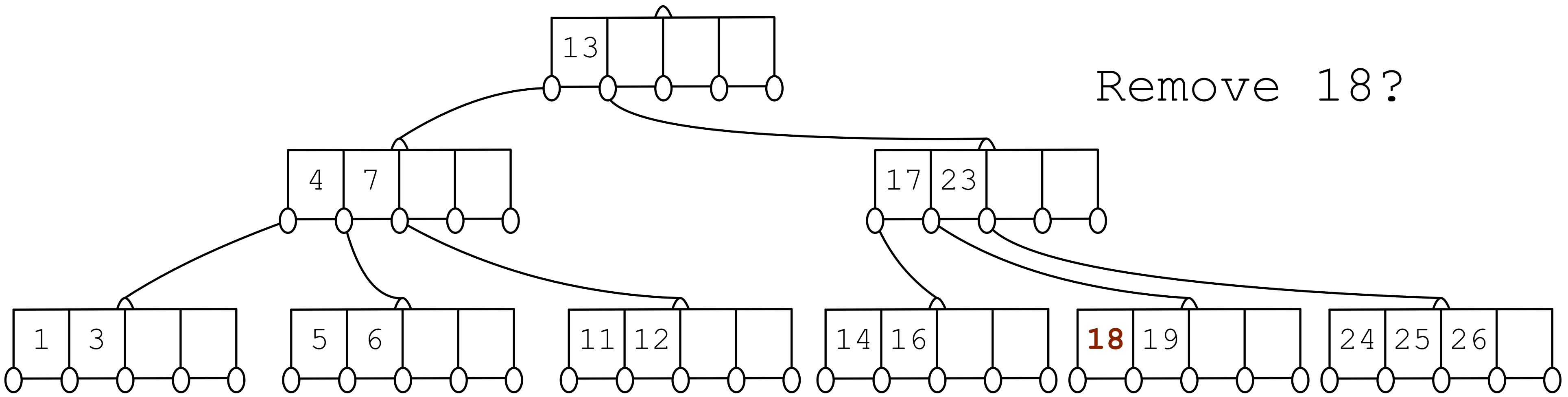


Remove 20. Use sort-order neighbor 23.

After:



B-Tree Operations: Remove

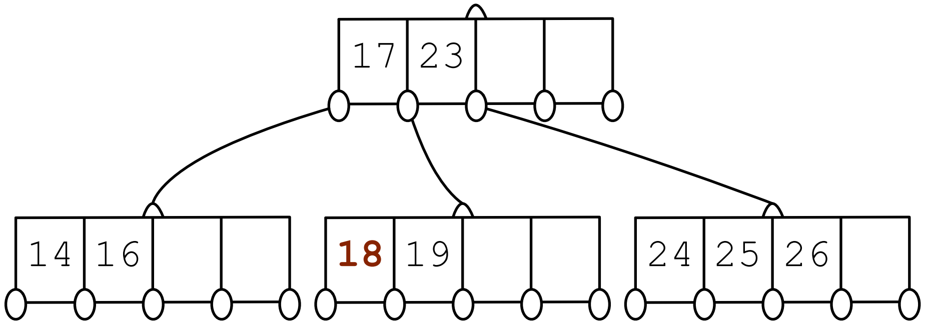


Remove 18?

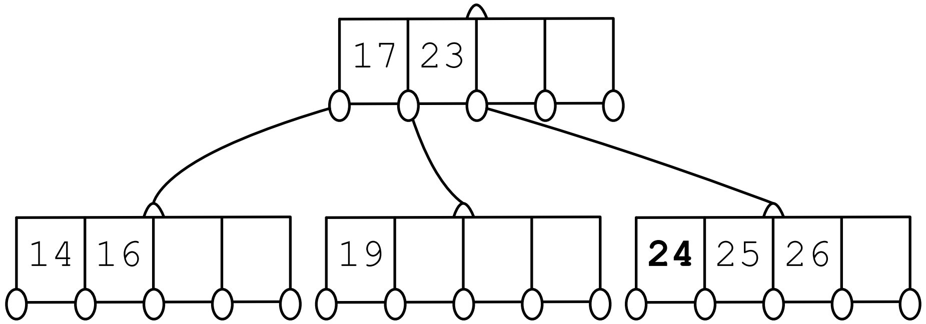
If removing from a node will result in too few keys, it might be possible to *borrow* from an adjacent sibling node. (Siblings share a parent.)

