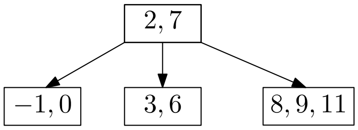
B-Trees and 2-3-4 Trees

Reading:

- "B-Trees" 18 CLRS
- "Multi-Way Search Trees" 3.3.1 and "(2,4) Trees" 3.3.2 GT

B-trees are an extension of binary search trees:

- They store more than one key at a node to divide the range of its subtree's keys into more than two subranges.
- Every internal node has between t and $2t$ children.
- Every internal node has one more child than key.
- All of the leaves must be at same level.



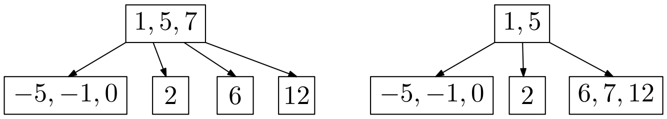
We will see shortly that the last two properties keep the tree balanced.

Most binary search tree algorithms can easily be converted to B-trees. The amount of work done at each node increases with t (e.g. determining which branch to follow when searching for a key), but the height of the tree decreases with t, so less nodes are visited. Databases use high values of t to minimize I/O overhead: Reading each tree node requires a slow random disk access, so high branching factors reduce the number of such accesses. We will assume a low branching factor of $t = 2$ because our focus is on balanced data structures. But the ideas we present apply to higher branching factors. Such B-trees are often called 2-3-4 trees because their branching factor is always 2, 3, or 4.

To guarantee a branching factor of 2 to 4, each internal node must store 1 to 3 keys. As with binary trees, we assume that the data associated with the key is stored with the key in the node. We assume that every 2-3-4 tree node N has the following fields.

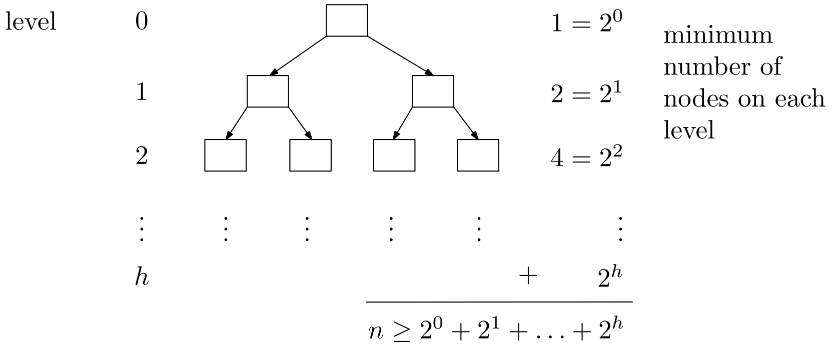
- The number $N.n$ of keys stored at this node.
- The keys $N.\text{key}[1 \ldots n]$ sorted so that $N.\text{key}[i] < N.\text{key}[j]$, for $i < j$.
- The indicator $N.\text{leaf}$ which is set to True if and only if N is a leaf node.
- The references $N.\text{child}[1 \ldots (N.n + 1)]$ pointing to each of N's children.

The example on the right shows that different 2-3-4 trees may hold the same keys.



Lemma. *Let n denote the total number of keys in a 2-3-4 tree. If $n \geq 1$, the height is $\leq \log_2 n$.*

Proof. In the picture below, all of the leaves are on level h. Each internal node has at least 2 children, which lower-bounds the number of nodes in every tree of height h.


level
0
1
2
⋮
h
$1 = 2^0$
$2 = 2^1$
$4 = 2^2$
⋮
$+ \quad 2^h$
$n \geq 2^0 + 2^1 + \ldots + 2^h$
minimum number of nodes on each level

So n is bounded below by a geometric series. Using our formula for geometric series:

$$\begin{aligned} n &\geq \frac{2^{h+1}-1}{2-1} = 2^{h+1}-1 \\ n+1 &\geq 2^{h+1} = 2 \cdot 2^h \\ n+n &\geq 2 \cdot 2^h \\ n &\geq 2^h \\ \log_2 n &\geq h \end{aligned}$$

□

The algorithm for locating a key in a 2-3-4 tree is similar to the algorithm for a binary tree. It differs from the binary tree case in that we have up to 4 subtrees to choose from when branching.

