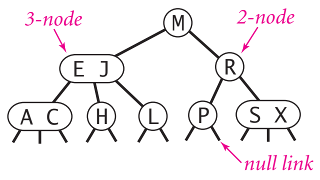
4.3 Balanced Trees

The BST algorithms in the previous section work well for a wide variety of applications, but they have poor worst-case performance. As we have noted, files already in order, files in reverse order, files with alternating large and small keys, or files with any large segment having such a structure all lead to linear insert and search times. We introduce in this section a type of search tree where costs are *guaranteed* to be logarithmic, no matter what sequence of keys is used to construct them.

Ideally, we would like keep our trees perfectly balanced, of height ~lg N, so that we can guarantee that all searches in an N-node tree can be completed in ~lg N compares, just as for binary search (see Proposition B). Unfortunately, maintaining perfect balance for dynamic insertions is too expensive. But we can relax the requirement to have *near*-perfect balance, where the height is guaranteed to be no larger than a constant times lg N, say 2 lg N. In this section, we consider a data structure that maintains such a property and provides guaranteed logarithmic performance for the *insert* and *search* operations in our symbol-table API. It also provides guaranteed logarithmic performance for all of the other operations (except range search).

2-3 search trees The primary step to get the flexibility that we need to guarantee balance in search trees is to allow the nodes in our trees to hold more than one key. Specifically, we refer to the nodes in a standard BST as *2-nodes* (they hold two links and one key) and also allow *3-nodes*, which hold three links and two keys. Every node has one link for each of the intervals subtended by its keys. As in BSTs, a 2-node has one link to a subtree containing keys smaller than its key and one link to a subtree containing keys larger than its key. A 3-node has one link to a subtree containing keys smaller than both its keys, one to a subtree containing keys in between its two keys, and one to a subtree containing keys larger than both its keys. Later, we shall see efficient ways to define and implement the basic operations on these extended nodes; for now, let us assume that we can manipulate them conveniently and see how they can be put together to form trees.



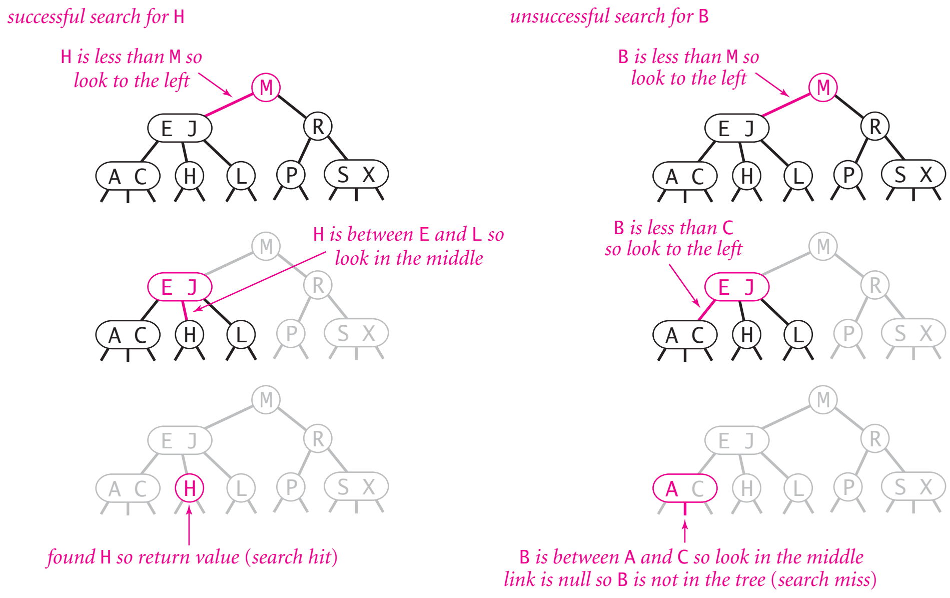
Anatomy of a 2-3 search tree

Definition *A 2-3 search tree is a tree that either is empty or:*
- *a 2-node, with one key and two links, a left link to a subtree with smaller keys, and a right link to a subtree with larger keys;*
- *a 3-node, with two keys and three links, a left link to a subtree with smaller keys, a middle link to a subtree with keys between the node's keys and a right link to a subtree with larger keys.*

As usual, we refer to a link to an empty tree as a null link.

A *perfectly balanced* 2-3 search tree is one whose null links are all the same distance from the root. To be concise, we use the term *2-3 tree* to refer to a perfectly balanced 2-3 search tree (the term denotes a more general structure in other contexts). Next, we describe the use of 2-3 trees as the underlying data structure for a symbol table.

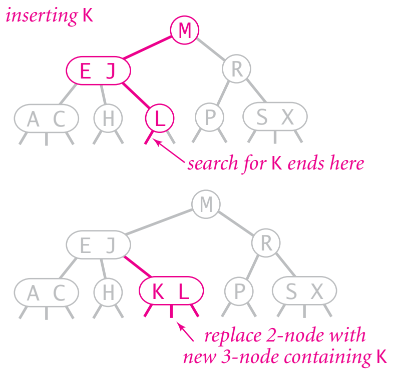
Search. The search algorithm for keys in a 2-3 tree directly generalizes the search algorithm for BSTs. To determine whether a key is in the tree, we compare it against the keys at the root: If it is equal to any of them, we have a search hit; otherwise, we follow the link from the root to the subtree corresponding to the interval of key values that could contain the search key, and then recursively search in that subtree.


successful search for H
H is less than M so look to the left
H is between E and L so look in the middle
found H so return value (search hit)
unsuccessful search for B
B is less than M so look to the left
B is less than C so look to the left
B is between A and C so look in the middle
link is null so B is not in the tree (search miss)
M
E J
R
A C
H
L
P
S X

Successful and unsuccessful search in a 2-3 tree

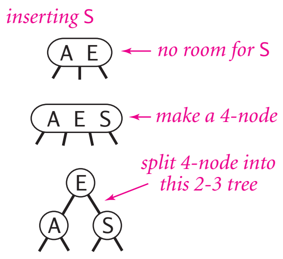
Balanced Trees

Insert into a 2-node. To insert a new node in a 2-3 tree, we might do an unsuccessful search and then hook on the node at the bottom, as we did with BSTs, but the new tree would not remain perfectly balanced. The primary reason that 2-3 trees are useful is that we can do insertions and still maintain perfect balance in the tree. It is easy to accomplish this task if the node at which the search terminates is a 2-node: We just replace the node with a 3-node containing its key and the new key to be inserted. If the node where the search terminates is a 3-node, we have more work to do.

