CPS222 Lecture: Balanced Binary Search Trees

last revised March 23, 2015

Objectives:

1. To introduce (2,4) (also known as 2-3-4) trees and their red-black representation

Materials:

1. Slide show showing various operations on a Red-Black tree

I. Introduction to Balanced Binary Search Trees

A. We have seen that all operations on a binary search tree (lookup, insert, and delete) can be done in O(h) time, where h is the height of the tree. However, depending on the order in which insertions are done, h can vary from a minimum of log n to a maximum of n, where n is the number of nodes. Optimum performance requires that we somehow ensure that the tree is balanced, or nearly so.

B. Actually, there are two different ways of approaching balancing a tree:

1. Height balancing attempts to make both subtrees of each node have equal - or nearly equal - height. It results, then, in a tree whose height differs only slightly from the optimal value of log n.

2. Weight balancing takes into account the PROBABILITIES of accessing the different nodes - assuming such information is known to us. It puts the nodes that are accessed most frequently nearest the root of the tree.

3. In general, weight balancing is an appropriate optimization only for static trees - i.e. trees in which the only operations performed after initial construction are lookups (no inserts, deletes.) Such search trees are common, though, since programming languages, command interpreters and the like have lists of reserved or predefined words hat need to be searched regularly. Of course,

weight balancing also requires advance knowledge of probability distributions for the accesses.

C. For now, we will first consider various structures and algorithms for height-balanced trees. We will consider an example of a weight-balanced binary search tree later in the course.

D. One further note: you recall that earlier in the course we mentioned that there are two different ways of defining the height of a tree:

1. Measure height in terms of the number of EDGES on the longest path from root to a leaf - hence a one-node tree has height = 0 (and an empty tree has height -1).

2. Measure height in terms of the number of NODES on the longest path from root to a leaf - hence a one-node tree has height = 1 (and an empty tree has height 0.)

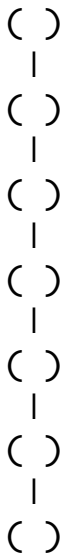
3. The author of your text uses the first definition. For discussions of binary search trees, the second definition is actually a bit more appropriate - since we do one key comparison at each NODE, the height of the tree, defined this way, turns out to be the maximum number of key comparisons needed when searching for a given value in the tree. (This also gives a more intuitive definition!)

II. Height-Balanced Binary Search Trees

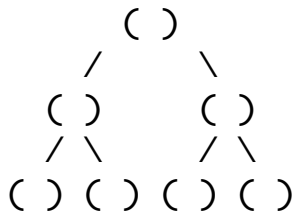
A. It is easy to show that a binary search tree containing n nodes can have a height that could be as great as n and as small as ceiling(log(n+1)).

Example: n = 7

Worst case



Best case



height = 7 (nodes) Height = 3 (nodes)

1. It is also easy to show, by experiment, that inserting keys into a tree in random order using an algorithm such as we have discussed previously typically results in a tree whose height is some small constant multiple of log n. (You will experiment with this in lab)

2. The problem comes in when we insert keys in a non-random order

   If we use the standard binary tree insertion algorithm we looked at when we studied BSTS earlier, what orders of insertion would result in a tree like our worst case?

   ASK

   There are many, but the most obvious are insertion in either strict alphabetic order or strict reverse alphabetic order! While the latter is unlikely, the former is quite possible if we are using sorted data to begin with!

3. Our goal, then, is to come up with some strategy that guarantees logarithmic height regardless of insertion order - typically by performing some rebalancing operation from time to time during

insertion (which operation must not increase the complexity of insertion beyond O(h)).

B. Several strategies have been devised for maintaining binary search trees with height O(log n), regardless of the order in which the keys are inserted. Three are discussed in the book:

1. The AVL tree strategy, so-called after its two inventors: Adelson-Velski and Landis.

2. The Splay tree strategy, which does not guarantee.that every individual operation will be O(log n), but guarantees that the average cost of all operations (the amortized cost) is O(log n).

3. The Red-Black tree strategy. (You'll see the reason for the name soon.)

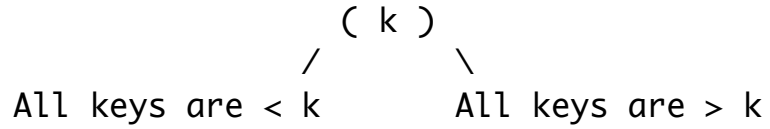
C. Since time does not permit discussing all the options in depth, we will only discuss the latter. This strategy is of particular interest because it is the one used by the TreeMap and TreeSet collections in Java and by the STL map, set, multimap and multiset containers.

III. (2,4) (Also Known as 2,3,4) Trees

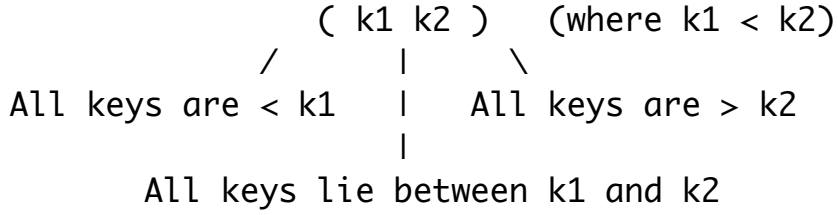
A. As it turns out, the best way to approach the Red-Black tree strategy is by first considering a search tree that is not binary called a 2-3-4 tree. (Your book calls this a (2,4) tree. We will then show how 2-3-4 trees can be realized by Red-Black binary trees, which are what is actually used in practice.

