The binary tree algorithms of the previous section work very well for a wide variety of applications, but they do have the problem of bad worst-case performance. What's more, as with Quicksort, it's embarrassingly true that the bad worst case is one that's likely to occur in practice if the person using the algorithm is not watching for it. Files already in order, files in reverse order, files with alternating large and small keys, or files with any large segment having a simple structure can cause the binary tree search algorithm to perform very badly.

With Quicksort, our only recourse for improving the situation was to resort to randomness: by choosing a random partitioning element, we could rely on the laws of probability to save us from the worst case. Fortunately, for binary tree searching, we can do much better: there is a general technique that will enable us to guarantee that this worst case will not occur. This technique, called balancing, has been used as the basis for several different "balanced tree" algorithms. We'll look closely at one such algorithm and discuss briefly how it relates to some of the other methods that are used.

As will become apparent below, the implementation of balanced tree algorithms is certainly a case of "easier said than done." Often, the general concept behind an algorithm is easily described, but an implementation is a morass of special and symmetric cases. Not only is the program developed in this chapter an important searching method, but also it is a nice illustration of the relationship between a "high-level" algorithm description and a "low-level" Pascal program to implement the algorithm.

**Top-Down 2-3-4 Trees**

To eliminate the worst case for binary search trees, we'll need some flexibility in the data structures that we use. To get this flexibility, let's assume that we can have nodes in our trees that can hold more than one key. Specifically, we'll
allow 3-nodes and 4-nodes, which can hold two and three keys respectively. A 3-node has three links coming out of it, one for all records with keys smaller than both its keys, one for all records with keys in between its two keys, and one for all records with keys larger than both its keys. Similarly, a 4-node has four links coming out of it, one for each of the intervals defined by its three keys. (The nodes in a standard binary search tree could thus be called 2-nodes: one key, two links.) We'll see below some efficient ways of defining and implementing the basic operations on these extended nodes; for now, let's assume we can manipulate them conveniently and see how they can be put together to form trees.

For example, below is a 2-3-4 tree which contains some keys from our searching example.

```
  E
 / \ 
A   S

A   C   H   I

S
```

It is easy to see how to search in such a tree. For example, to search for O in the tree above, we would follow the middle link from the root, since O is between E and R then terminate the unsuccessful search at the right link from the node containing H and I.

To insert a new node in a 2-3-4 tree, we would like to do an unsuccessful search and then hook the node on, as before. It is easy to see what to if the node at which the search terminates is a 2-node: just turn it into a 3-node. Similarly, a 3-node can easily be turned into a 4-node. But what should we do if we need to insert our new node into a 4-node? The answer is that we should first split the 4-node into two 2-nodes and pass one of its keys further up in the tree. To see exactly how to do this, let's consider what happens when the keys from A S E A R C H I N G E X A M P L E are inserted into an initially empty tree. We start out with a 2-node, then a 3-node, then a 4-node:

```
  E
 / \   |
A   S   A E S

A   C   H   I

S
```

Now we need to put a second A into the 4-node. But notice that as far as the search procedure is concerned, the 4-node at the right above is exactly equivalent to the binary tree:

```
  E
 /   |
A   S

A   C   H

I

S
```

If our algorithm "splits" the 4-node to make this binary tree before trying to insert the A, then there will be room for A at the bottom:

```
  E
 /   |
A   S

A   E

S
```

Now R, C, and the H can be inserted, but when it's time for I to be inserted, there's no room in the 4-node at the right:

```
  E
 /   |
A   S

A   A

R S
```

Again, this 4-node must be split into two 2-nodes to make room for the I, but this time the extra key needs to be inserted into the father, changing it from a 2-node to a 3-node. Then the N can be inserted with no splits, then the G causes another split, turning the root into a 4-node:

```
  E
 /   |
A   S

A   C   H

I

S
```

But what if we were to need to split a 4-node whose father is also a 4-node? One method would be to split the father also, but this could keep happening all the way back up the tree. An easier way is to make sure that the father of any node we see won't be a 4-node by splitting any 4-node we see on the way down the tree. For example, when E is inserted, the tree above first becomes
These transformations are purely "local": no part of the tree need be examined or modified other than what is diagrammed. Each of the transformations passes up one of the keys from a 4-node to its father in the tree, restructuring links accordingly. Note that we don’t have to worry explicitly about the father being a 4-node since our transformations ensure that as we pass through each node in the tree, we come out on a node that is not a 4-node. In particular, when we come out the bottom of the tree, we are not on a 4-node, and we can directly insert the new node either by transforming a 2-node to a 3-node or a 3-node to a 4-node. Actually, it is convenient to treat the insertion as a split of an imaginary 4-node at the bottom which passes up the new key to be inserted. Whenever the root of the tree becomes a 4-node, we’ll split it into three 2-nodes, as we did for our first node split in the example above. This (and only this) makes the tree grow one level “higher.”