Can we rearrange the keys among neighboring siblings and the parent so the minimum number of keys are present in the leaf level?

B-Tree Operations: Remove

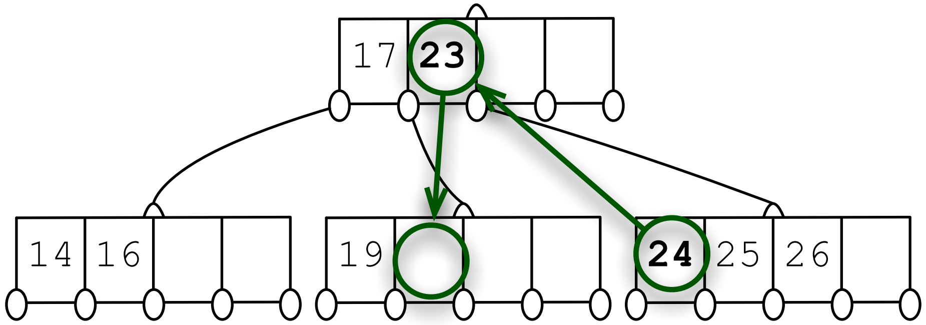


Removing 18.



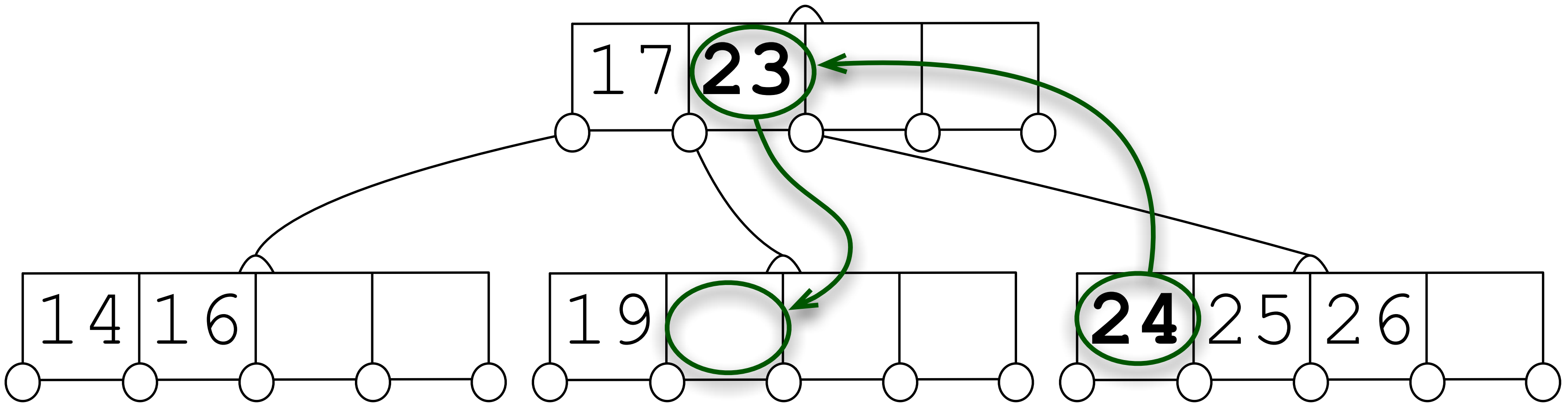
✕ **Node w/ 1 key is bad.**

Can't borrow from left neighbor. Right neighbor has an extra key.