B. A 2-3-4 tree is a search tree in which each node has 2, 3, or 4 children and contains 1, 2, or 3 keys - e.g.

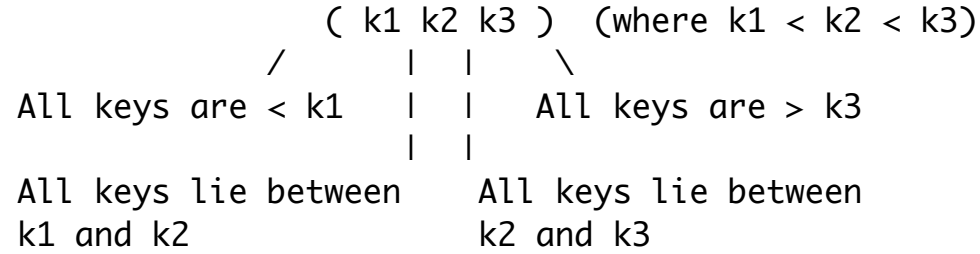
a) A 2 node:



2. A 3 node:



3. A 4 node:

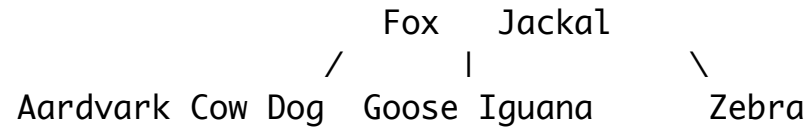


C. Such a tree can be searched by an extension of our binary search tree algorithms. We won't consider this, though, because we will eventually use a different representation for the tree.

D. Of more interest is the problem of maintaining the tree.

E. As always, we will insert new keys at the bottom of the tree. But now we have several possibilities:

1. If we encounter a 2 node or a 3 node whose children are all NULL, instead of creating a new node we will add the key to the existing node. Example - Insert Raccoon into:

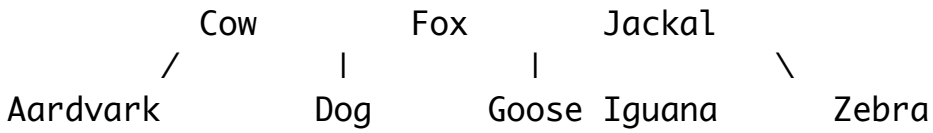


The rightmost node becomes: Raccoon Zebra

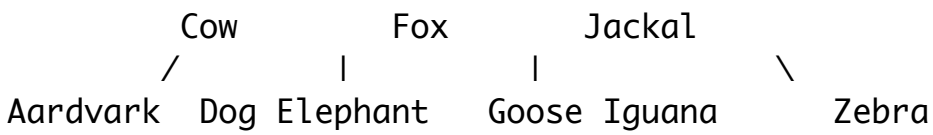
If, instead, we inserted Hippo, the center node would become Goose Hippo Iguana

2. If we encounter a 4 node whose children are all NULL, we need to SPLIT the 4 node into two 2 nodes, and pass the middle key up to its parent, and then insert the new key in the appropriate half - e.g. Insert Elephant the above tree (starting with the original tree):

   Split the four node, moving the middle key into the parent:



   Insert the new key into the appropriate half of the split node:



   (Note that the split is done BEFORE the insertion.)

3. However, there is one place where this strategy can get us into trouble. Suppose, when we split a four node, its parent is also a four node? Then, we have no room to insert the key in the parent.

   a) This can be handled by simply splitting the parent and promoting one of its keys - which in turn could cause another split etc. This can get messy.

   b) A cleaner alternative is to adopt a policy of ALWAYS SPLITTING any four node we encounter going down the tree on a search.

      (1) Even if we don't have to do so immediately, we will probably have to do so eventually.

      (2) This ensures whenever we split a node that its parent will be either a 2 node or a 3 node, with room for the promoted key.

4. One special case remains - the root. If the root of the tree is a four node, what do we do? (Where do we put the promoted key?)

   a) The answer is that we create a new two node to become the root of the tree, which adopts the two halves of the original root - e.g.

      Before:

      ```
      Cow   Fox   Jackal
      ```

      After:

      ```
            Fox
          /     \
      Cow         Jackal
      ```

   b) When this occurs, the overall tree increases its height by one. However, this will be a rather rare occurrence, since two promotions from below (each the result of splitting a four node) will have to occur before the root again needs to split.

5. One interesting property of a 2-3-4 tree is that it is ALWAYS height-balanced. In particular:

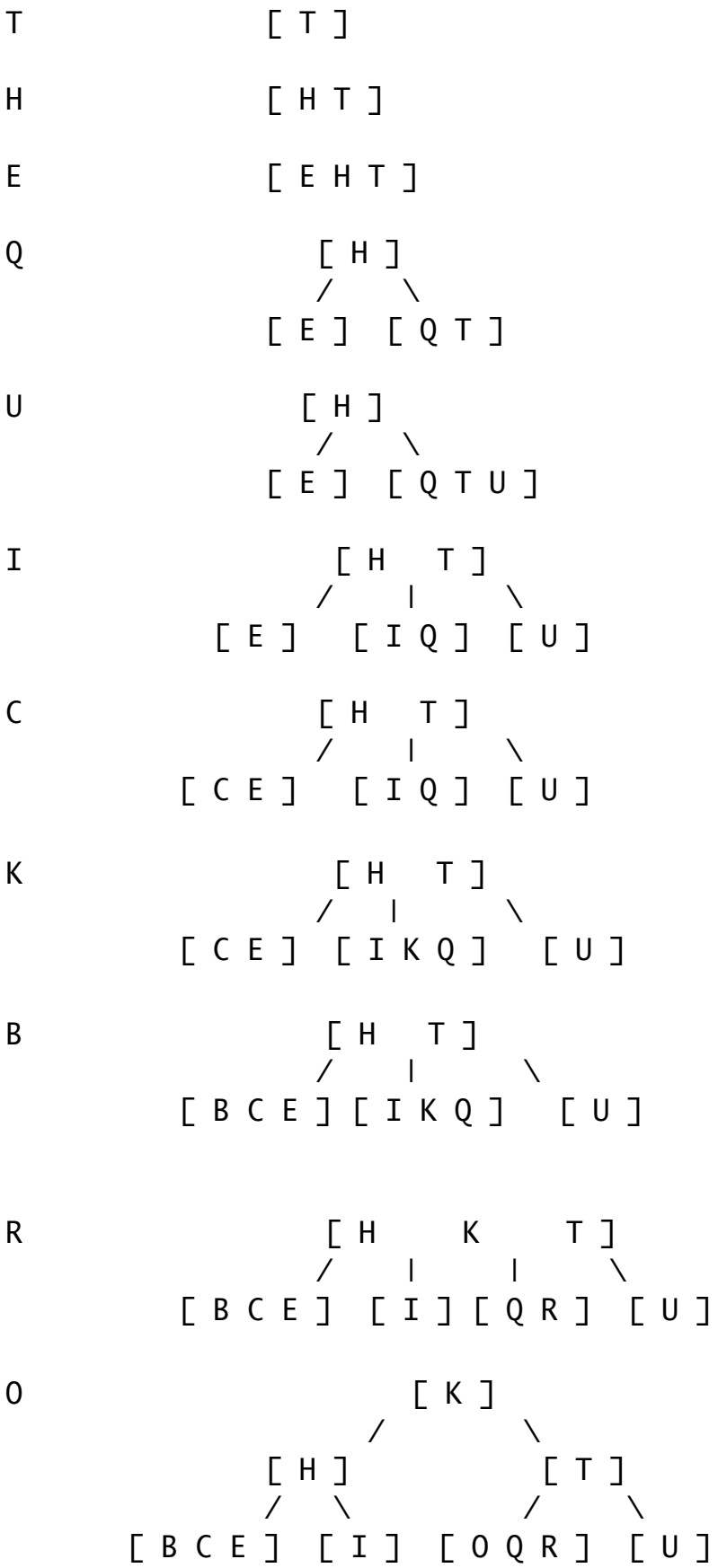
   a) Either all the children of a node are NULL or none of them are.