The algorithm sketched in the previous paragraph gives a way to do searches and insertions in 2-3-4 trees; since the 4-nodes are split up on the way from the top down, the trees are called top-down 2-3-4 trees. What’s interesting is that, even though we haven’t been worrying about balancing at all, the resulting trees are perfectly balanced! The distance from the root to every external node is the same, which implies that the time required by a search or an insertion is always proportional to \(\log N\). The proof that the trees are always perfectly balanced is simple: the transformations that we perform have no effect on the distance from any node to the root, except when we split the root, and in this case the distance from all nodes to the root is increased by one.

The description given above is sufficient to define an algorithm for searching using binary trees which has guaranteed worst-case performance. However, we are only halfway towards an actual implementation. While it would be possible to write algorithms which actually perform transformations on distinct data types representing 2-, 3-, and 4-nodes, most of the things that need to be done are very inconvenient in this direct representation. (One can become convinced of this by trying to implement even the simpler of the two node transformations.) Furthermore, the overhead incurred in manipulating the more complex node structures is likely to make the algorithms slower than standard binary tree search. The primary purpose of balancing is to provide “insurance” against a bad worst case, but it would be unfortunate to have to pay the overhead cost for that insurance on every run of the algorithm. Fortunately, as we’ll see below, there is a relatively simple representation of 2-, 3-, and 4-nodes that allows the transformations to be done in a uniform way with very little overhead beyond the costs incurred by standard binary tree search.
Red-Black Trees

Remarkably, it is possible to represent 2-3-4 trees as standard binary trees (2-nodes only) by using only one extra bit per node. The idea is to represent 3-nodes and 4-nodes as small binary trees bound together by “red” links which contrast with the “black” links which bind the 2-3-4 tree together. The representation is simple: 4-nodes are represented as three 2-nodes connected by red links and 3-nodes are represented as two 2-nodes connected by a red link (red links are drawn as double lines):

(Either orientation for a 3-node is legal.) The binary tree drawn below is one way to represent the final tree from the example above. If we eliminate the red links and collapse the nodes they connect together, the result is the 2-3-4 tree from above. The extra bit per node is used to store the color of the link pointing to that node: we’ll refer to 2-3-4 trees represented in this way as red-black trees.

The “slant” of each 3-node is determined by the dynamics of the algorithm to be described below. There are many red-black trees corresponding to each 2-3-4 tree. It would be possible to enforce a rule that 3-nodes all slant the same way, but there is no reason to do so.

These trees have many structural properties that follow directly from the way in which they are defined. For example, there are never two red links in a row along any path from the root to an external node, and all such paths have an equal number of black links. Note that it is possible that one path (alternating black-red) be twice as long as another (all black), but that all path lengths are still proportional to log N.

A striking feature of the tree above is the positioning of duplicate keys. On reflection, it is clear that any balanced tree algorithm must allow records with keys equal to a given node to fall on both sides of that node: otherwise, severe imbalance could result from long strings of duplicate keys. This implies that we can’t find all nodes with a given key by repeated calls to the searching procedure, as in the previous chapter. However, this does not present a real problem, because all nodes in the subtree rooted at a given node with the same key as that node can be found with a simple recursive procedure like the treeprint procedure of the previous chapter. Or, the option of requiring distinct keys in the data structure (with linked lists of records with duplicate keys) could be used.

One very nice property of red-black trees is that the tree search procedure for standard binary tree search works without modification (except for the problem with duplicate keys discussed in the previous paragraph). We’ll implement the link colors by adding a boolean field red to each node which is true if the link pointing to the node is red, false if it is black; the treeseach procedure simply never examines that field. That is, no “overhead” is added by the balancing mechanism to the time taken by the fundamental searching procedure. Each key is inserted just once, but might be searched for many times in a typical application, so the end result is that we get improved search times (because the trees are balanced) at relatively little cost (because no work for balancing is done during the searches).