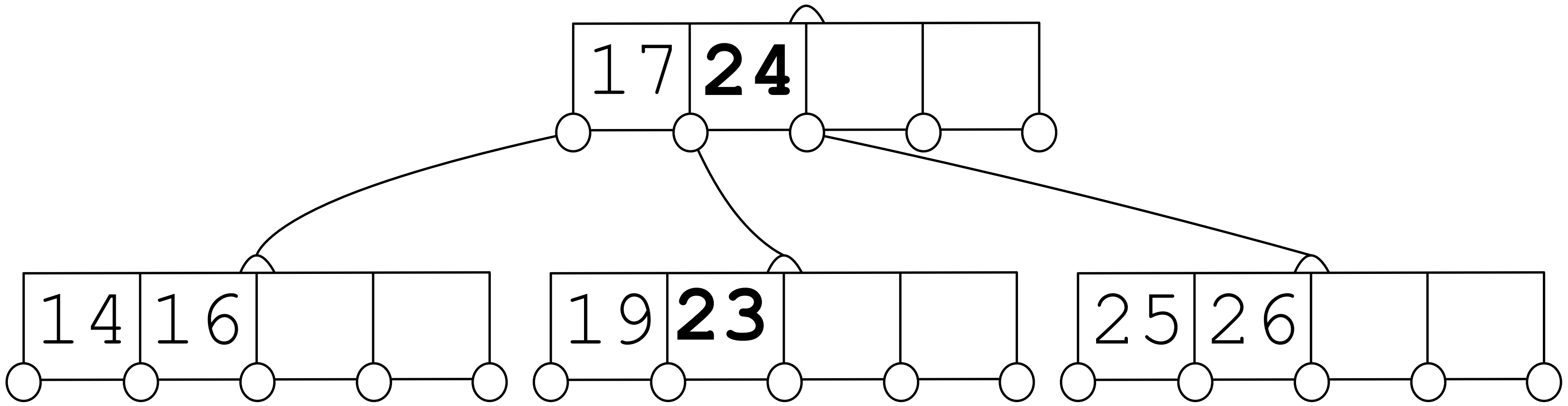


Rearrange keys with parent and lending neighbor to fill vacant slot in under-full node.

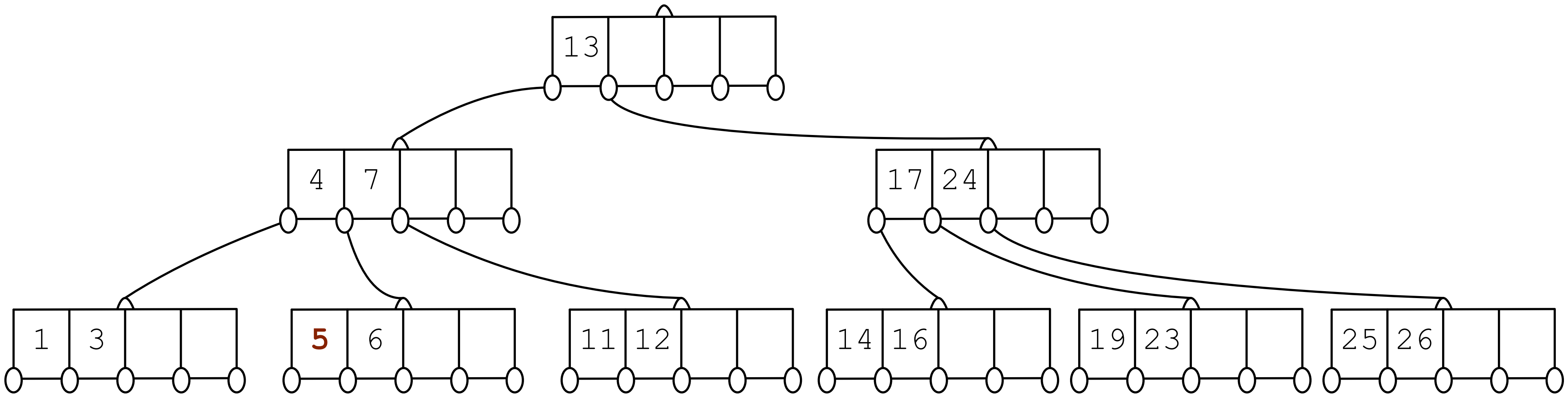
B-Tree Operations: Remove



Closeup! Notice that sort order is preserved, and the keys in all related nodes fill from the left-most slot. Here's the result:



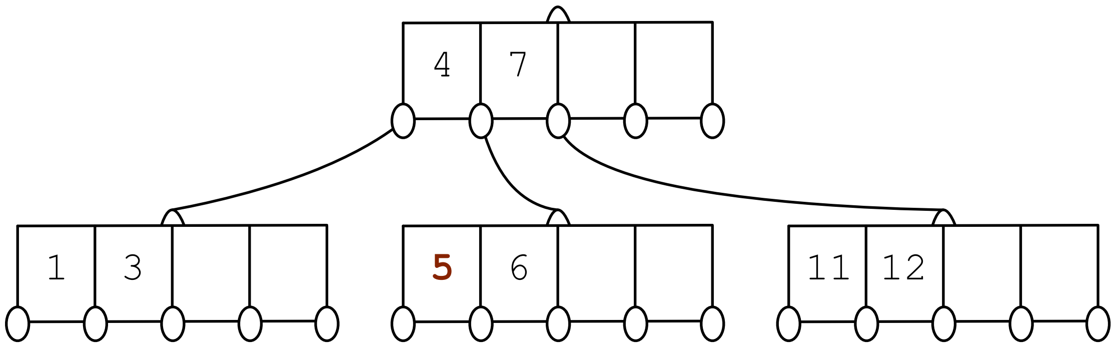
B-Tree Operations: Remove



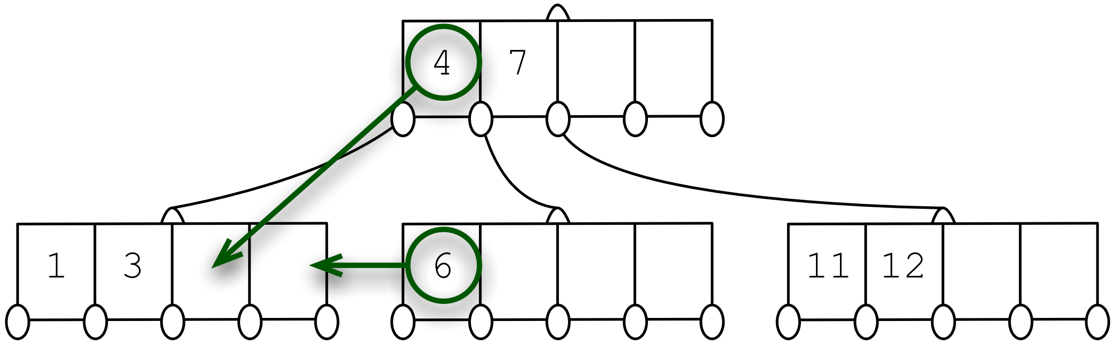
Removing 5 leads to all sorts of interesting issues. Removing it leaves an under-full node, and the siblings don't have any spare keys to lend.

The solution is to *combine* the under-full node with keys from the parent and one of the barely-full neighboring nodes.

B-Tree Operations: Remove

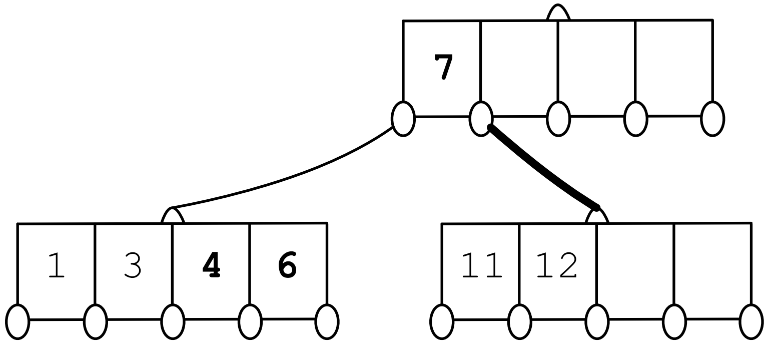


Deleting key 5.



✗ under-full node!

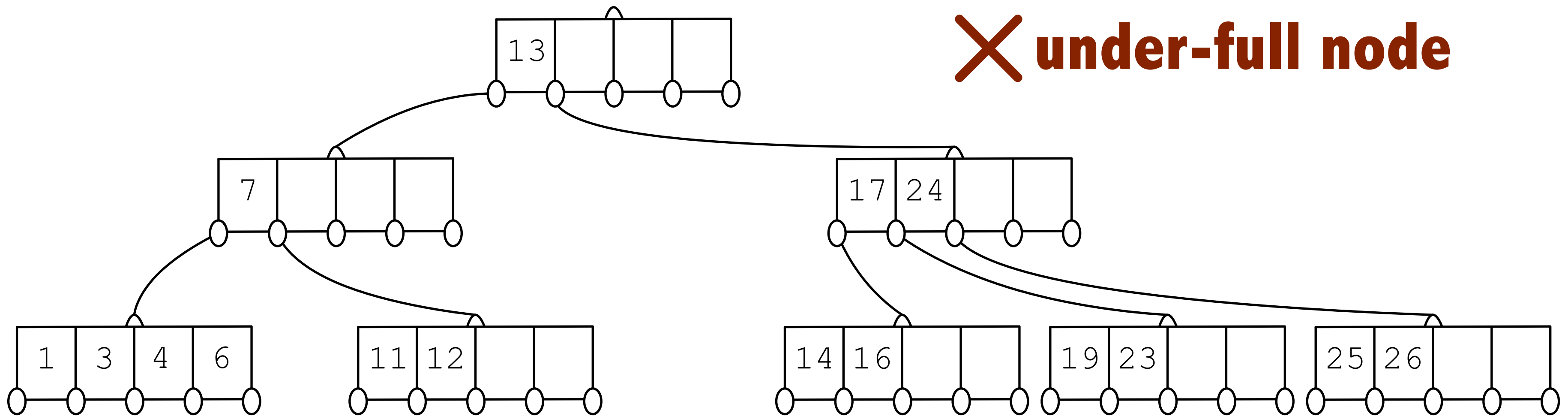
Neighbors can't spare a key. Combine to make a new node.