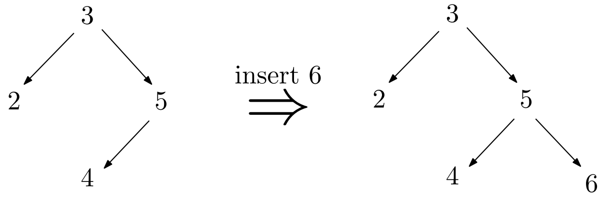
```
Algorithm 2-3-4-Find($T$,$k$)
   if ($T$ = nil) then
      return nil

   for $i \leftarrow 1$ to $T.n$ do
      if ($k < T$.key[$i$]) then
         return 2-3-4-Find($T$.child[$i$],$k$)
      if ($k = T$.key[$i$]) then
         return ($T$,$i$)

   return 2-3-4-Find($T$.child[$T.n+1$],$k$)
```

Run-time analysis: We do $\Theta(1)$ work per level to find the appropriate subtree (or key). If we recurse, we drop a level in the tree. There are $O(\log n)$ levels. So we do at most $\Theta(1) \times O(\log n) = O(\log n)$ work.

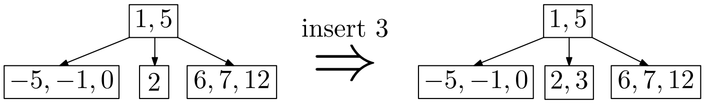
Exercise: Write `2-3-4-Max`

Adding a Node.

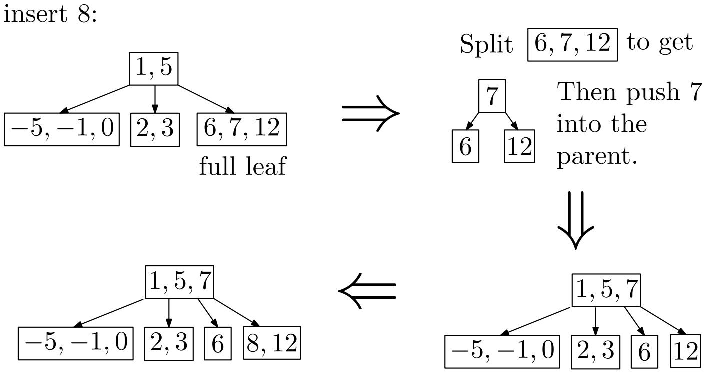
With an unbalanced binary search tree, adding a node was easy: We searched for the key in the tree until we fell off; then the last node that we visited became the parent of a leaf node storing the key.



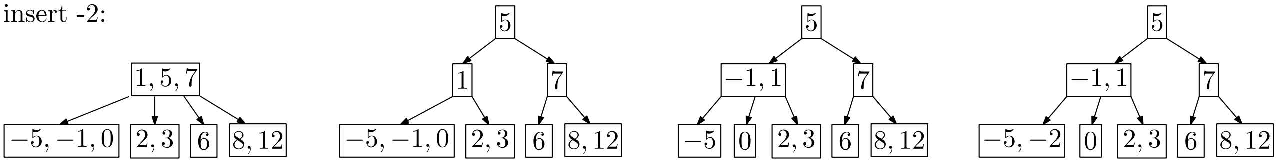
But we can not do this with a 2-3-4 tree because all of the leaves must be on the last level. However, unlike binary trees, 2-3-4 leaf nodes can contain more than one key. So we may still be able to insert a key at a leaf node.



What if the leaf node to which we want to add is full? Then we split the leaf node and push the split key up a level.



What if the parent does not have room for the split node? Then we must split the parent before the child.



initial tree

split the root

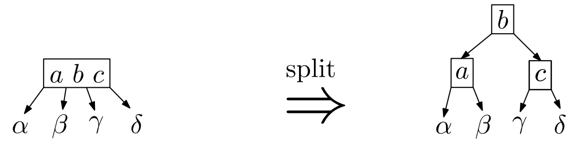
split the leaf

insert the key

Notice that splitting the root increases the height of the 2-3-4 tree. This is the only way that a 2-3-4 tree's height increases. The height of a 2-3-4 tree grows by adding a new root, whereas the height of a binary search tree grows by adding new leaves.