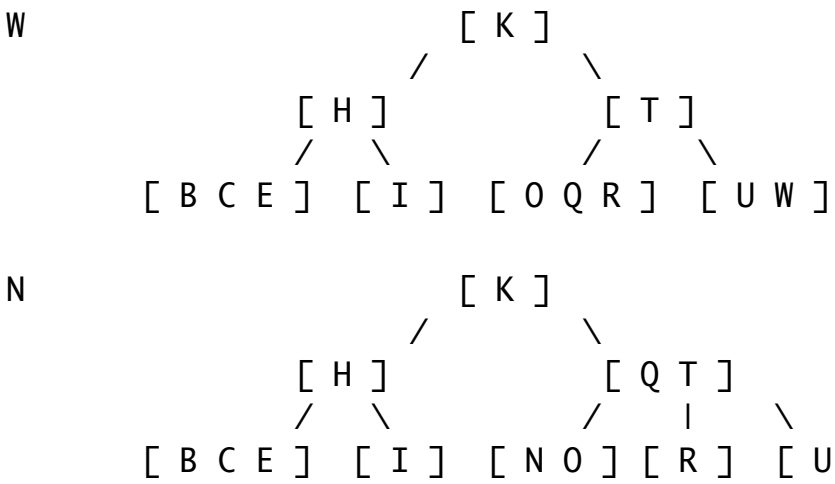
   b) All the nodes with NULL children are at the same level.

      This follows from the fact that we never insert new nodes into the tree; rather, we insert into or split existing nodes. The only way the height of the tree ever increases is when the root splits, and this affects all leaves equally.

      Example: insert T H E Q U I C K B R O W N into an initially empty 2-3-4 tree.

      Develop with class


T
[T]
H
[H T]
E
[E H T]
Q
[H]
[E] [Q T]
U
[H]
[E] [Q T U]
I
[H T]
[E] [I Q] [U]
C
[H T]
[C E] [I Q] [U]
K
[H T]
[C E] [I K Q] [U]
B
[H T]
[B C E] [I K Q] [U]
R
[H K T]
[B C E] [I] [Q R] [U]
O
[K]
[H] [T]
[B C E] [I] [O Q R] [U]



Question: what sequence of keys would most quickly lead to another insertion into the root?

Answer: X Y M P

Observe that this would not lead to splitting the root - which would require a great many more insertions

F. Let's analyze the efficiency of operations on a 2-3-4 tree.

1. Clearly insert and locate are O(h). (We don't consider delete here, because it is considerably more complex to implement; however, it too is O(h)).

2. What is the relationship between the number of KEYS in a 2-3-4 tree (n) and its height (h)? (Note: we worry about keys, not number of nodes.)

   a) To answer this question, we can consider the related problem of determining the MINIMUM number of keys in a 2-3-4 tree of height h. Clearly, such a tree would have every node a two node. Thus, we would have:

Level	Number of nodes at level	Number of keys at level	Total number of keys
1	1	1	1
2	2	2	3
3	4	4	7
4	8	8	15

So in the worst case, in a 2-3-4 tree of height h, the the total number of keys at all levels is equal to $2^h - 1$. That is, h is $O(\log_2 n)$ (since the -1 becomes vanishingly insignificant as h increases).

b) By similar reasoning on a MAXIMAL tree we get

Level	Number of nodes at level	Number of keys at level	Total number of keys
1	1	3	3
2	4	12	15
3	16	48	63
4	64	192	255

So in the best case, in a 2-3-4 tree of height h, the the total number of keys at all levels is equal to $4^h - 1$. That is, h is $O(\log_4 n)$ (since the -1 becomes vanishingly insignificant as h increases). But since $\log_4 n$ is $0.5 * \log_2 n$, we can regard h - and hence the effort for lookup, insert, and delete - as $O(\log_2 n)$ for any 2-3-4 tree, regardless of whether it is minimal, maximal, or somewhere in between.

IV. Red-Black Trees

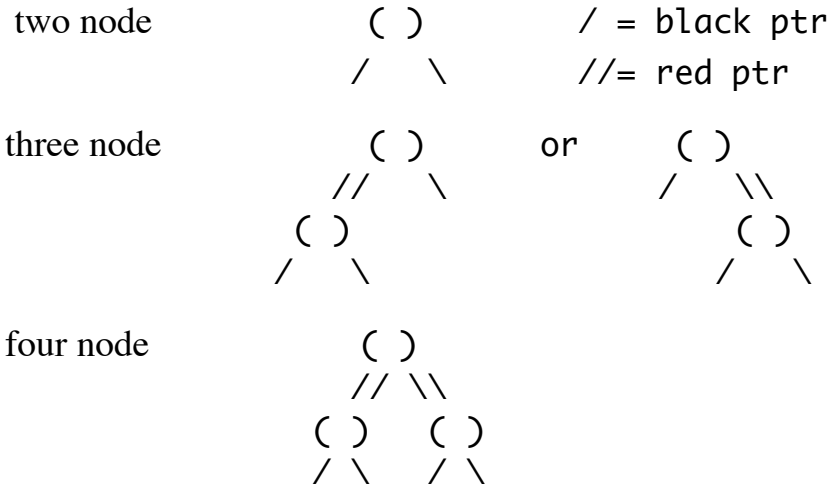
A. We have seen that 2-3-4 trees represent a nice way of maintaining a balanced search tree. Unfortunately, the code to maintain and use them is made complex by the fact that we have to deal with three different types of node.

