ISAM – Indexed Sequential Access Method

ISAM is a static index structure – effective when the file is not frequently updated. Not suitable for files that grow and shrink. When an ISAM file is created, index nodes are fixed, and their pointers do not change during inserts and deletes that occur later (only content of leaf nodes change afterwards). As a consequence of this, if inserts to some leaf node exceed the node's capacity, new records are stored in overflow chains. If there are many more inserts than deletions from a table, these overflow chains can gradually become very large, and this affects the time required for retrieval of a record.

B+ tree – a dynamic structure that adjusts to changes in the file gracefully. It is the the most widely used structure because it adjusts well to changes and supports both equality and range queries.

It is a balanced tree in which the internal nodes direct the search and the leaf nodes contain the data entries. The leaf nodes are organized into a doubly linked list allowing us to easily traverse the leaf pages in either direction.

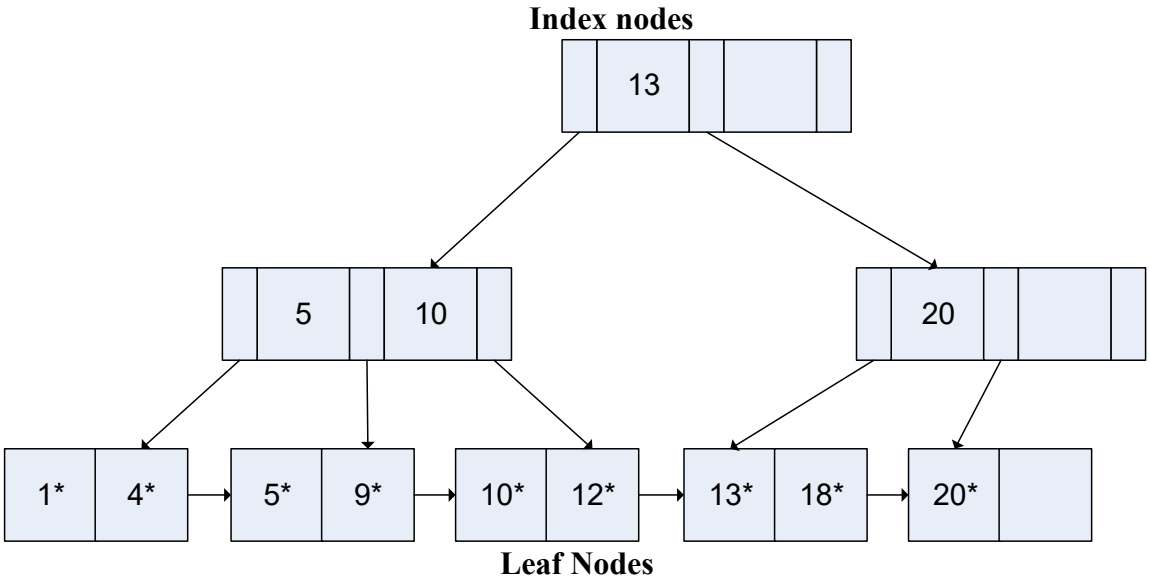
Main characteristics of a B+ tree:

- Operations (insert, delete) on the tree keep it balanced. $\text{Log}_f N$ cost where f=fanout, N = # of leaf pages.
- Minimum occupancy of 50% is guaranteed for each node except the root node if the deletion algorithm we will present is used. (in practice, deletes just delete the data entry because files usually grow, not shrink). Each node contains m entries where $d <= m <= 2d$ entries. d is referred to as the **order** of the tree.
- Search for a record is just a traversal from the root to the appropriate leaf. This is the height of the tree – because it is balanced is consistent. Because of the high fan-out, the height of a B+ tree is rarely more than 3 or 4.

