A LaTeX'ed solution to B-tree insertion and deletion.

Mitchell Scott (mtscot4)

September 27, 2024

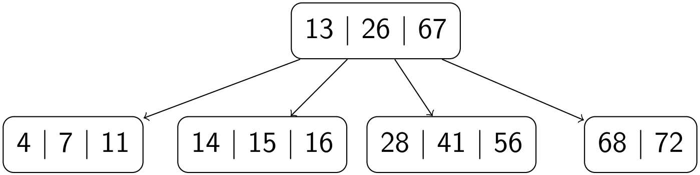
B-Tree Insertion and Deletion Assignments

Assignment 3-1: Insert

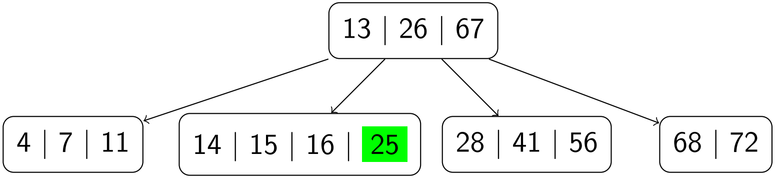
Assignment 3-1: Deletion

Table of Contents

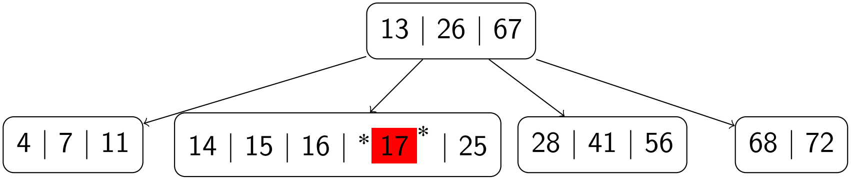
For $m = 2$, and the below B-tree, insert the keys 25, 17, 42 and 29, in this order.



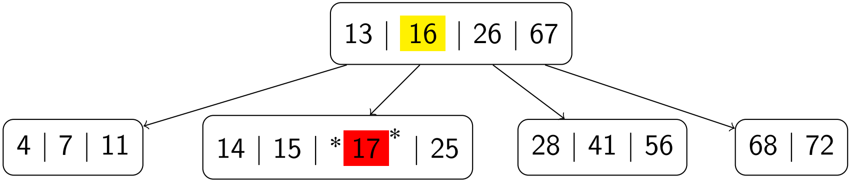
25 is inserted into currentNode, currentNode is checked for overflow. Since $4 \leqslant 2m = 4$, there is no overflow.



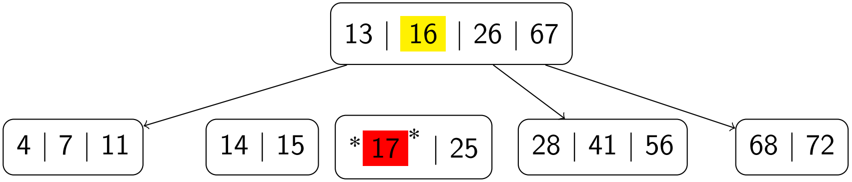
17 is inserted into currentNode, currentNode is checked for overflow. $5 > 2m = 4$, so overflow, as it violates the B-tree property.



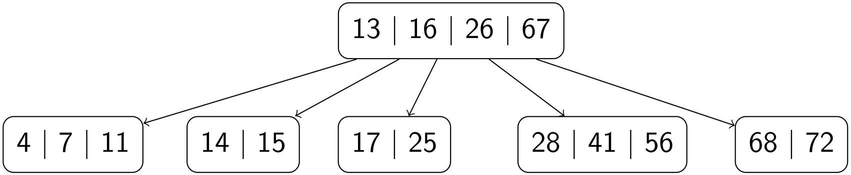
16 is found to be the median of the keys of currentNode, so we move 13 from currentNode to Parent.



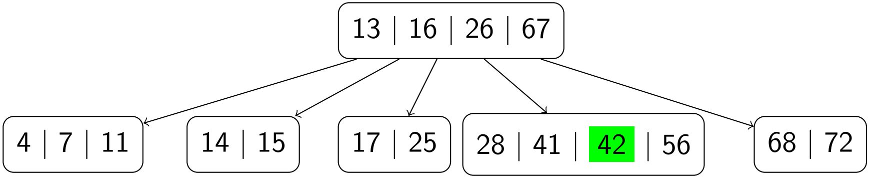
Now, `parent` has the wrong number of pointers, as four keys should have five pointers. We split ‘`currrentNode`’ into ‘`leftNode`’ and ‘`rightNode`’ .