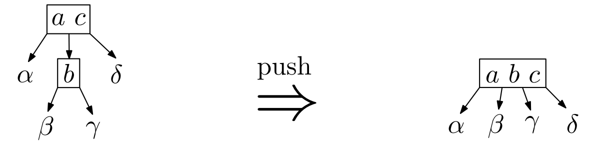
When adding a key to a 2-3-4 tree, we traverse from the root to the leaf where we insert the key. At each node that we visit, we split the node if it has three keys. This guarantees we can always push a key into a node's parent because we visited its parent first. In particular, we will always be able to split a leaf to make room for a new key.

Is splitting correct? We just have to check that it is order preserving.



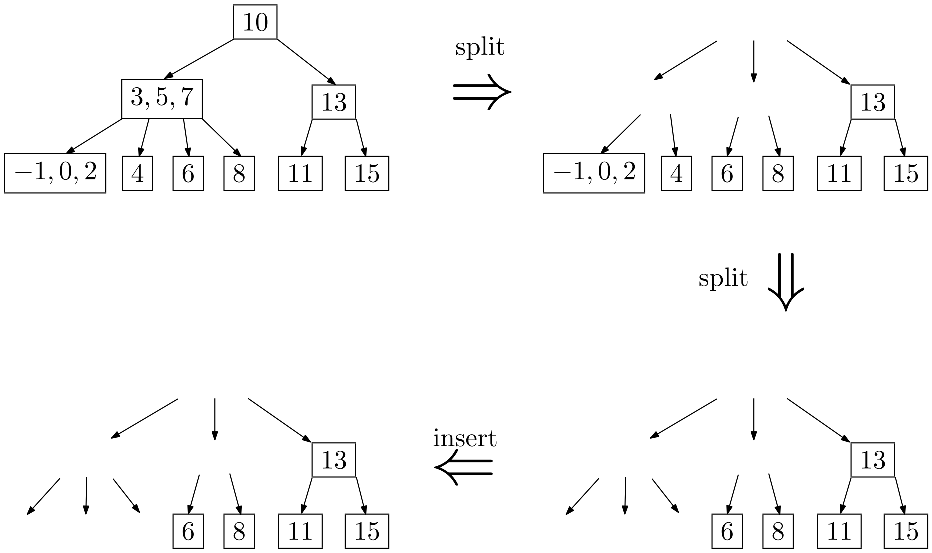
$\alpha < a < \beta < b < \gamma < c < \delta$ $\quad$ $\boxed{\alpha < a < \beta} < b < \boxed{\gamma < c < \delta}$

Is pushing correct? Again, check that it is order preserving.

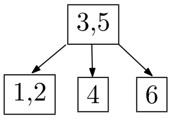


$\alpha < a < \boxed{\beta < b < \gamma} < c < \delta$ $\quad$ $\alpha < a < \beta < b < \gamma < c < \delta$

Practice: Insert -2 into the 2-3-4 tree below.



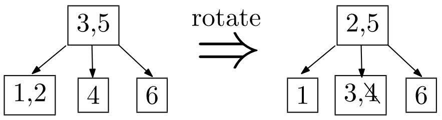
Deleting keys: In binary search trees, deleting a leaf was the easiest case: We just removed the leaf. This is the easiest case for 2-3-4 trees, as well, so we consider it first. Just as we had to be sure that there was room for a new key when inserting into a leaf, we must make sure that a leaf is not empty after deleting a key.



For example, it is easy to delete 1: just change $\boxed{1,2}$ to $\boxed{2}$. But deleting 4 is complicated because we forbid empty leaves.

When key deletion would create an empty leaf, we look at the leaf's immediate siblings (i.e. to the left and then right) and try to borrow a key. In this example, we look at $\boxed{4}$'s siblings $\boxed{1,2}$ and $\boxed{6}$. Clearly, $\boxed{1,2}$ is the only candidate lender. How do we borrow a key? We cannot just transfer a key directly because that would violate the ordering property of the nodes relative to their parents. So we pass it through the common parent.

