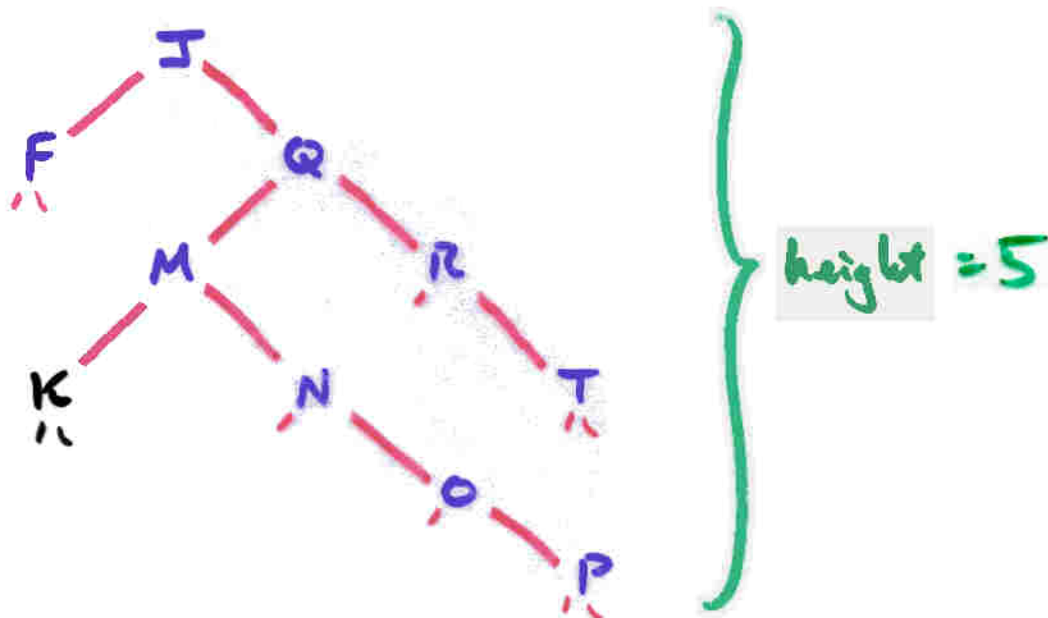
Balancing Trees

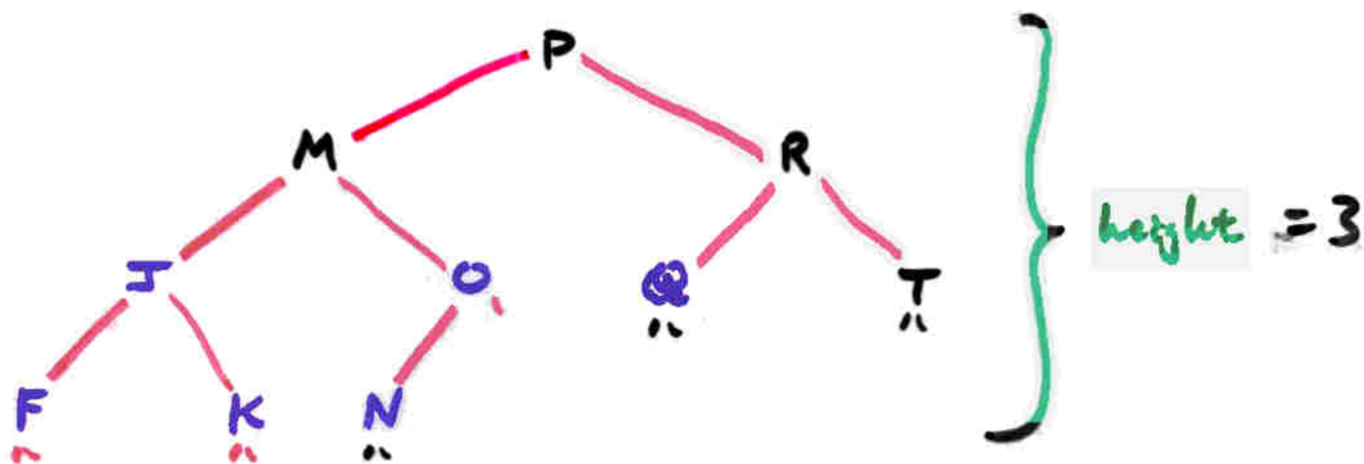
Consider the sequence ...

, Q, R, M, K, F, N, O, P, T

inserted into an ordinary BS Tree ...



This is not great — the ideal might be ...



We could obviously have been more extreme in our choice of sequence!

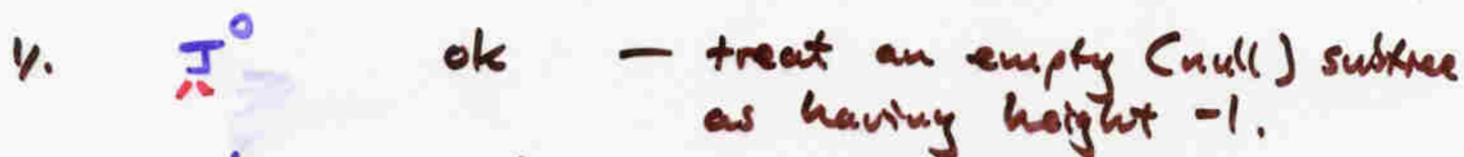
Since the whole point of binary search trees is that searching is only of order the height of the tree, and a well-balanced tree will have height roughly $\log n$, if the tree is badly out of balance all our nice estimates blow up!!

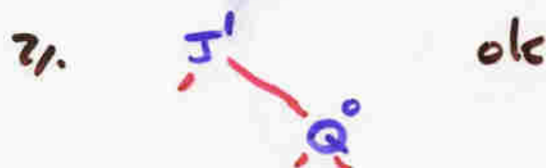
The two main approaches are either to maintain some reasonable approximation of balance throughout the tree manipulation, or to periodically 'zap' the tree by a separate balancing algorithm. We'll start by looking at two flavours of the first option.

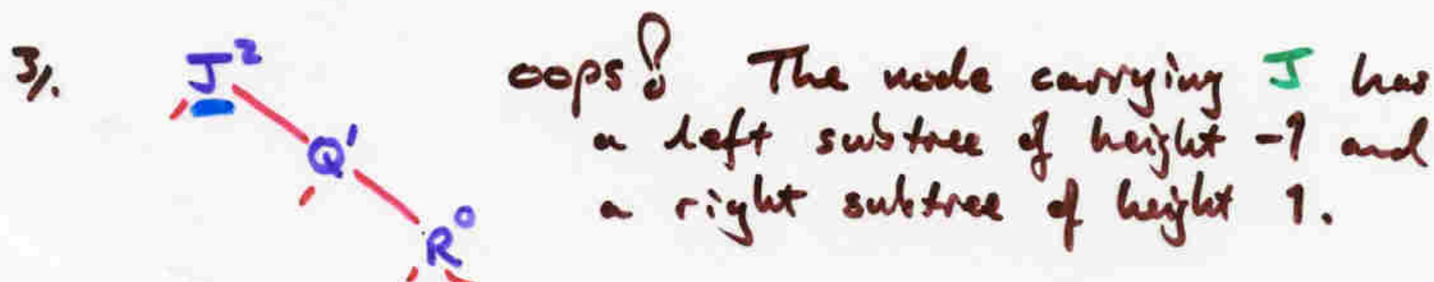
AVL Trees

This approach, due to Adelson-Velskii and Landis, insists that at each node, the difference in height between the left and right subtrees never exceed 1. (We store the height information of each subtree in that subtree's root node.)

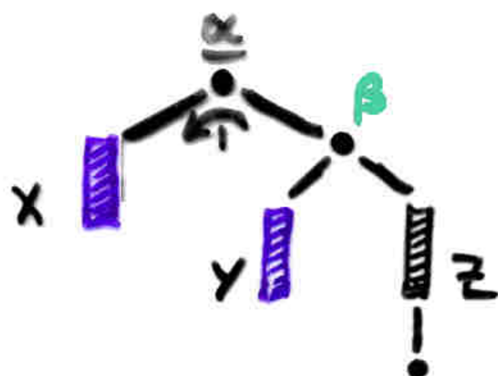
Pouring our sequence into an AVL tree gives ...

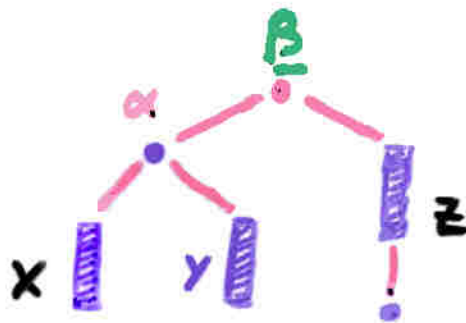






We handle this problem by rotating Q into J's location. More precisely, a single left rotation is performed as follows ...





Here we treat each of these purple blocks as though they have height h, so the purple block with the extra 'dot' represents a block of height h-1. The actual rotation is done by ...

1/. move α together with its left subtree X down to its left.

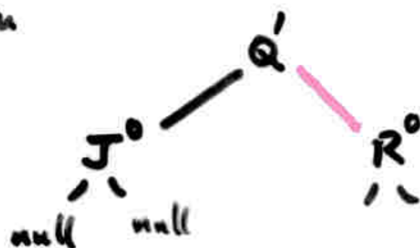
2/. move β together with its right subtree Z up to where α used to be.

3/. change Y from being β's left subtree to being α's right subtree.

Of course, when coding this, we would need to use some temporary iterator to avoid 'losing' a node! A single right rotation is similar. Thus step 3/. becomes ...

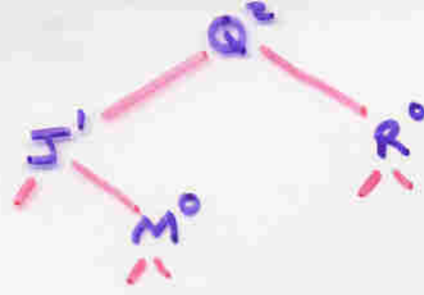
3/.

single L rotn



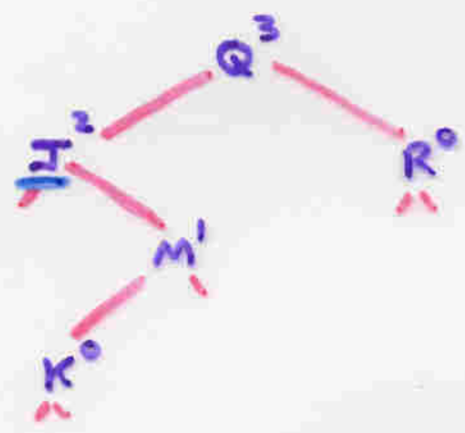
Here we've labelled explicitly the two null nodes which match X and Y in our explanation — Z is the R subtree.

4/.

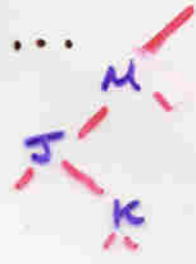


- adding "M" causes no complications.

5/.