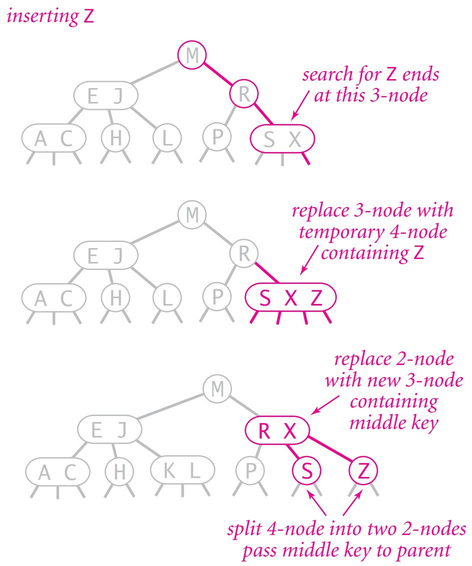


Insert into a 2-node

Insert into a tree consisting of a single 3-node. As a first warm-up before considering the general case, suppose that we want to insert into a tiny 2-3 tree consisting of just a single 3-node. Such a tree has two keys, but no room for a new key in its one node. To be able to perform the insertion, we temporarily put the new key into a *4-node*, a natural extension of our node type that has three keys and four links. Creating the 4-node is convenient because it is easy to convert it into a 2-3 tree made up of three 2-nodes, one with the middle key (at the root), one with the smallest of the three keys (pointed to by the left link of the root), and one with the largest of the three keys (pointed to by the right link of the root). Such a tree is a 3-node BST and also a perfectly balanced 2-3 search tree, with all the null links are at the same distance from the root. Before the insertion, the height of the tree is 1; after the insertion, the height of the tree is 2. This case is simple, but it is worth considering because it illustrates height growth in 2-3 trees.

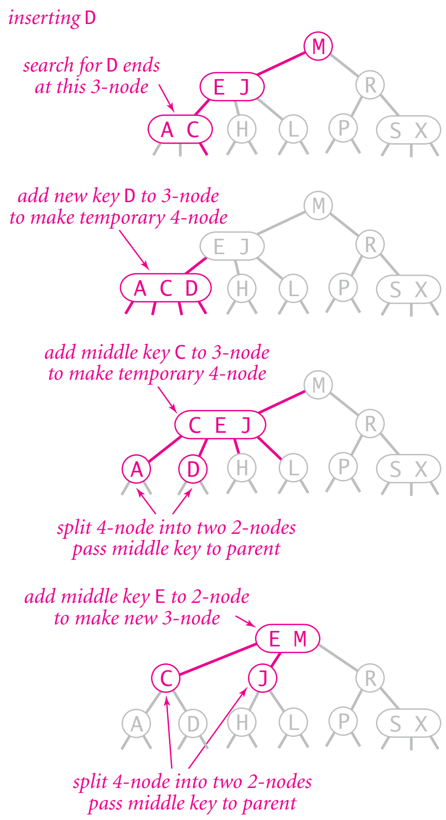


Insert into a single 3-node

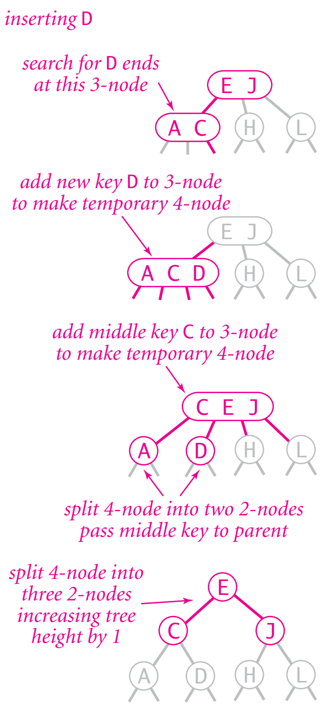

inserting Z
M
E J
R
A C
H
L
P
S X
search for Z ends at this 3-node
M
E J
R
A C
H
L
P
S X Z
replace 3-node with temporary 4-node containing Z
M
E J
R X
A C
H
K L
P
S
Z
replace 2-node with new 3-node containing middle key
split 4-node into two 2-nodes pass middle key to parent

Insert into a 3-node whose parent is a 2-node

Insert into a 3-node whose parent is a 2-node. As a second warm-up, suppose that the search ends at a 3-node at the bottom whose parent is a 2-node. In this case, we can still make room for the new key *while maintaining perfect balance in the tree*, by making a temporary 4-node as just described, then splitting the 4-node as just described, but then inserting the middle key in the node's parent (instead of putting it in its own 2-node). You can think of the transformation as replacing the link to the old 3-node in the parent by the middle key with links on either side to the new 2-nodes. By our assumption, there is room for doing so in the parent: the parent was a 2-node and becomes a 3-node. Be certain that you understand this transformation—it is the crux of 2-3 tree dynamics.