One could use a single type of node with a small integer field that would store 2, 3, or 4 to indicate the number of children (or 1, 2, or 3 to indicate the number of keys). But that would result in wasting a lot of space - e.g. all nodes would need room for 3 keys, but a 2-node would only hold 1 key etc.

B. We now consider a type of tree called a red-black tree that implements the principle of the 2-3-4 tree more easily / efficiently.

1. A red-black tree is a binary tree in which we associate a COLOR (red or black) with each link. (This can be implemented by two extra bits in each node, or perhaps by a single tag bit on each pointer, using a strategy similar to that used for threaded trees.)

2. We can represent a 2-3-4 tree by a red-black tree as follows:



That is, a 2-3-4 tree node is represented by a binary tree of height 1 or 2, with red pointers used to maintain the internal structure and black pointers used to refer to other nodes.

3. This kind of tree has several interesting properties:

   a) Our lookup algorithm is the same as for an ordinary binary search tree.

   b) The number of black links on any path from root to leaf is the same for all paths in the tree.

   c) On any path from root to leaf, we never encounter two successive red links. Thus, the total number of links on any path from root to leaf is at most twice the number of links in the corresponding 2-3-4 tree. (This relates to our demonstration that the height of a 2-3-4 tree lies between $\log_2 n$ and $\log_4 n$.

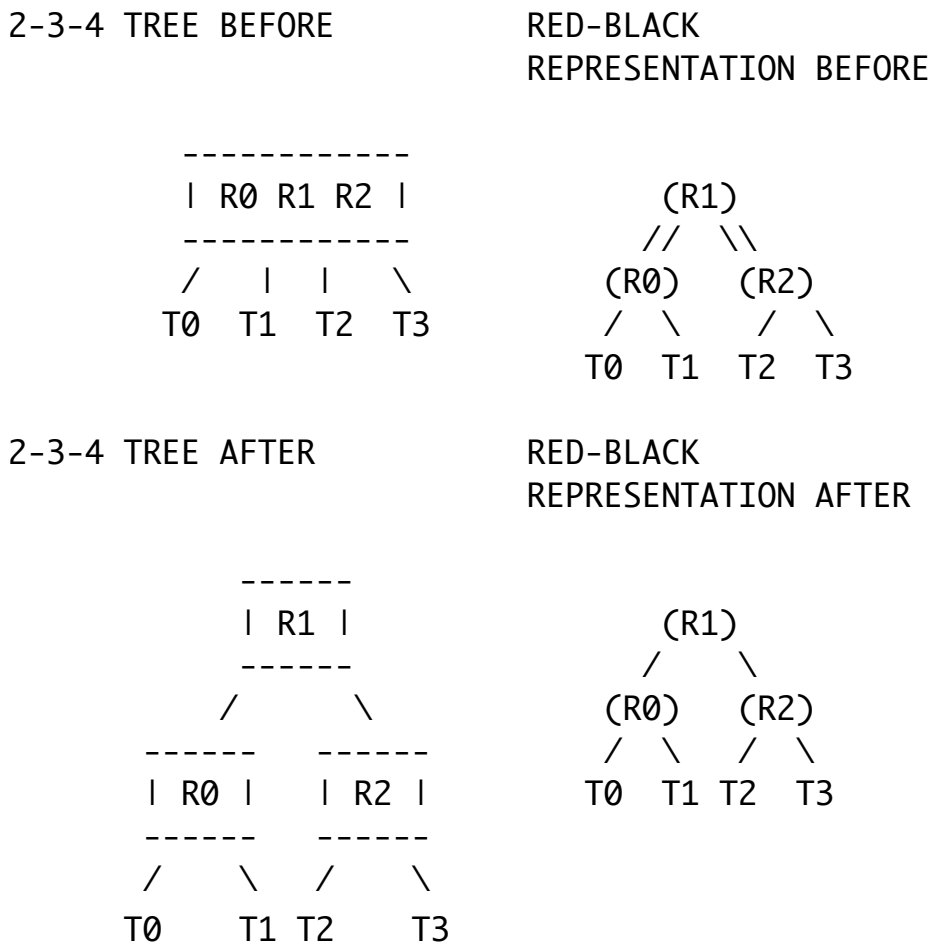
4. On insert, when going down the tree, if we encounter a node with two red children, it represents a four node and should be split. There are four cases, depending on the parent of the four node

   a) Note: most cases below are actually two separate cases, depending on whether the node being split is the left child or the right child of its parent. We consider only the left child case; the right child one is a mirror image.

   b) In the drawings, we use the following abbreviations

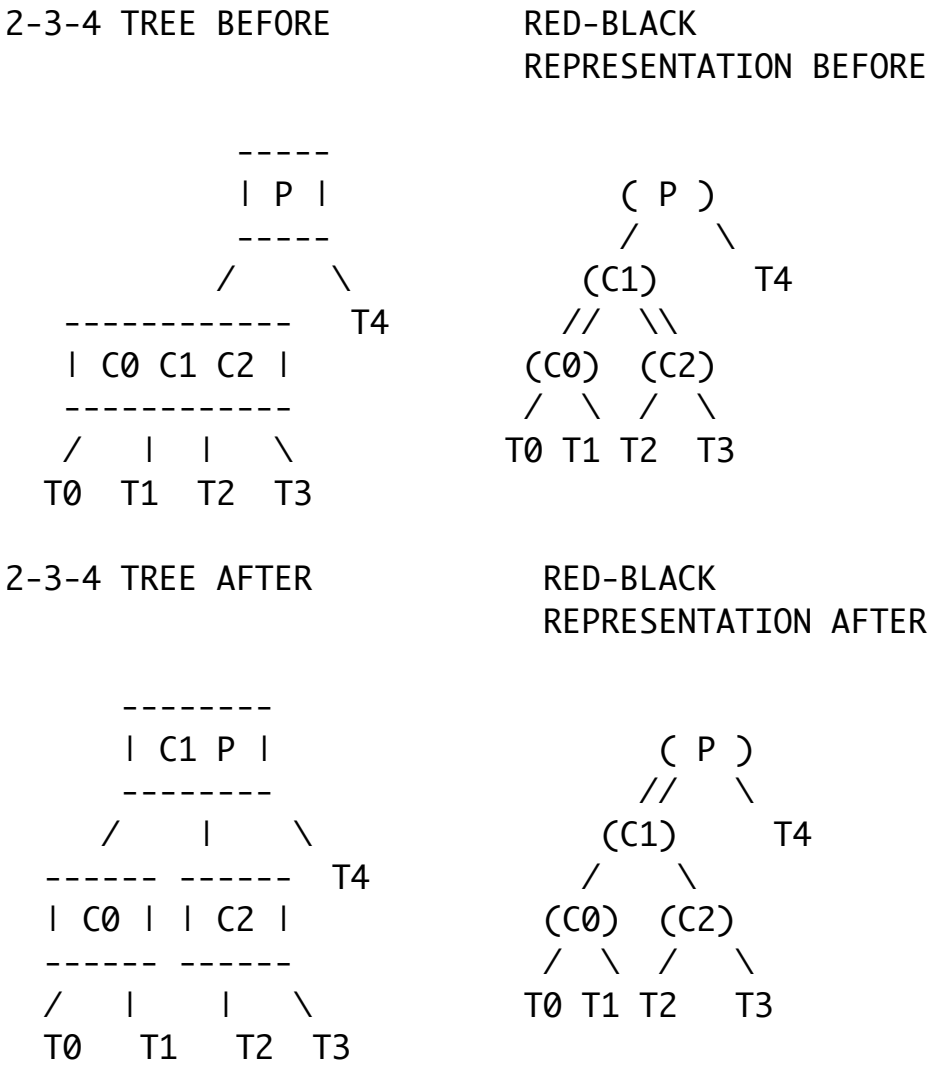
      R0, R1, R2: Keys in the root when it is being split
      C, C0, C1, C2: Keys in a child when it is being split
      P, P0, P1: Keys in the parent of a child being split
      T0, T1, T2, T3, T4, T5: subtrees of a node being split
      (can be of any allowed size)

c) There is no parent - the node being split is the root of the whole tree.



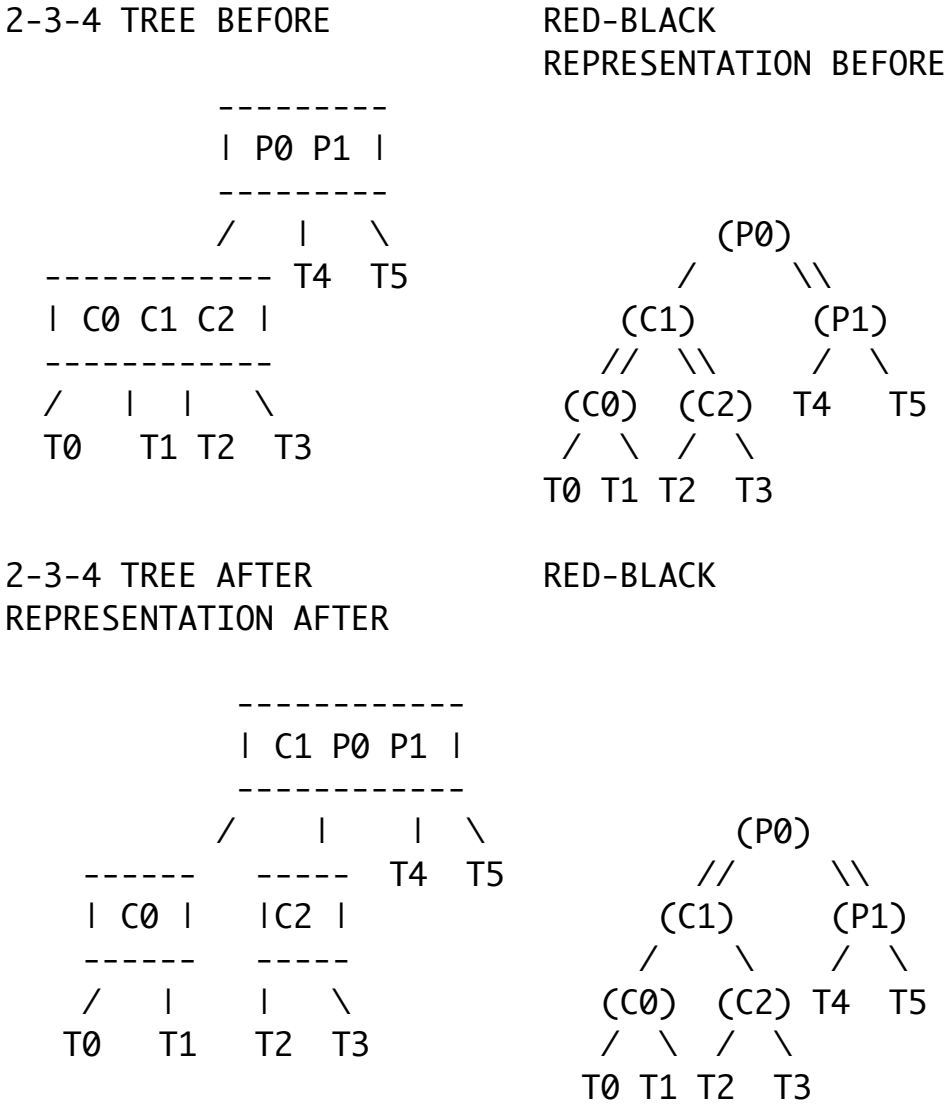
Observe: The only change is to convert the two red child pointers to black!

d) The parent is a two node. (The drawing assumes child being split is left child of parent. The case for the child being the right child is symmetrical)



Observe: the only change needed to implement the split is to change the pointer from the parent to the node being split to red, and to change the two child pointers of the node being split to black!

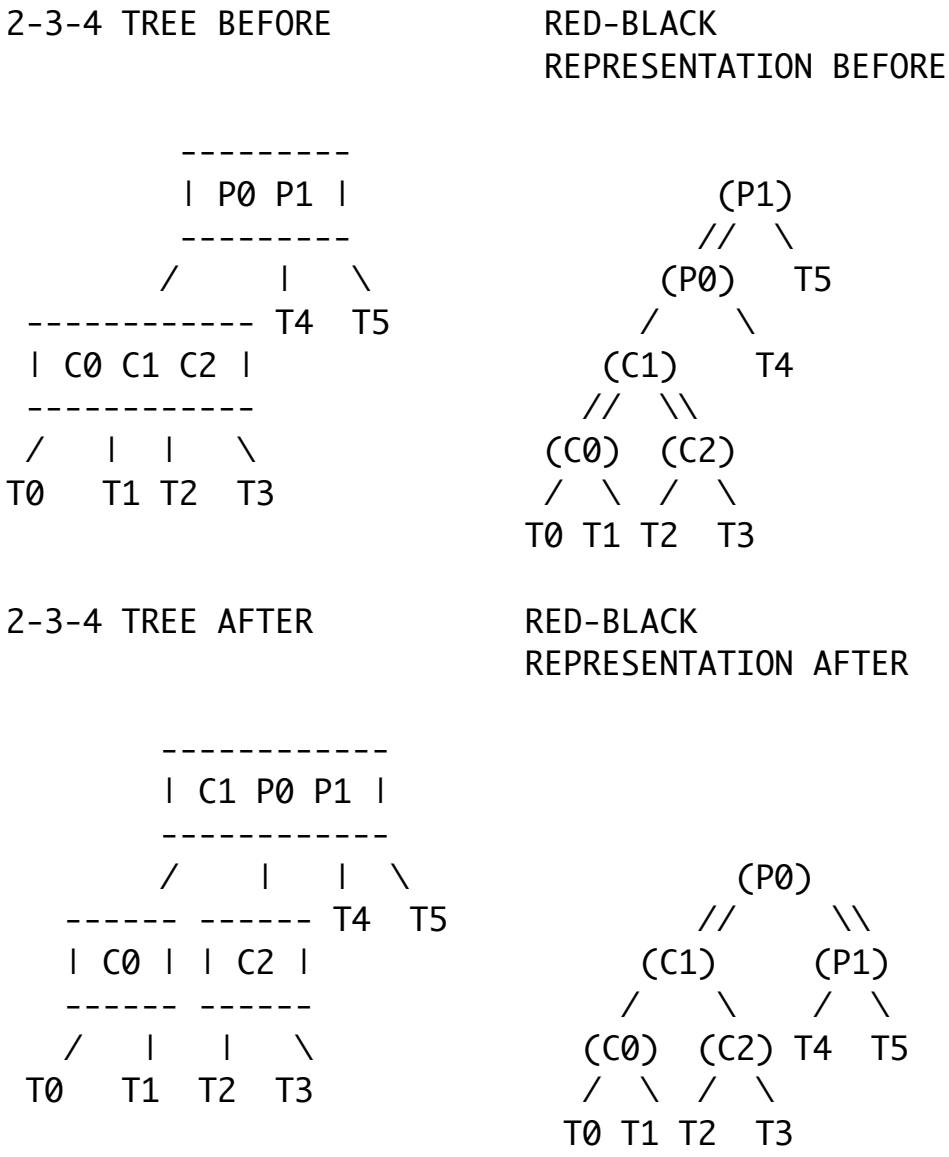
e) The parent is a three node with node being split on its "black" side. (The drawing assumes child being split is left child of parent. The case for the child being the right child is symmetrical)



Observe: This case is basically the same as the previous one.

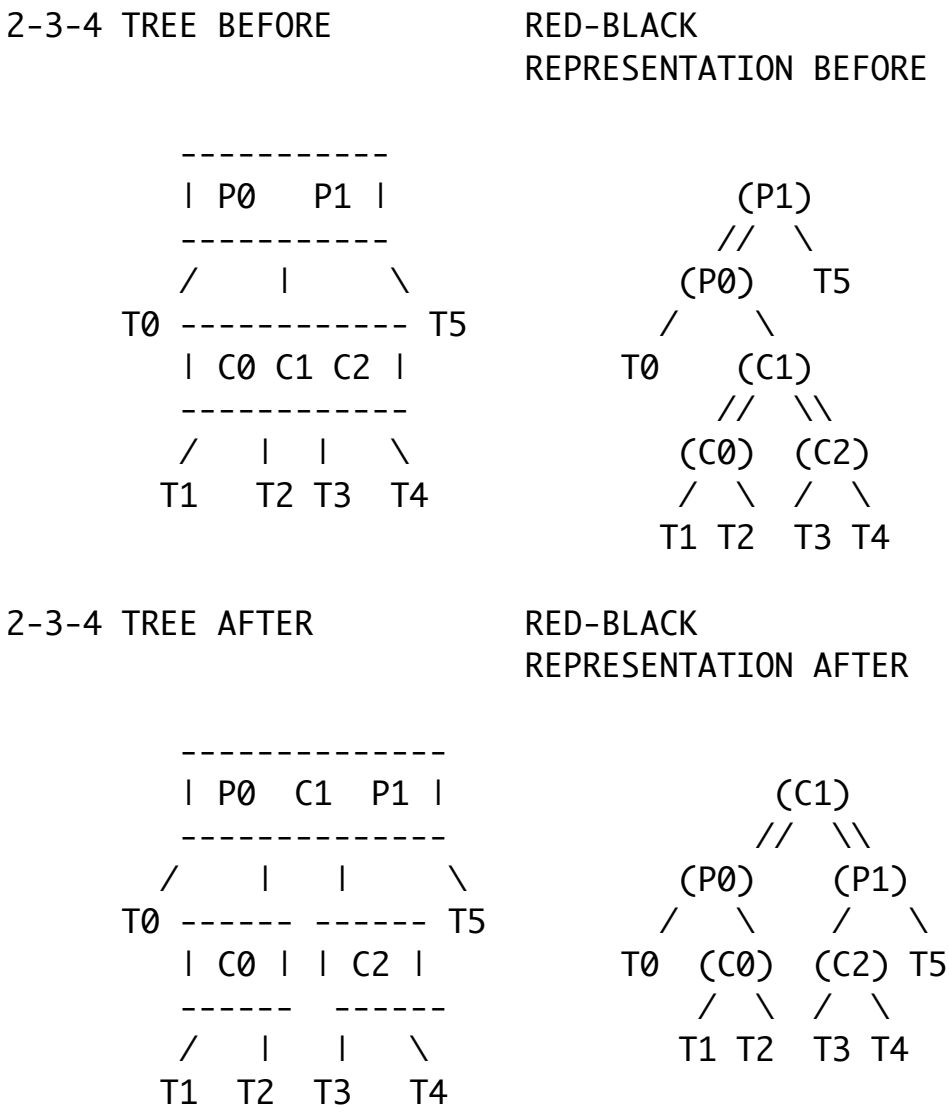
f) The parent is a three node with node being split on its "red" side. We have two subcases:

i. Node being split is "outer" child of parent. (The drawing assumes child being split is left child of parent. The case for the child being the right child is symmetrical)



Observe: This has required a right rotation around the root of the parent. The root of the subtree is now P0, not P1.

ii. Node being split is middle child of parent. (The drawing assumes child being split is in left subtree of parent. The case for the child being in the right subtree is symmetrical)



Observe: This has required a double rotation - left around the left child of the parent, then right around the root of the parent. The root of the subtree is now C1, not P1.

g) The parent can never be a four node; if it were, it would have already been split when passing through it to the child we are now splitting. Thus, we have covered all possible cases (except for mirror images.)

5. Finally, we must consider what happens on insert when we reach the bottom of the tree.

   a) In the 2-3-4 tree, we never add nodes at the bottom of a tree - we simply insert a new key into a leaf node. (Recall that our splitting strategy guarantees that each node on the path from the root down will be at biggest a three, so there will always be room.)

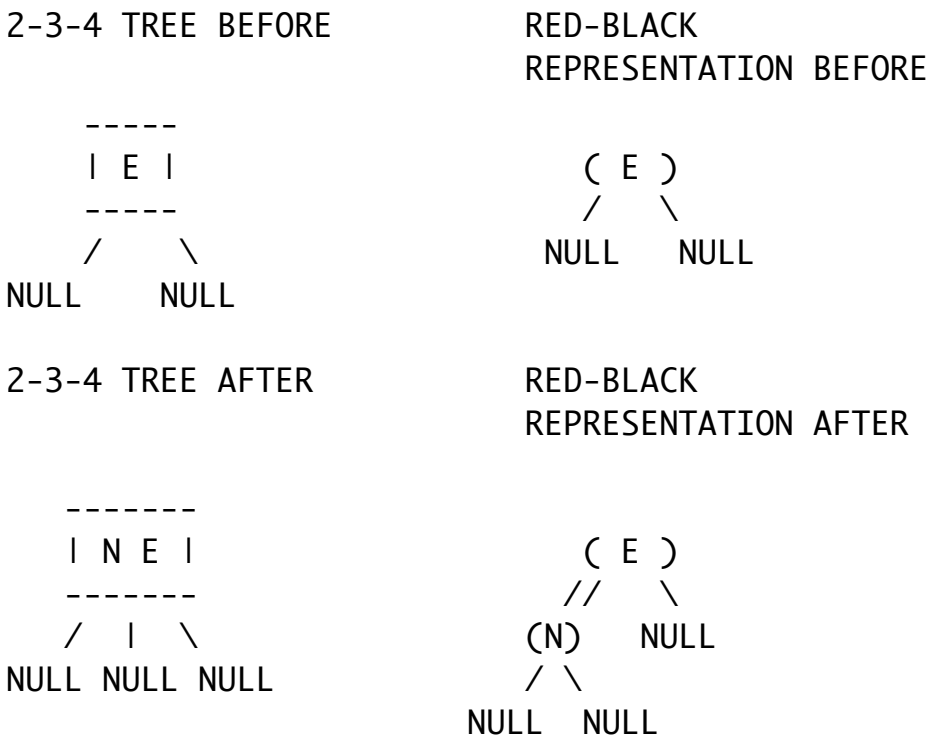
   b) However, in the red-black implementation we actually DO add a node in some cases - e.g. when converting a 2-node to a 3-node or a 3-node to a 4-node.

   c) Again there are several cases dependent on the leaf into which the key is to go. (Again, we consider insertion on the left. Insertion on the right is the mirror image.)

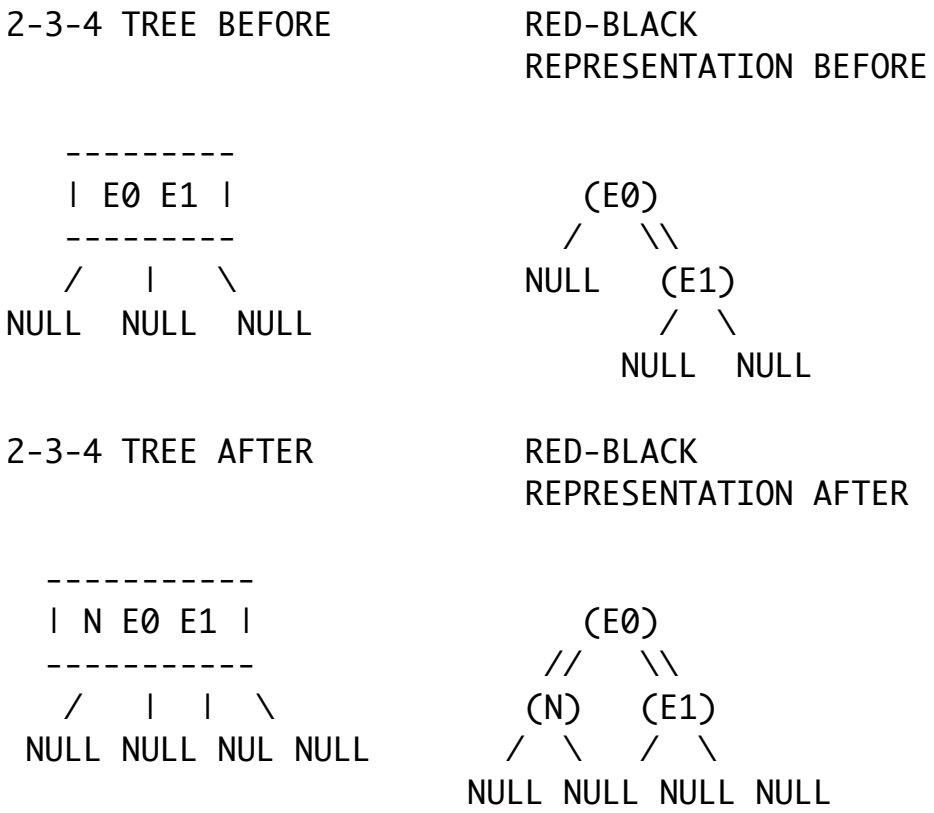
      In the drawings, we use the following abbreviations