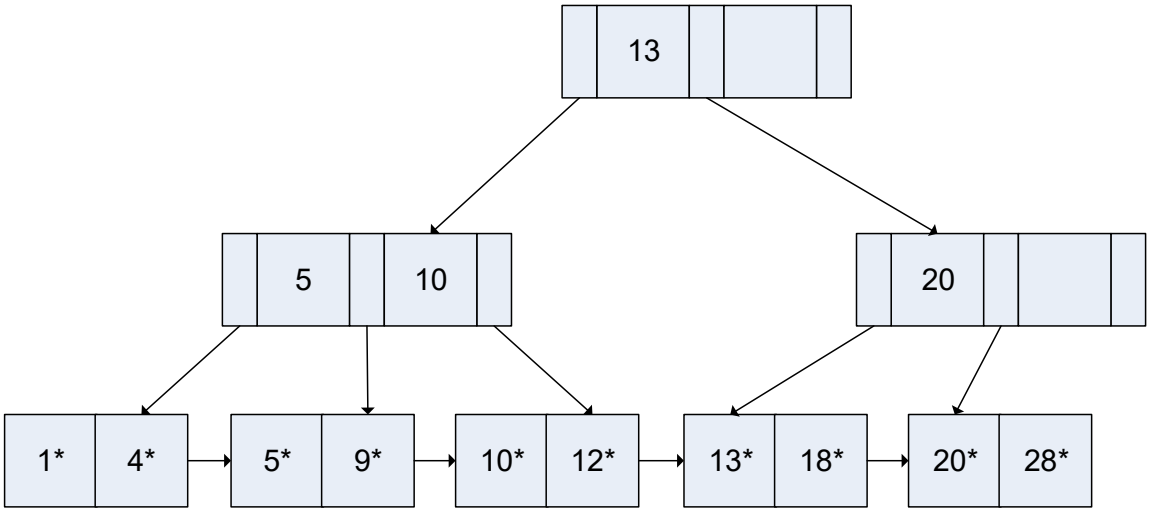
The `insert` algorithm for B+ Trees:

Leaf page full?	Index page full?	Action
No	No	Place the record in sorted position in the appropriate leaf page
Yes	No	1. Split the leaf page 2. Place Middle Key in the index page in sorted order. 3. Left leaf page contains records with keys below the middle key. 4. Right leaf page contains records with keys equal to or greater than the middle key.
Yes	Yes	1. Split the leaf page. 2. Records with keys < middle key go to the left leaf page. 3. Records with keys >= middle key go to the right leaf page. 4. Split the index page. 5. Keys < middle key go to the left index page. 6. Keys > middle key go to the right index page. 7. The middle key goes to the next (higher level) index. IF the next level index page is full, continue splitting the index pages.

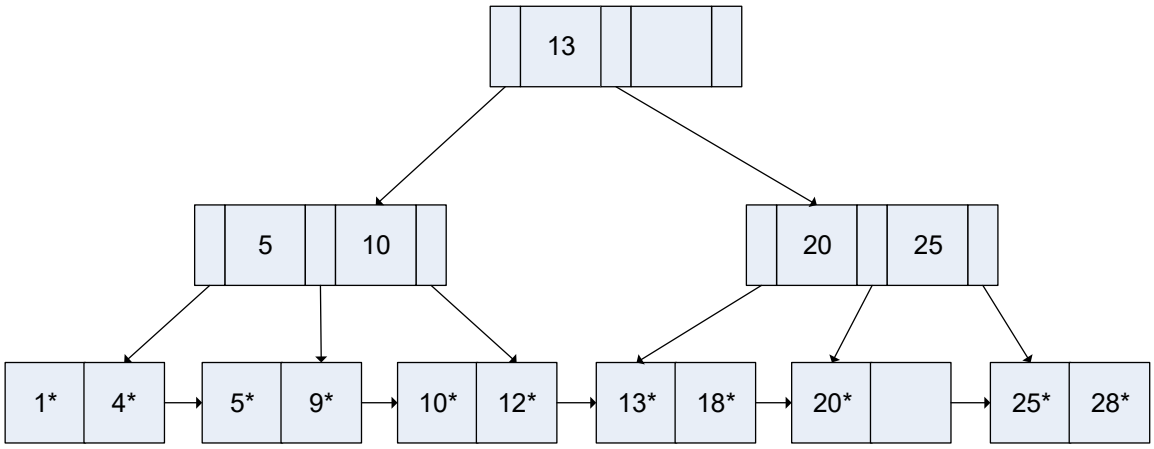
Examples of insertion with B+ tree with order = 1. Starting with a tree looking like this:



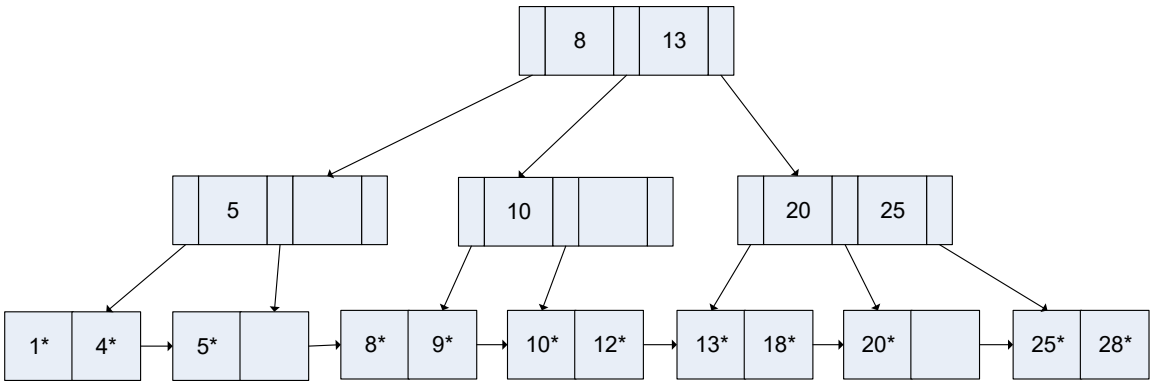
Our first insertion has an index of 28. We look at the leaf node to see if there is room. Finding an empty slot, we place the index in node in sorted order.



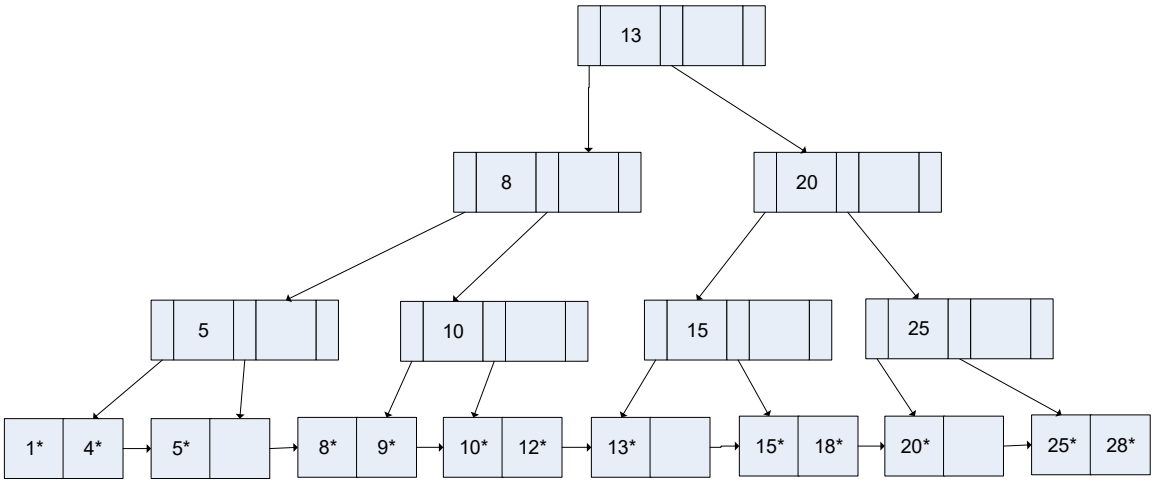
Our next insertion is at 25. We look at the leaf node it would go in and find there is no room. We split the node, and roll the middle value to the index mode above it.



Our next case occurs when we want to add 8. The leaf node is full, so we split it and attempt to roll the index to the index node. It is full, so we must split it as well.