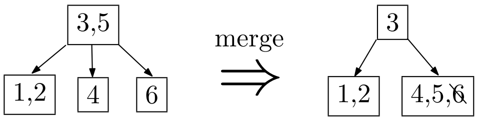
delete 4:



The restructuring caused by borrowing is equivalent to the rotation used by AVL and red-black trees. After borrowing from $\boxed{1,2}$, we can safely delete 4.

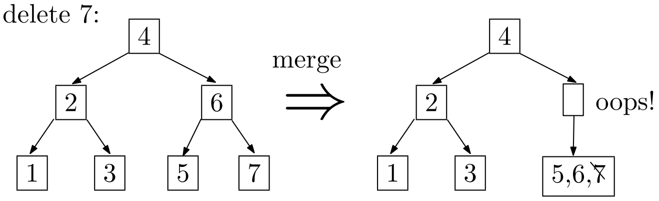
What about deleting 6 from our first example? In this case, there is no immediate sibling from which to borrow a key. So we steal a key from our parent by doing the opposite of a split: We merge. Just as we tried to borrow from the left first, we try to merge on the left first.

delete 6:



Merging has a pitfall similar to splitting: When splitting, we needed room in the parent to push a key up; when merging, we need a key from the parent to pull down.

delete 7:

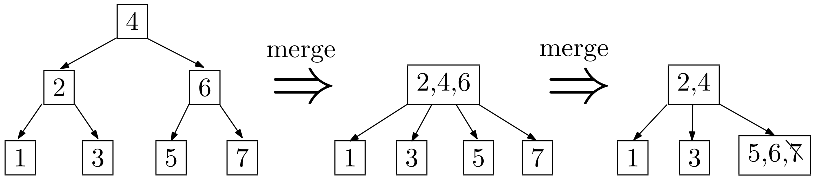


How do we make sure that we have a key to pull down? While adding, we traversed the tree maintained the invariant that the parent of our current node had at *most* two keys. Now we maintain the invariant that the parent has at *least* two keys.

When we were adding, we split every node with three keys that we visited. Now that we are deleting, we ensure that every node that we visit has at least two keys. But we have a strong preference: first we attempt to borrow from a sibling (left and then right), and if that fails, we steal from a parent.

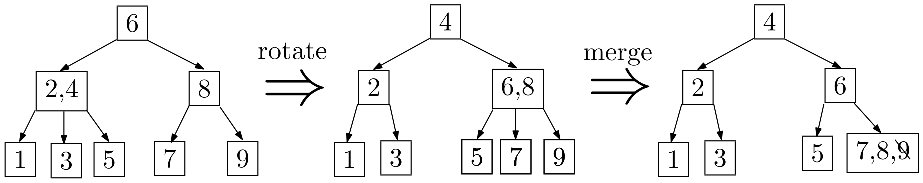
Why do we always try to borrow first? So that we can merge, if needed: If we cannot rotate, then both siblings have one key and we can merge with either of them.

delete 7:



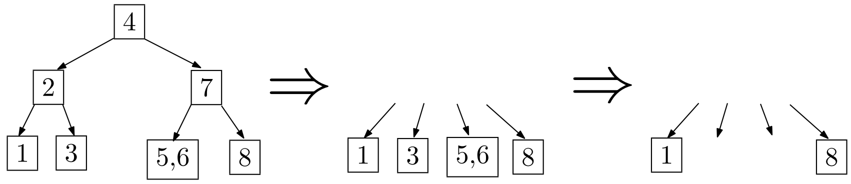
So 2-3-4 trees shrink from the root as well as grow from the root!

delete 9:

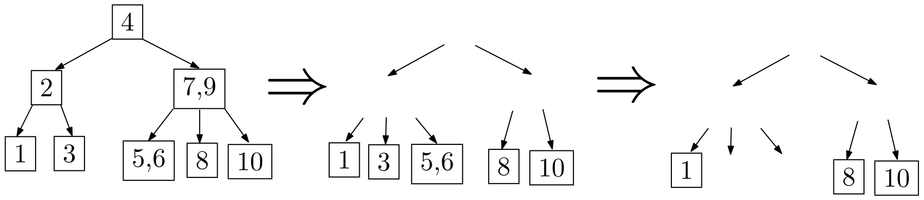


Practice:

delete 3:



delete 3:



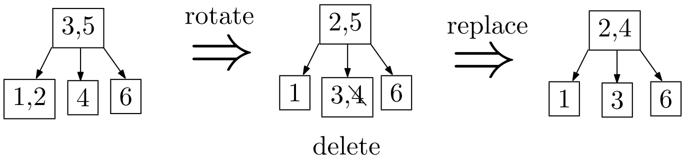
We now know how to delete keys from leaf nodes: We traverse the tree maintaining the invariant that each node that we visit has at least two keys. We do this by borrowing (rotating), and when we can't borrow, we steal (merge).

What about deleting keys from internal nodes. Recall how we did this with binary search trees:

- If the node has zero children, delete it.
- If the node has one child, splice it out.
- If the node has two children, we remove its predecessor and swap keys.

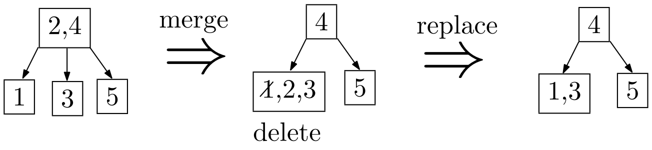
The first two strategies do not apply to internal nodes of a B-tree. What about the third strategy?

delete 5: first remove predecessor 4, and then swap 4 for 5



Let's try it again. Notice that the key that we wish to delete may move when we delete its predecessor.

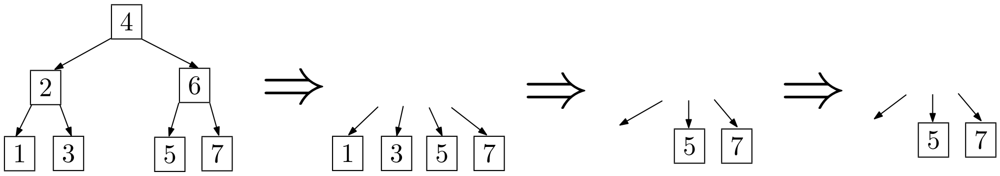
delete 2: first remove predecessor 1, and then swap 1 for 2



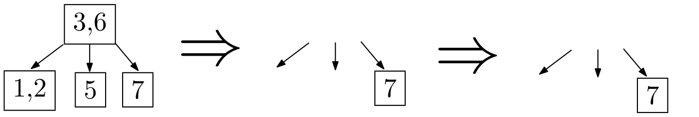
In both examples, the strategy of removing the predecessor and then swapping worked because the predecessor was in a leaf node, which we know how to remove. Is this true in general? Yes because every key in the internal node of a B-tree has a left subtree. The predecessor is the maximum in this subtree, which is the leaf on the rightmost path.

Practice:

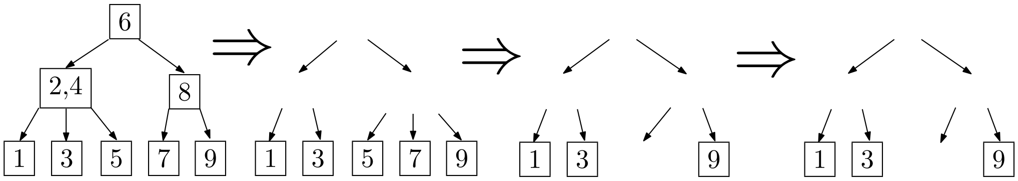
delete 4:



delete 6:



delete 8:



Deletion Algorithm: Find the key to delete, and

- if it is in a leaf, delete it,
- otherwise, find the predecessor, delete the predecessor, find the key to delete again (it may have moved), and swap keys.

This is simpler than the routine in CLRS, but a constant factor slower.

IMPORTANT: Use this routine! And show intermediate steps to guarantee partial credit.