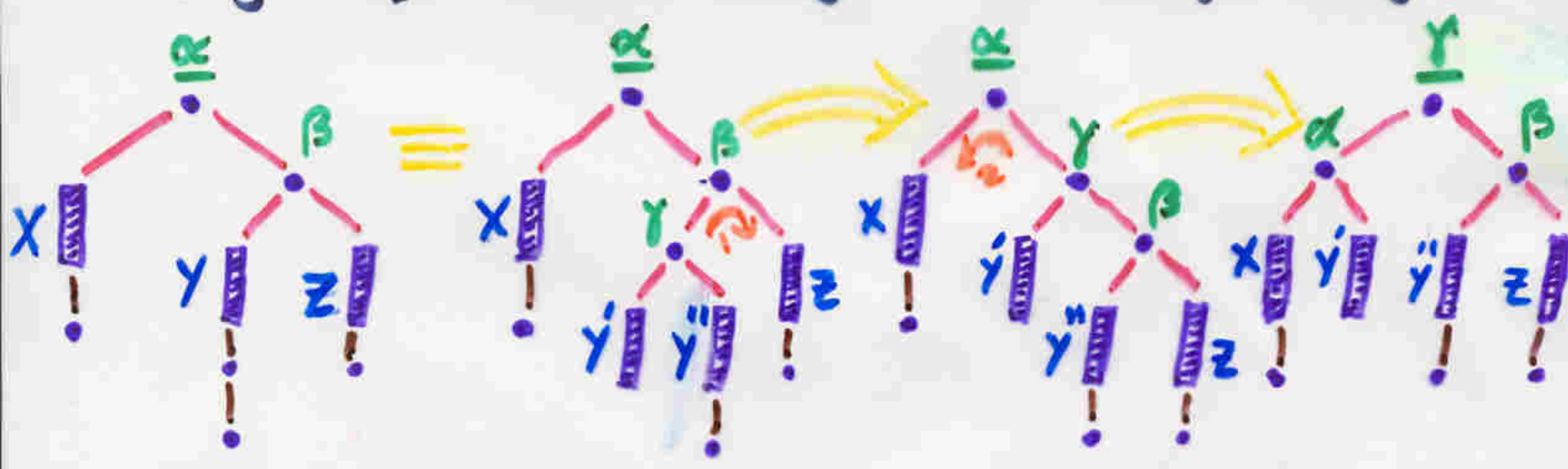


- adding "K" causes problems of excessive imbalance at the nodes carrying J and Q. Our approach is to attempt to resolve problems at the lowest level, in this case J. However, a single rotation would give ...



which is an equally bad problem!!

We handle this situation by performing two rotations: the first a single right rotation "through" M, the second a single left rotation "through" J. More generically ...



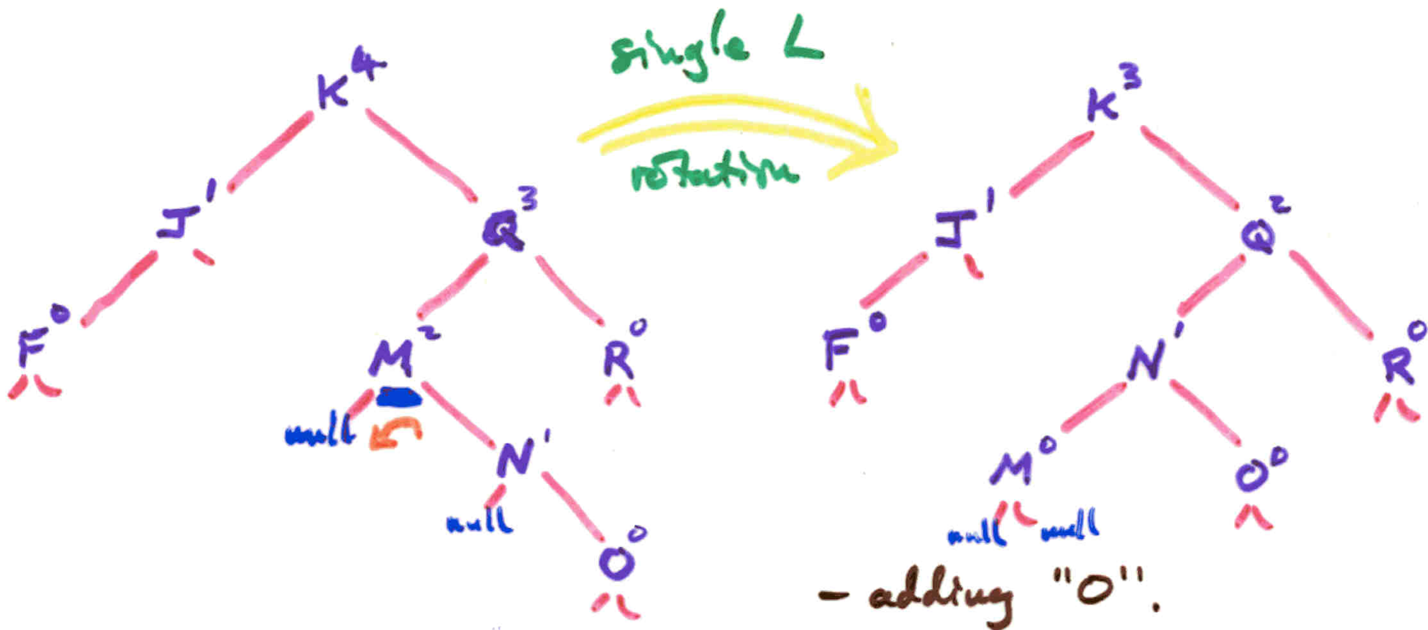
It doesn't matter whether Y' or Y'' is regarded as longer (they could both be equally long). So a double left rotation is really a single right rotation at the child level followed by a single left rotation at the parent level. Thus step 5/. becomes ...

5/.

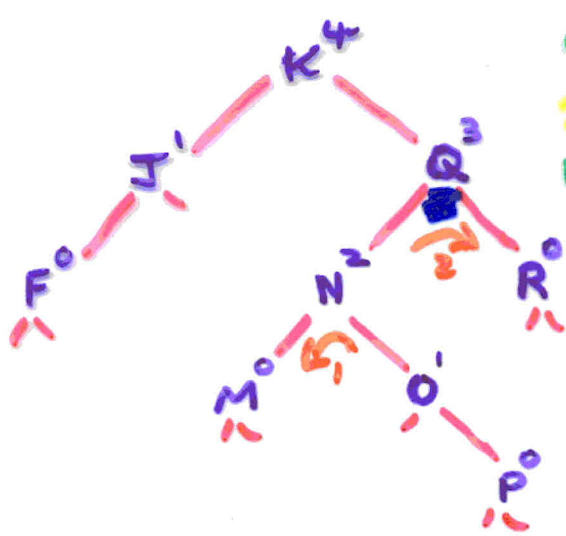
6/.

7/.

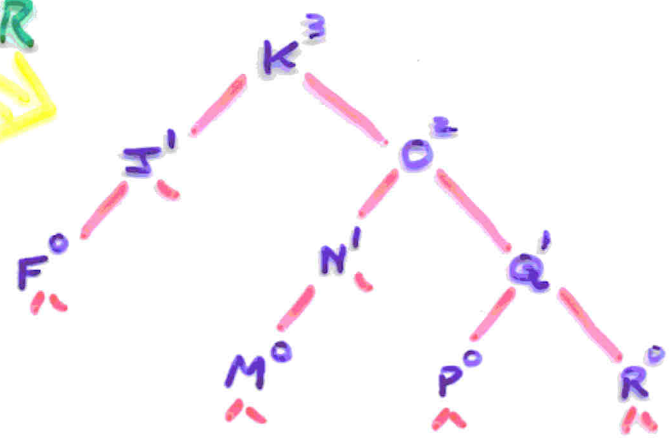
8/.



9/.

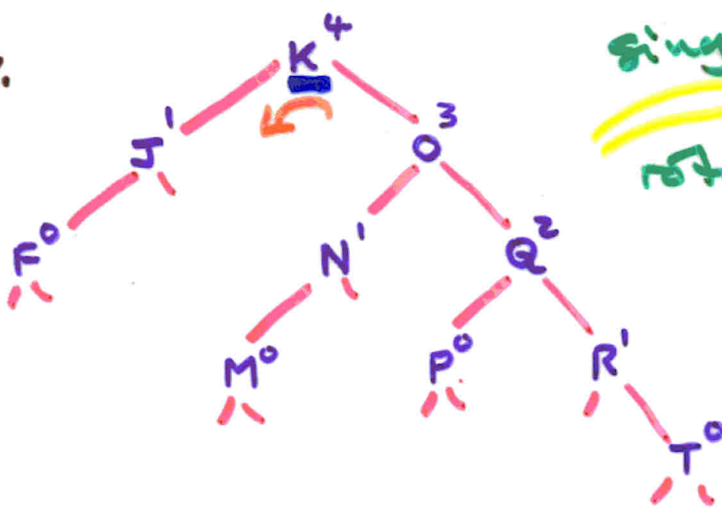


double R rotation

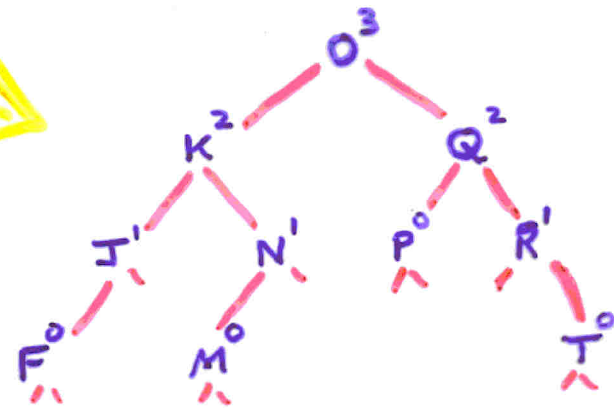


– adding "P".

10/.



single L rotation



– adding "T".

At each stage we underlined (in blue) the 'lowest' node which had excessively unbalanced subtrees. The natural question is how do decide whether to expect to do a single or double rotation. The effective answer is to perform a single rotation if the additional length is to the <u>outside</u> of the child of the affected node, and to perform a double rotation if it's to the <u>inside</u> of that child.

All these operations are $O(1)$ since the adjustments are complete when the single or double rotation is done. The overall bound on the height of an AVL tree of n terms is $\leq 1.44 \log(n+2)$.

Red-Black Trees

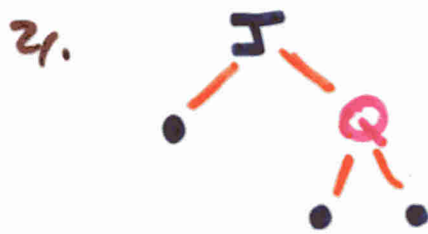
This provides another algorithm for maintaining approximate balance while manipulating a BS Tree. We start by modifying our BS Tree so that in place of any null children we have a formal external node; a non-null object which can hold values. Such a BS Tree is said to be extended. Then we require...

1/. — root node is black.
2/. — all external nodes are black.
3/. — no root to external node path has consecutive red nodes.
4/. — all root to external node paths have the same number of black nodes.