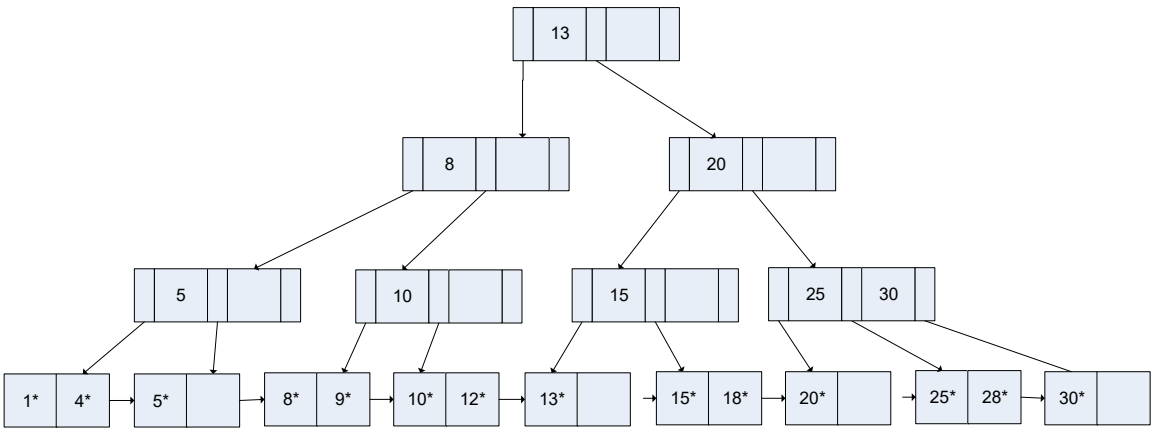
Our last case occurs when we want to add 15. This is going to result in the root node being split. The leaf node is full, as are the two index nodes above it. This gives us:



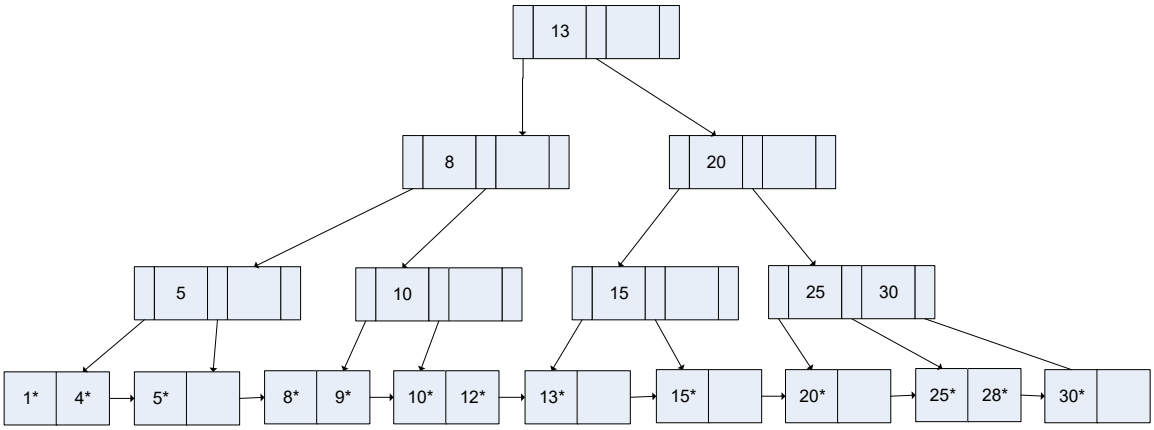
The delete algorithm:

No	No	Delete the record from the leaf page. Arrange keys in ascending order to fill void. If the key of the deleted record appears in the index page, use the next key to replace it.
Yes	No	Combine the leaf page and its sibling. Change the index page to reflect the change.
Yes	Yes	1. Combine the leaf page and its sibling. 2. Adjust the index page to reflect the change. 3. Combine the index page with its sibling. Continue combining index pages until you reach a page with thecorrect fill factor or you reach the root page.