Moreover, the overhead for insertion is very small: we have to do something different only when we see 4-nodes, and there aren’t many 4-nodes in the tree because we’re always breaking them up. The inner loop needs only one extra test (if a node has two red sons, it’s a part of a 4-node), as shown in the following implementation of the insert procedure:
function rbtreeinsert(v: integer; x:link): link;
  var gg, g, f, l: link;
begin
  f := x;  g := x;
  repeat
    gg := g;  g := f;  f := x;
    if v < x↑.key then x := x↑.l else x := x↑.r;
    if x↑.l↑.red and x↑.r↑.red then x := split(v, gg, g, f, x);
    until x = g;
  new(x);  x↑.key := v;  x↑.l := z;  x↑.r := z;
  if v < f↑.key then f↑.l := x else f↑.r := x;
  rbtreeinsert := x;
  x := split(v, gg, g, f, x);
end;

In this program, x moves down the tree as before, with gg, g, and f kept
pointing to x's great-grandfather, grandfather, and father in the tree. To see
why all these links are needed, consider the addition of Y to the tree above.
When the external node at the right of the 3-node containing S and X is
reached, gg is R, g is S, and f is X. Now, Y must be added to make a 4-node
containing S, X, and Y, resulting in the following tree:

```
    I
   / \   
  E   R
 /     /
A     B
 / \   / \ 
C   G   N  X
 / \   / \   / \
A   E   H   M   P   S   Y
```

We need a pointer to R (gg) because R's right link must be changed to point
to X, not S. To see exactly how this comes about, we need to look at the
operation of the split procedure.

To understand how to implement the split operation, let's consider the
red-black representation for the two transformations we must perform: if we
have a 2-node connected to a 4-node, then we should convert them into a
3-node connected to two 2-nodes; if we have a 3-node connected to a 4-node,
we should convert them into a 4-node connected to two 2-nodes. When a
new node is added at the bottom, it is considered to be the middle node of
an imaginary 4-node (that is, think of z as being red, though this never gets
explicitly tested).

The transformation required when we encounter a 2-node connected to a
4-node is easy:

This same transformation works if we have a 3-node connected to a 4-node in
the "right" way:

Thus, split will begin by marking x to be red and the sons of x to be black.
This leaves the two other situations that can arise if we encounter a 3-node
connected to a 4-node:
In this diagram, $T_1$ represents the tree containing all the records with keys less than $A$, $T_2$ contains all the records with keys between $A$ and $B$, and so forth. The transformation switches the orientation of the 3-node containing $A$ and $B$ without disturbing the rest of the tree. Thus none of the keys in $T_1, T_2, T_3,$ and $T_4$ are touched. In this case, the transformation is effected by the link changes $s_l := gs_l.r; gs_l.r := s_s; y_l := gs$. Also, note carefully that the colors of $A$ and $B$ are switched. There are three analogous cases: the 3-node could be oriented the other way, or it could be on the right side of $y$ (oriented either way).

Disregarding the colors, this single rotation operation is defined on any binary search tree and is the basis for several balanced tree algorithms. It is important to note, however, that doing a single rotation doesn’t necessarily improve the balance of the tree. In the diagram above, the rotation brings all the nodes in $T_1$ one step closer to the root, but all the nodes in $T_3$ are lowered one step. If $T_3$ were to have more nodes than $T_1$, then the tree after the rotation would become less balanced, not more balanced. Top-down 2-3-4 trees may be viewed as simply a convenient way to identify single rotations which are likely to improve the balance.