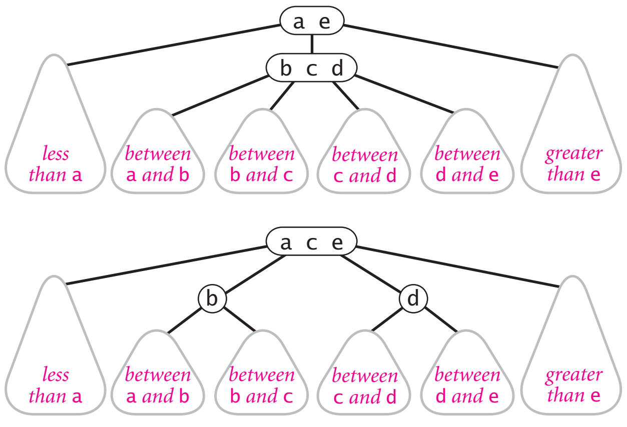

inserting D
search for D ends at this 3-node
add new key D to 3-node to make temporary 4-node
add middle key C to 3-node to make temporary 4-node
split 4-node into two 2-nodes pass middle key to parent
add middle key E to 2-node to make new 3-node
split 4-node into two 2-nodes pass middle key to parent
Insert into a 3-node whose parent is a 3-node

Insert into a 3-node whose parent is a 3-node. Now suppose that the search ends at a node whose parent is a 3-node. Again, we make a temporary 4-node as just described, then split it and insert its middle key into the parent. The parent was a 3-node, so we replace it with a temporary new 4-node containing the middle key from the 4-node split. Then, we perform *precisely the same transformation on that node*. That is we split the new 4-node and insert its middle key into *its* parent. Extending to the general case is clear: we continue up the tree, splitting 4-nodes and inserting their middle keys in their parents until reaching a 2-node, which we replace with a 3-node that does not to be further split, or until reaching a 3-node at the root. If we end up with a temporary 4-node at the root, we split it into three 2-nodes, increas-


inserting D
search for D ends at this 3-node
add new key D to 3-node to make temporary 4-node
add middle key C to 3-node to make temporary 4-node
split 4-node into two 2-nodes pass middle key to parent
split 4-node into three 2-nodes increasing tree height by 1
Splitting the root

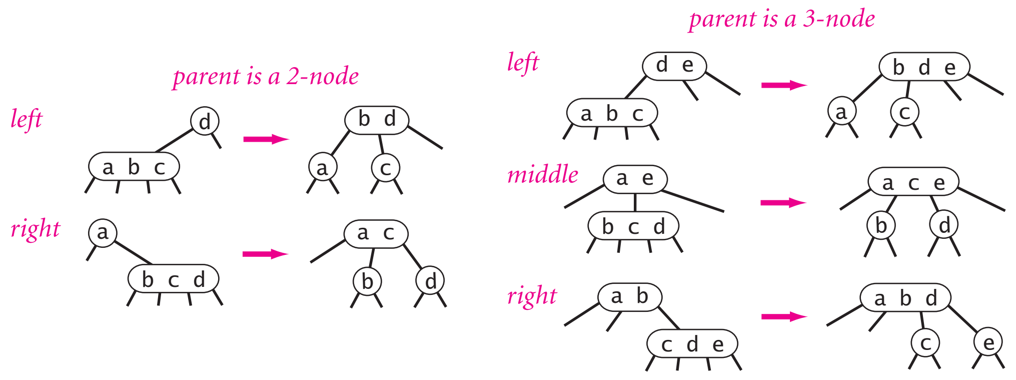
ing the height of the tree by 1, in just the same way as when inserting into a tree consisting of a single 3-node. Note that this transformation preserves perfect balance in the tree only when it is performed at the root.

Local transformations. Splitting a temporary 4-node in a 2-3 tree involves one of five transformations, summarized at the top of the next page. The 4-node may be the left child or the right child of a 2-node, or it may be the left child, middle child, or right child of a 3-node. The basis of the algorithm is that all of these transformations are purely *local*: No part of the tree needs to be examined or modified other than the specified nodes. The number of links changed for each transformation is bounded by a small constant. In particular, the transformations are effective when we find the specified patterns *anywhere* in the tree, not just at the bottom. Each of the transformations passes up one of the keys from a 4-node to that node's parent in the tree, and then restructures links accordingly.



Splitting a 4-node is a local transformation

Global properties. Moreover, these *local* transformations preserve the *global* properties that the tree is ordered and balanced. For reference, a complete diagram illustrating this point for the case that the 4-node is the middle child of a 3-node is shown here. If you are not fully convinced, you are encouraged to work EXERCISE 4.3.X, which asks you to extend the diagrams for the other four cases at the top of the next page to illustrate the same point. Before the split, if the height of the subtree rooted at any node in the tree is h, then the height of the subtree rooted at its parent is $h+1$. *Each transformation preserves this property*, even while splitting the 4-node into two 2-nodes and while changing the parent from a 2-node to a 3-node. or from a 3-node into a temporary 4-node, or when splitting the root into three 2-nodes and increasing the height of the whole tree by 1. Understanding that every transformation preserves order and perfect balance in the whole tree is the key to understanding the algorithm.



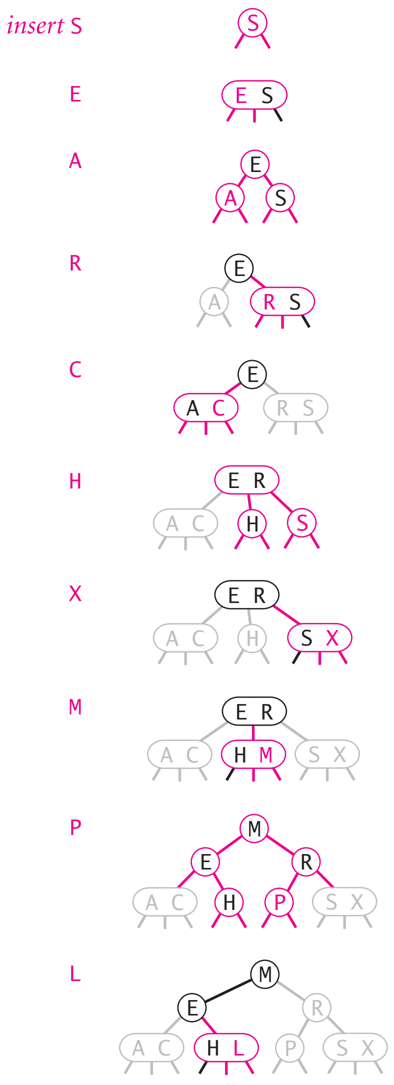
Splitting a temporary 4-node in a 2-3 tree (summary)

Unlike standard BSTs, which grow down from the top, 2-3 trees grow up from the bottom. If you take the time to carefully study the sequence of trees that are produced by our standard indexing test client and the sequence of trees that are produced when the same keys are inserted in increasing order, you will have a good understanding of the way that 2-3 trees are built. Recall that in a BST, the increasing-order sequence for 10 keys results in a worst-case tree where a search might involve examining all the keys. In the 2-3 trees, all keys can be found in every case by examining at most three nodes.