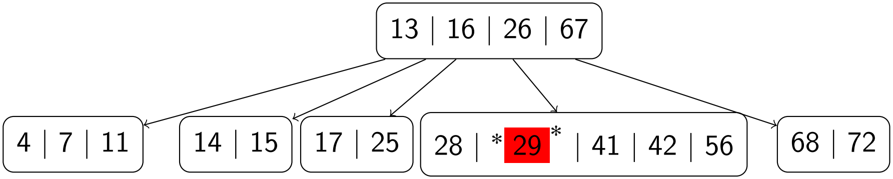
We add the left and right pointer to `leftNode` and `rightNode`, respectively. Lastly, since the root has grown, we check for overflow. Since $4 \leqslant 2m = 4$ keys, there is no overflow. We have finally inserted 17.



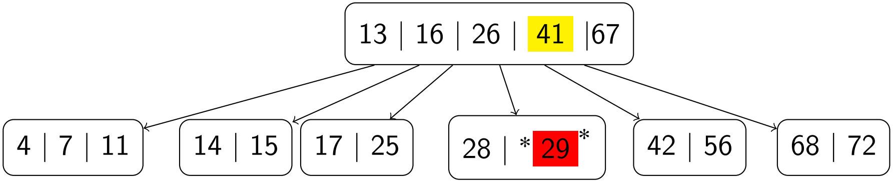
Now we seek to insert 42. Since the new `currentNode` has $4 \leqslant 2m = 4$ keys, we don't have overflow, so we are done.



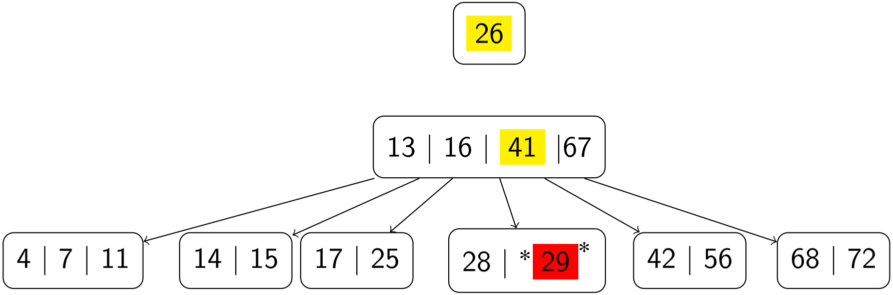
Lastly, we seek to add 29. We insert into `currentNode`, but observe that `currentNode` is overflowing as it has $5 > 2m = 4$ keys.



41 is the median of `currentNode` so we move it to `parent`. The rest of the node is split as we need another pointer coming from `parent`.



Lastly, we have overflow at the root, so we follow the same precedure, where we find the median, and move to `parent`, or becoming the new root in this case.



We split the remaining node, and they are now children of `newRoot`. A keen observer, such as RANDOM CAT, might suspect underflow at the new root, but the root doesn't have a minimum number of keys, so we are done!

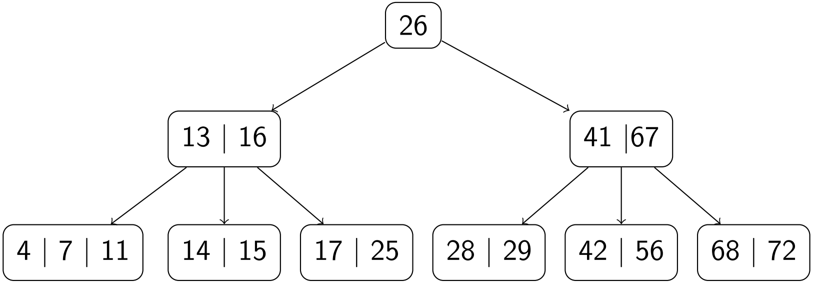
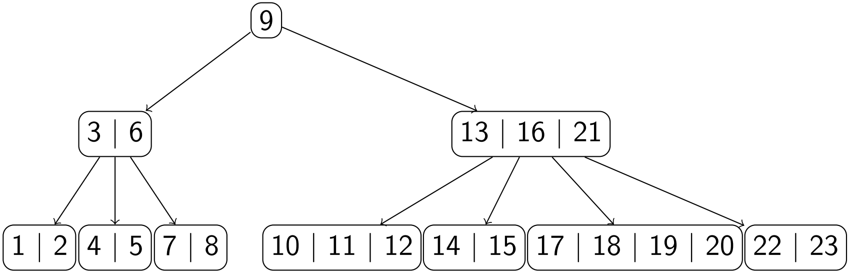
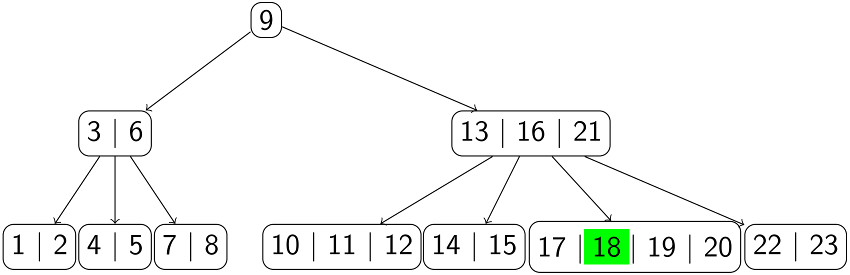