Combine to form a new node using parent/sibling keys. Update child links to account for combined node.

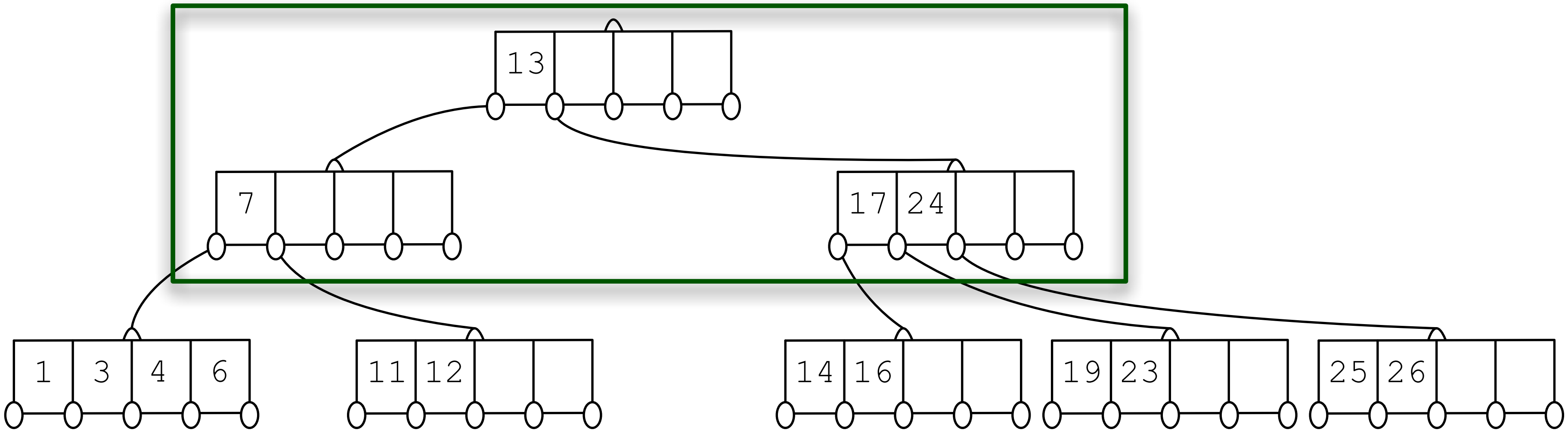
✗ under-full node, again!