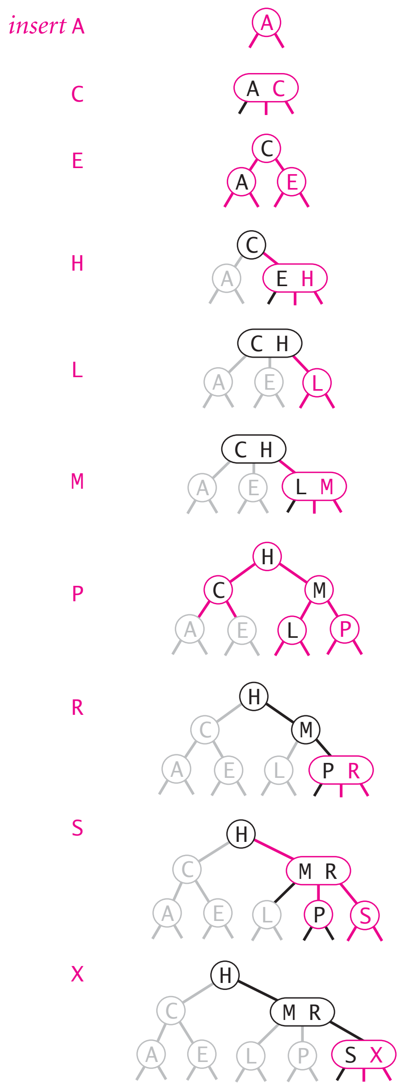
The preceding description is sufficient to define a symbol-table implementation with 2-3 trees as the underlying data structure. Analyzing 2-3 trees is different from analyzing BSTs because our primary interest is in *worst-case* performance, as opposed to average-case performance (where we analyze expected performance under an assumption that key values come from some random source). In symbol-table implementations, we normally have no control over the order in which clients insert keys into the table and worst-case analysis is one way to provide performance guarantees.

Property F. *Search and insert operations in 2-3 trees are guaranteed to complete in logarithmic time.*

Proof. The height of a N-node 2-3 tree is between $\log_3 N = (\lg N)/(\lg 3)$ (if the tree is all 3-nodes) and $\lg N$ (if the tree is all 2-nodes) (see Exercise 4.3.X). The amount of time required at each node by each of the operations is bounded by a constant, and both operations examine nodes on just one path. ■



standard indexing client

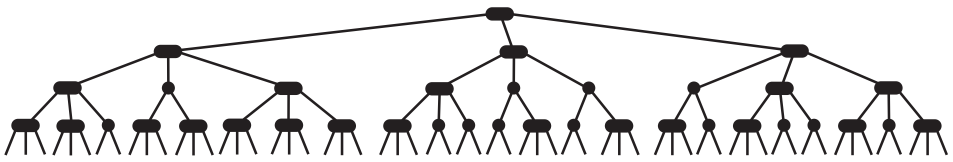


same keys in increasing order

2-3 tree construction traces

Thus, we can guarantee good worst-case performance with 2-3 trees. As you can see from examining the tree depicted at the bottom of this page, a perfectly balanced tree strikes a remarkably flat posture. For example, the height of a 2-3 tree that contains 1 billion keys is between nineteen and thirty. It is quite remarkable that we can guarantee to perform arbitrary search and insertion operations among 1 billion keys by examining less than thirty nodes.

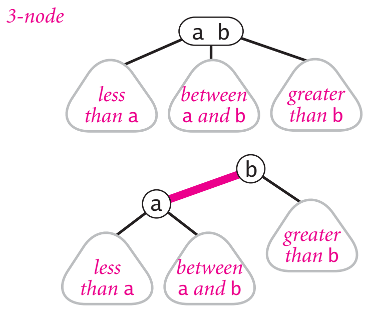
However, we are only part of the way to an implementation. Although it would be possible to write code that performs transformations on distinct data types representing 2- and 3-nodes, most of the tasks that we have described are inconvenient to implement in this direct representation because there are numerous different cases to be handled. We would need to maintain two different types of nodes, compare search keys against each of the keys in the nodes, copy links and other information from one type of node to another, convert nodes from one type to another, and so forth. Not only is there a substantial amount of code involved, but the overhead incurred could make the algorithms slower than standard BST search. The primary purpose of balancing is to provide insurance against a bad worst case, but we would prefer the overhead cost for that insurance to be low. Fortunately, as you will see, we can do the transformations in a uniform way using little overhead beyond the costs incurred by standard BST search.



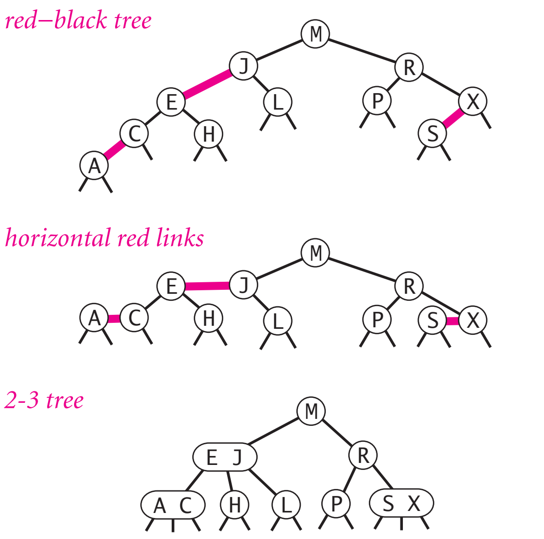
Typical 2-3 tree built from random keys

Red–Black Trees The insertion algorithm for 2-3 trees just described is not difficult to understand; now, we will see that it is also not difficult to implement. We will consider a simple representation known as *red-black trees* that leads to a natural implementation. In the end, not much code is required, but understanding how and why the code gets the job done requires a careful look.