Table of Contents

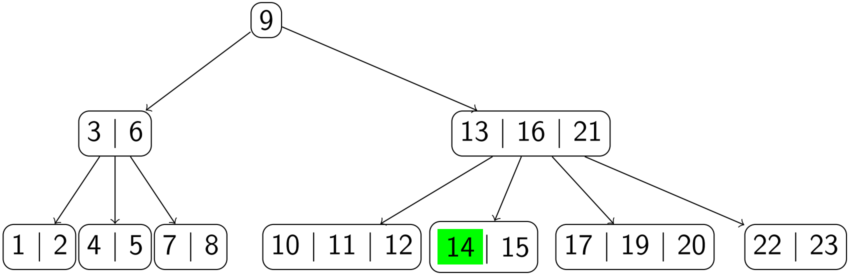
For $m = 2$, and the below B-tree, delete the keys 18,14,21, and 1, in this order.



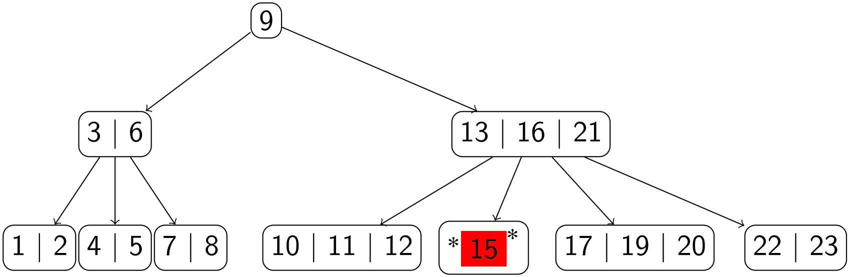
Since 18 is a leaf node, it is deleted at `currentNode`. Then we check `currentNode` for underflow. Since $3 > m = 2$, there is no underflow, so no additional work to delete 18.



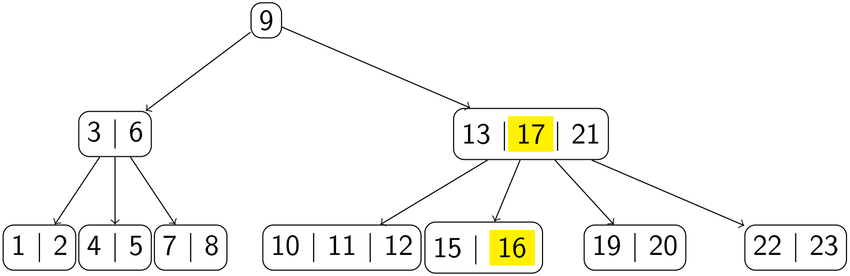
14 is deleted at `currentNode`, so we check `currentNode` for underflow.



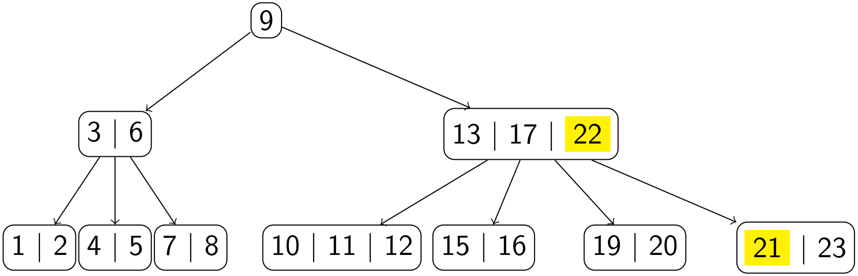
`currentNode` is a leaf node that is underflowing as $1 < m = 2$ key. We observe that `rightSibling` has $3 \geqslant m = 2$ nodes, so we borrow from it.



