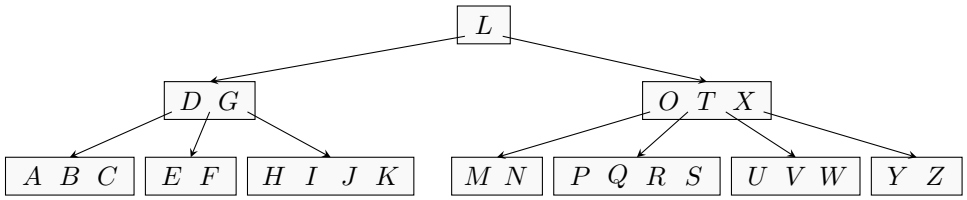
1. Without dwelling on the structure of the nodes and on the positional relationship between keys and subtrees (for which an example picture will be sufficient), give an otherwise complete and concise definition of a B-tree of minimum degree t, listing all the defining structural properties.

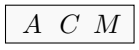
   The features of a B-Tree with minimum degree t are as follows:

   - Any node has no fewer than $t-1$ elements (with the exception of the root when the whole tree has fewer than $t-1$ elements.
   - No node has more than $2t-1$ elements.
   - A node with n elements should have $n+1$ children.
   - Every route from any node to a leaf passes through the same number of nodes.
   - Insertion into a B-Tree is done at a leaf.
   - When inserting an element into a B-Tree you should split any nodes you pass through which have $2t-1$ elements.
   - When deleting an element from a B-Tree you should fill any nodes you pass through so that they have at least t elements.

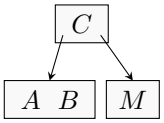


2. Explain how to insert a new item into a B-tree, showing that the algorithm preserves the B-tree properties you gave above. Then insert the following values, in this order, into an initially empty B-tree whose nodes hold at most three keys each: C A M B R I D G E X. Produce a frame-by-frame "movie" in which you redraw the tree whenever it changes in any way.

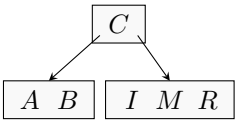
   The first three insertions (C, A, M) are simple – we do not reach the largest node size and so simply insert them into the root node without any issues.



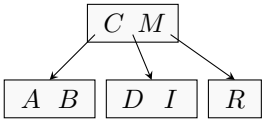
   Inserting B is more problematic. The root node is full. So when we reach the root node, we split it, promoting the middle element (C) to the new root node and placing the other two elements in new nodes – each the children of C. Now we insert B without issue, seeing it is less than C but greater than A so insert it to the right of A.



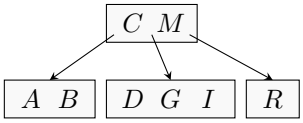
   We insert I and R without having to take any action to preserve the properties of the B-Tree. These insertions are simple traversal.



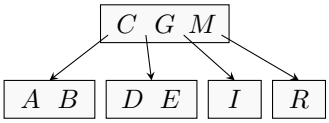
When we insert D, we notice that the node containing I, M, R is full, and so split it, promoting the middle element M into the parent node. We then insert D into the correct child node.



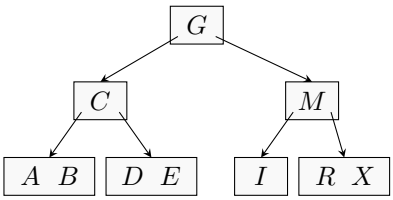
G is inserted in a simple traversal.



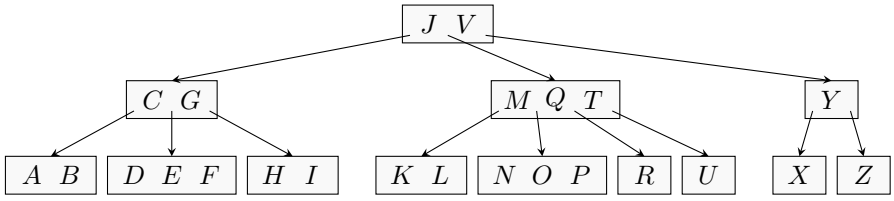
when inserting E, we reach the node D, G, I – which is full. So we must split this node, promoting the middle node into the parent node.



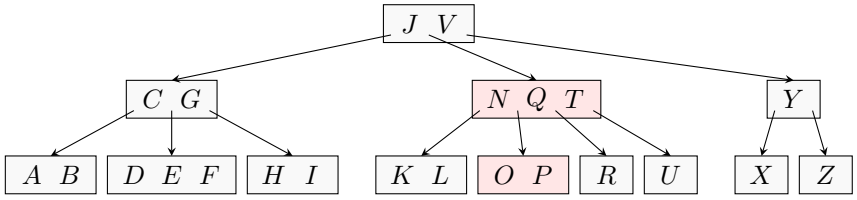
The next insertion finds that the root is full and so splits in, promoting the middle element G to the new root node. We then insert X in a trivial insertion.



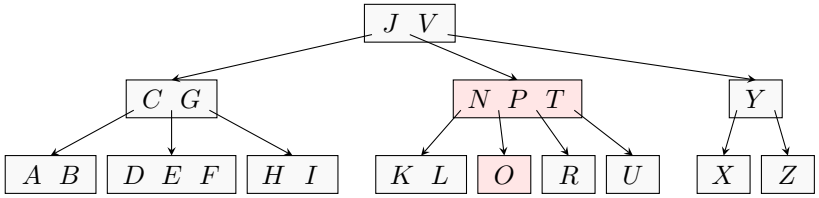
3. Explain how to delete a value from a B-tree and illustrate your algorithm by showing frame-by-frame movies for the deletion of M, Q, Y, in that order, from the following B-tree. As before, each node holds at most three keys.



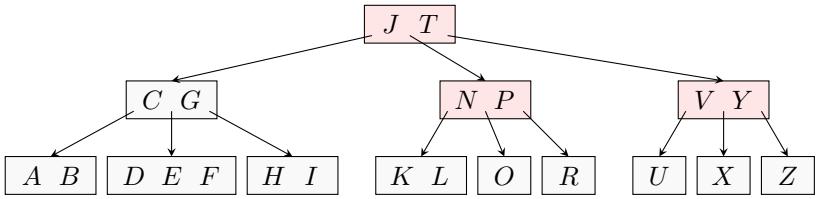
Since M is an internal node, we cannot delete M without manipulating the tree slightly. Since the successor of M (N) is in a node which is not of the minimum degree, we can replace M with it's successor and then delete the successor without further manipulation. The tree after this has happened is shown below with the affected nodes highlighted.



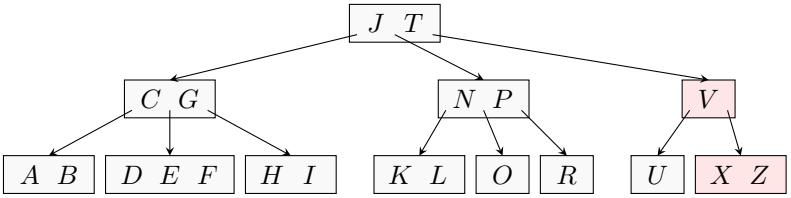
To delete Q we notice that the successor (R) is in a node of minimum degree – so we cannot replace Q with R. Next we check the predecessor – in this case P. We notice that P is in a node which is not of the minimum degree and so we can replace Q with P and then delete P from the node it was in while still satisfying all the B-Tree properties.



In order to delete Y we notice that Y is in a node of minimum degree. So we cannot delete Y without violating the properties of a B-Tree. However, since Y has a sibling – in this case it's neighbour N, P T – which has more than the minimum degree. So we can "shuffle" across nodes from the sibling. We do this by moving the rightmost element in the sibling into the parent and moving the successor down into the original node (with Y in it). The diagram of this is shown as below. We must also move U across in order to preserve the properties of the B-Tree. Now we are deleting from a node with more than the minimum degree.



We notice both Y's successor and predecessor are in nodes of minimum degree. This means we cannot promote either of them otherwise we would have a node with less than the minimum degree. So we merge Y's children and delete Y. This preserves the properties of the B-Tree.



4. Briefly explain what a binary search tree (BST) is, listing its properties.

   A binary search tree is a tree where each node has 0 - 2 children. Every node in the left subtree is smaller than the parent and every node in the right subtree is greater than the parent. Duplicate elements are not allowed in a Binary *Search* Tree.

5. Describe an optimally efficient algorithm to find the predecessor of a given node n in a BST and explain why it works.

   There are two cases:

   (a) The node n has a left child.

   If the node n has a left child then you must traverse into the left subtree and then traverse into the right subtrees until you reach a node which has no right subtree. This is the predecessor to the node n.

If n has a left child then every node to the left is less than n. Let the predecessor to n be p. Assume that the p is not in the left subtree. Then there are three subcases:

- p is in the right subtree of n. However this violates the BST property that all elements in the right subtree are less than the element. So p cannot be in the right subtree.
- There is some ancestor a of n such that p is in the left subtree and n is in the right subtree. Since all elements in the left subtree are less than a and all elements in the right subtree are greater than a; this means that $p < a < n$. However, since there is an element a that is between p and n, this means that p cannot be the predecessor to n. Which is a contradiction.
- There is some ancestor a of n such that n is in the left subtree and p is in the right subtree. All elements in the right subtree are greater than the root which is greater than all elements in the left subtree. So $p > n$. This means p cannot be the predecessor to n. Which is a contradiction.

So if n has a left child, then the predecessor to n must be in the left child.

The predecessor must be the largest element in the left subtree of n – else there is an element larger than p and that would be the predecessor. To find the largest element in a subtree you traverse to the right until you reach a node which does not have a right node. So we have found the predecessor to n.

(b) The node n has no left child.

If the node n has no left child then you must go up the tree until you are coming from a right child. The node at which this happens is the predecessor to n. If you traverse to the root coming only from the left child then node n is the smallest element in the tree.

We can show using the same logic as part (a) n is the successor of this node. And if no such node exists then n has no successor and so n is the smallest node in the tree.

6. Describe an optimally efficient algorithm for deleting a node d from a BST when neither of d's subtrees is empty. Explain why it works and prove that what remains is still a BST.

   Replace d with it's successor s. Then delete s from it's original position. s is guaranteed to have one or zero subtrees (it must not have a left subtree – else the successor to d would be in that subtree and not s). This means that deletion of the successor only involves changing the left pointer of the successor's parent to the pointer to the successors right child (whether it is a node or Null).

   The properties of a BST for every node n; every node in the left subtree of n is less than n which is less than every node n's right subtree.

   So after replacing the node d with s, in order for the Binary Search Tree to remain a Binary Search Tree; we must satisfy that every node in d's left subtree is less than s and every node in d's right subtree (excluding s) is greater than s.

   s is greater than d so if every node in d's left subtree is less than d; then it must also be less than s. We also know that s is the smallest element in d's right subtree. If there is an element (excluding s) in d's right subtree which is less than s; that means that s is not the smallest element in d's right subtree and so s is not d's successor.

7. Assume that node l, whose key is k_l, is a leaf of a BST and that its parent is node p, with key k_p. Prove that, of all the keys in the BST, k_p is either the smallest key greater than k_l or the largest key smaller than k_l.

   Trivially a is the successor of $b \iff b$ is the predecessor of a.

Let us start at p. This problem has two cases.

(a) l is the left child of p.

In this case consider the predecessor of p. p has a left subtree and so the predecessor of p is the largest node in p's left subtree. l is trivially the rightmost node in p's left subtree and so l is the predecessor of p.

Taking the inverse of this gives: p is the successor of l. This means that k_p is the smallest key greater than k_l.

(b) l is the right child of p.

In this case consider the predecessor of p. p has a right subtree and so the successor of p is the smallest node in p's right subtree. l is trivially the leftmost node in p's right subtree and so l is the successor of p.

Taking the inverse of this gives: p is the predecessor of l. This means that k_p is the largest key smaller than k_l.

So k_p is either the smallest key greater than k_l or the largest key smaller than k_l as required.

8. State the five invariants of Red–Black Trees and briefly explain the advantages and disadvantages of Red–Black Trees over Binary Search Trees.

   The five invariants of red-black trees are:

   - All nodes are either red or black
   - The root is black
   - The number of black nodes from any node to any leaf in any subtree of which it is the root is constant.
   - If a node is red, both it's children are black
   - All leaves are black and contain no key-value pairs

   Red-Black Trees have a guaranteed $\Theta(\lg n)$ depth in terms of the number of nodes in the tree. This is in contrast to non-balanced Binary Search Trees which have a depth of $O(n)$. This means that for Red-Black Trees, insertion and deletion operations are both $\Theta(\lg n)$. So all operations for Red-Black Trees are guaranteed $\Theta(\lg n)$ time.

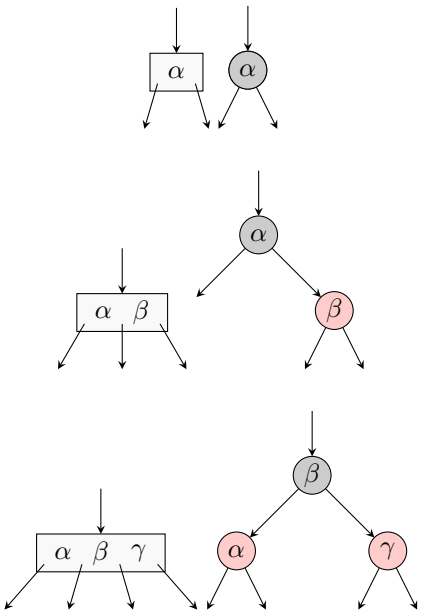
   However, this hides a higher constant than with Binary Search Trees. And while the worst-case complexity of a Binary Search Tree is $O(n)$, the average case complexity is $O(\lg n)$. An argument against this is that in most situations the average case complexity with a lower constant factor is better than the guaranteed $\Theta(\lg n)$ complexity.

   Another feature that Red-Black Trees require is an additional bit per node – this stores whether a node is red or black. Binary Search Trees do not have such a requirement. However, one bit per node is usually negligible – analysis of node sizes in python gives this: each node has 96 bits in pointers to children and parents plus a standard 48 bits of class information and 32 bits for a pointer to the data it is storing. In a high-level programming language this means a node will have ≈ 176 bits. Adding an additional bit is not meaningful. Using a different language could reduce the amount of storage needed in metadata. However, the pointers are constant and the conclusion is the same.

   I would recommend the use of Red-Black Trees in general – however Binary Search Trees can be used if the data is known to be pure random or a fast response time is not required.

9. For each of the possible types of 2–3–4-tree nodes, draw an isomorphic node cluster made of Red-Black nodes. The node clusters you produce must have the same number

of keys and external links as the 2–3–4 nodes they replace and they must respect all the Red–Black tree rules when composed with other node clusters.



10. What are the minimum and maximum possible number of nodes of a Red-Black tree with black-height h? Justify your answer.

    The minimum possible height of a Red-Black tree with black height h is h. This is the case where every node is a black node. In this case the Red-Black tree is a full tree of height h – and so has $2^h - 1$ nodes.

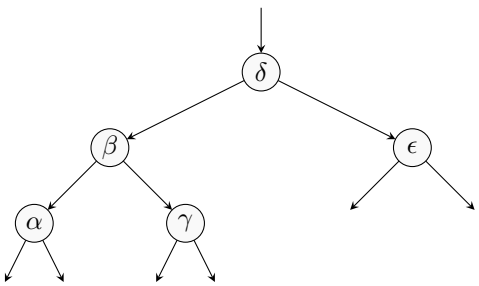
    The maximum possible size of a Red-Black tree with black height h is $2h$. This happens when every black node has two red children. In this case the Red-Black tree is a full tree of height $2h$ and so has $2^{2h} - 1$ nodes ($4^h - 1$ nodes).

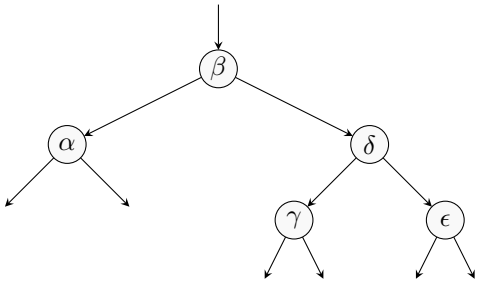
    Although I question how it would be possible to *build* a tree in either case.

11. Explain, with clear pictures, what a "rotation" operation is, in the context of Binary Search Trees.

    A rotation is a change in some pointers which "rotates" an edge so that the original child becomes the parent and order in the BST is preserved.

    Take the tree below as an example. I will rotate the edge β – δ:





There are two types of rotation – left rotations and right rotation. What I demonstrated was a right-rotation: a left rotation is the inverse of that and would be the same as going from the second tree to the first.

12. Given an arbitrary n-node Binary Search Tree containing n distinct keys, show how you can restructure it into a balanced binary tree in-place (i.e. using O(1) memory additional to the input tree itself). Can your procedure be extended to restructure any tree into any desired target shape?

    I have two solutions to this question. Each of them have a slightly different interpretation of what "in-place" means and have different time and space complexities.

    - The first interprets "in-place" as "requiring negligible auxiliary memory". This algorithm sweeps across the tree converting it section-by-section into a red-black tree and deallocating the memory used to store the colours of the nodes at the earlierst opportunity. This algorithm can has a $\Theta(n)$ time complexity and a $\Theta(\lg n)$ space complexity (specifically $4 \lg n$ bits – negligible).
    - The second algorithm interprets "in-place" to mean "requiring a strictly constant amount of memory" – this means no recursion since the logarithmic number of stack frames are unnacceptable, no storing the size of the balanced tree since the $\lg \lg n$ bits required to do so are unnacceptable. This algorithm flattens the tree using the parent pointers to convert it into a linked list, then promotes the median of the list to the parent and repeats on the subtree $\Theta(n \lg n)$ and a space complexity of $\Theta(1)$.

    For simplification, both algorithm take a trees where each node has a parent, left and right pointers. I do, however discuss how to change either algorithm to work on nodes which only have left and right pointers without adversely affecting their complexities.

    (a) In the first algorithm I will make the simplifying assumption that "in-place" means "do not extract from the tree or do anything remotely memory-intensive". This allows me to convert the tree into a red-black tree – which has a non-constant but completely negligible memory requirement – $4 \lg n$ bits.

    Naïvely you may think that the cost to red-black tree-ify would be n bits. This is not true. After we have red-black tree-ified every subtree and every subtree of every sibling of a node, we never need to consider the colour of that node again. This means we can deallocate the memory used to store the colour of the node. So given a tree we only need to know the colours of a number of nodes proportional to the maximum height of the tree – the most colours we need to store is $4h - 1$ where h is the black-height of the red-black tree (which is logarithmic in terms of the number of nodes n in the tree). So the space complexity of the algorithm is $\Theta(\lg n)$ (with a *very* low constant). However, if you wished, you could perform the algorithm without deallocating the colours and have a red-black tree at the end of it.

    A maximally unbalanced red-black tree of black-height h will have $2^{h+1} - 1$ nodes. In this case, the most colours we need is $4h - 1$. So $n = 2^{h-1} - 1$. For large n:

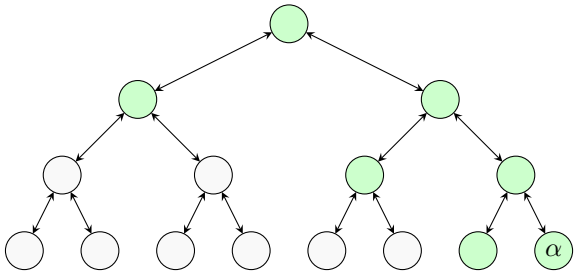
$n \approx 2^{h-1}$. $\lg n + 1 \approx h$. So the most colours we need is $\approx 4 \lg n + 4 - 1 \approx 4 \lg n + 3$. we can add one for the colour of the node we are inserting. Each colour is one bit. So we need at worst $4 \lg n + 4$ bits to colour the tree during the red-black tree-ification.

For reference if we had 1 billion nodes (taking 20GB to store) we would require at worst ≈ 230 bits to colour the tree. This is negligible. I would also like to point out that this is the *worst* case. The average case is linearly less than this. Although the space complexity is not constant I feel that it is so low that in any real-world application there would be no issue.

Informally; when we recurse down the right subtree we only ever rotate left – which does not bring any new nodes into the tree from the right subtree. So we no longer need to know the colours of the nodes in the right subtree (except for the root). Vice-versa for when we recurse left.

This is easier to represent visually.

In the tree below: to balance the node α or any subchildren of α, the only nodes in the tree we could possibly need to know the colour of are highlighted.



The algorithm goes as follows.

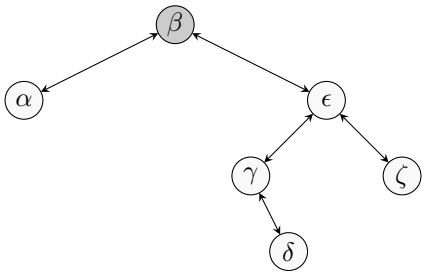
Set the root to be black. The algorithm for insertion into red-black trees is designed to recurse up the tree. This means that it works if every node in the relation has children. We can use this to keep the top part of the tree "balanced" but not the bottom part of the tree.

When I discuss a dfs I mean a custom pre-order dfs which keeps a pointer to one node and a boolean indicating whether we came *up* to the tree from the left or *down* to it (we only need one boolean for the whole algorithm – not per-node). We should then traverse to the next unvisited node. Firstly, we should start a dfs down the tree to the first violation of the properties of a red-black tree – which (assuming the tree is larger than size 3) will be that there is a red node which is the child of a red node.

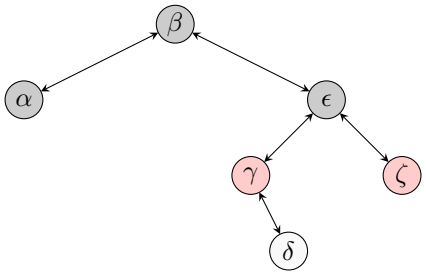
We then rebalance the tree as if we are inserting this new red node keeping a pointer to the original node we found. Once the tree has been rebalanced we continue with the dfs. We only require one dfs to red-black tree-ify the whole tree. We should set nodes to be red when we reach them or their sibling with the dfs and deallocate the memory for the node when we go through the parent with the dfs.

The reason we are guaranteed to red-black tree-ify in a single dfs is: rebalancing does not change the order of the tree. If we scan from left-to-right (even if we rebalance or rotate), the successor be the same. So we will never miss a node. The case of un-inserted uncles is discussed later – for now it is enough to know they are also not problematic.

I will demonstrate the balancing on the tree below.

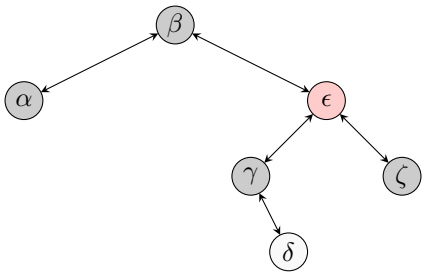


We start the depth-first search. We traverse to α, notice no violations and so start traversing into the right subtree. When we reach γ we find a violation. To resolve this we follow the default implementation for this case in a red-black tree – setting α and ϵ to be black, then fixing β – turning it black again.

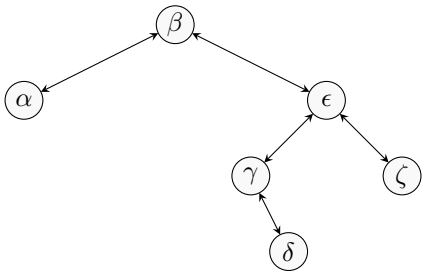


Next we reach δ and balance starting at δ. Note that at this point, δ has an uncle (ζ) which has not been inserted properly and so could be in violation of the red-black properties (as could it's children – however in no case would we ever check ζ's children). This does not matter. If the parent of δ is black then the insertion is done without checking ζ. If the parent of δ is red then the grandparent of δ (and the parent of ζ) is black and so ζ is not in violation of the red-black tree properties. This argument is crucial and allows us to red-black tree-ify using a dfs rather than a bfs (which has a non-negligible memory requirement of $O(n)$ pointers).

We now balance the tree using the normal method in this case. We set ζ and γ to be black, change the grandparent to be red and recurse onto the grandparent. ϵ is not in violation of any red-black tree properties – so we continue with the dfs. This leaves us in the state below:



We now finish the dfs by checking ζ, notice it does not violate any red-black tree properties, then traverse back up the tree de-allocating the colours in any nodes (and chilren) that we pass. The tree is now in the state below:



We have now balanced the tree in-place in linear time with minimal space requirements. Each balancing in a Red-Black Tree takes $O(\lg n)$ time. A naïve analysis would therefore conclude that the complexity of rebalancing is therefore $O(n \lg n)$. However, since consecutive insertions make the tree in a more consistent state, the total cost of doing n rebalancings is $\Theta(n)$. This means that the cost of converting the tree into a red-black tree is $\Theta(n)$ (since we do a dfs – linear, and rebalance n nodes – also linear). This is asymtotically optimal – an informal proof goes: "if you have n nodes – then to know a tree is balanced you must pass over $\Omega(n)$ nodes – therefore any balancing algorithm must be $\Omega(n)$".

This procedure cannot be extended to restructure any tree into any desired target shape. It works exclusively for conversion of a Binary Search Tree into a Red-Black Tree.

In order to convert this algorithm into one which worked on nodes containing only left and right pointers, you would add and deallocate parent pointers similarly to colouring and de-colouring nodes. Except we need linearly fewer parent pointers – ie if we look at the sibling of a node when doing insert_fixup we already know its parent from its siblings parent and do not need to store it again. If $|p|$ is the number of bits in a pointer then this method would require an additional $2|p| \lg n$ bits. This conversion does not affect either the space or time complexity.

(b) The second algorithm is much less cool. However it is parallelisable – something the first is not.

Firstly we perform an in-order depth-first-search on the tree changing the parent pointer of the node we leave to its successor and setting both of its child nodes to null when we leave it (we should also set its childs parent pointer to the parent of this node. This means that we can convert the tree to a linked list in guaranteed linear time and constant space complexity.

We now have the tree in a linked-list representation with a pointer to the head of the linked list. We can start by traversing down the list with two pointers – one which moves one node per cycle and the other which moves two. When the node which moves two hits the end of the list (initially indicated by a null pointer) we know that the pointer which only moves one has reached halfway down the list. This is the upper median.