Encoding 3-nodes The basic idea behind red-black trees is to encode 2-3 trees by starting with standard BSTs, which are made up of 2-nodes, and adding extra information to encode 3-nodes. We think of the links as being of two different types: *red* links, which bind together small binary trees that represent 3-nodes, and *black* links, which bind together the 2-3 tree. Specifically, we represent 3-nodes as two 2-nodes connected by a single red link. To define a 1-1 correspondence, we require that red links are always left links. One advantage of using such a representation is that it allows us to use our `get()` code for standard BST search *without modification*., while we never fully specified the search process within 3-nodes for 2-3 trees. Given any 2-3 tree, we can immediately derive a corresponding BST, just by converting each node as specified. We refer to BSTs that represent 2-3 trees in this way as (balanced) *red–black BSTs*.



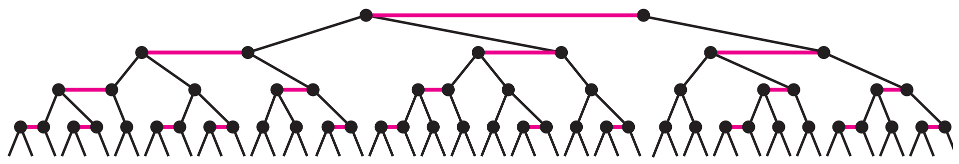
Encoding a 3-node with two 2-nodes connected by a left-leaning red link



1–1 correspondence between red-black and 2-3 trees

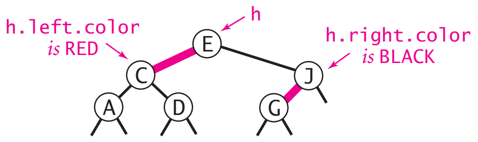
An equivalent definition Another way to proceed is to *define* red–black BSTs as BSTs having red and black links and satisfying the following three restrictions:

- Red links lean left.
- No node has two red links connected to it.
- The tree has *perfect black balance*: every path from the root to a null link has the same number of black links.


A red-black tree with horizontal red links is a 2-3 tree

There is a 1-1 correspondence between red-black BSTs defined in this way and 2-3 trees: if we draw the red links horizontally in a red–black BST, all of the null links are the same distance from the root, and if we then collapse together the nodes connected by red links, the result is a 2-3 tree. For convenience, we define the *black height* of a red–black BST as the number of black links on the path from the root to every null link (the height of the corresponding 2-3 tree). Whichever way we choose to define them, red–black BSTs are *both* BSTs and 2-3 trees. Thus, if we can implement the balanced 2-3 tree insertion algorithm by maintaining the 1-1 correspondence, then we get the best of both worlds: the simple and efficient search method from standard BSTs and the efficient insertion–balancing method from 2-3 trees.

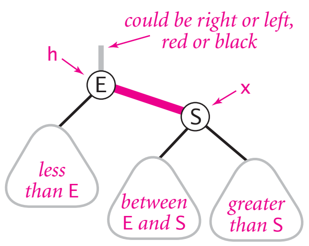
Representation For convenience, since each node is pointed to by precisely one link (from its parent), we encode the color of links in *nodes* (not links), by adding a `boolean` instance variable `color` to our `Node` data type, which is `true` if the (left) link from the parent is red and `false` if the link from the parent to the node is black (not red). For clarity in our code, we define constants `RED` and `BLACK` for use in setting and testing this variable. For reasons that will later become more clear, new nodes are always `RED`. We use a private method `isRed()` to test node color (instead of having an instance method in `Node`) to avoid having to test for `null` in client code.



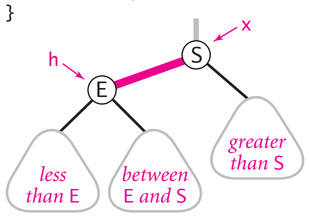
```
private static final boolean RED   = true;
private static final boolean BLACK = false;

private class Node
{
   Key key;           // key
   Value val;         // associated data
   Node left, right;  // subtrees
   int N;             // # nodes in this subtree
   boolean color;     // color of link from
                      //   parent to this node

   Node(Key key, Value val, int N, boolean color)
   {
      this.key   = key;
      this.val   = val;
      this.N     = 1;
      this.color = RED;
   }
}

private boolean isRed(Node x)
{
   if (x == null) return false;
   return x.color == RED;
}
```

Node representation for red–black trees

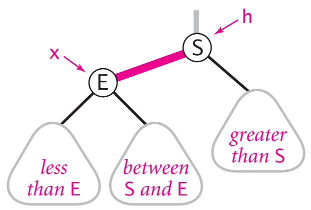
Rotations When you saw the definition of red–black BSTs, you may have asked yourself why the red links should all lean to the left. Why not allow some of them to lean to the right? This is a reasonable question that is addressed in the Q&A at the end of this section. Indeed, the standard implementation of red–black BSTs that has been in use for decades allows for right-leaning red links (see Exercise 4.3.TODO). In the implementation that we will consider, we also allow temporary right-leaning red links during an operation, but always make them lean to the left before the operation completes. Next, we consider the code that we need to switch orientation of red links. This code is crucial: by changing links in the trees, it brings them into better balance. First, suppose that we have a right-leaning red link that needs to be rotated to lean to the left (see the diagram at left). This operation is called a *left rotation*. We organize the computation as a method that takes a link to a red-black BST as argument, and assuming that link to be to a `Node h` whose right link is red, makes the necessary adjustments and returns a link to a node that is the root of a red–black BST for the same set of keys whose *left* link is red. If you think of the corresponding 3-node as you check each of the lines of code against the before/after drawings in the diagram, you will find this operation is easy to understand: we are switching from having the smaller of the two keys at the root to having the larger of the two keys at the root. Implementing a *right rotation* that converts a left-leaning red link to a right-leaning one amounts to the same code, with left and right interchanged. Whether left or right, every rotation leaves us with a link. Note that this link may be red or black—both `rotateLeft()` and `rotateRight()` take care to preserve its color by setting `x.color` to `h.color`. One implication of this decision is that our algorithms might temporarily allow two red links in a row to occur within the tree. Indeed, we use rotations precisely to correct this condition when it arises. In our algorithms, we always use the link returned by `rotateRight()` or `rotateLeft()` to reset the appropriate link in the parent (or the root of the tree). That may