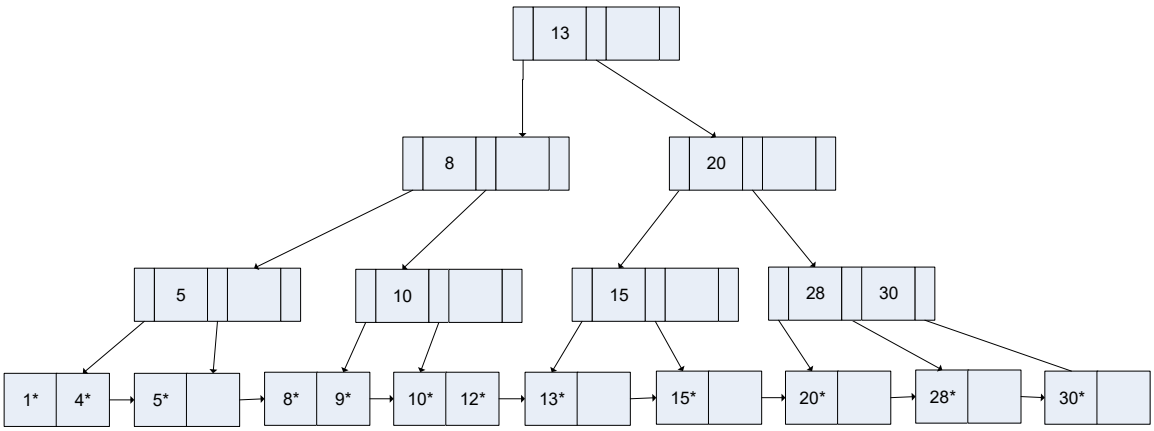
Let's take our tree from the insert example with a minor modification (we have added 30 to give us an index node with 2 indexes in it:



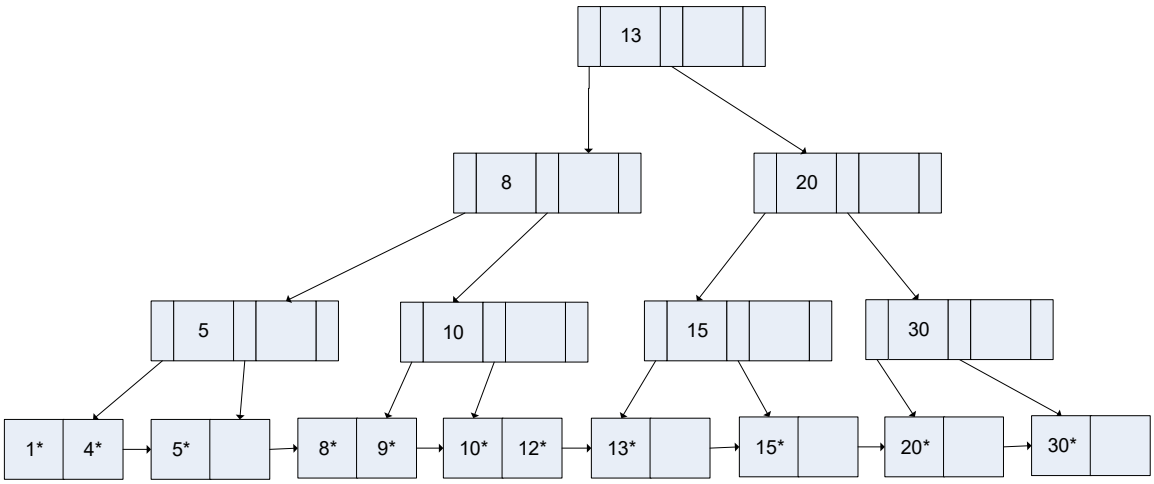
Our first delete is of 18. Simplest case is that it is not an index and in a leaf node that deleting it will not take you below d.



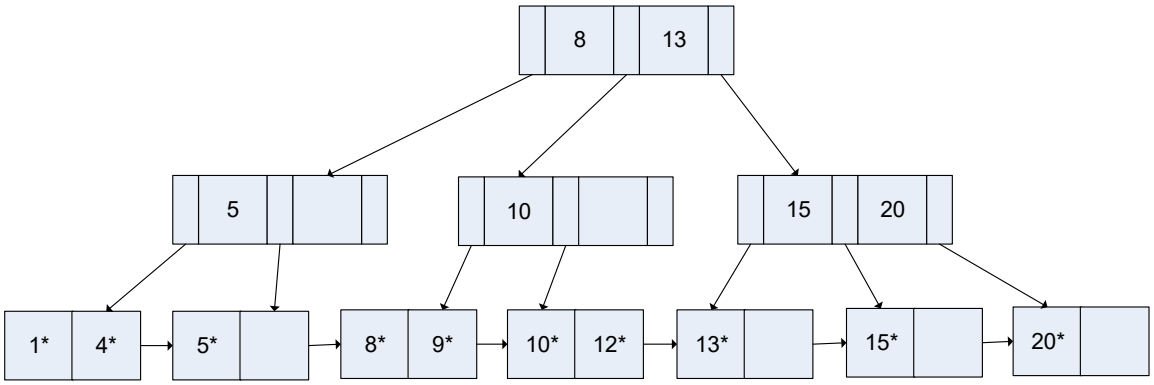
Our next delete is similar, except the index appears in a index node. In that case, the next index replaces the one in the index node. Let's delete 25.