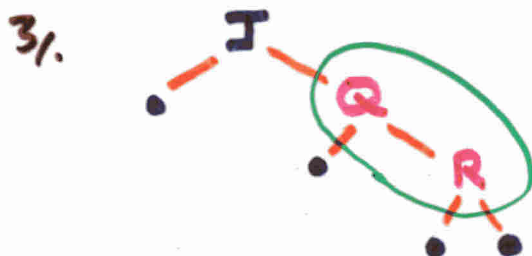
This approach will yield a tree of height $\leq 2 \log(n+1)$. Illustrating with the same sequence as before...

1/.

— adding "J". The root node has to be black, and we denote external nodes by big dots.

2/. 

— adding "Q". We always start by selecting red for newly inserted nodes since otherwise we would guarantee to break condition #4.

3/. 

— adding "R" has broken condition #3. Resolving this depends on classifying the nature of the imbalance.

Problem :- ∃ two consecutive red nodes.

Classification :-

- call these nodes p(n) and n, both red. ← parent of node n
- let g(n) be the grandparent of n. This must exist and must be black.
- there are two cases:

case IR if the other child of g(n) is red.

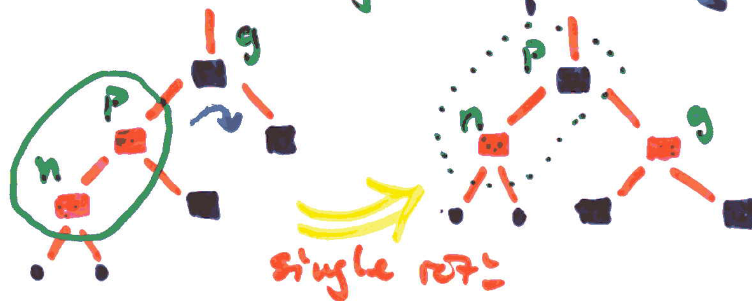
- toggle the colours of g(n), p(n) and g(n)'s other child. This correction might propagate up the tree.

The same recolouring will work independently of whether n is <u>inside</u> or <u>outside</u>.

$O(\log n)$

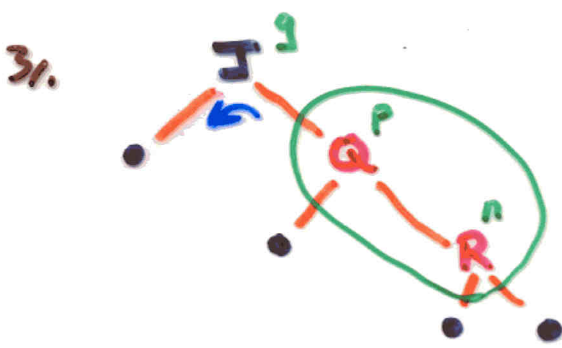
case IB if the other child of g(n) is black.

- rotate singly or doubly as for AVL trees, swapping the colours of p(n) and g(n) or n and g(n) respectively. This correction does <u>not</u> propagate up the tree.



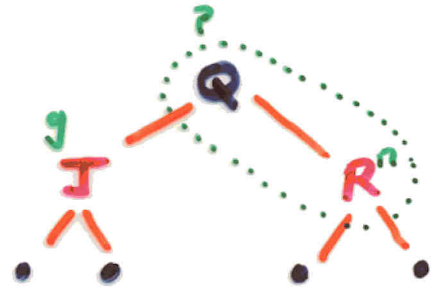
Returning to our example . . .

3/.



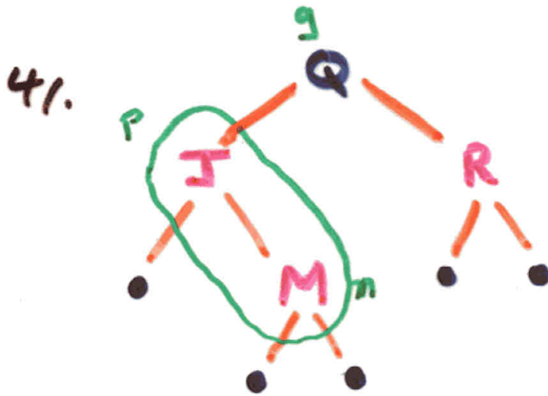
case IB

single

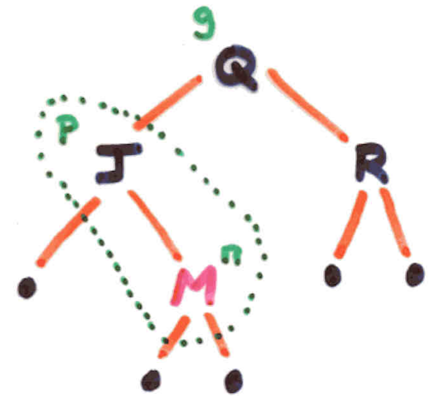


– adding "R".

4/.



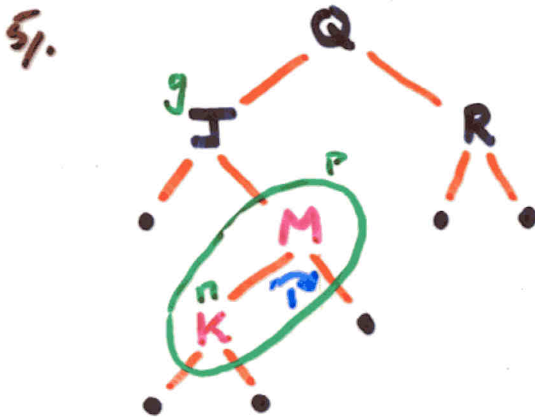
case IR



– adding "M".

Here we can't change the colour of g since it's the root node, so we must make J black, but then R must also be black by condition #4.

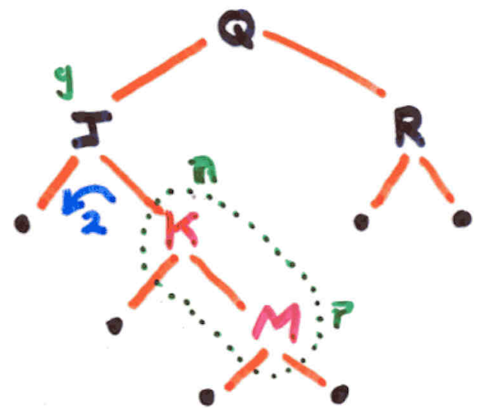
5/.



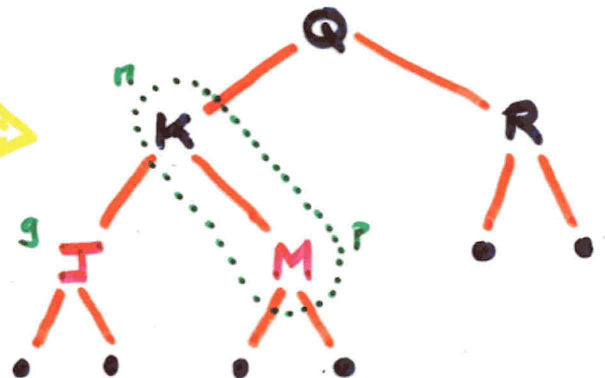
case IB

double

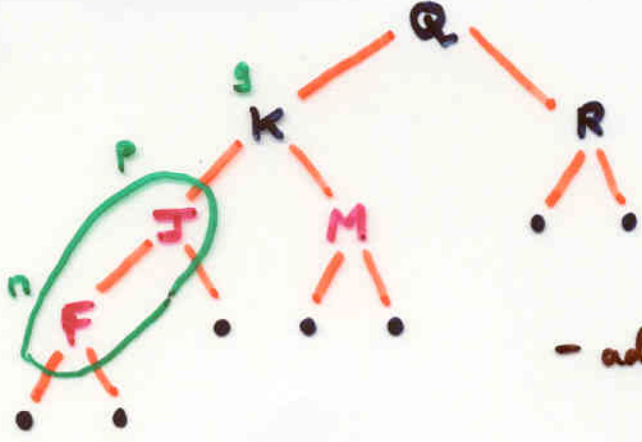
– adding "K".



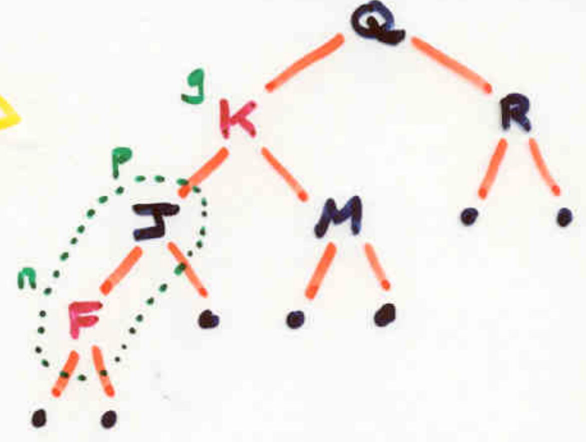
continued



6/.

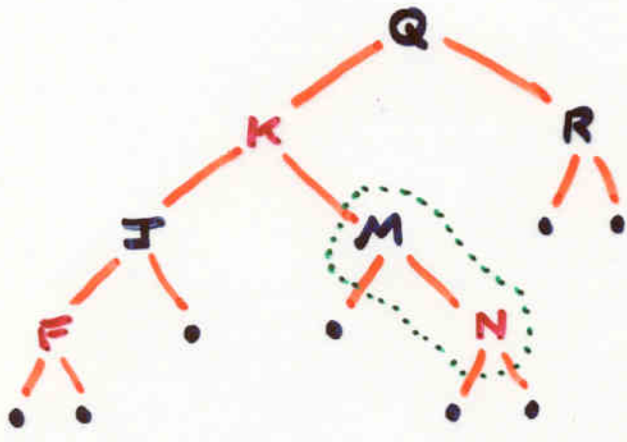


case IR



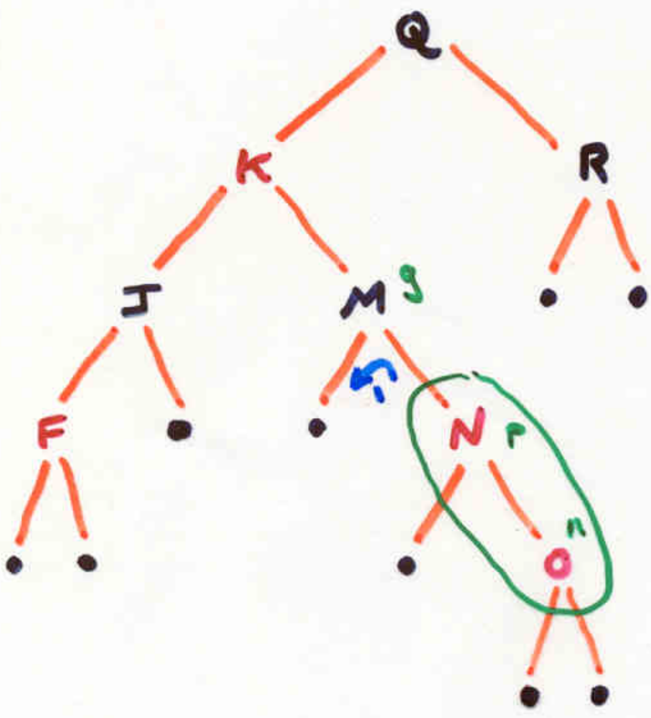
– adding "F".