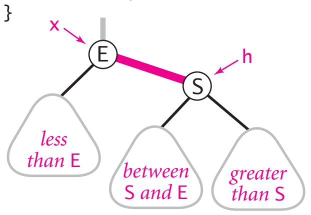
```
Node rotateLeft(Node h)
{
   x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```



Left rotate (right link of `h`*)*



```
Node rotateRight(Node h)
{
   x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```

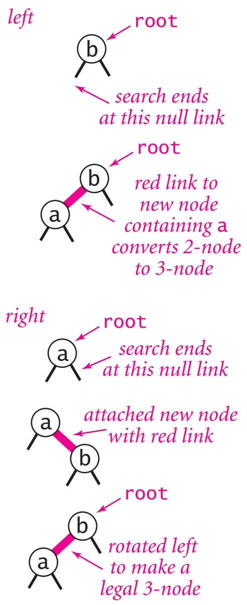


Right rotate (left link of `h`*)*

be a right or a left link, but we can always use the returned link to reset the argument link. For example, the code

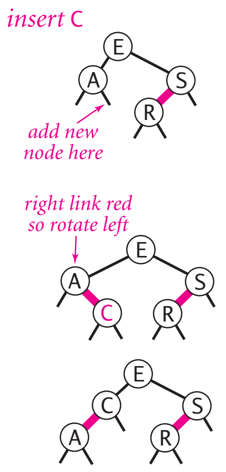
```
    h = rotateLeft(h);
```

rotates left a right-leaning red link that is to the left of node `h`, setting h to point to the root of the resulting subtree (which contains all the same nodes as the subtree pointed to by `h` before the rotation, but a different root). The ease of writing this type of code is the primary reason we use recursive implementations of BST methods, as it makes doing rotations an easy addition to normal search, as you will see. The reason that we can use rotations to help maintain the 1-1 correspondence between 2-3 trees and red–black BSTs as new keys are inserted is that they preserve a very important property: *they preserve order and perfect black balance*. That is, we can use rotations on a red–black BST without have to worry about losing its order and its perfect black balance. As just mentioned, our approach is to use rotation to eliminate connected red links. As before, we warm up with some easy cases.


left
root
b
search ends at this null link
root
b
a
red link to new node containing a converts 2-node to 3-node
right
root
a
search ends at this null link
a
attached new node with red link
b
root
b
a
rotated left to make a legal 3-node

Insert into a single 2-node

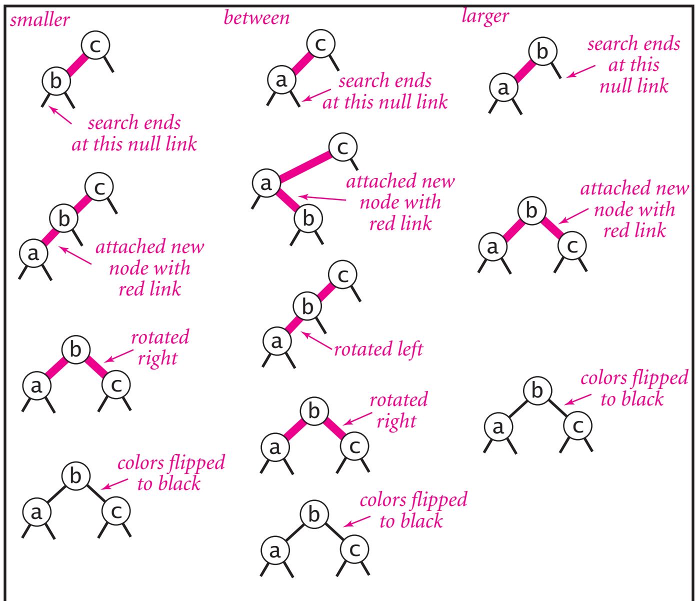
Insert into a single 2-node A red-black tree with 1 key is just a single 2-node. Inserting the second key immediately shows the need for having a rotation operation. If the new key is smaller than the key in the tree, we just make a new (red) node with the new key and we are done: we have a red-black tree that is equivalent to a single 3-node. But if the new key is larger than the key in the tree, then attaching a new (red) node gives a right-leaning red link, and the code `root = rotateLeft(root);` completes the insertion by rotating the red link to the left and updating the tree root link. The result in both cases is the red-black representation of a single 3-node, with two keys, one left-leaning red link and black height 1.



Insert into a 2-node at the bottom

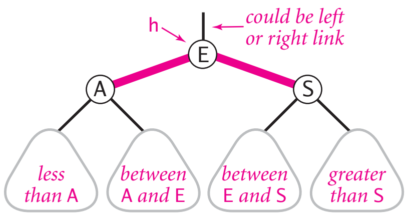
Insert into a 2-node at the bottom We insert keys into a red-black tree as usual into a BST, adding a new node at the bottom (respecting the order), but always connected to its parent with a red link. If the parent is a 2-node, then the same two cases just discussed are effective. If the new node is attached to the left link, the parent simply becomes a 3-node; if it is attached to a right link, we have a 3-node leaning the wrong way, but a left rotation finishes the job.