Our next case takes the node below d. Let's delete 28. For this one we combine the leaf page (in our case it is empty) with its sibling and update the index appropriately. That gives us:

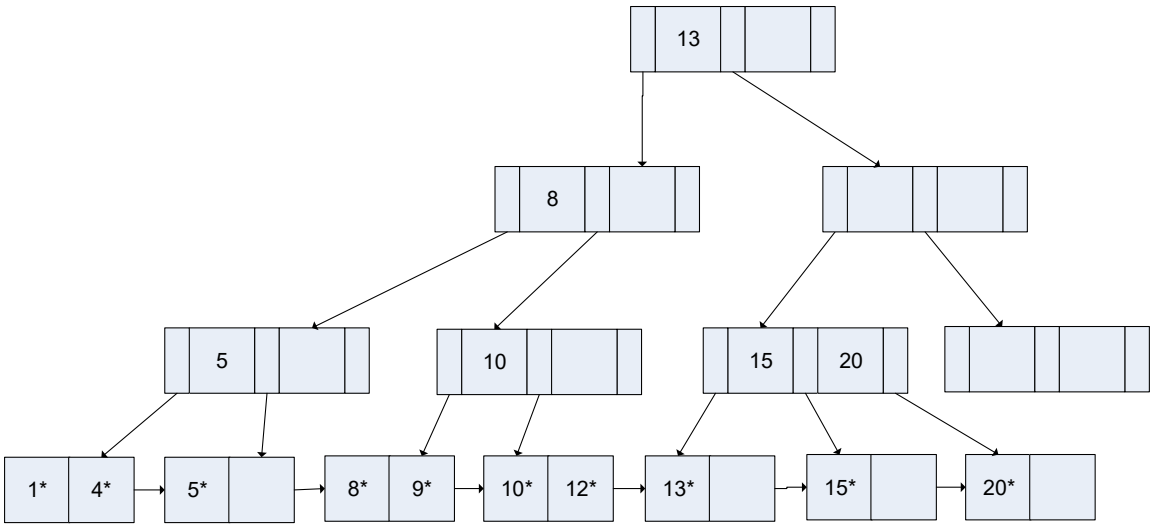


Next we delete 30. This takes us below d for the index. We combine the indexes, which has the effect of taking the index above below d. This continues to the root.

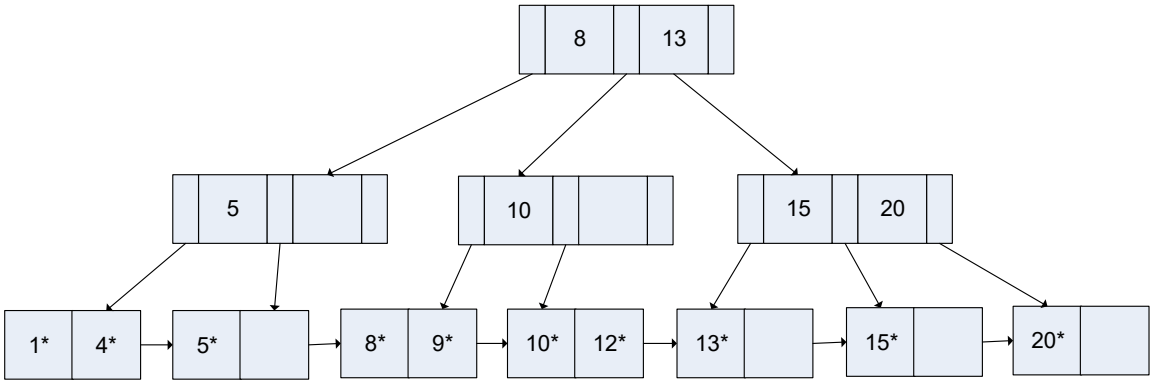


Woah. That seemed like magic. What process got us to that?