7/.



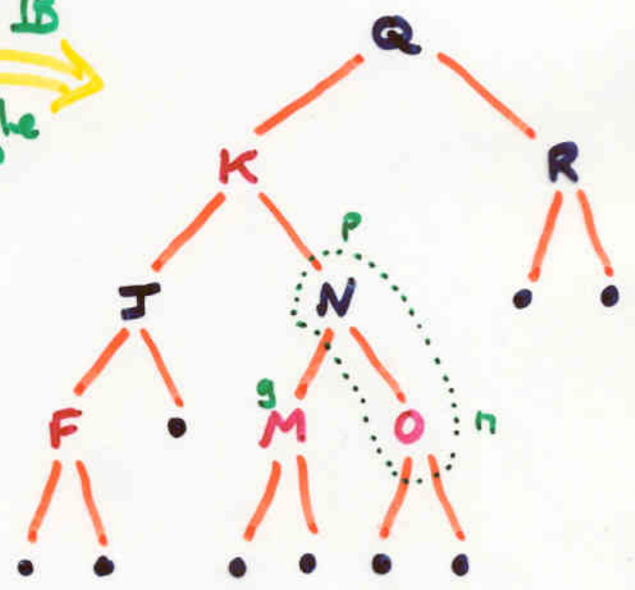
– adding "N". No further adjustments needed this time.

8/.



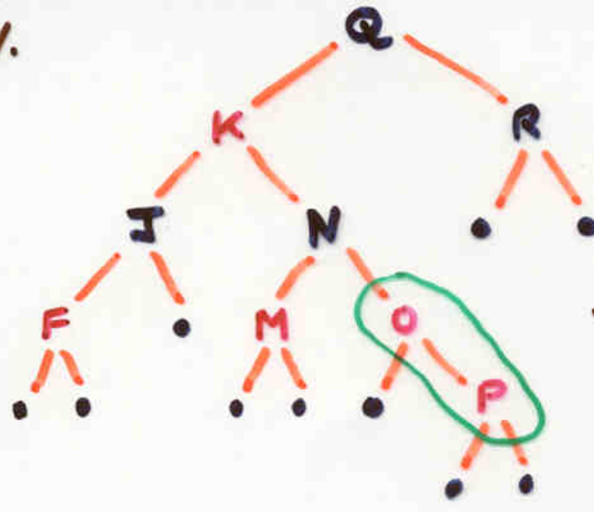
case IIB

single



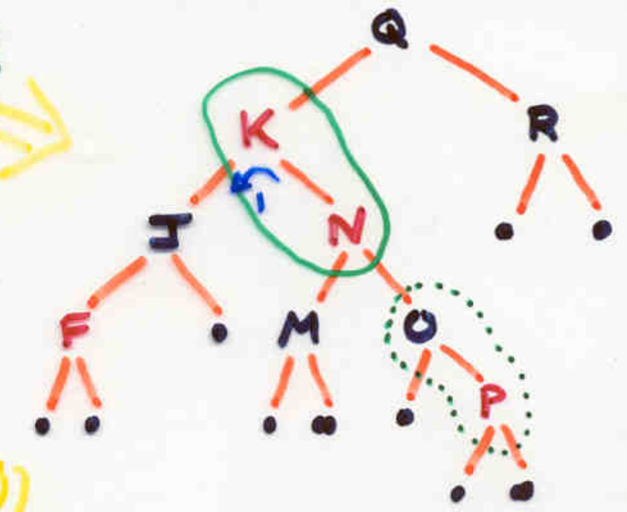
– adding "O".

9/.



case R

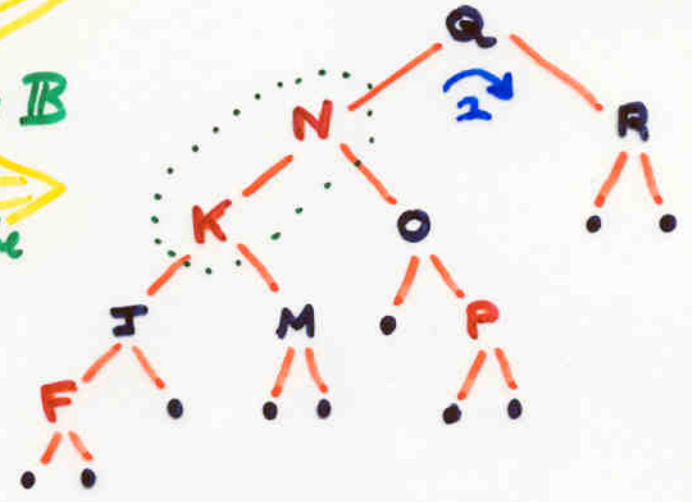
– adding "P".



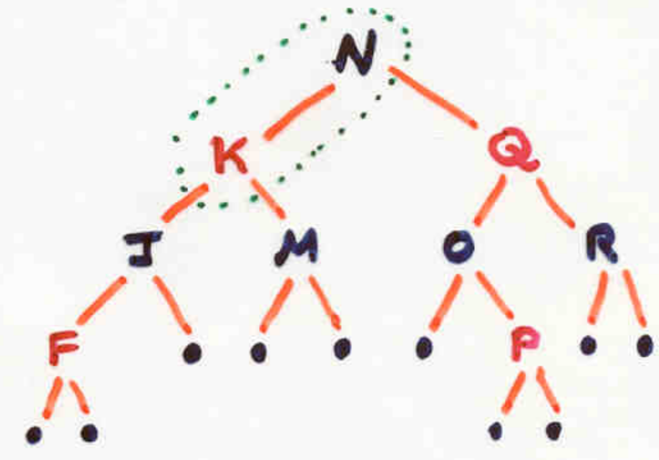
Notice that this is an example of the re-colouring causing the problem to propagate up the tree.

case B

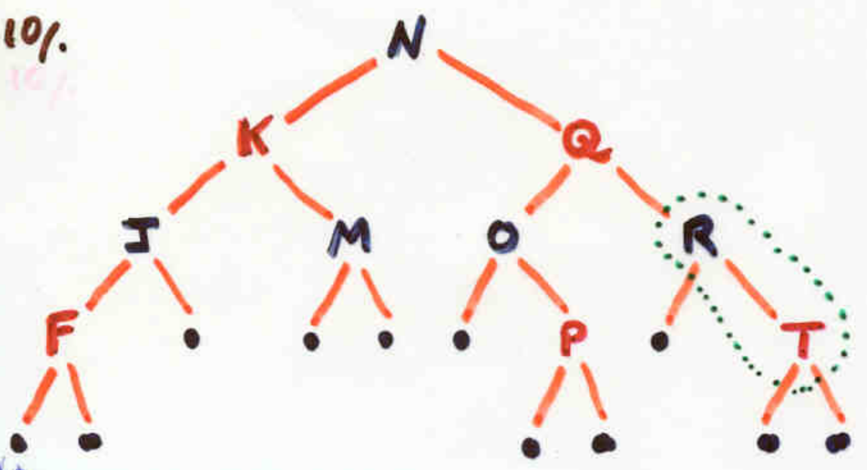
double



(continued)



10/.



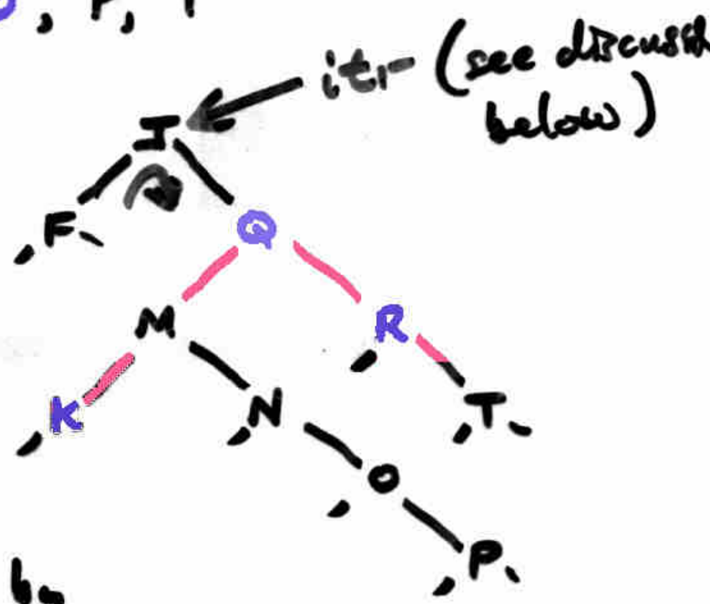
– adding "T". No further adjustments needed.

Deletions in red-black trees behave similarly, though watch out for condition #4.

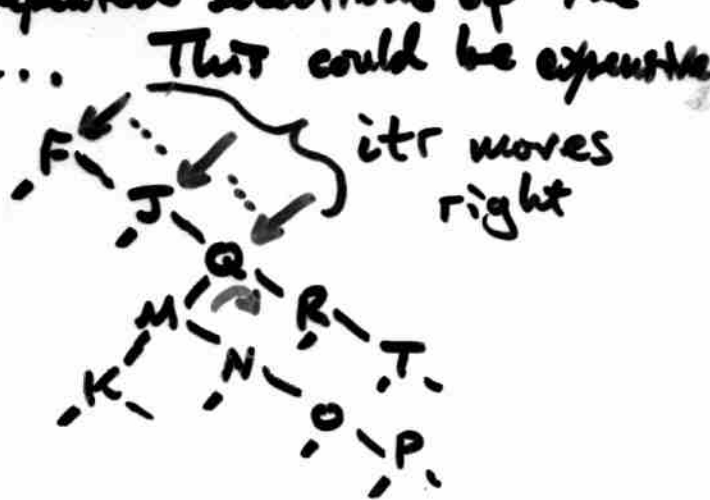
How about global re-balancing of a BS Tree?

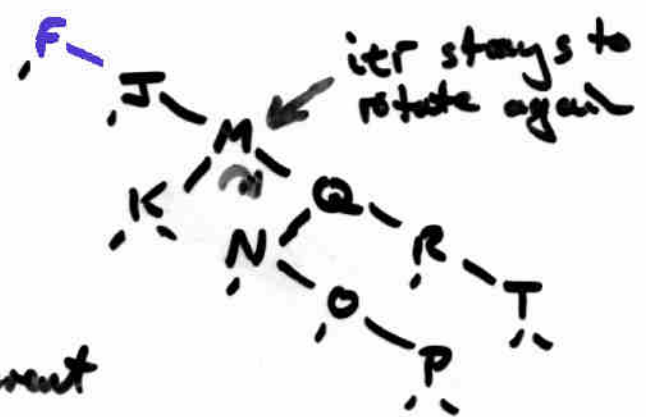
J, Q, R, M, K, F, N, O, P, T

gives the default BS Tree ...



