Binary search trees and Balanced binary search trees

Binary search tree (BST):

It's a rooted tree, which is there's a root, the root has children and so on, until the last nodes which are called leafs.

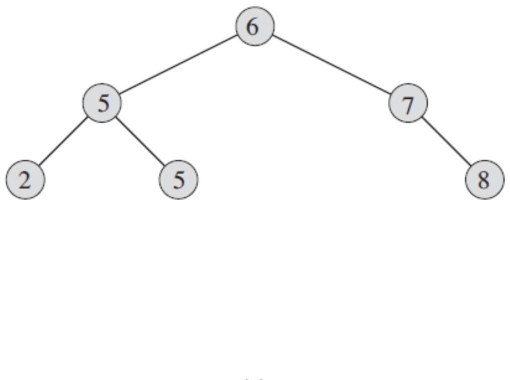
This is a completely ordered data structure. Methods:

- Key(x) – the value stored in the node.
- Left(x) – pointer to the left child.
- Right(x) – pointer to the right child.
- P(x) – pointer to the parent of x.

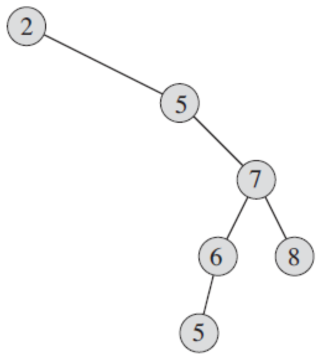
A completely ordered structured:

In a heap, a partially ordered structure, there's a relation between the parent and the children. Here there's also a relation between the two children: for each x: $left(x) \leq x \leq right(x)$, where by right and left means all elements in these sub trees (not like in a heap, where the two children are smaller than the parent).

The degree for each node would be 0,1,2.



(a)



(b)

The tree could be in worst case simply a linked list (each node has 1 child except for the leaf) – a linked list. The height of such tree could be n. In contrast, a balanced tree is a tree that is more similar to (a) above, with minimum height of $\lg n$.

Operations on trees:

Traverse tree:

Go through the entire tree and perform some operation. For instance, print the tree's elements in order:

```
$in - order - tree - walk(x)$:
1.  $If\ x! = nil\ then$
    1.1.  $In - order - tree - walk(left(x))$
    1.2.  $Print(x)$
    1.3.  $In - order - tree - walk(right(x))$
```

The running time would be $\Theta(n)$.

For a tree with root 1, $|T_l| = q, |T_r| = n - q + 1 \Rightarrow$ the total would be $T(n) = T(q) + T(n-q) + 1 = \Theta(n)$.

The print example above will output the tree's elements **sorted**.

Two more traversals are:

- Preorder traversal: first process the parent, then left and right.
- Post order traversal: first process left, right and then parent.

The running time of all the traversal algorithms in **linear**.

Find-key:
Given a key k, you simply apply a binary search, since we know that at each node the left subtree is entirely smaller than the right subtree.
The running time of the algorithm is $O(height\ of\ the\ tree) = O(n)$ – in case the tree is simply a linked list.
Insertion:
Look for the correct place to insert it, which would be a leaf. The running time is again $O(h)$ (which can be $O(n)$).
Minimum / Maximum:
The minimum is the node all the way to the left. Running time $O(h)$.
The maximum is the node all the way to the right. Running time $O(h)$.
Creation of a tree:
It can't be linear since that would infer we can create + travers = sort in $O(n)$. So creating the tree, if it's not balanced, would cost $O(n^2)$. The lease would be $O(n \lg n)$.
So ideally we want a tree of height $O(\lg n)$ – a balanced tree.
Successor:
Given x, we want to find the node with smallest key greater than $key(x)$.

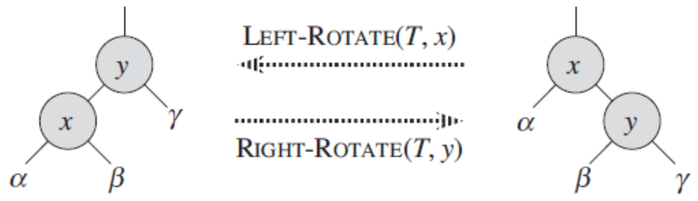
- If x has a right subtree: That would be the **minimum** of the **right** subtree of x.
- If x **doesn't** have a right subtree: the successor would be the **shallowest ancestor** that has a **left** child.

Finding the successor would be also $O(h)$.
Delete a node x:

- If x has no children: just delete it.
- If not, that means x either have a successor that is a leaf, or a predecessor that is a leaf. Otherwise we're in case 1. Once you get to a leaf you push all those on the way back up one level up (to eventually switch another node instead of the one deleted). Total: down and up = $O(h)$.

Red-Black Tree:

It is a balanced tree so each operation will take $O(\lg n)$ time. Each subtree is itself a subtree.
Rotations:



The rotations keep the structure a valid search tree. The operations done are only a few pointer substitutions, a total cost of a rotation is $O(1)$.
The reason to perform a right rotation is in order to lower the height of the α subtree (if it gets too deep).

