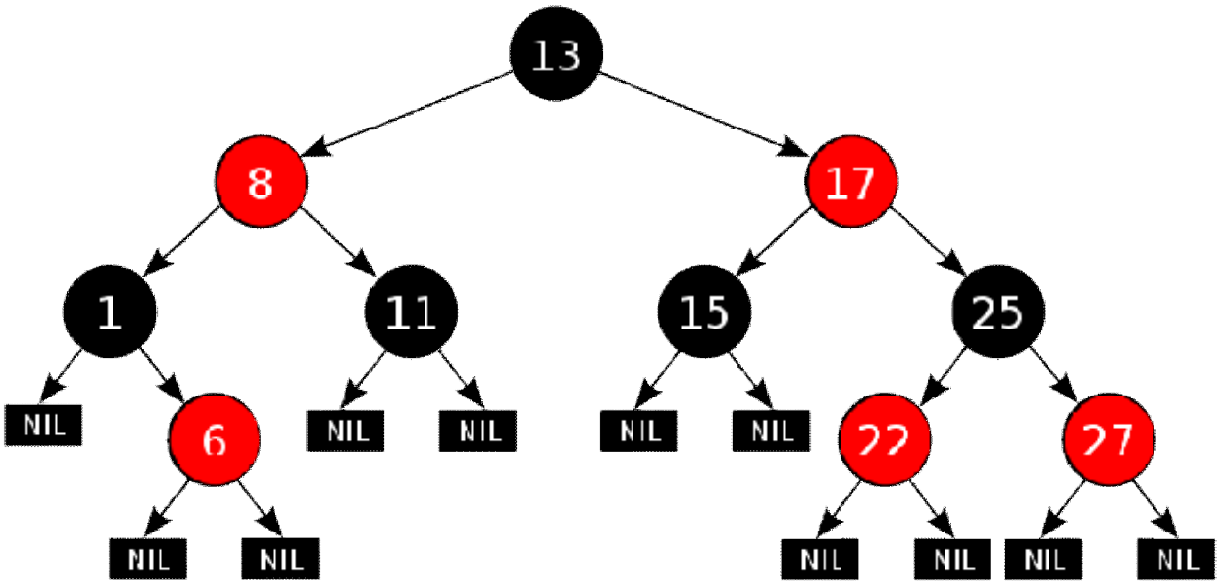
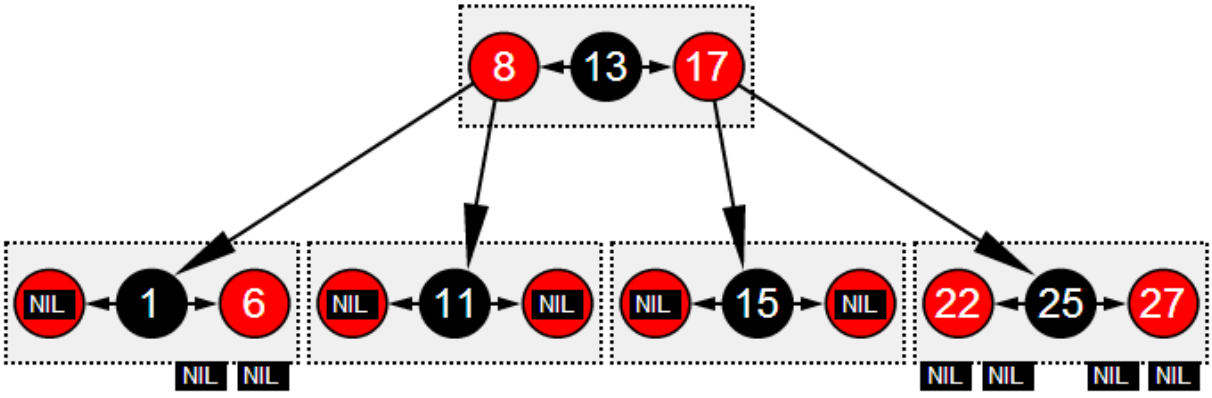
Actually Wikipedia has a great article that shows every RB-Tree can easily be expressed as a B-Tree. Take the following tree as sample:



now just convert it to a B-Tree (to make this more obvious, nodes are still colored R/B, what you usually don't have in a B-Tree):



Same is true for any other RB-Tree. It's taken from this article:

http://en.wikipedia.org/wiki/Red-black_tree
To quote from this article:

The red-black tree is then structurally equivalent to a B-tree of order 4, with a minimum fill factor of 33% of values per cluster with a maximum capacity of 3 values.

