instead of > specifically to ensure that single rotations are done in these cases rather than double rotations. We leave verification of the remaining cases as an exercise.

4.5 Splay Trees

We now describe a relatively simple data structure known as a **splay tree** that guarantees that any M consecutive tree operations starting from an empty tree take at most $O(M \log N)$ time. Although this guarantee does not preclude the possibility that any *single* operation might take $\Theta(N)$ time, and thus the bound is not as strong as an $O(\log N)$ worst-case bound per operation, the net effect is the same: There are no bad input sequences. Generally, when a sequence of M operations has total worst-case running time of $O(Mf(N))$, we say that the **amortized** running time is $O(f(N))$. Thus, a splay tree has an $O(\log N)$ amortized cost per operation. Over a long sequence of operations, some may take more, some less.

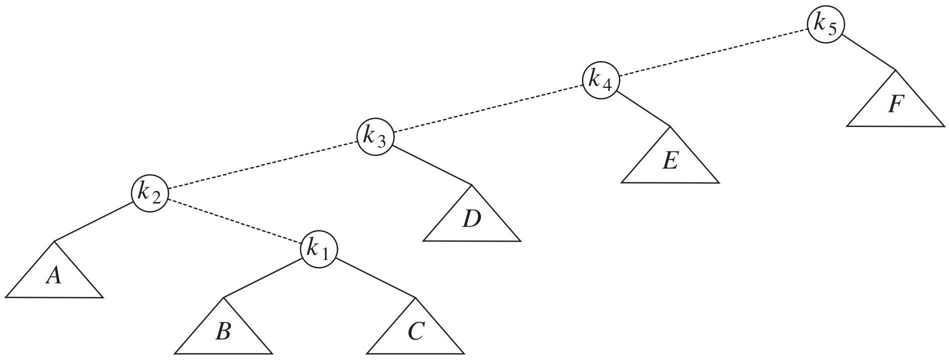
Splay trees are based on the fact that the $O(N)$ worst-case time per operation for binary search trees is not bad, as long as it occurs relatively infrequently. Any one access, even if it takes $\Theta(N)$, is still likely to be extremely fast. The problem with binary search trees is that it is possible, and not uncommon, for a whole sequence of bad accesses to take place. The cumulative running time then becomes noticeable. A search tree data structure with $O(N)$ worst-case time, but a *guarantee* of at most $O(M \log N)$ for any M consecutive operations, is certainly satisfactory, because there are no bad sequences.

If any particular operation is allowed to have an $O(N)$ worst-case time bound, and we still want an $O(\log N)$ amortized time bound, then it is clear that whenever a node is accessed, it must be moved. Otherwise, once we find a deep node, we could keep performing accesses on it. If the node does not change location, and each access costs $\Theta(N)$, then a sequence of M accesses will cost $\Theta(M \cdot N)$.

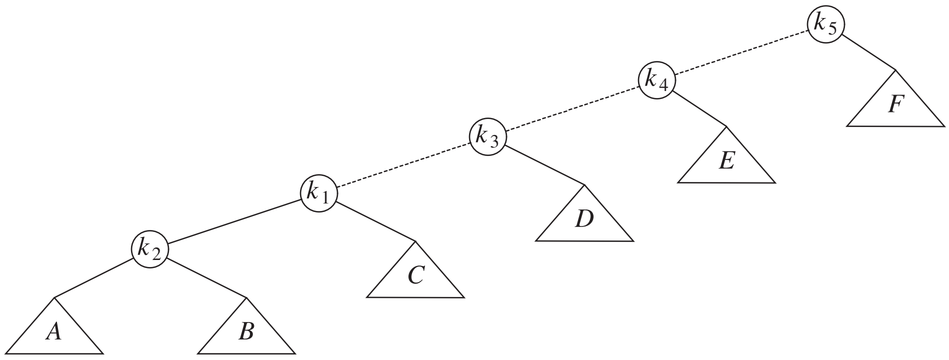
The basic idea of the splay tree is that after a node is accessed, it is pushed to the root by a series of AVL tree rotations. Notice that if a node is deep, there are many nodes on the path that are also relatively deep, and by restructuring we can make future accesses cheaper on all these nodes. Thus, if the node is unduly deep, then we want this restructuring to have the side effect of balancing the tree (to some extent). Besides giving a good time bound in theory, this method is likely to have practical utility, because in many applications, when a node is accessed, it is likely to be accessed again in the near future. Studies have shown that this happens much more often than one would expect. Splay trees also do not require the maintenance of height or balance information, thus saving space and simplifying the code to some extent (especially when careful implementations are written).