One obvious approach would be to decant this tree into a sorted list (by repeated calls to findMin() or by an initial call to findMin() followed by a separately written L-R traversal method), and then build the balanced tree by repeated selections of the median, quartiles, octiles, ... This could be expensive for memory, so there's a nifty algorithm ascribed to Day, Stout and Warren which works with the tree in situ by using successions of single rotations.





The first step is to create a spine, essentially by...

```
if (itr.left != null)
    rotate right
    itr = child which became parent
else itr = itr.right
repeat until itr == null
```

Notice that the DSW algorithm really uses a tree iterator moving through the tree.

itr moves right

These will be the pivot locations for the first set of left rotations in step 2, starting with the lowest

null

Now that we have the spine we were after, we move to the second step, which is a collection of left rotations on odd-numbered nodes (starting counting at 1). The very first run is (possibly) exceptional since it deals with what will become the bottom partial level if the number of terms isn't exactly triangular.

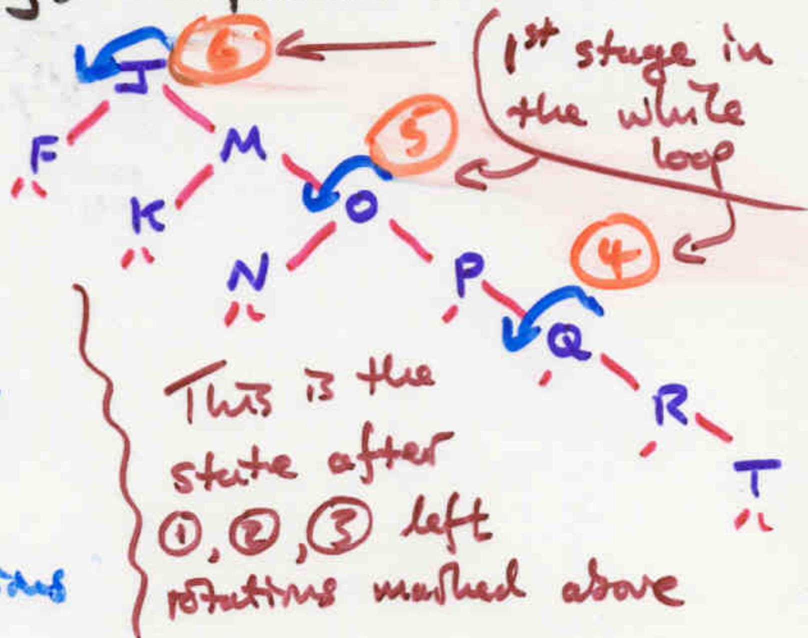
$m = 2^{\text{floor}(\log(n+1))} - 1$

make $n - 2^m$ odd rotations

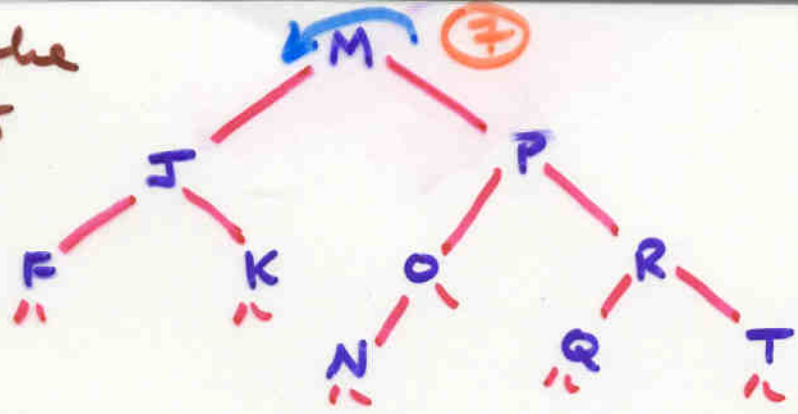
while ($m > 1$)

  $m = m/2$

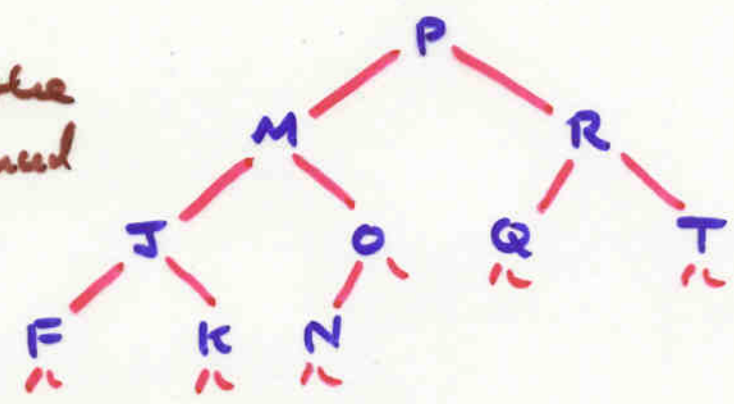
  make m odd rotations



This is the state after ④, ⑤, ⑥ left rotations marked above



This is the final balanced tree after ⑦ left rotation as marked above



Notice that it's quite important to do the first few left rotations before embarking on the while loop. Also, it's important to work backwards once the rotation pivots have been selected, if we're to finish with a perfectly balanced tree (or at least as well-balanced as possible).

Generalising BS Trees

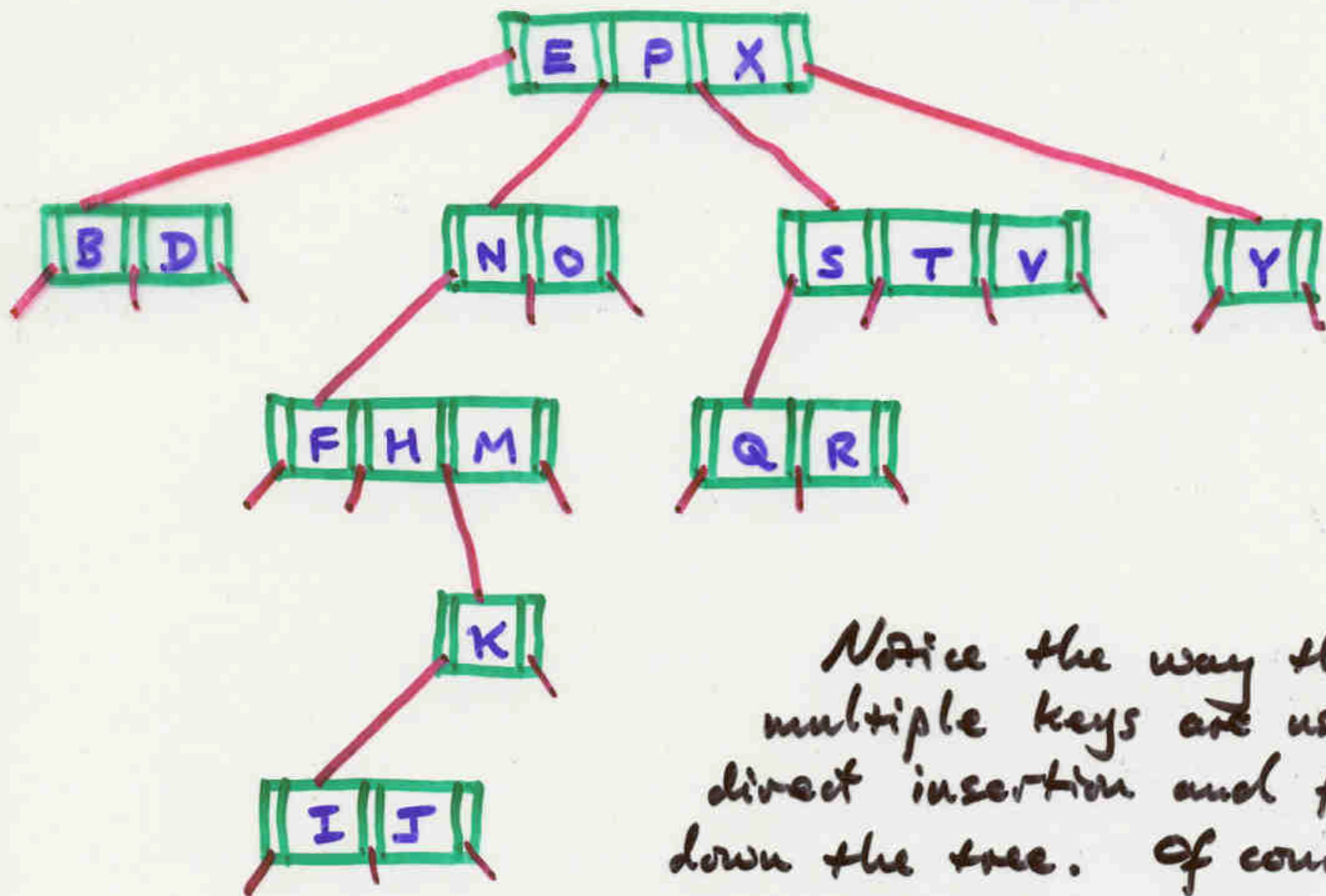
When dealing with very large data sets, there are compelling reasons to prefer broader yet shallower data structures. If we think of a BS Tree having nodes, each of which has ≤ 2 children and ≤ 1 key data; then a natural generalisation would be to have nodes, each of which would have $\leq m$ children and $\leq m-1$ key data values.

Such an animal is called an **m-way search tree**.

For example, if we put the sequence ...

E, B, P, N, O, X, D, S, F, H, Q, M, T, K, R, V, I, Y, J

into a 4-way search tree, we would get ...



Notice the way that multiple keys are used to direct insertion and find down the tree. Of course, this structure raises the same questions about balance as we saw with the 2-way (binary) search trees.

It is worth noting that if an m-way tree is balanced and full, then it too will have log (n) height, but the base of the log will now be m. This can have a dramatic effect on the height; for example, if n = 1000, a 2-way (binary) tree would have height about 10 whereas a 10-way tree would have height about 3 !!!

There are some important special cases. An m-way search tree is a **B tree of order d** if ...

1/. the root has ≥ 2 subtrees (unless it's a leaf)

2/. non-root non-leaf nodes each have k references (pointers) to subtrees (and hence $k-1$ key values) where

round up to next int if needed

$$d = \lceil m/2 \rceil \leq k \leq m = 2d - (m \% 2)$$

3/. non-root leaf nodes each have $k-1$ key values, where k is as above.

4/. all leaves are on the same level.

The value d above is also called the minimum degree of the B-tree, and it's unfortunately worth noting that some authors define m to be the order of the tree.

The effect of all this is that a B tree is perfectly balanced, relatively shallow, and more than half-full. (Note that the 1st and 3rd of these statements might not always be true!)

We'll build a B tree of minimum degree 3 (so it's at max a 5-way tree) using the same longer sequence we used for the 4-way search tree example a couple of pages back.

1/.

– note that this is both root and a leaf.

2/.

– note the 'correct' placing of the newly arrived "B" relative to "E".