I found no data that one of both is significantly better than the other one. I guess one of both had already died out if that was the case. They are different regarding how much data they must store in memory and how complicated it is to add/remove nodes from the tree.

Update:

My personal tests suggest that **B-Trees are better when searching for data**, as they have better data locality and thus the CPU cache can do compares somewhat faster. The higher the order of a B-Tree (the order is the number of children a note can have), the faster the lookup will get. **On the other hand, they have worse performance for adding and removing new entries the higher their order is**. This is caused by the fact that adding a value within a node has linear complexity. As each node is a sorted array, you must move lots of elements around within that array when adding an element into the middle: all elements to the left of the new element must be moved one position to the left or all elements to the right of the new element must be moved one position to the right. If a value moves one node upwards during an insert (which happens frequently in a B-Tree), it leaves a hole which must be also be filled either by moving all elements from the left one position to the right or by moving all elements to the right one position to the left. These operations (in C usually performed by memmove) are in fact O(n). **So the higher the order of the B-Tree, the faster the lookup but the slower the modification**. On the other hand if you choose the order too low (e.g. 3), a B-Tree shows little advantages or disadvantages over other tree structures in practice (in such a case you can as well use something else). Thus I'd always create B-Trees with high orders (at least 4, 8 and up is fine).

File systems, which often base on B-Trees, use much higher orders (order 200 and even a lot more) - this is because **they usually choose the order high enough so that a note (when containing maximum number of allowed elements) equals either the size of a sector on harddrive or of a cluster of the filesystem**. This gives optimal performance (since a HD can only write a full sector at a time, even when just one byte is changed, the full sector is rewritten anyway) and optimal space utilization (as each data entry on drive equals at least the size of one cluster or is a multiple of the cluster sizes, no matter how big the data really is). Caused by the fact that the hardware sees data as sectors and the file system groups sectors to clusters, B-Trees can yield much better performance and space utilization for file systems than any other tree structure can; that's why they are so popular for file systems.

When your app is **constantly updating the tree, adding or removing values from it, a RB-Tree or an AVL-Tree may show better performance on average** compared to a B-Tree with high order. Somewhat worse for the lookups and they might also need more memory, but therefor modifications are usually fast. **Actually RB-Trees are even faster for modifications than AVL-Trees**, therefor AVL-Trees are a little bit faster for lookups as they are usually less deep.

So as usual it depends a lot what your app is doing. My recommendations are:

1. **Lots of lookups, little modifications: B-Tree (with high order)**
2. **Lots of lookups, lots of modifiations: AVL-Tree**

3. Little lookups, lots of modifications: RB-Tree

An alternative to all these trees are AA-Trees. As this PDF paper suggests, AA-Trees (which are in fact a sub-group of RB-Trees) are almost equal in performance to normal RB-Trees, but they are much easier to implement than RB-Trees, AVL-Trees, or B-Trees. Here is a full implementation, look **how tiny it is**(the main-function is not part of the implementation and half of the implementation lines are actually comments).

As the PDF paper shows, a **Treap** is also an interesting alternative to classic tree implementation. A Treap is also a binary tree, but one that doesn't try to enforce balancing. To avoid worst case scenarios that you may get in unbalanced binary trees (causing lookups to become O(n) instead of O(log n)), a Treap adds some randomness to the tree. Randomness cannot guarantee that the tree is well balanced, but it also makes it highly unlikely that the tree is extremely unbalanced.