4.5.1 A Simple Idea (That Does Not Work)

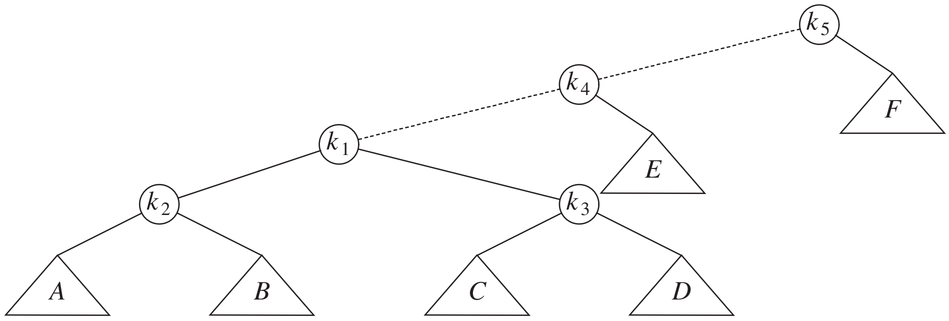
One way of performing the restructuring described above is to perform single rotations, bottom up. This means that we rotate every node on the access path with its parent. As an example, consider what happens after an access (a `find`) on k_1 in the following tree:



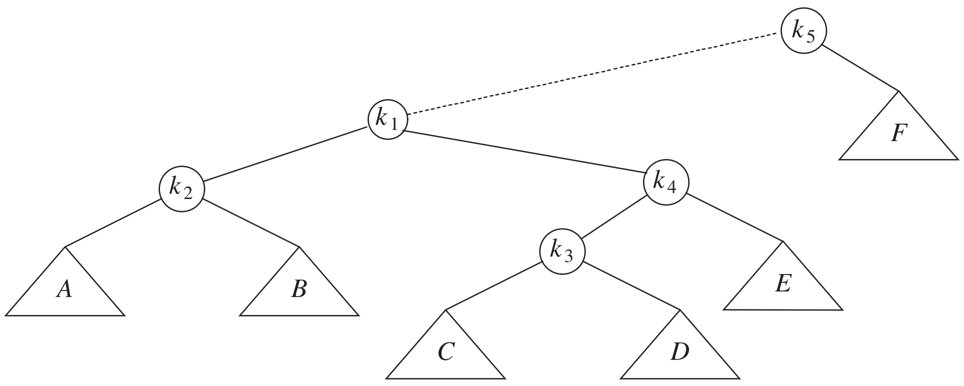
The access path is dashed. First, we would perform a single rotation between k_1 and its parent, obtaining the following tree:

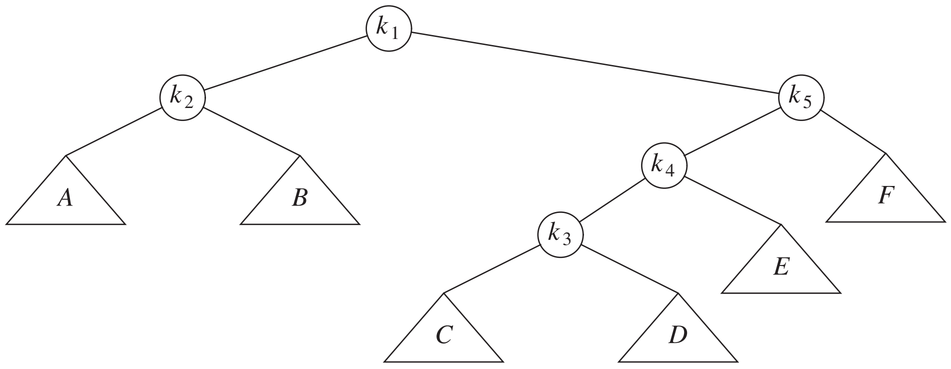


Then, we rotate between k_1 and k_3, obtaining the next tree:



Then two more rotations are performed until we reach the root:





These rotations have the effect of pushing k_1 all the way to the root, so that future accesses on k_1 are easy (for a while). Unfortunately, it has pushed another node (k_3) almost as deep as k_1 used to be. An access on that node will then push another node deep, and so on. Although this strategy makes future accesses of k_1 cheaper, it has not significantly improved the situation for the other nodes on the (original) access path. It turns out that it is possible to prove that using this strategy, there is a sequence of M operations requiring $\Omega(M \cdot N)$ time, so this idea is not quite good enough. The simplest way to show this is to consider the tree formed by inserting keys $1, 2, 3, \ldots, N$ into an initially empty tree (work this example out). This gives a tree consisting of only left children. This is not necessarily bad, though, since the time to build this tree is $O(N)$ total. The bad part is that accessing the node with key 1 takes N units of time, where each node on the access path counts as one unit. After the rotations are complete, an access of the node with key 2 takes N units of time, key 3 takes $N - 1$ units, and so on. The total for accessing all the keys in order is $N + \sum_{i=2}^{N} i = \Omega(N^2)$. After they are accessed, the tree reverts to its original state, and we can repeat the sequence.

4.5.2 Splaying

The splaying strategy is similar to the rotation idea above, except that we are a little more selective about how rotations are performed. We will still rotate bottom up along the access

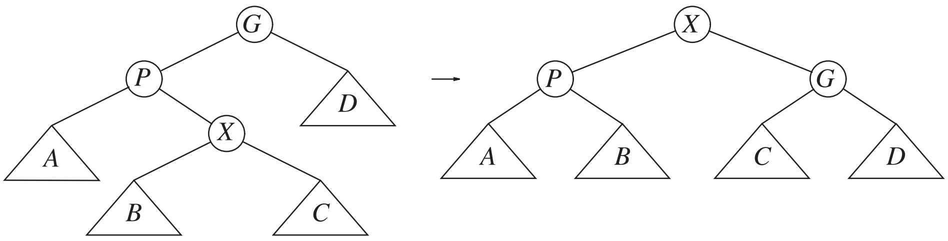


Figure 4.48 Zig-zag

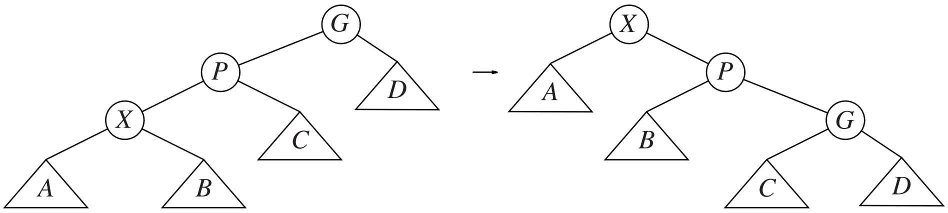
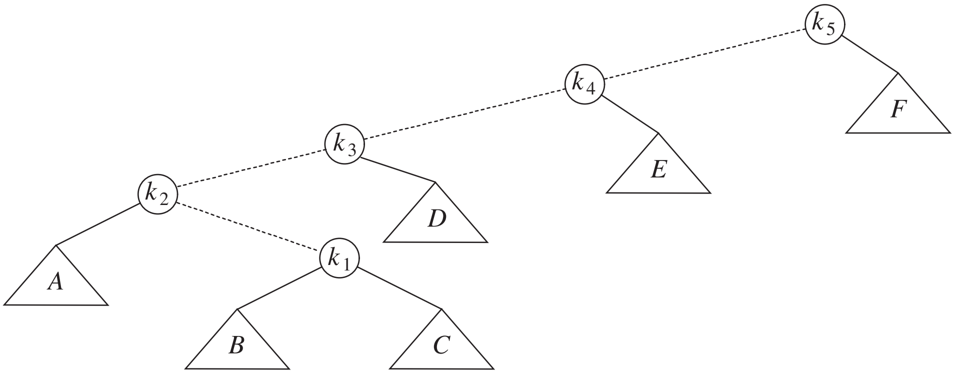