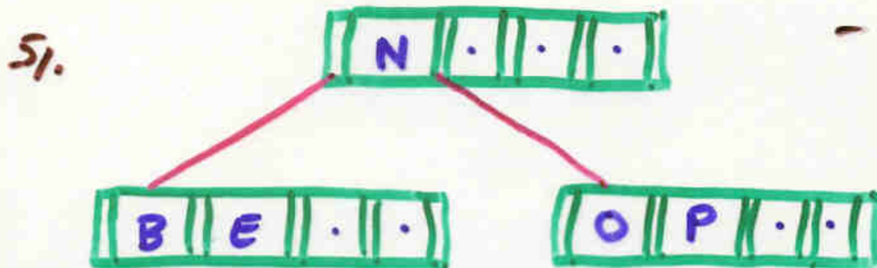
3/.

– adding "P".

4/.

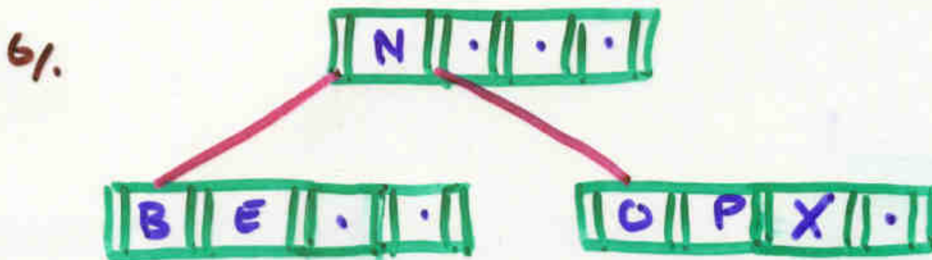
– adding "N".

5/.



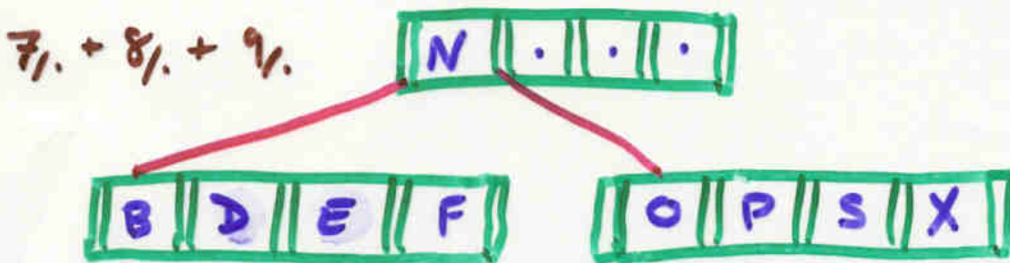
– adding "O". Note that to handle the overflow in the leaf we spawned a new leaf plus a parent node – forced by #1 & #2

6/.



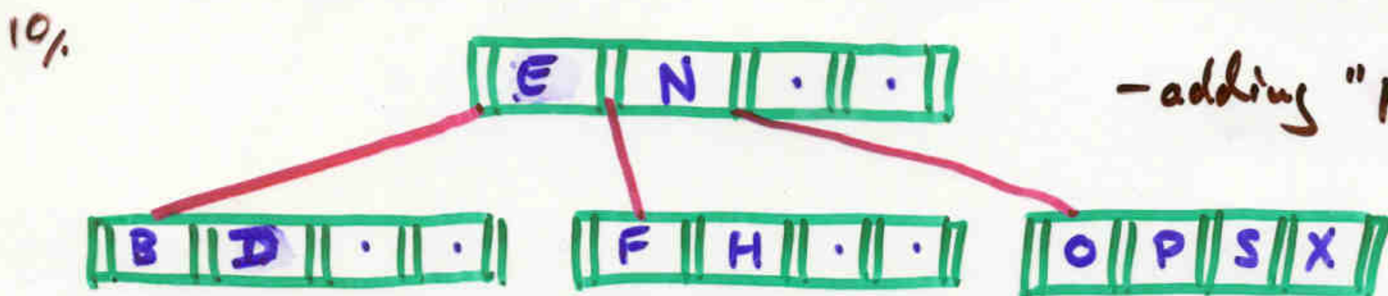
– adding "X". This time we've followed the usual search tree approach to find the appropriate node.

7/. + 8/. + 9/.



– adding "D", "S" and "F" is similar to 6/.

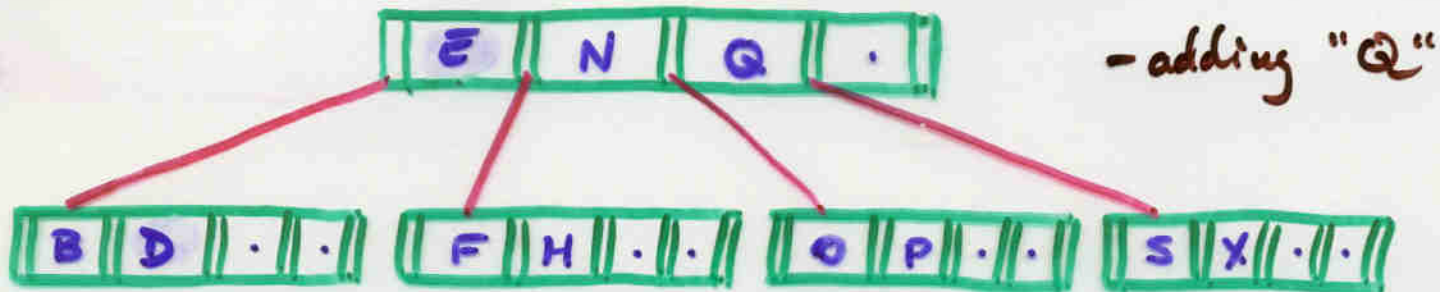
10/.



– adding "H"

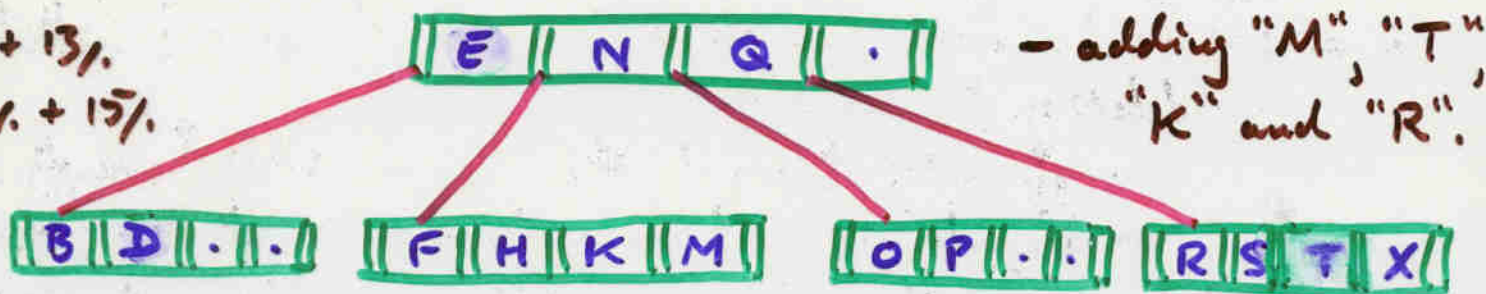
Note that this is similar to adding "O" in 5/., but that we didn't need to create the parent this time.

11/. – adding "Q"

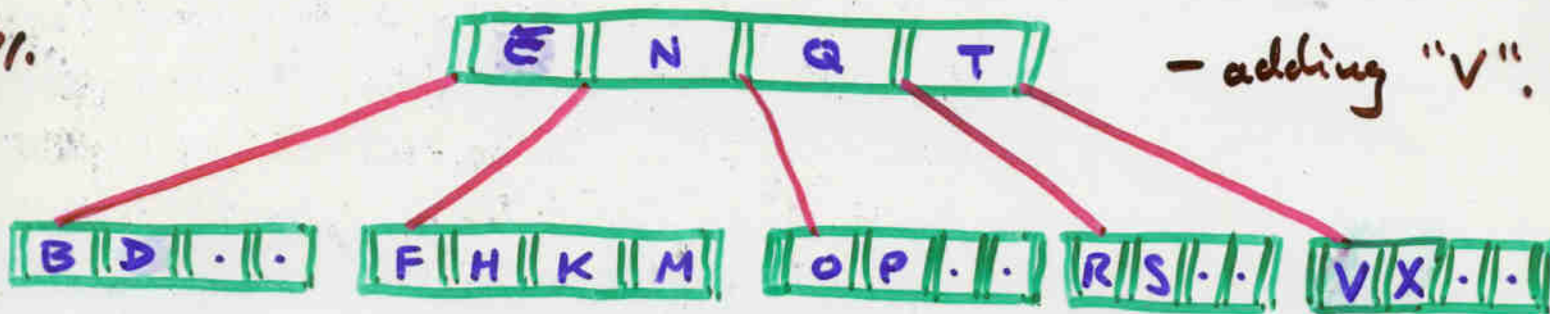


Note that adding "Q" caused an overflow in the third leaf in 10/., forcing the creation of a new leaf and the 'promotion' of the median as before. If m is even (it's 5 in our example!), then we spawn a new leaf immediately before adding the overflowing term, and then assign that term to whichever child node it should now go to.

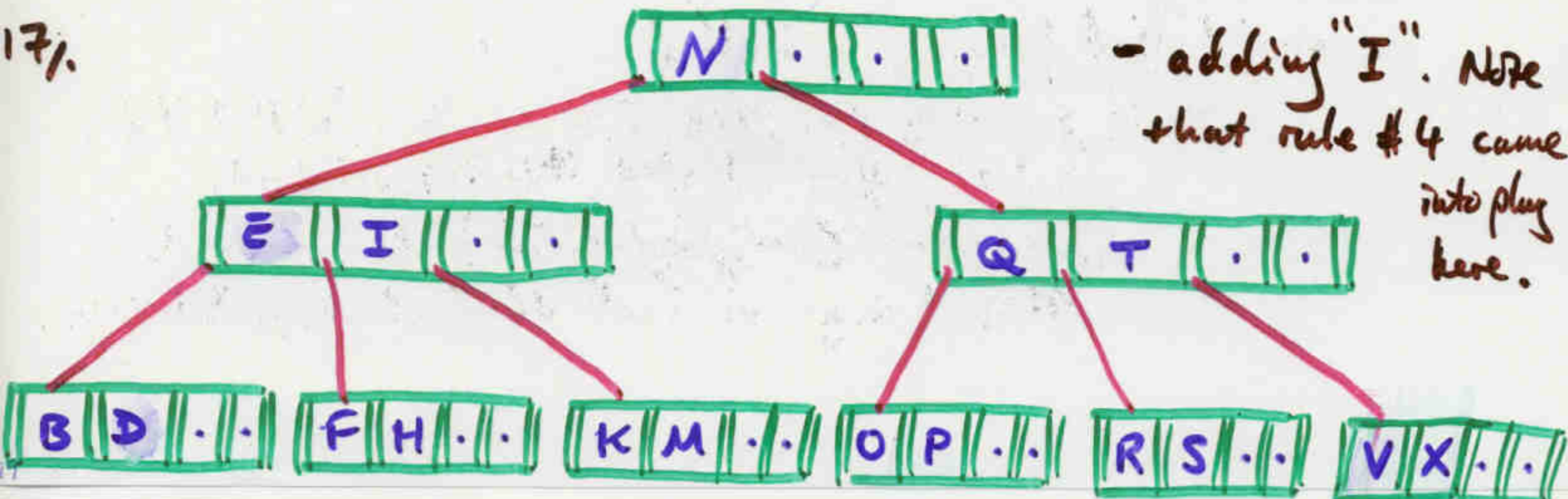
12/. + 13/. + 14/. + 15/. – adding "M", "T", "K" and "R".