Insert into a tree with 2 keys (in a 3-node) This case reduces to three subcases, depending on whether the new key is less than both keys in the tree, between them, or greater than both of them. Each of the cases introduces a node with two red links connected to it: our goal is to correct this condition. The simplest of the three cases is when the new key is larger than the two in the tree and is therefore attached on the rightmost link of the 3-node, making a balanced tree with the middle key at the root, connected with red links to nodes containing smaller and a larger key. If we flip the colors of those two links from red to black, then we have a three-node balanced tree, of height 2, exactly what we need to maintain our 1-1 correspondence to 2-3 trees. The other two cases eventually reduce to this case. If the new key is smaller than the keys in the tree and goes on the left link, then we have two red links in a row, both leaning to the left, which we can reduce to the previous case by rotating the top link to the right. If the new key goes in the middle, we again have two red links in a row, a right-leaning one below a left leaning one,which we can reduce to the previous case by rotating left the bottom link. In summary, we achieve the desired result by doing zero, one, or two rotations followed by flipping the colors of the two children of the root. As with 2-3 trees, *be certain that you understand these transformations*, as they are the key to red-black tree dynamics.

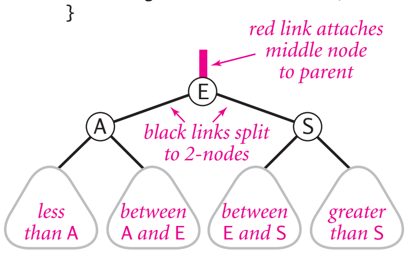

smaller
between
larger
search ends at this null link
search ends at this null link
search ends at this null link
attached new node with red link
attached new node with red link
attached new node with red link
rotated right
rotated left
rotated right
colors flipped to black
colors flipped to black
colors flipped to black
a
b
c

Insert into a single 3-node (three cases)

Flipping colors To flip the colors of the children of a node, we use a method `flipColors()`, shown at right. In addition to flipping the colors of the children from red to black, we also flip the color of the parent from black to red. (In the case just considered, this will color the root red—by convention, we color the root black after each insertion, which increases the black height of the tree by 1 if it was red.) A critically important characteristic of this operation is that, like rotations, it is a local transformation that *preserves perfect black balance* in the tree. Moreover, this convention immediately leads us to a full implementation, as we describe next.