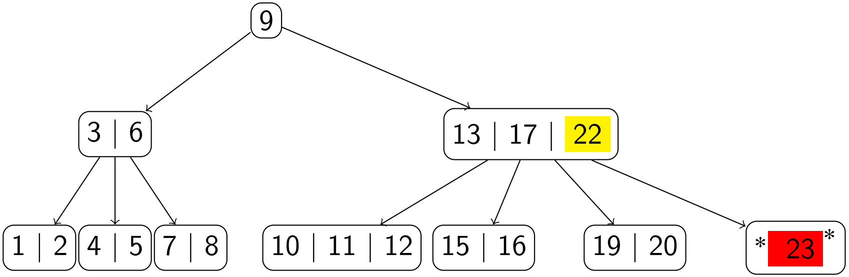
We perform a rotation to borrow from `rightSibling`. 17 goes from `rightSibling` to `parent`, which allows 16 to go from `parent` to `currentNode`. Done.



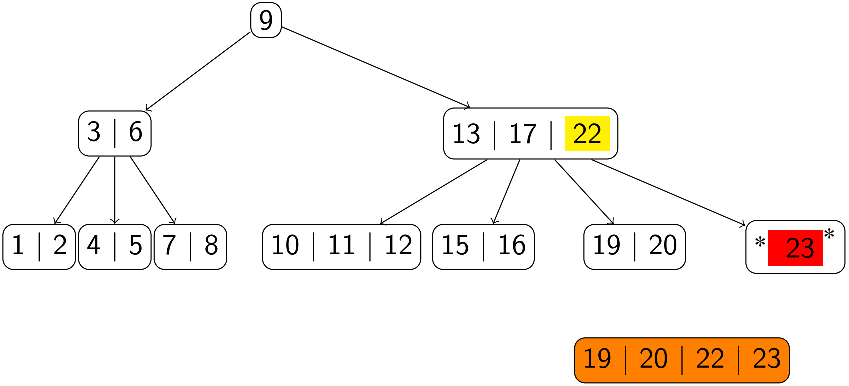
To delete 21, which is a branch node, we must switch it with the next largest key, 22, which is guarenteed to be a leaf node.



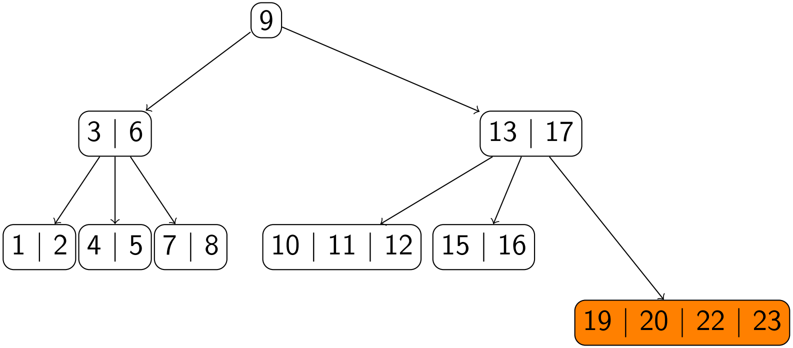
We now delete key 21, which causes an underflow in `currentNode`. Since `currentNode` has no right sibling, we turn to `leftSibling` to attempt to balance.



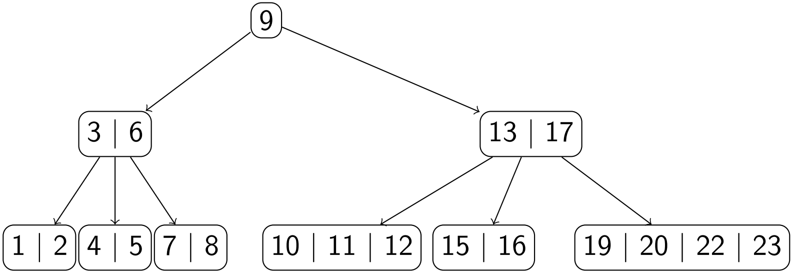
Observing that `leftSibling` has exactly $m = 2$ keys, we cannot rotate, so we construct a `newNode` with exactly $2m = 4$ keys by collapsing `currentNode` and `leftSibling`.