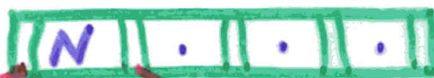
16/. – adding "V".



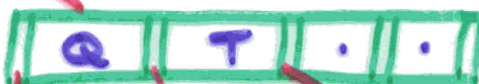
17/. – adding "I". Note that rule #4 came into play here.



18%. + 19%.



– adding "Y" and "J".



Note that each addition is passed all the way down to a leaf — the non-leaf nodes are only created as needed to hold medians of overflows.

Deletion from a B tree follows a similar pattern, where care has to be exercised if there is a danger of underflow.

- deletion from a leaf
  - if rules #2 and #3 are not violated, simply delete
  - if rule #3 will fail (the node will be too empty)
    - if a L or R sibling has terms to spare then redistribute through the parent key.
    - if neither adjacent sibling has enough terms, then merge two siblings together with the parent key. Note that this approach might propagate upwards.

- deletion from a non-leaf
  - to preserve the tree organisation, do this by replacing the value by its immediate successor (or predecessor) from a leaf. This brings us back to leaf key deletion.

So far, all our attention has been focussed on building various data structures to hold objects. Of course, moving objects around is relatively expensive — it would be far better to build a system of indices, with the data structure manipulating an index rather than a collection of objects.

There can be three levels of detail: the object itself, a key value associated with that object, and in the middle possibly a data entry to facilitate object access. The three standard approaches to this, assuming k to be some key value, are that the data entry be ...

1/. k* , the actual data object.

2/. (k, id) , a pair where id is the record id of a data object.

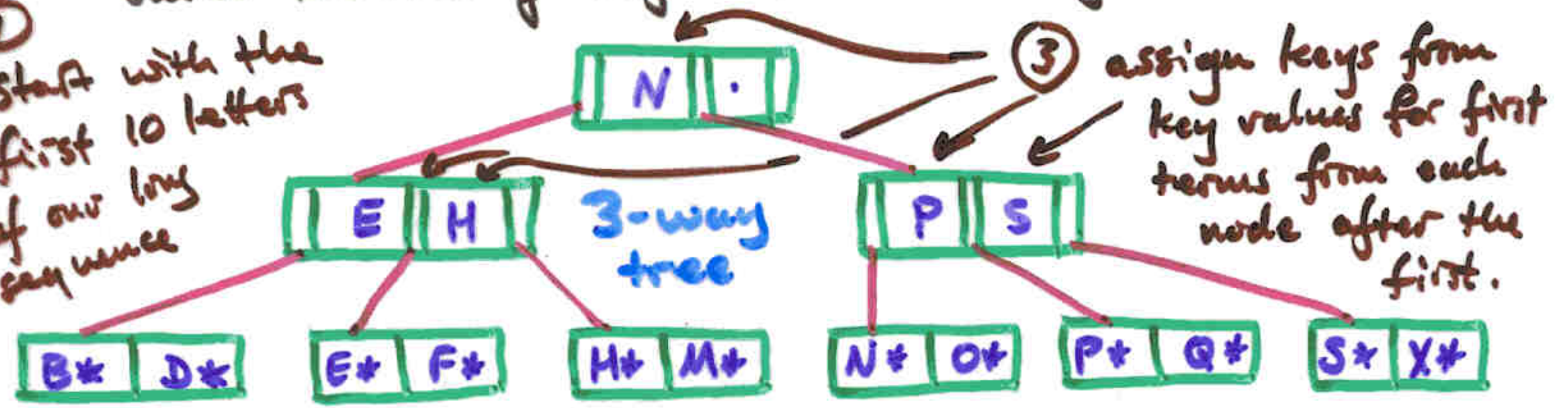
3/. (k, list) , a pair where list is a list of record ids of data objects.

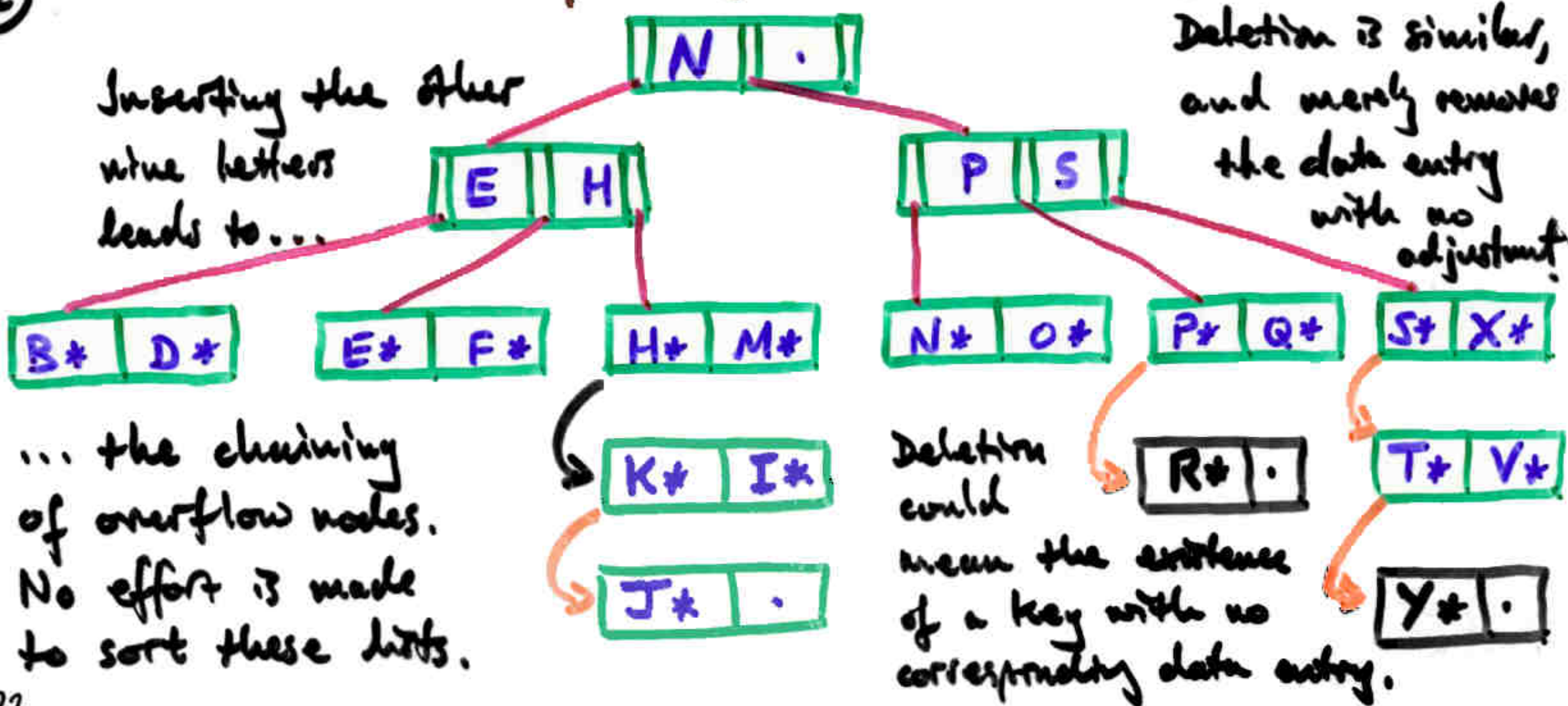
The intent is that searching for a key eventually yields a data entry in the index which leads us to the object. If we choose option 1/., then putting the keys k into a search tree will effectively sort the objects k*. However, if we then want to use a different key k′ for these objects, the new search tree will find the objects k* chaotically sited in memory.

There are many circumstances where one option wins over the others — it all depends on context. It's also worth thinking about the time complexities using a search tree to select data within some range of values of k. We'll consider two very standard structures ...

1. ISAM Trees

Here, ISAM stands for indexed sequential access method, and it works quite well in situations where the underlying data changes very little after initial construction. The crucial thing to bear in mind is that we are going to build a tree of key values (for non-leaf nodes) directing us to data entries in all the leaf nodes which point to the actual objects residing somewhere. The point of this structure is that once it's created, searching is efficient, so we create by first sorting and allocating sequentially the index leaves for all the then known data entries. After that, we build the tree of keys as an m-way search tree...



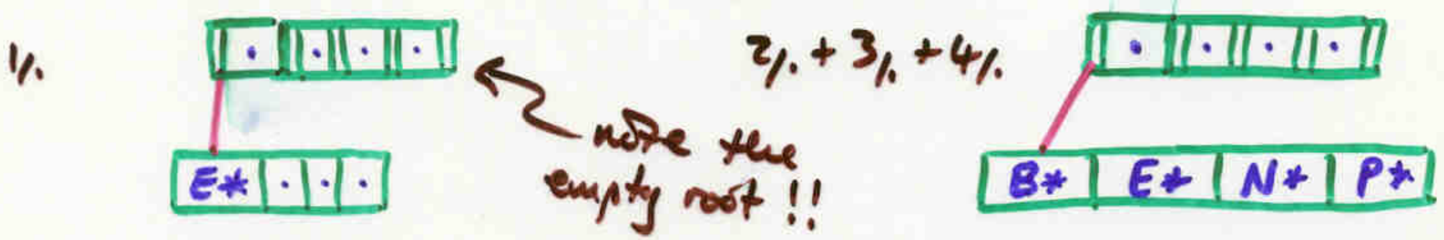


2/. B+ trees

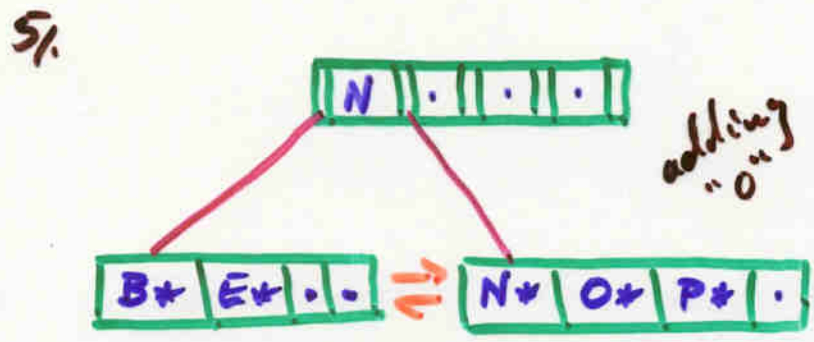
These provide a dynamic way of overcoming the static nature of ISAM trees, and are essentially an indexed version of B trees.

One of the big advantages of the sequential allocation in ISAM trees is that acquiring data entries within a range of values of the keys is very fast — simply run along the leaves once you've found the starting point. We mimic this for B+ trees by building a B tree of keys over the data entries (which again resides in the leaves), and then making the leaf nodes into a doubly-linked list.

Note that there is a slight twist on the B tree key-building for B+ trees due to the leaves carrying the only appearance of the data entries — if a leaf node fills up & spawns a new one, the data entries have to remain at the leaf level, even as the corresponding key gets promoted . . .

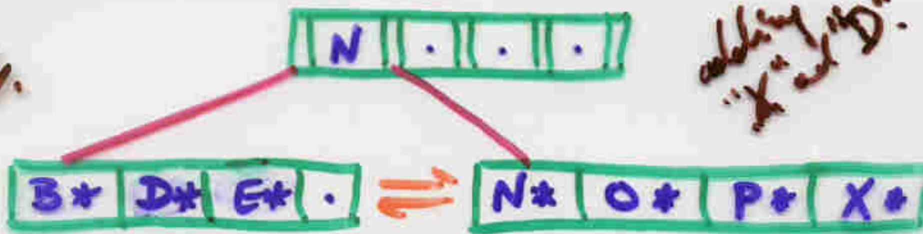


5/.



Notice that "N" lives in the key node and "N*" lives in the data entry node.

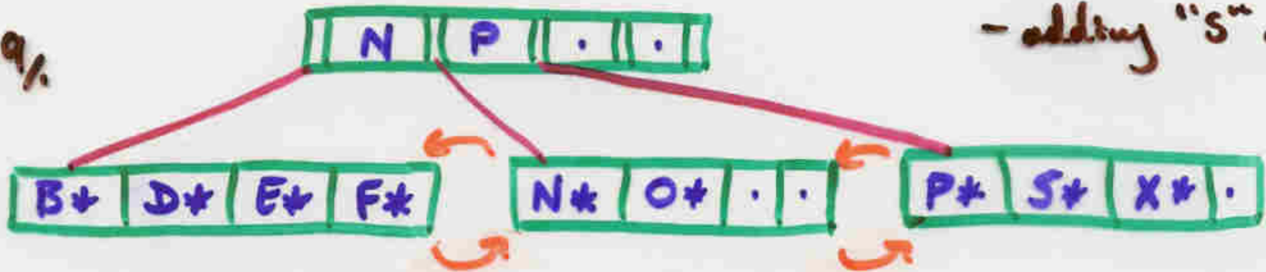
6/. + 7/.



adding "X" and "D".

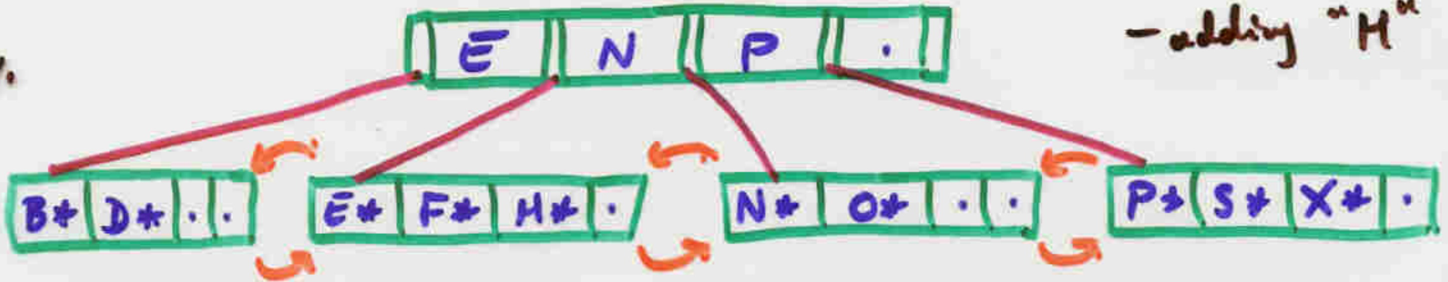
Notice that this leaf node has filled up sooner.

8/. + 9/.



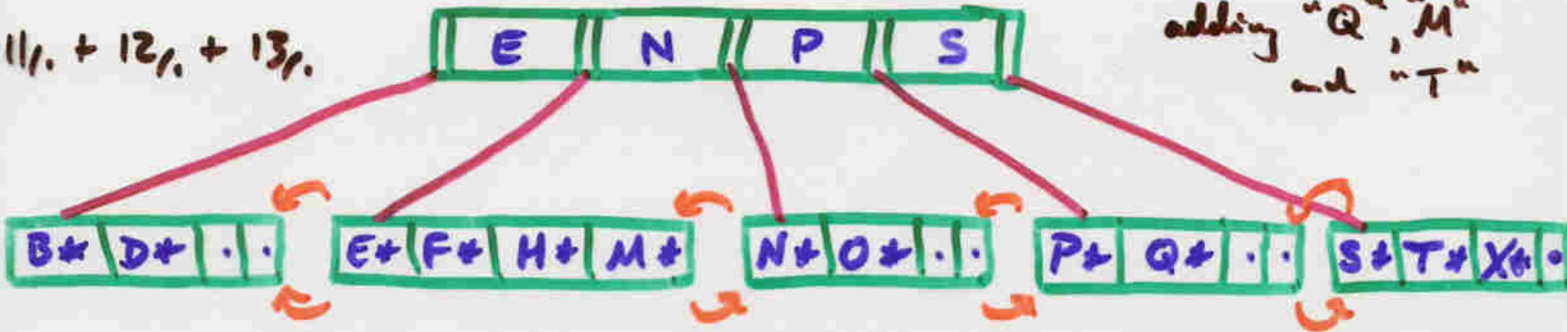
– adding "S" and "F".

10/.



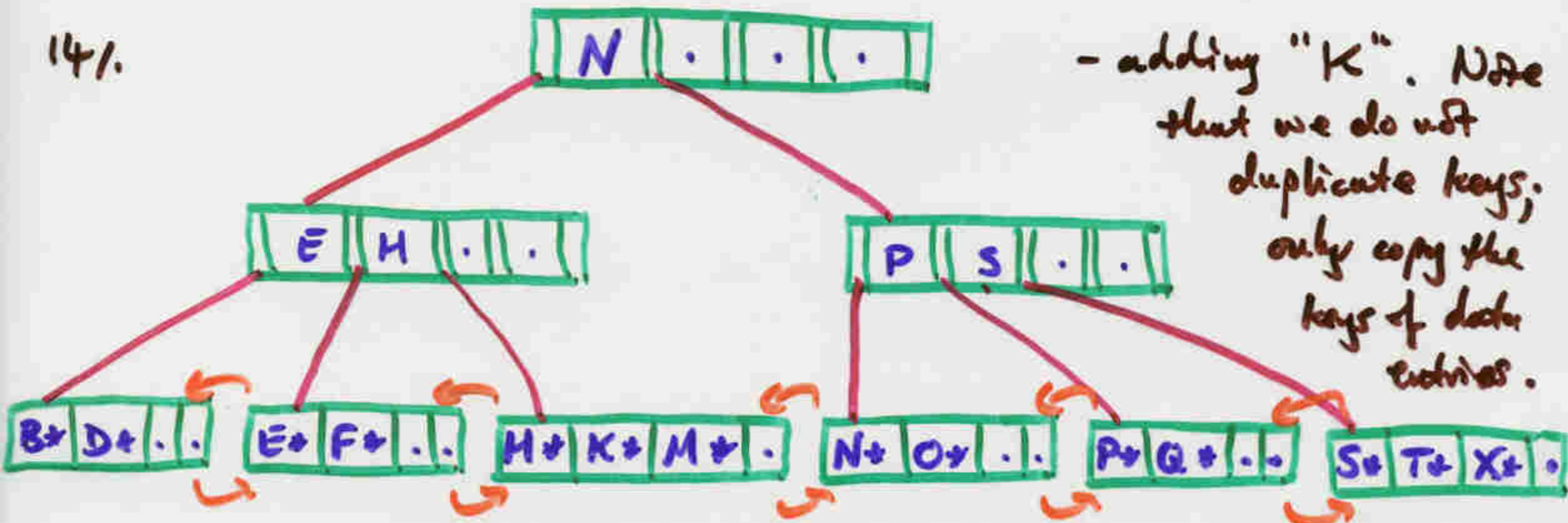
– adding "H"

11/. + 12/. + 13/.

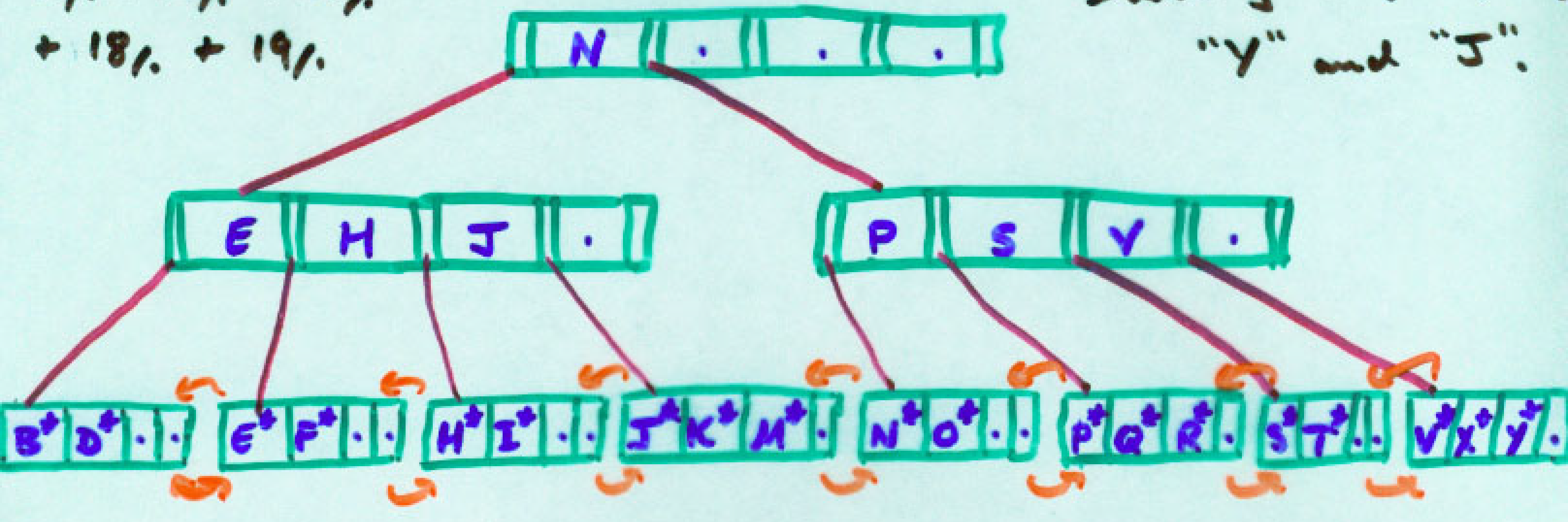


adding "Q", "M" and "T"

14/.



– adding "K". Note that we do not duplicate keys; only copy the keys of data entries.



Compare this with the previous B-tree representation.

Handling deletion is similar to the B tree case interpreted in this context.