Doing a single rotation involves structurally modifying the tree, something that should be done with caution. A convenient way to handle the four different cases outlined above is to use the search key $v$ to “rediscover” the relevant son ($s$) and grandson ($gs$) of the node $y$. (We know that we’ll only be reorienting a 3-node if the search took us to its bottom node.) This leads to somewhat simpler code that the alternative of remembering during the search not only the two links corresponding to $s$ and $gs$ but also whether they are right or left links. We have the following function for reorienting a 3-node along the search path for $v$ whose father is $y$:

```java
function rotate(v: integer; y: link): link;
var s, gs: link;
begin
  if v < y.r.key then s := y.r.l; else s := y.r.r;
  if v < s.key then begin gs := s.r.l; s.r := gs.r; gs.l := s end
    else begin gs := s.r.r; s.r := gs.r; gs.l := s end;
  if v < y.r.key then y.r.l := gs else y.r.r := gs;
  rotate := gs;
end;
```

If $s$ is the left link of $y$ and $gs$ is the left link of $s$, this makes exactly the link transformations for the diagram above. The reader may wish to check the
other cases. This function returns the link to the top of the 3-node, but does not do the color switch itself.

Thus, to handle the third case for split, we can make g red, then set x to rotate(v, gg), then make x black. This reorients the 3-node consisting of the two nodes pointed to by g and f and reduces this case to be the same as the second case, when the 3-node was oriented the right way.

Finally, to handle the case when the two red links are oriented in different directions, we simply set f to rotate(v, g). This reorients the "illegal" 3-node consisting of the two nodes pointed to by f and x. These nodes are the same color, so no color change is necessary, and we are immediately reduced to the third case. Combining this and the rotation for the third case is called a double rotation for obvious reasons.

This completes the description of the operations which must be performed by split. It must switch the colors of x and its sons, do the bottom part of a double rotation if necessary, then do the single rotation if necessary:

```plaintext
function split(v: integer; gg, g, f, x: link): link;
begin
    x/.red := true; x/.l/.red := false; x/.r/.red := false;
    if f/.red then
        begin
            g/.red := true;
            if (v<g/.key)<<v<f/.key) then f := rotate(v, g);
            x := rotate(v, gg);
            x/.red := false
        end;
    head/.r/.red := false;
    split := x
end;
```

This procedure takes care of fixing the colors after a rotation and also restarts x high enough in the tree to ensure that the search doesn't get lost due to all the link changes. The long argument list is included for clarity; this procedure should more properly be declared local to rbtreeinsert, with access to its variables.

If the root is a 4-node then the split procedure will make the root red, corresponding to transforming it, along with the dummy node above it into a 3-node. Of course, there is no reason to do this, so a statement is included at the end of split to keep the root black.

Assembling the code fragments above gives a very efficient, relatively simple algorithm for insertion using a binary tree structure that is guaranteed to take a logarithmic number of steps for all searches and insertions. This is one of the few searching algorithms with that property, and its use is justified whenever bad worst-case performance simply cannot be tolerated. Furthermore, this is achieved at very little cost. Searching is done just as quickly as if the balanced tree were constructed by the elementary algorithm, and insertion involves only one extra bit test and an occasional split. For random keys the height of the tree seems to be quite close to \( \log N \) (and only one or two splits are done for the average insertion) but no one has been able to analyze this statistic for any balanced tree algorithm. Thus a key in a file of, say, half a million records can be found by comparing it against only about twenty other keys.

Other Algorithms

The "top-down 2-3-4 tree" implementation using the "red-black" framework given in the previous section is one of several similar strategies than have been proposed for implementing balanced binary trees. As we saw above, it is actually the "rotate" operations that balance the trees: we've been looking at a particular view of the trees that makes it easy to decide when to rotate. Other views of the trees lead to other algorithms, a few of which we'll mention briefly in this section.

The oldest and most well-known data structure for balanced trees is the AVL tree. These trees have the property that the heights of the two subtrees of each node differ by at most one. If this condition is violated because of an insertion, it turns out that it can be reinstated using rotations. But this requires an extra loop: the basic algorithm is to search for the value being inserted, then proceed up the tree along the path just travelled adjusting the heights of nodes using rotations. Also, it is necessary to know whether each node has a height that is one less than, the same, or one greater than the height of its brother. This requires two bits if encoded in a straightforward way, though there is a way to get by with just one bit per node.

A second well-known balanced tree structure is the 2-3 tree, where only 2-nodes and 3-nodes are allowed. It is possible to implement insert using an "extra loop" involving rotations as with AVL trees, but there is not quite enough flexibility to give a convenient top-down version.

In Chapter 18, we'll study the most important type of balanced tree, an extension of 2-3-4 trees called B-trees. These allow up to \( M \) keys per node for large \( M \), and are widely used for searching applications involving very large files.