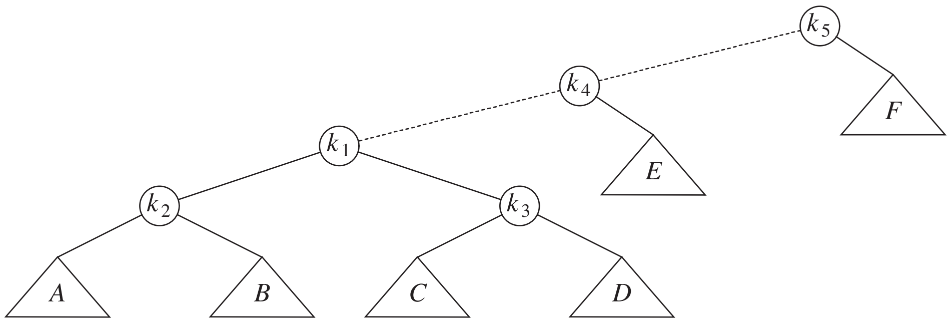
Figure 4.49 Zig-zig

path. Let X be a (non-root) node on the access path at which we are rotating. If the parent of X is the root of the tree, we merely rotate X and the root. This is the last rotation along the access path. Otherwise, X has both a parent (P) and a grandparent (G), and there are two cases, plus symmetries, to consider. The first case is the **zig-zag** case (see Fig. 4.48). Here X is a right child and P is a left child (or vice versa). If this is the case, we perform a double rotation, exactly like an AVL double rotation. Otherwise, we have a **zig-zig** case: X and P are both left children (or, in the symmetric case, both right children). In that case, we transform the tree on the left of Figure 4.49 to the tree on the right.

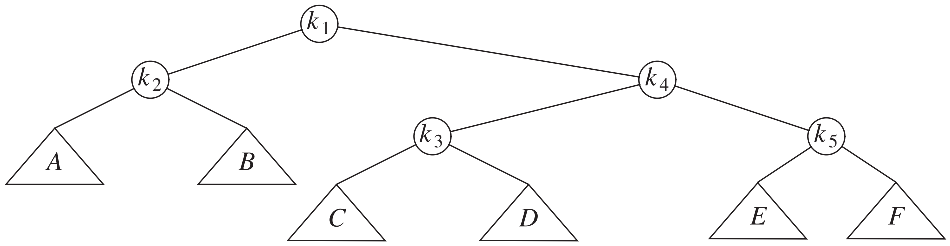
As an example, consider the tree from the last example, with a `contains` on k_1:



The first splay step is at k_1 and is clearly a *zig-zag*, so we perform a standard AVL double rotation using k_1, k_2, and k_3. The resulting tree follows:



The next splay step at k_1 is a *zig-zig*, so we do the *zig-zig* rotation with k_1, k_4, and k_5, obtaining the final tree:



Although it is hard to see from small examples, splaying not only moves the accessed node to the root but also has the effect of roughly halving the depth of most nodes on the access path (some shallow nodes are pushed down at most two levels).

To see the difference that splaying makes over simple rotation, consider again the effect of inserting items $1, 2, 3, \ldots, N$ into an initially empty tree. This takes a total of $O(N)$, as before, and yields the same tree as simple rotations. Figure 4.50 shows the result of splaying at the node with item 1. The difference is that after an access of the node with item 1, which

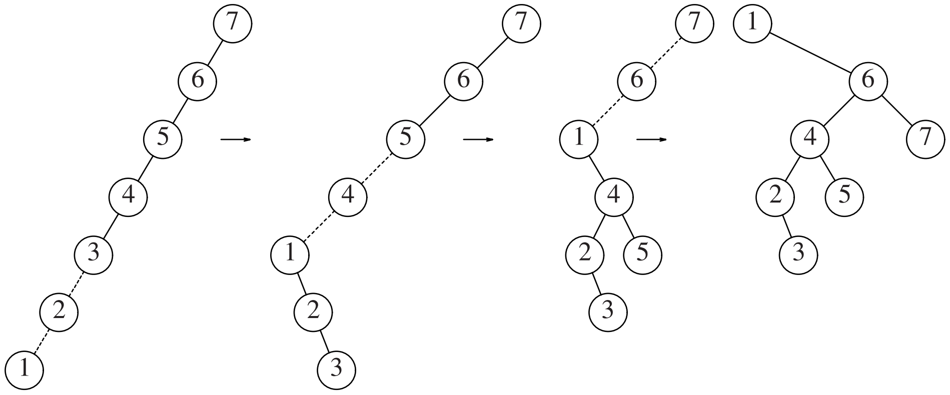


Figure 4.50 Result of splaying at node 1

takes N units, the access on the node with item 2 will only take about $N/2$ units instead of N units; there are no nodes quite as deep as before.