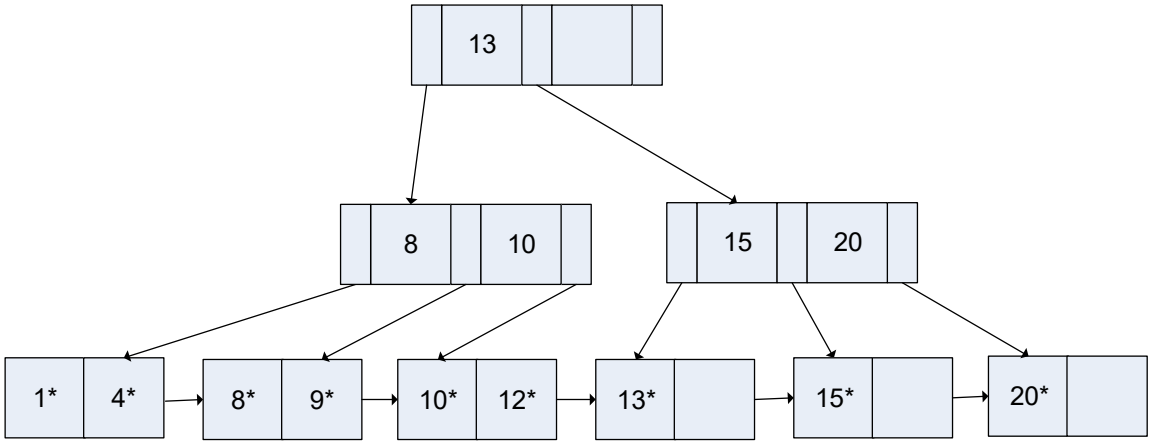
Ok – let's go through it. When we deleted 30, which took the data entry node that 30 was in below d. Now we have to merge with the sibling. When we merge – it's to the sibling on the left, which means pointer in the index above is no longer valid. We remove it, (which leaves it less than d), pull down the index from above and merge the index node with its sibling.



Repeating the process gets us back to



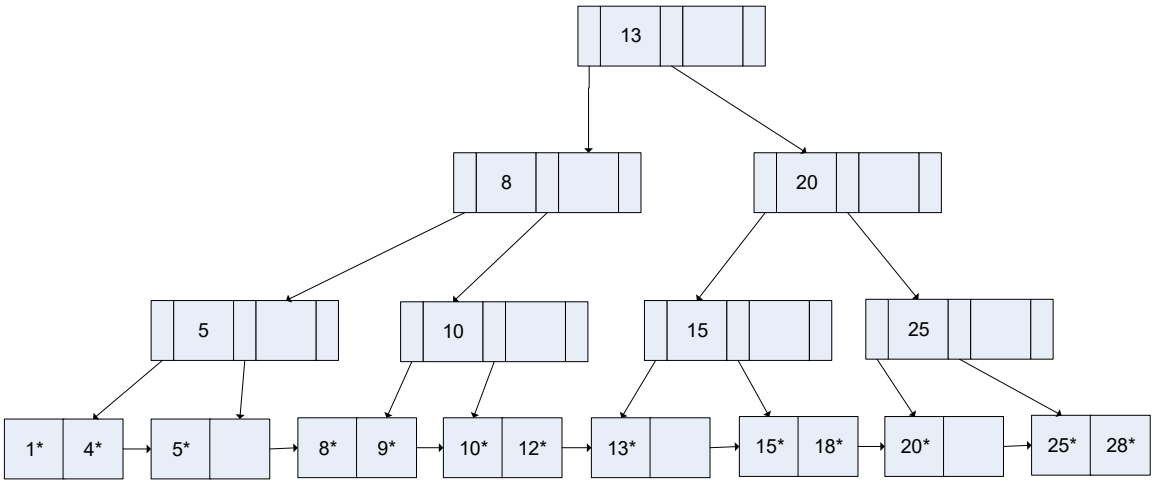
Our last example deletes 5. This takes the node and the index above it below d. We remove the leaf node and combine the index with its neighbor.