The first time we do this we should set the pointer to the first node to point to the median (initially this is the root pointer, but on subsequent iterations this is the parents left or right pointer), the left child in the median to be the first element in the sublist and the last element in the left sublist to be the median (and the same for the right). This means that we now have a cyclical linked list formed by the medians child pointers and the parent pointers in the children. We repeat the median-finding algorithm however now the terminatation condition is finding the pointer that moves two pointers per-turn finds a pointer to the parent rather than a null pointer. This allows us to further subdivide the linked list. We repeat this until we find the base case where the pointer which moves twice per turn finds the parent after moving once (we can have a boolean indicator variable which we set to be True at the start of the loop and change to false if the pointer

that moves twice moves succesfully).

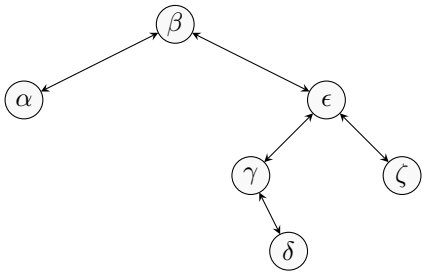
Then we come up to the median and repeat for the right subtree. Note that this algorithm does not need to be recursive since we can return to the parent from the children.

This algorithm forms the recurrence relation:

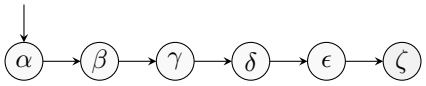
$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \tag{1}$$

Which has the solution $T(n) \in \Theta(n \lg n)$.

An example of this is given below:

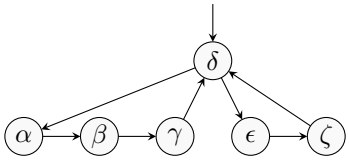


Firstly we perform an inorder traversal on the tree and convert it into a linked list in-place.

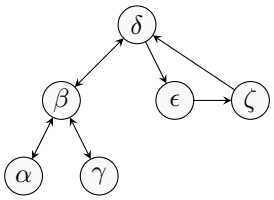


Now we must find the median of this linked list. We move a pointer forward one – to α and another forward two to β. Repeat two more times until the second pointer hits a null pointer. At this point the first pointer si at δ.

So we set δ to be the root and change the the root pointer and change its child pointers.

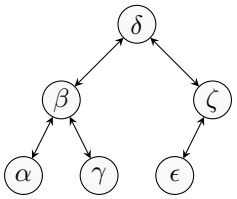


Now we traverse into the left subtree, to find the median of α, β, γ. We perform the same algorithm except terminating when we find a pointer to the parent rather than a null pointer. We find now that the median is β.



Now we traverse into α, notice that it is a linked list of length one – so this is the base case, traverse back to β, then traverse into γ, notice again that it is the base case and so traverse up. Since this time we came to β from the bottom we should traverse up into δ, then deal with the right subtree of δ.

we move the first pointer one forward to ϵ and the second to ζ. Since the second pointer has not hit a the pointer to δ yet we move forward again. The first pointer is at ζ, the second now finds δ so we know ζ is the upper median and rearrange the tree accordingly.



Now we traverse into ϵ, notice that it is a linked list of length one and so is the base case. We now traverse back up the tree, hit δ from the right and so we are done.

We have now fully balanced the tree absolutely in-place in $\Theta(n \lg n)$ time.

This procedure can be extended for rebalancing the tree into an arbitrary shape – say by moving the second pointer by three insteal of two we could have a tree where the right subtree was twice the size of the left.

A footnote; to convert this algorithm into one where every node only had a left and right pointer would mean changing the linked list to use the right pointers and perform a search for the node when travelling to the parent – the complexity is unchanged. Assume every node is at the bottom of the tree – this costs more than the travelling to the parents would and so provides an upper bound. Traversing to the bottom of the tree is $\Theta(\lg n)$ and so n parent operations would cost $\Theta(n \lg n)$ – which is the same complexity as the balancing and so would not change the overall complexity.