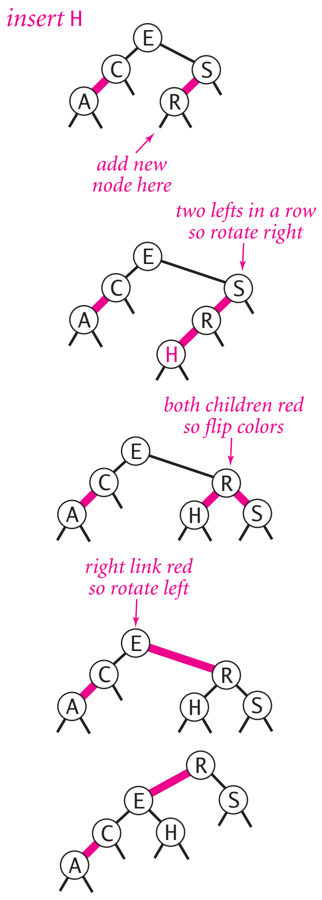


```
void flipColors(Node h)
{
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```


red link attaches middle node to parent
E
A
S
black links split to 2-nodes
less than A
between A and E
between E and S
greater than S

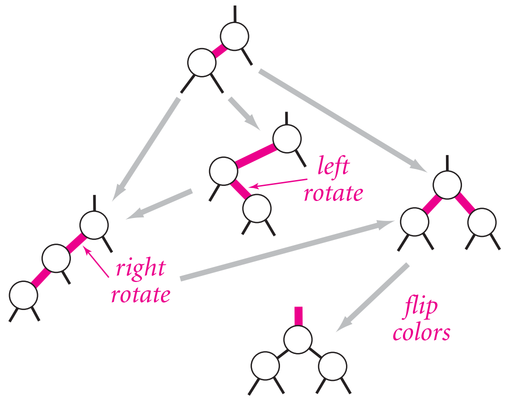
Flipping colors to split a 4-node

Insert into a 3-node at the bottom Now suppose that we add a new node at the bottom that is connected to a 3-node. The same three cases just discussed arise. Either the new link is connected to the left link of the 3-node (in which case we need to rotate the top link right and flip colors) or to the right link of the 3-node (in which case we just flip colors) or to the middle link of the 3-node (in which case we rotate left the bottom link, then rotate right the top link, then flip colors). Flipping the colors makes the link to the middle node red, which amounts to passing it up to its parent, putting us back in the same situation with respect to the parent, which we can fix by moving up the tree.


insert H
E
C
S
A
R
add new node here
two lefts in a row so rotate right
E
C
S
A
R
H
both children red so flip colors
E
C
R
A
H
S
right link red so rotate left
E
C
R
A
H
S
R
E
S
C
H
A

Insert into a 3-node at the bottom

Passing a red link up the tree The 2-3 insertion algorithm calls for us to split the 3-node, passing the middle key up to be inserted into its parent, continuing until encountering a 2-node or the root. In every case we have considered, we precisely accomplish this objective: after doing any necessary rotations, we flip colors, which turns the middle node to red. From the point of view of the parent of that node, that link becoming red can be handled in precisely the same manner as if the red link came from attaching a new node: we pass up a red link to the middle node. The three cases summarized in the diagram on the next page precisely capture the operations necessary in a red-black tree to implement the key operation in 2-3 tree insertion: to insert into a 3-node, create a temporary 4-node, split it, and pass a red link to the middle key up to its parent. Continuing the same process, we pass a red link up the tree until reaching a 2-node or the root.



Passing a red link up a red-black tree

In summary, we can maintain our 1-1 correspondence between 2-3 trees and red–black trees during insertion by judicious use of three simple operations: left rotate, right rotate, and color flip. we can accomplish the insertion by performing the following operations, one after the other, on each node as we pass up the tree from the point of insertion:

- If the right child is red and the left child is not red, rotate left.
- If both the left child and its left child are red, rotate right.
- If both children are red, flip colors.

It certainly is worth your while to check that this sequence handles each of the cases just described. Note that the first operation handles both the rotation necessary to lean the 3-node to the left when the parent is a 2-node and the rotation necessary to lean the bottom link to the left when the new red link is the middle link in a 3-node.

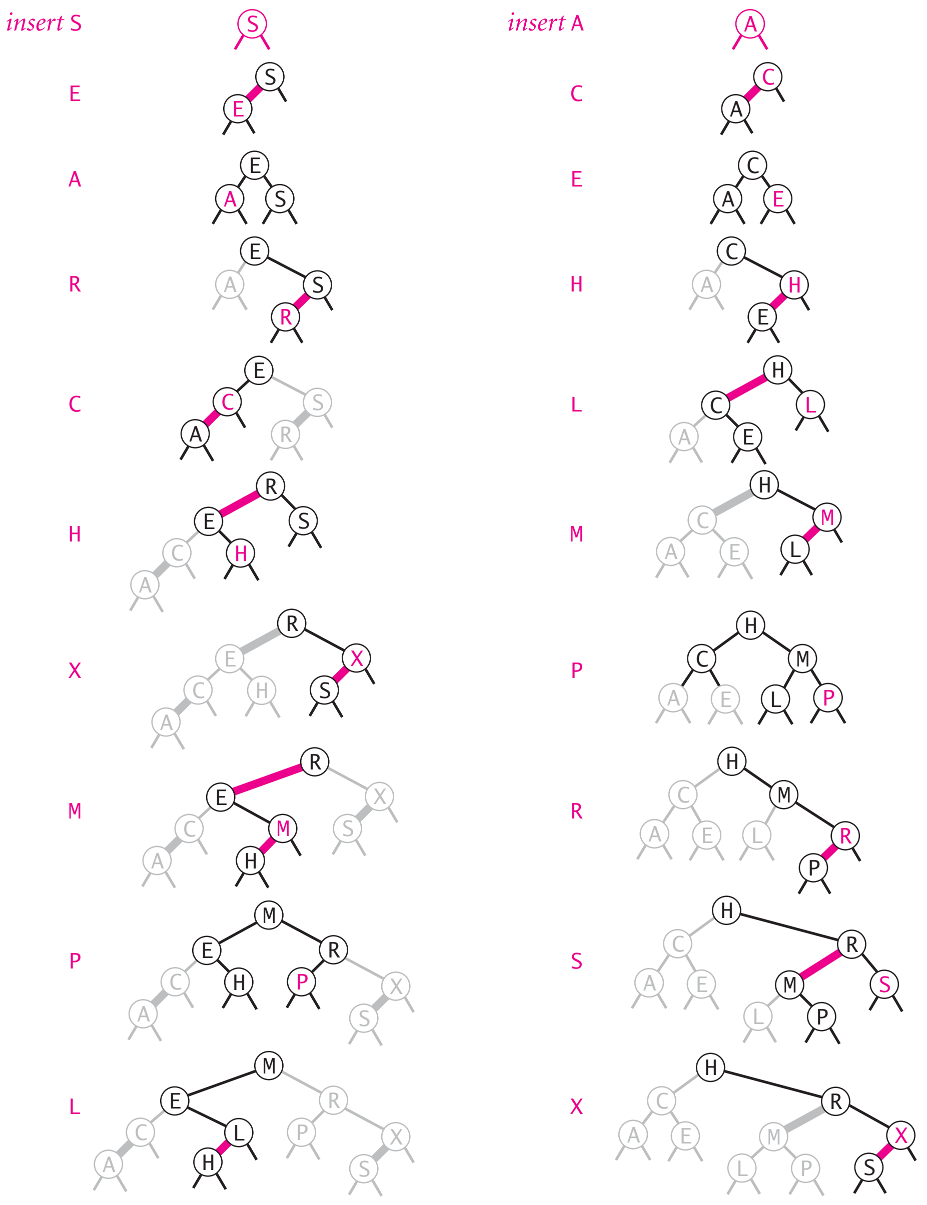
Implementation Since the balancing operations are to be performed on the way *up* the tree from the point of insertion, implementing them is easy in our standard recursive implementation: we just do them after the recursive calls, as shown in Algorithm 4.4. The three operations listed in the previous paragraph are encapsulated in a method `fixUp()`. Even though it involves a small amount of code, this implementation would be quite difficult to understand without the two layers of abstraction that we have developed (2-3 trees and red-black trees) to implement it. At a cost of testing three to five link colors (and perhaps doing a rotation or two or flipping colors when a test succeeds), we get trees that have nearly perfect balance.

The traces for our standard indexing client and for the same keys inserted in increasing order is given on the facing page. Considering these examples simply in

Algorithm 4.4 Insert for red–black BSTs

```
public class RedBlackBST<Key extends Comparable<Key>, Value>
{
   private class Node
   // Standard BST Node with color bit added -- see text.

   private boolean isRed(Node h)
   private Node rotateLeft(Node h)
   private Node rotateRight(Node h)
   private void flipColors(Node h)
   // See text for implementations of these methods.

   private Node fixUp(Node h)
   {
      if (isRed(h.right) && !isRed(h.left))    h = rotateLeft(h);
      if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);
      if (isRed(h.left) && isRed(h.right))     flipColors(h);

      h.N = size(h.left) + size(h.right) + 1;
      return h;
   }

   public void put(Key key, Value val)
   {  root = put(root, key, val);  }

   private Node put(Node h, Key key, Value val)
   {
      if (h == null)
         return new Node(key, val, 1, RED);

      int cmp = key.compareTo(h.key);
      if      (cmp < 0) h.left  = insert(h.left,  key, val);
      else if (cmp > 0) h.right = insert(h.right, key, val);
      else h.val = val;

      return fixUp(h);
   }
}
```

The code for the recursive `put()` *for red-black BSTs is identical to* `put()` *in elementary BSTs except for the* `fixUp()` *method, which uses a color bit in* `Node` *to provide near-perfect balance in the tree by maintaining a 1-1 correspondence with 2-3 trees.*



standard indexing client

same keys in increasing order

red-black tree construction traces

terms of our three operations on red–black trees, as we have been doing, is an instructive exercise. Another instructive exercise is to check the correspondence with 2-3 trees that the algorithm maintains (using the figure for the same keys given earlier in this section). In both cases, you can test your understanding of the algorithm by considering the transformations (two color flips and two rotations) that are needed when `P` is inserted into the tree.