B-Tree Operations: Remove

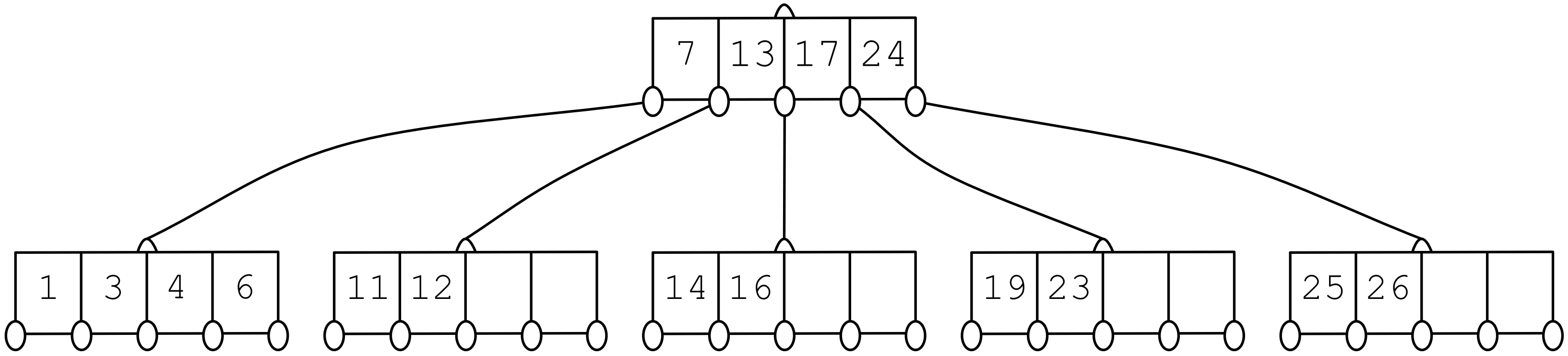


After the previous step of combining nodes, we are left with a broken invariant: The node with key 7 is under-full. The last time we had this situation we borrowed from a neighboring sibling. But the only sibling has no keys to lend. So, break out the last-ditch scenario: reducing the height of the tree by combining keys and removing whole nodes.

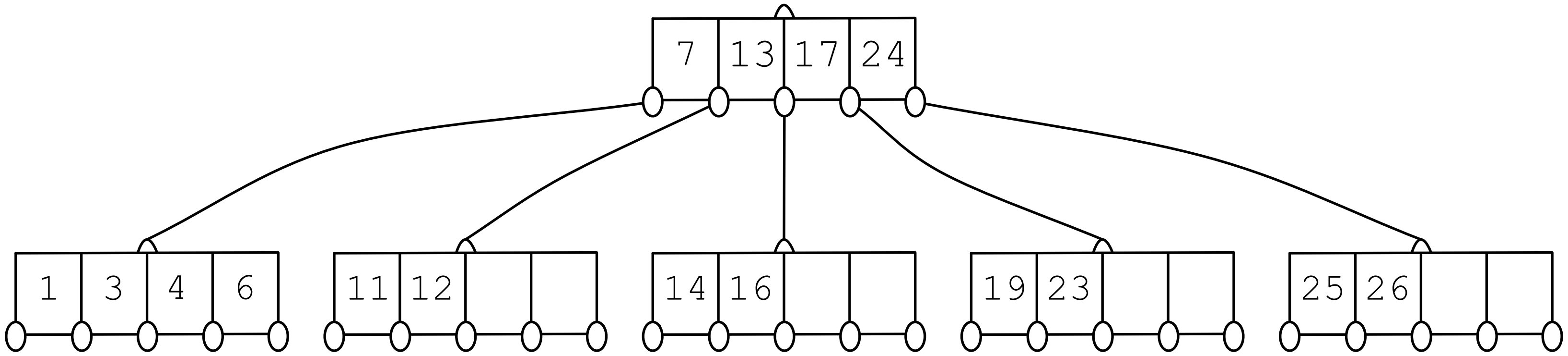
B-Tree Operations: Remove



Combine the top three nodes, since there are a total of 4 keys, this gives us a full node that becomes the new root of the B-Tree.



B-Tree Operations: Query



Recursive Python code for querying.

```
def query(btree, val):
    if btree.contains_key(btree.keys, val):
        return True
    else:
        subtree = find_subtree(btree, val)
        return subtree.query(subtree, val)
```

Notice that it uses helper functions:

contains_key(keys, val)
find_subtree(btree, val)

B-Tree Operations: Query

Those helper functions could be like this:

```
def contains(keys, val):
    for k in keys:
        if k == val:
            return True
    return False

def find_subtree_for(btree, val):
    if btree.is_leaf:
        return None
    idx = 0
    for key in btree.keys:
        if key > val:
            idx = idx + 1
        else:
            break
    return btree.children[idx]
```

B-Tree Design Strategy

As I have said approximately 400 times, the goal of this homework assignment is to give you a very hard task that forces you to think critically about what you are being asked to do, as well as about how you are going to go about solving it. This should involve the steps enumerated on the next page.

B-Tree Design Strategy

1) Understand the invariants.
2) Draw diagrams of B-Tree configurations.
3) Identify tree configurations that must be treated differently.
4) Write (and use!) test code. I won't give you any of this.
5) Decompose high-level tasks (insert) into a series of possibly reusable steps (split, combine).
6) Write your own function invariants for steps like splitting. What does it return? What states can it leave the tree?
7) Fail fast, fail early, and *characterize your mistakes*.