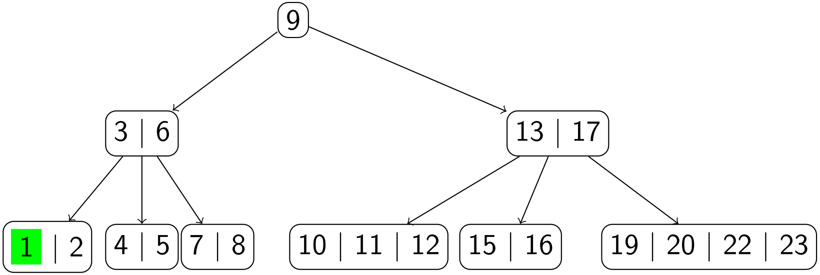
We then delete `currentNode`, delete `leftSibling` and delete the separater from `parent`. Then we link the gap in `parent` to `newNode`.



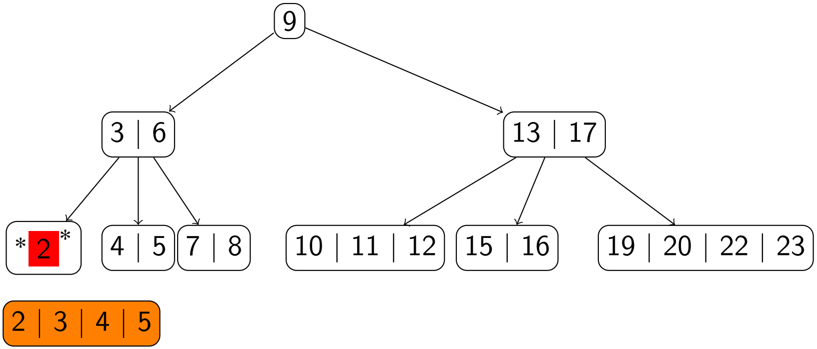
Lastly, we check for underflow at `parent`, but since $2 \geqslant m = 2$, there is no underflow, and we are done.



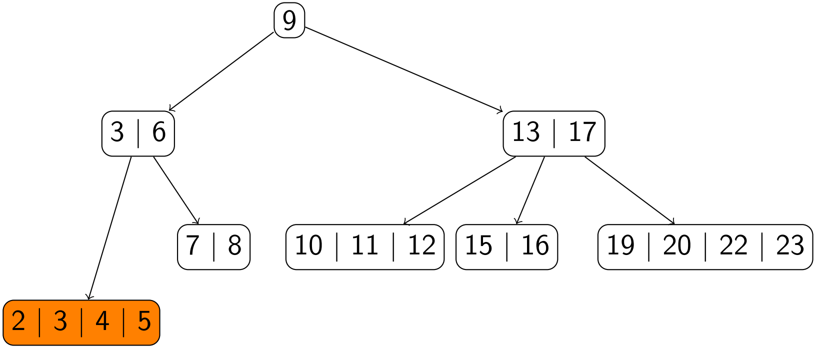
Next we delete key 1, which we can do as it is a leaf node. This leaves underflow at `currentNode`. We look to `rightSibling` and see that it doesn't have more than $m = 2$ keys.



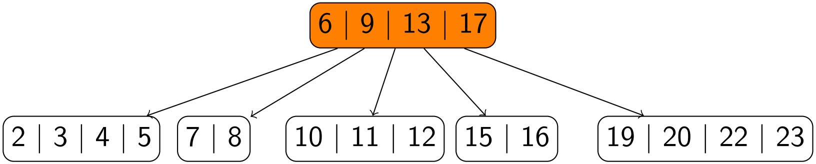
Since a rotation is not possible, we collapse `currentNode` and `rightSibling`.



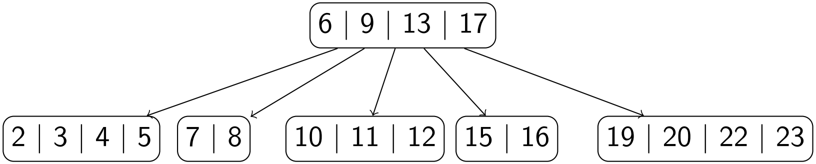
When we are done linking the gap from `newNode` to `parent`, we then check overflow for `parent`, which it unfortunately is underflowing.



Since we are underflowing at `parent`, and `rightSibling` cannot help, we must collapse `parent` and `rightSibling`, to finally arrive at the finished B-tree.



Lastly, we perform a sanity check to see that every node has between 2 and $2m = 4$ keys, and the `parent` has 4 nodes and 5 children, so we are done.


6 | 9 | 13 | 17
2 | 3 | 4 | 5
7 | 8
10 | 11 | 12
15 | 16
19 | 20 | 22 | 23