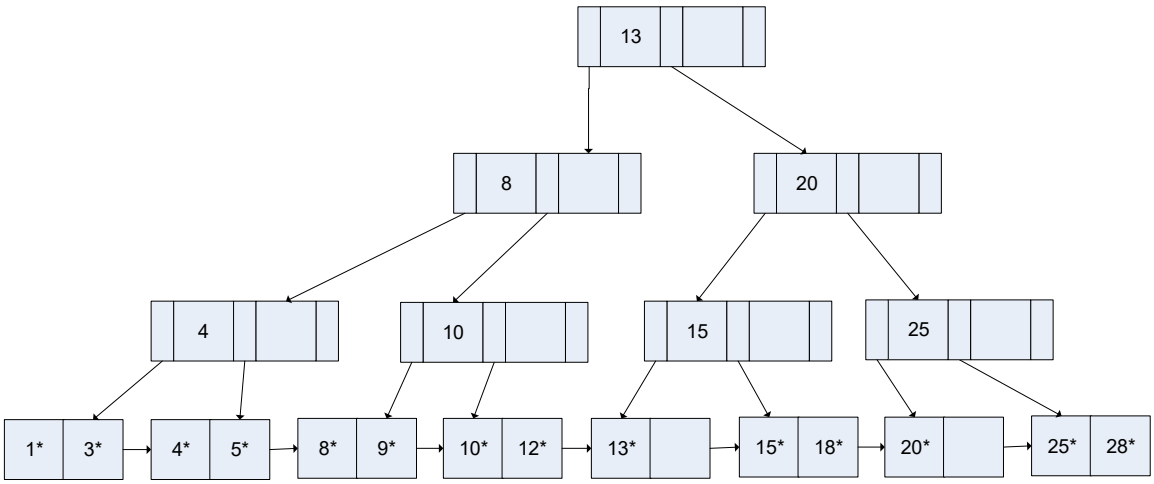
In this case, deleting 5 caused a merge with the data entry node containing (8,9). Eliminating the index node with 5 forced a merge with its sibling and pulled 8 down out from the parent node.

Rotation

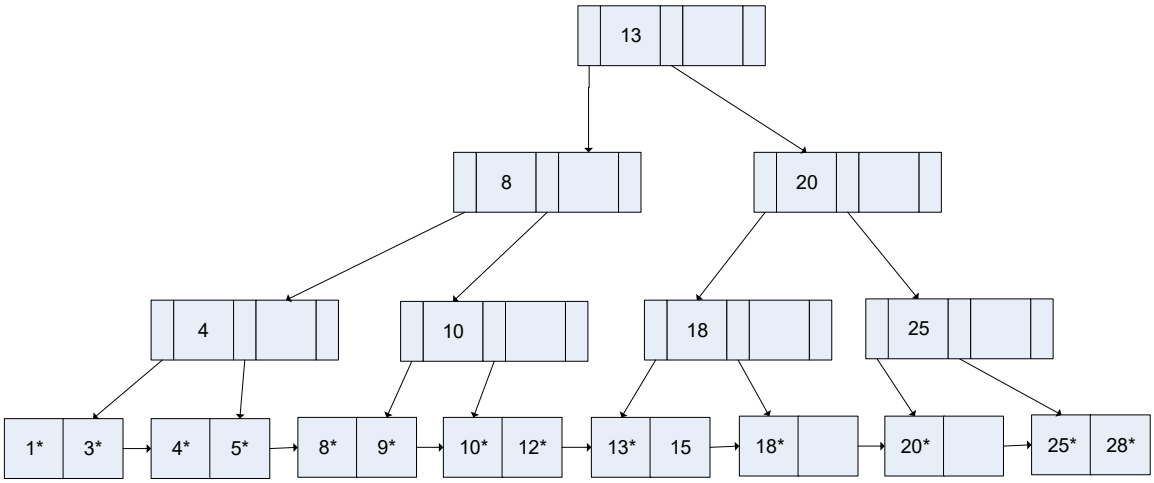
It is also possible to rebalance a tree to reduce the number of splits – called rotation. If you are trying to insert, and a leaf page is full, but its sibling isn't – you can move an index to a sibling and avoid splitting. Let's go back to a tree from our insert example:



We want to add 3 – but in this case we check the sibling to see if it has room. It does, so we move a record to it adjusting the index. Now we have :



The same concept works with deletes. If we took the above tree and deleted 13, you can re-distribute from the sibling:



and then do the delete: