

A Brief Introduction  
to Red/Black Trees

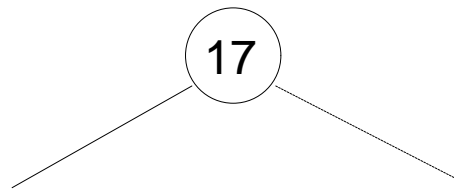
*by*

*Peter M. Maurer*

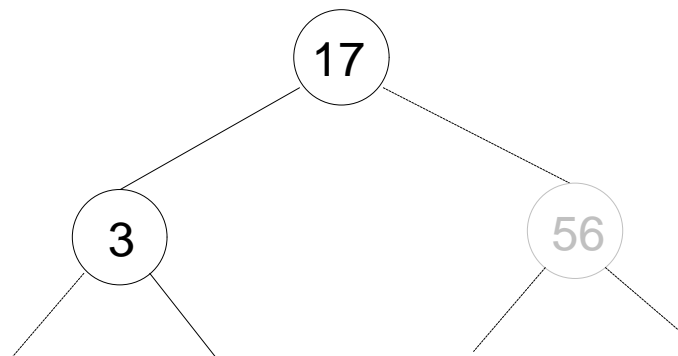
# Search Trees

## 1. Binary Search Trees.

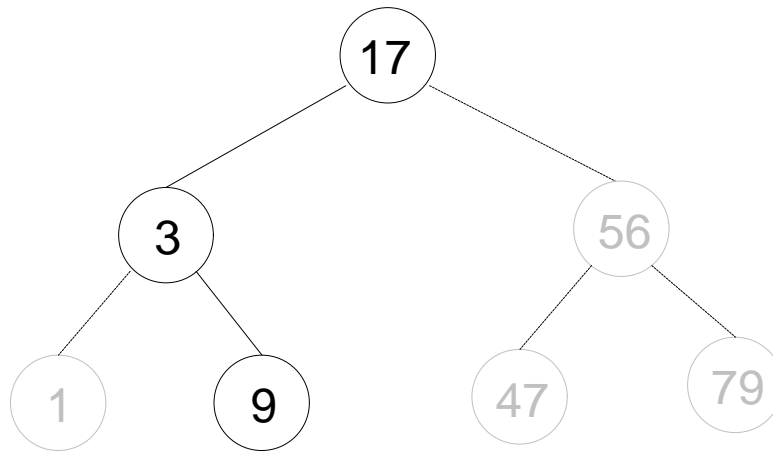
As we all know, the Binary Search of an ordered list is much more efficient than a linear search of the same list. For a Binary Search, it takes on the order of  $\lg n$  comparisons to discover that a particular element is not in the list, while for linear search it would take  $n$  comparisons to discover the same thing. For large values of  $n$ ,  $\Theta(\lg n)$  is much faster than  $\Theta(n)$ . The interesting thing about binary search, is that it can be encoded as a tree. Suppose, for example that we are searching for the value 12 in the list of the following values. 1, 3, 9, 17, 47, 56, 79. The first thing we do is compare 12 to 17. Grapically, we can represent this as follows.



Since 12 is less than 17, we follow the left branch. If 12 were greater than 17 (which of course it is not) then we would follow the right branch. The next step in the binary search would be to compare 12 to 3, which can be represented graphically as follows. Since 12 is larger than 3, we follow the right branch out of vertex 3 rather than the left..

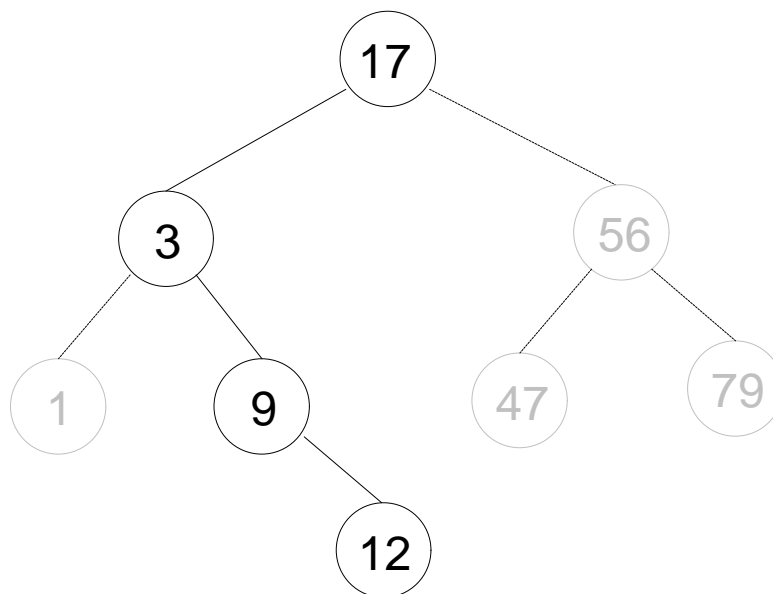


Finally we compare the value 12 to the value 9, which is represented graphically as follows.

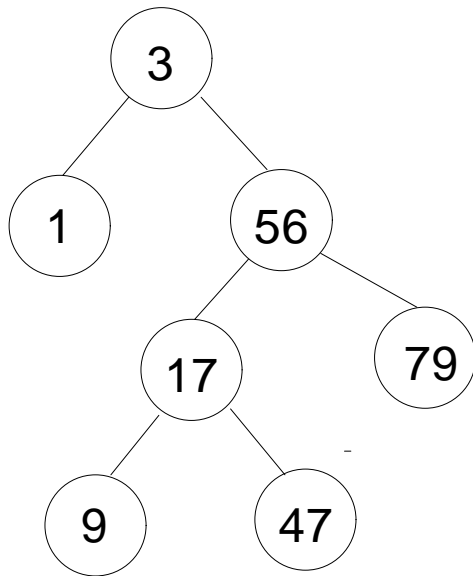


If the vertex containing 9 had a right child, we would visit that child. However, it does not, so we must report that the search tree does not contain the value 12.

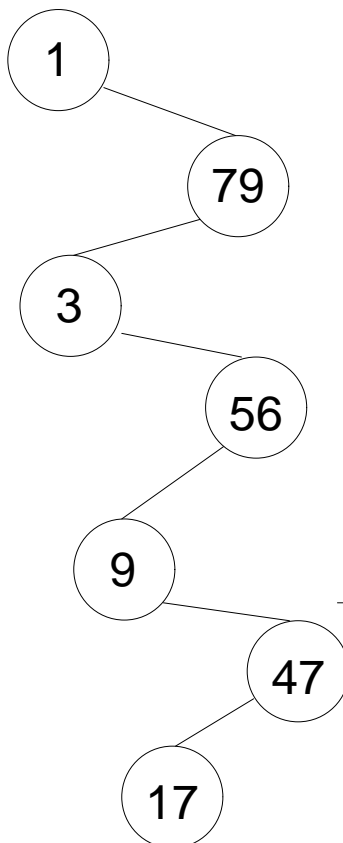
This graphical representation of binary search suggests that we could use binary trees to perform efficient searches. This is especially important when values are added and deleted dynamically from the search trees. Adding a new value to an ordered list is a costly operation, which may entail moving all  $n$  existing keys to make room for the new one. But if a binary tree were constructed using pointers, it would be an easy matter to add a new key. For example, we could add a vertex containing 12 to the existing tree by performing a normal binary search for the value 12. Once it was discovered that the vertex containing 9 had no right child, we could create such a child, and add the value 12 to it, as illustrated in the following example.



This would seem to be a nice way to guarantee efficient searches when it was necessary to add and delete keys dynamically. Unfortunately, it doesn't always work so nicely. The problem is that there is more than one binary search tree containing the values 1, 3, 9, 17, 47, 56, and 79. The following tree for example.

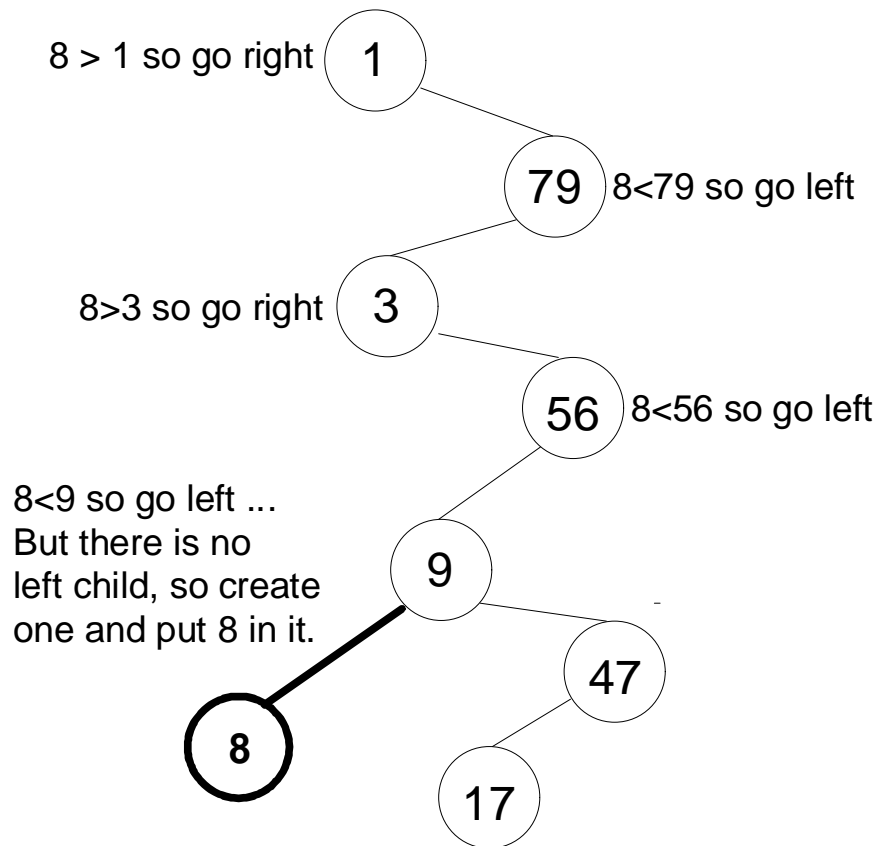


The following example is even worse.



All of these trees are legitimate binary search trees, and each one of them could have been created on the fly given the proper order of the input values. To review the proper

method of inserting a new element into a binary search tree, let us imagine that 8 is being added to the previous tree.



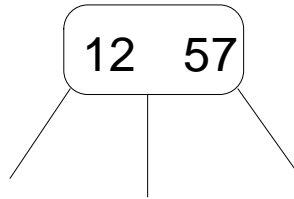
It should be obvious from this example that the structure of the tree is heavily dependent on the order of the input values. If we are lucky enough to create a balanced tree, then we will end up with an efficient searching algorithm, one that runs in  $\Theta(\lg n)$  time. If we are unlucky, we will create a tree that runs in  $\Theta(n)$  time. Given a random sequence of inputs, are we likely to create a balanced tree or an unbalanced one. Well, of course, we are most likely to create an unbalanced one, but how unbalanced? Studies have shown that when keys are added in random order, the resultant search process is  $\Theta(n)$  not  $\Theta(\lg n)$ .

The big problem with binary search trees is that the new vertices are always added at the bottom. Suppose we could start adding a bunch of vertices at the bottom, and when we finally got enough for another level, we added a new level just above the bottom level. This seems like it ought to keep the trees balanced, but it isn't at all clear how one would go about doing this sort of thing. Nevertheless, *it can be done!*

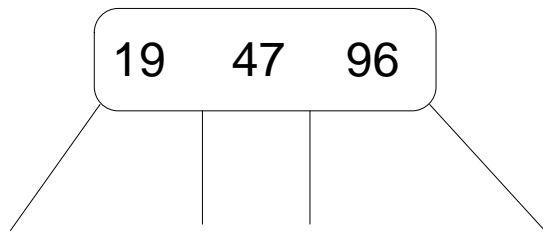
## 2. 2-3-4 Trees.

In a binary tree, every vertex has (at most) two children. Computer scientists are fond of asking weird questions like "Why just two children? Why not 3 or 4?" Suppose, for example, that you had a tree in which every vertex had three children. Now to be able to

decide which child to search next, it would be necessary for this vertex to have two keys, as in the following example.



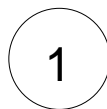
When this vertex is searched, for some number  $x$ , if  $x < 12$  then the left branch is taken. If the  $x > 57$  then the right branch is taken. if  $12 < x < 57$ , then the middle branch is taken. We can do the same thing with four output arcs, but this would require three keys, as in the following example.



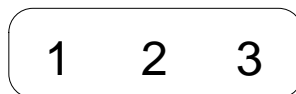
To search this type of node, first we test  $x$  against 47 if  $x < 47$ , then we test  $x$  against 19 if  $x > 47$  then we test  $x$  against 96. If  $x > 96$  then we take the right link. If  $x < 19$  then we take the left link. if  $19 < x < 47$  then we follow the second-to-the-left link, if  $47 < x < 96$  then we follow the second-to-the-right link.

We could certainly make a trinary and or a quaternary tree, but then we need to have a multiple of 2 keys or a multiple of 3 keys. Suppose, instead, we build a tree out of three different types of vertices. Some that have one key, some that have two, and some that have three. We will call these trees 2-3-4 trees. Let's see how we would build such a tree, and let's suppose that we want to add the keys from 1-15 in that order. If we were to build a binary tree from such input, the result would be the worst possible type of tree.

At first, everything seems to be going well. We add the key, 1, and end up with the following.

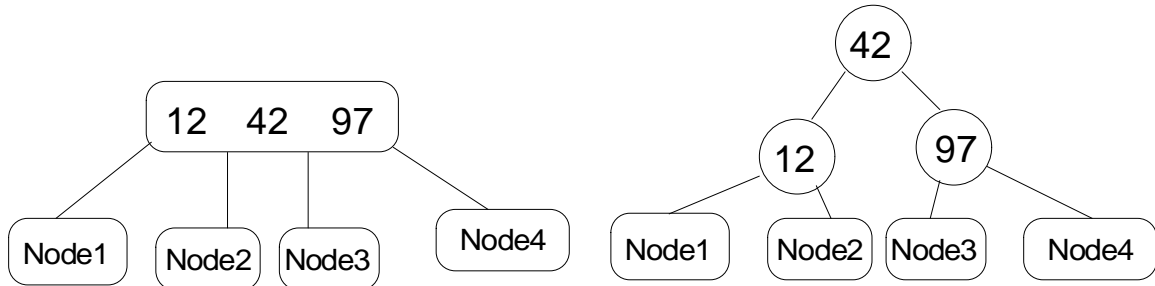


Adding the next two keys is also easy, since we just stick them into the node that's already there.



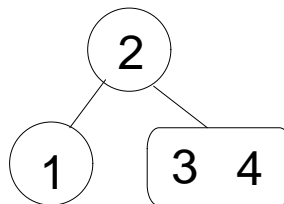
But, what about the value 4? We could create a right child and add a value to the new node, but this would just perpetuate the problem that we had with the binary trees. Instead, let's alter the rules just slightly. First, the procedure for finding the proper location for a new value, is to search the tree for the value, and if the key isn't found, insert it into the proper position. The rule is the same for 2-3-4 trees, but during the search we will add the following additional rule.

Before searching a 4-node, break the node into 3 2-nodes, as illustrated in the following procedure.

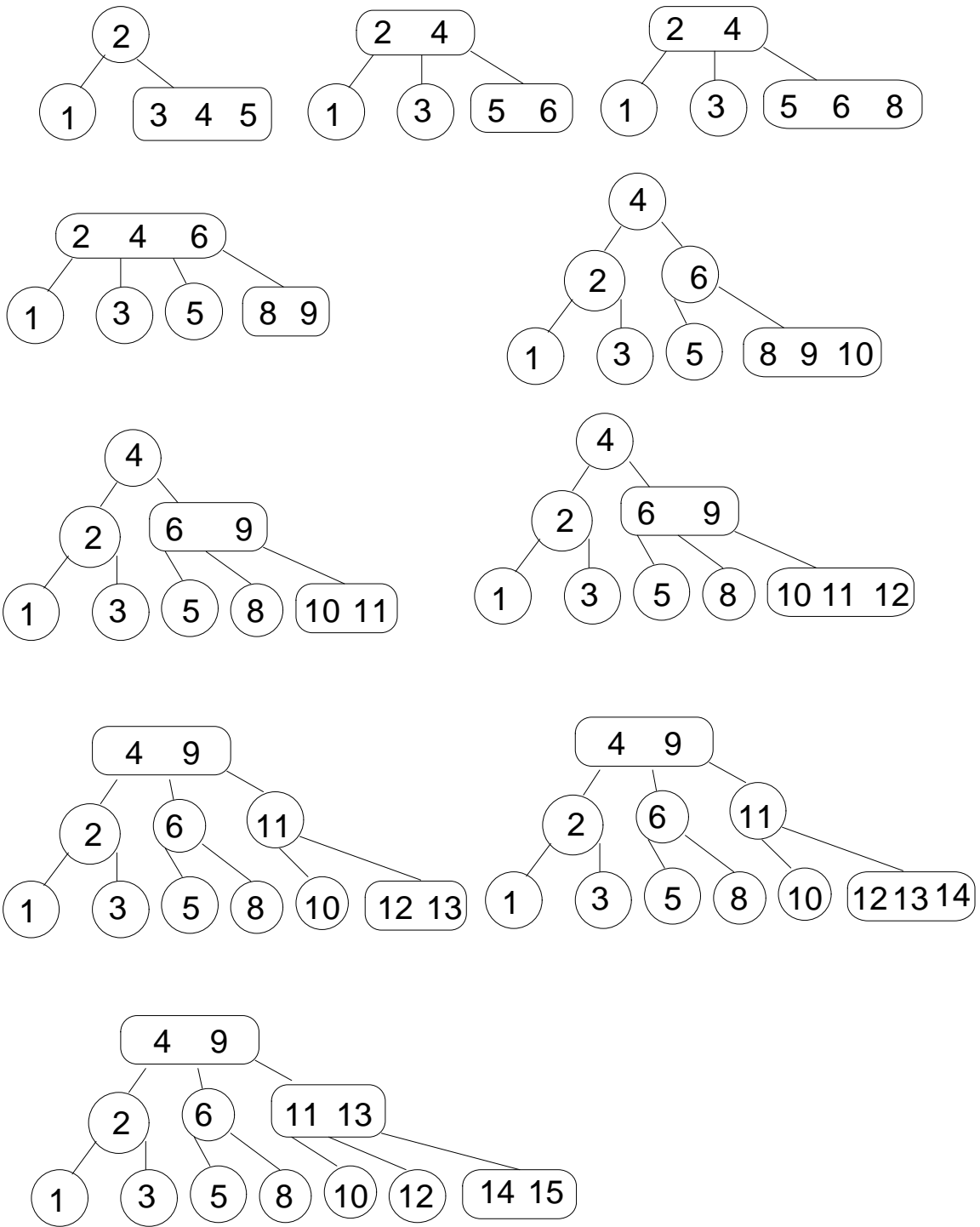


Remember that the breakup of the 4-node is done as soon as it is encountered, before comparing the search value against 42. The search actually begins with the vertices after they have been broken up. Since the 4-nodes have all been broken up before we search them, there will always be room in an existing vertex to add a new value. Furthermore, any new vertices are added at an existing level or at one level up from an existing level. This procedure is guaranteed to keep the tree balanced.

Once we add the value 4 to the above tree, we end up with the following tree.



The remainder of the insertion process is illustrated below.

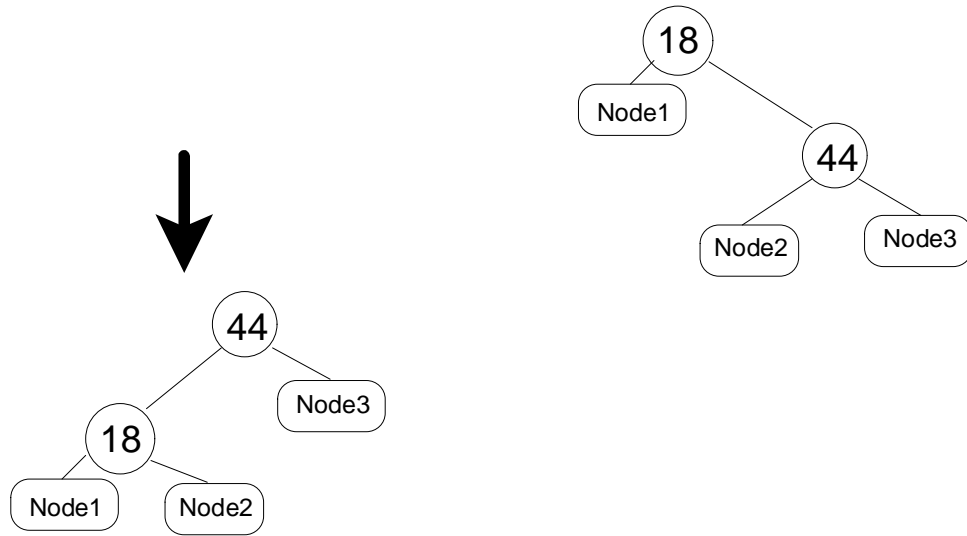


Notice that this tree ends up being perfectly balanced, and will remain perfectly balanced no matter in what order we add the keys.

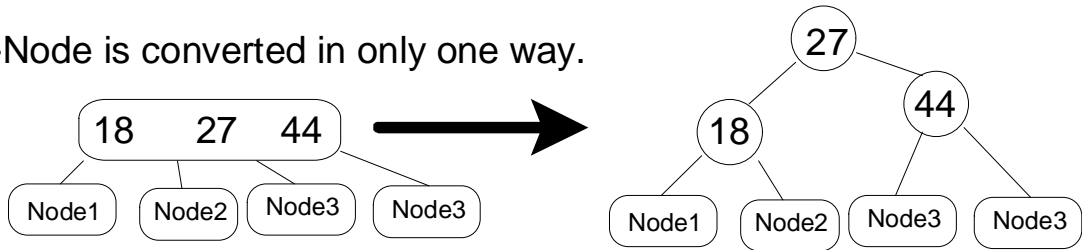
### 3. Red-Black Trees.

Surprisingly enough, the 2,3,4, trees can actually be created using ordinary binary search trees. The conversion is done as follows.

A 3-Node is converted in one of two ways:

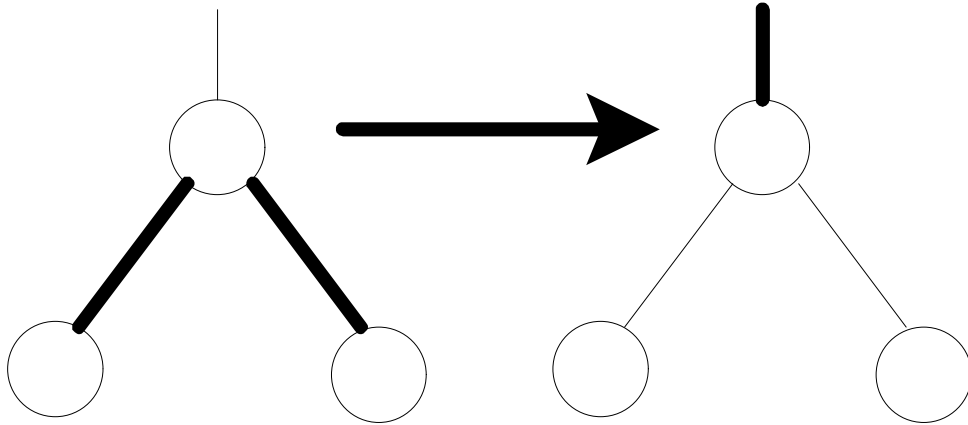


A 4-Node is converted in only one way.



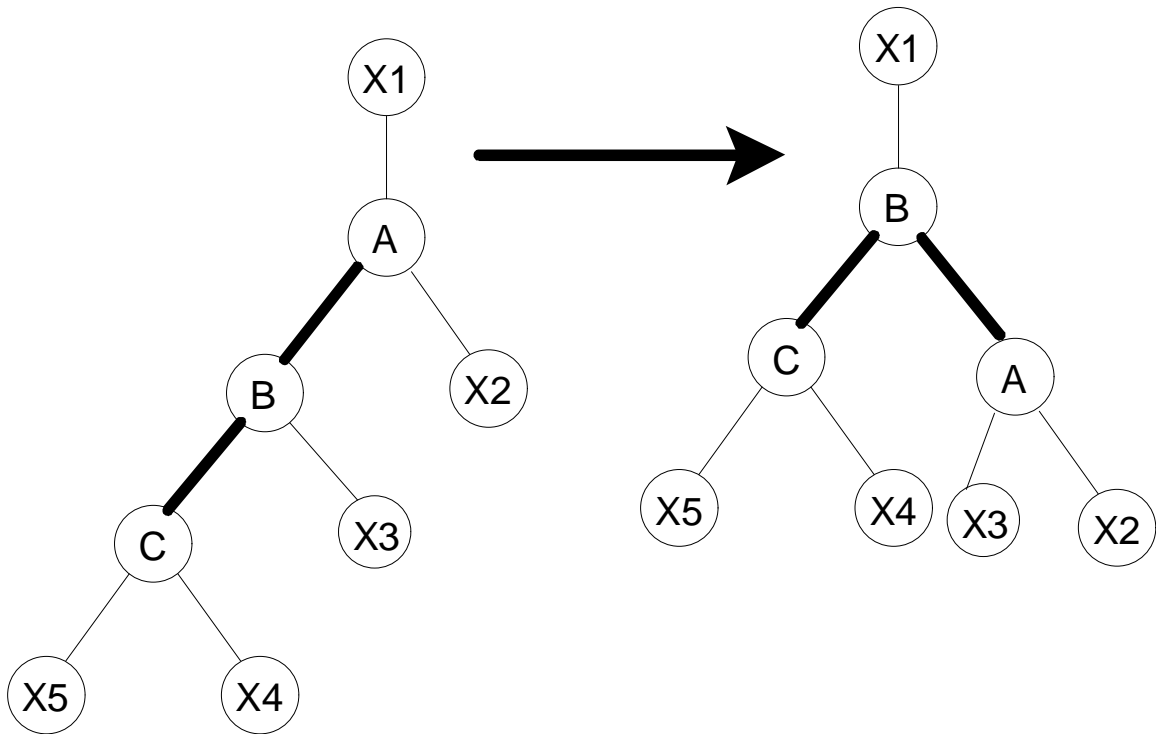
In the converted tree (which is a binary search tree) every edge is colored either red or black. The edges that came directly from the original tree are colored black, while the edges that were created during the conversion of the 3 and 4 nodes are colored red. In any path between the root and the leaf of a converted tree, at least half of the edges will be black. Since every path from the root to a leaf has exactly the same number of black edges, the resultant tree is almost balanced. The maximum length of any path is at most  $2 \cdot \lg n$ . Searching red-black trees is an  $\Theta(\lg n)$  process.

The problem arises in adding new edges to a red-black tree. In the following diagram, the fat edges are red, and the thin edges are black. To split a 4-node into 3 2-nodes the following process is used.

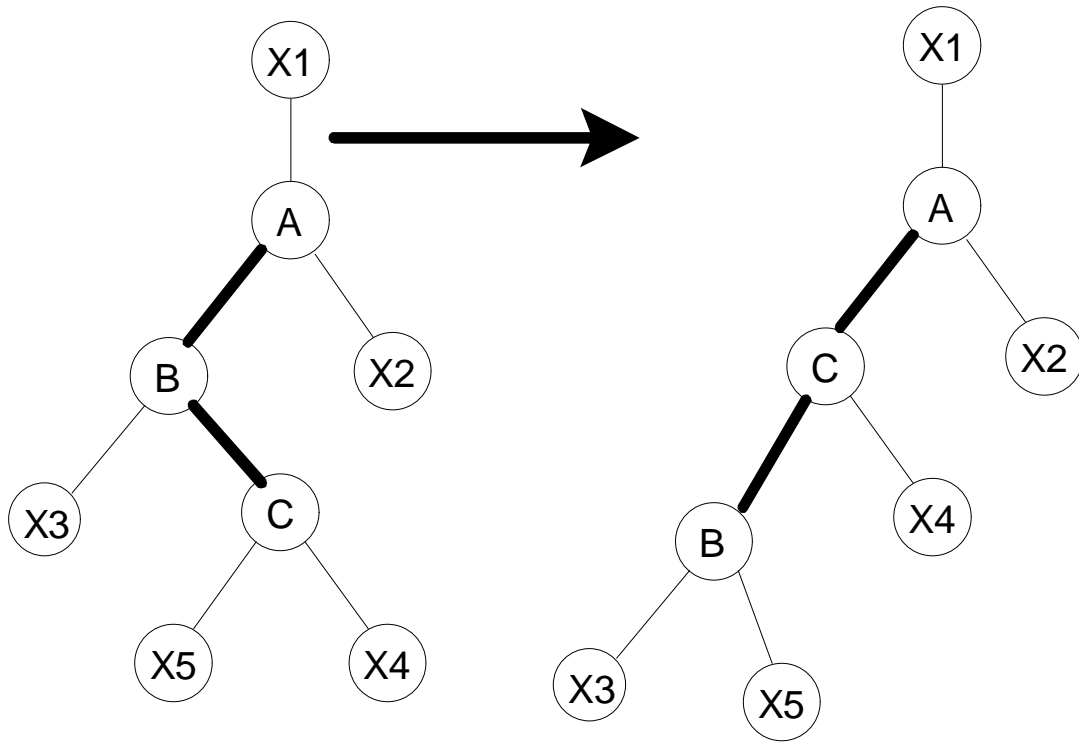


To split a 4-node in a red-black tree, the colors of the edges touching the root vertex are reversed, and no other change is made.

A red-black tree is a valid binary search tree and can be searched using the usual binary search-tree algorithm. When a vertex is added to a red-black tree, it is done in the same manner as for an ordinary binary search tree. Of course, all 4-nodes are color-reversed as they are encountered, and before they are searched. When the proper location is found, a new vertex and a new edge are created, and the new value is placed into the new vertex. The new edge is *always* colored red. Adding a new value can create either a 3-node or a 4-node. Creating a 3-node is no problem, but 4-nodes can be created in two different illegal configurations the first of which is illustrated below.



This diagram also indicates how the illegal configuration is corrected. This process is known as rotation, since the edge between the vertices A and B is rotated to correct the invalid configuration. The second illegal configuration is illustrated below.



The second illegal configuration cannot be corrected by a single rotation. First, the lower red edge must be rotated to create an illegal configuration of the first kind, which is corrected in the usual fashion.

It is the rotations in the red-black trees that keep the tree balanced.