      E, E0, E1: An existing leaf key
      N: the new key

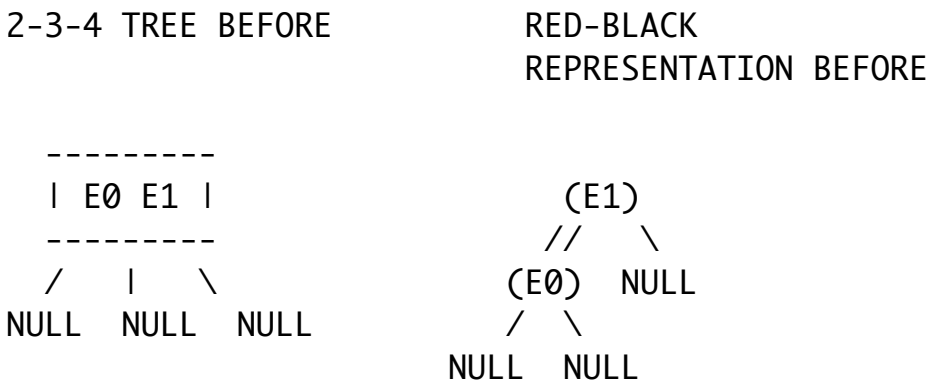
      i. The leaf is a two node:

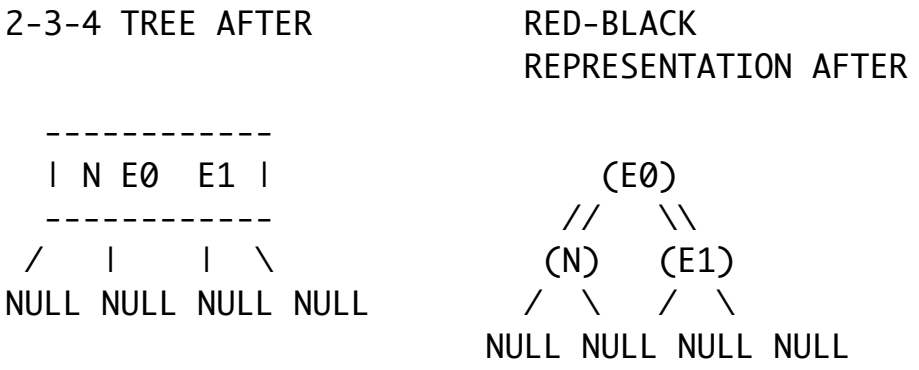


ii. The leaf is a three node, with new key going on the "black" side:



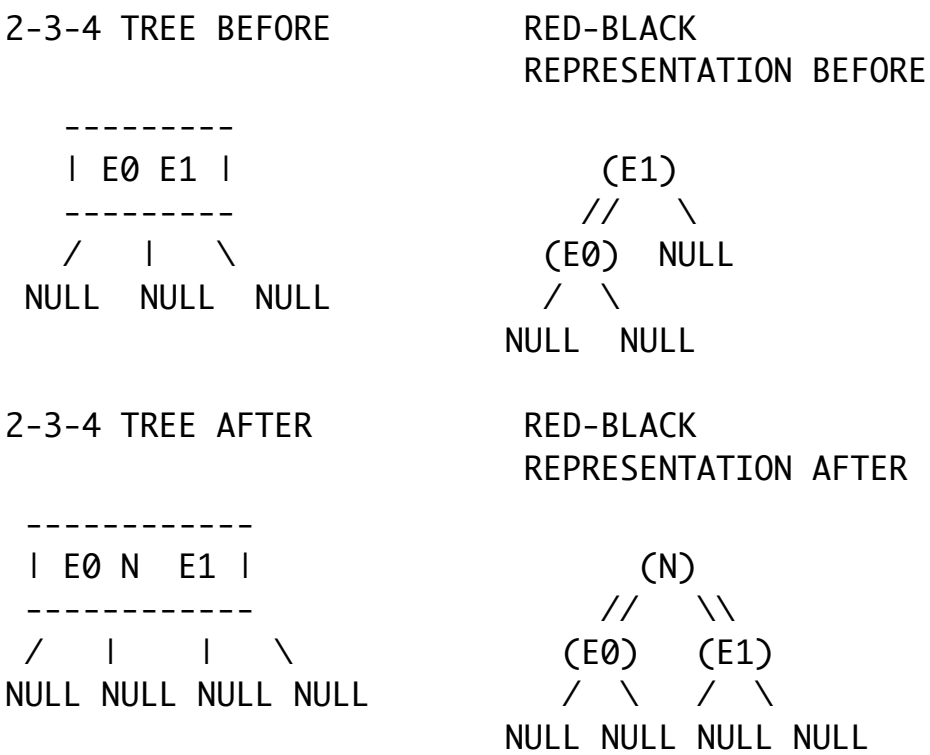
iii. The leaf is a three node, with new key going on the outside of "red" side:





Note rotation around root of subtree.

iv. The leaf is a three node, with new key going on the inside of "red" side:



Note double rotation (new node is originally added as a child of E0, then E0 subtree is rotated left, then main subtree is rotated right.

v. Of course, the leaf could never be a 4-node, since it would already have been split when working down to it.