In RB trees the same relationship is as in regular BST. In addition there are conditions:

- All nodes are colored either **RED** or **BLACK**.
- The root is always **BLACK**.
- All nodes but the leafs contain keys. The leafs contain NIL, and are colored **BLACK**.
- If a node is **RED**, its children have to be **BLACK**.
- The **BLACK** height of every leaf is the same. This condition actually keeps the tree balanced. This is called the black height of the tree. (It includes both the leaf and the root in the count).

The shortest path would then be h and the longest would be $2h$. Notice that if we start from the root and merge all nodes with their children if they are colored differently, we turn the tree to a <2,3>-tree – each node may be a super-node containing 1, 2 or 3 values in it. Then – ALL leafs are at the same level, which is the black height of the original BR tree. The degree of the super node would be at most 4. The height is now logarithmic (in at most base 4). The actual height is just $2\lg n$.

Insertion:

- Find the correct leaf to insert the new value, and put it there. When putting it, color it **RED** and give it 2 **BLACK** children. The reason we color it red is to keep the black-height condition (given its new leafs are black).
  BUT, we may violate the RED rule – as the new node's father may already be red (and thus we have 2 reds in a row).
- RED violation – move the problem up the tree if it cannot be solved locally. The insertion is $\lg n$, fix if needed is also $\lg n$ – total $\lg n$. If the root became red eventually, we just color it BLACK – and the black height is increased by 1.

<u>There are 6 (3 pairs of symmetric cases) possible cases for each iteration</u>:

1)

Inserting X, X's parent is RED.

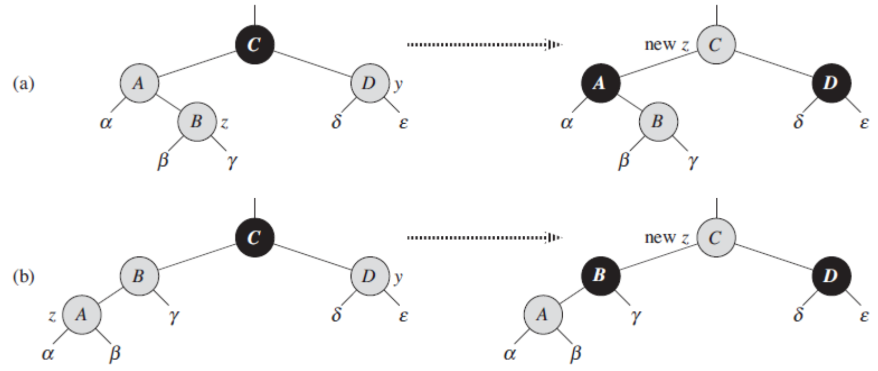
Since before X's insertion the tree was valid, that means the grandfather of X is BLACK, and the second child of the grandparent (the "uncle") has to be RED like X's father (since before the black height condition was kept).

<u>Solution</u>:

Color X's father and uncle BLACK and color X's grandparent RED.

- The black height is kept: instead of (father+undle) RED and grandparent BLACK, we switched them in color.
- No two consecutive REDs, as X's father is no longer RED like X.

This might have created a violation upper in the tree, but that will be solved there.

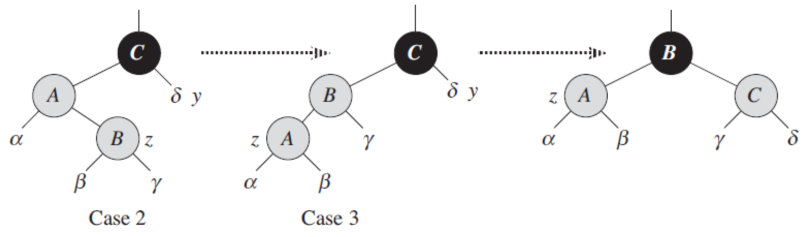


In the image: we moved the problem from A to C.

Locally we fix the problem in $O(1)$ time.

2)

X's parent is RED, but the grandparent and uncle are both BLACK.



Solution:

- First rotate to a **canonical form**. In the case above: rotate to the LEFT around the FATHER of X.
- Then rotate everything to the RIGHT around the GRANDPARENT. In this rotation when the father moves instead of the gradpa, it is colored BLACK. The grandpa that moves instead of the uncle, is colored RED.
  In the image: B is colored black, C is colored red.

This way the black height is kept and we fixed the red-red violation. We fixed the problem locally.
The cost for the fix is $O(1)$ – only rotations and recoloring.

3)
This is actually the “case 3” label in (2)’s image – we start from the canonical form.

So back to case 1, which moves the problem up, the total fix might take $\lg n$ violation fix. Each fix level takes constant time, so the whole process will end up $\lg n$. If up to the root is then colored red and its children are red, we just color it black and increase the black height by 1.