An access on the node with item 2 will bring nodes to within $N/4$ of the root, and this is repeated until the depth becomes roughly $\log N$ (an example with $N = 7$ is too small to see the effect well). Figures 4.51 to 4.59 show the result of accessing items 1 through 9 in a 32-node tree that originally contains only left children. Thus we do not get the same bad behavior from splay trees that is prevalent in the simple rotation strategy. (Actually, this turns out to be a very good case. A rather complicated proof shows that for this example, the N accesses take a total of $O(N)$ time.)

These figures highlight the fundamental and crucial property of splay trees. When access paths are long, thus leading to a longer-than-normal search time, the rotations tend to be good for future operations. When accesses are cheap, the rotations are not as good and can be bad. The extreme case is the initial tree formed by the insertions. All the insertions were constant-time operations leading to a bad initial tree. At that point in time, we had a very bad tree, but we were running ahead of schedule and had the compensation of less total running time. Then a couple of really horrible accesses left a nearly balanced tree, but the cost was that we had to give back some of the time that had been saved. The main theorem, which we will prove in Chapter 11, is that we never fall behind a pace of $O(\log N)$ per operation: We are always on schedule, even though there are occasionally bad operations.

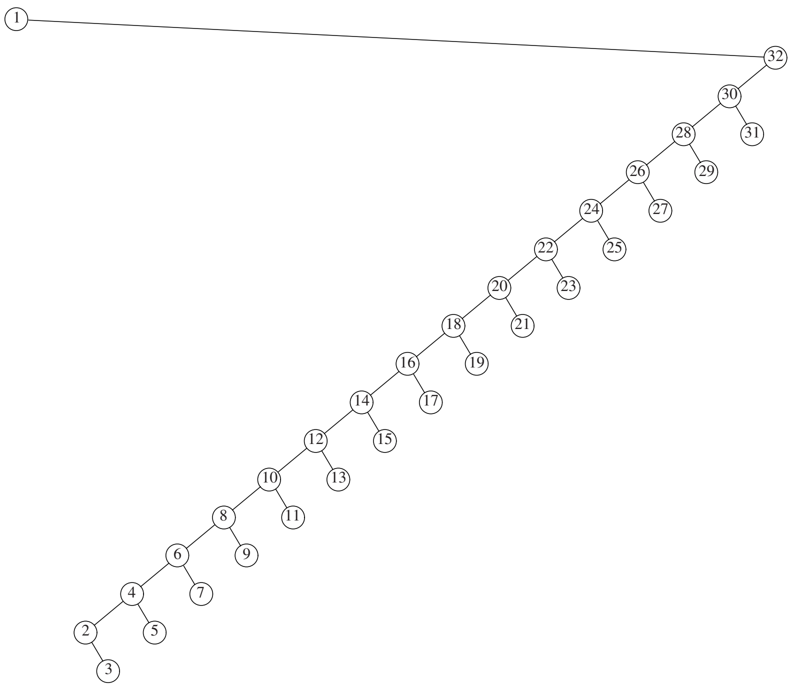


Figure 4.51 Result of splaying at node 1 a tree of all left children

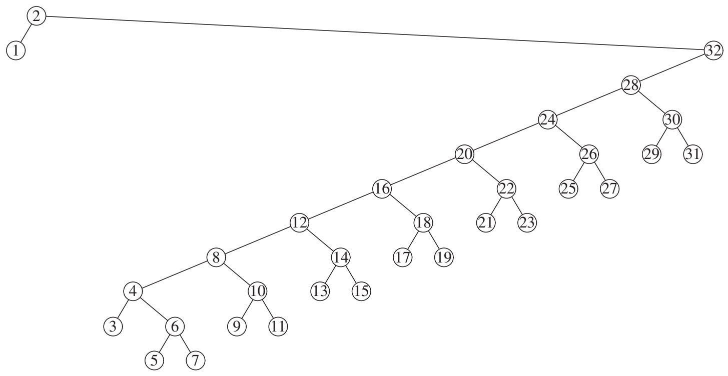


Figure 4.52 Result of splaying the previous tree at node 2

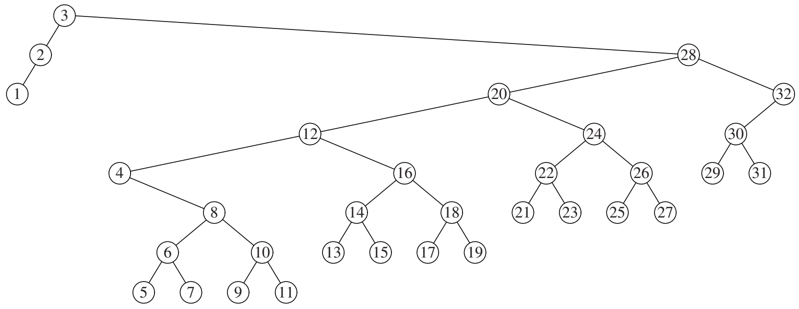


Figure 4.53 Result of splaying the previous tree at node 3

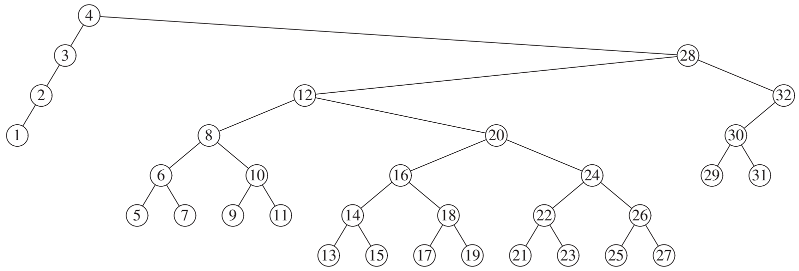


Figure 4.54 Result of splaying the previous tree at node 4

We can perform deletion by accessing the node to be deleted. This puts the node at the root. If it is deleted, we get two subtrees T_L and T_R (left and right). If we find the largest element in T_L (which is easy), then this element is rotated to the root of T_L, and T_L will now have a root with no right child. We can finish the deletion by making T_R the right child.

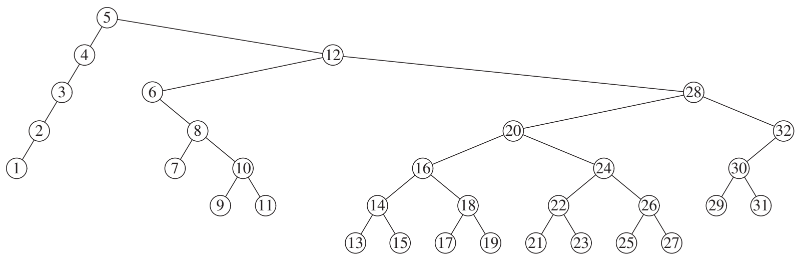


Figure 4.55 Result of splaying the previous tree at node 5

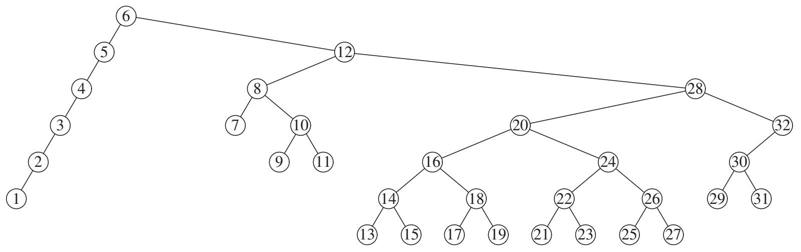


Figure 4.56 Result of splaying the previous tree at node 6

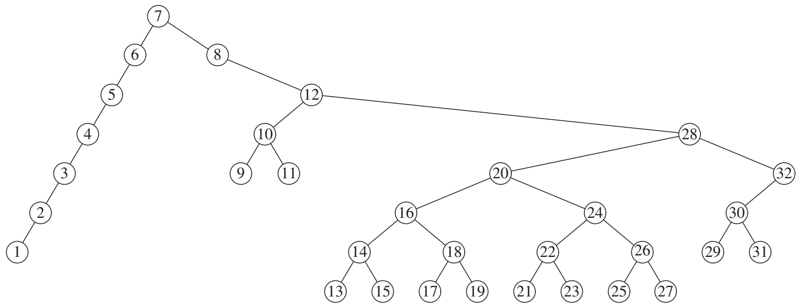


Figure 4.57 Result of splaying the previous tree at node 7

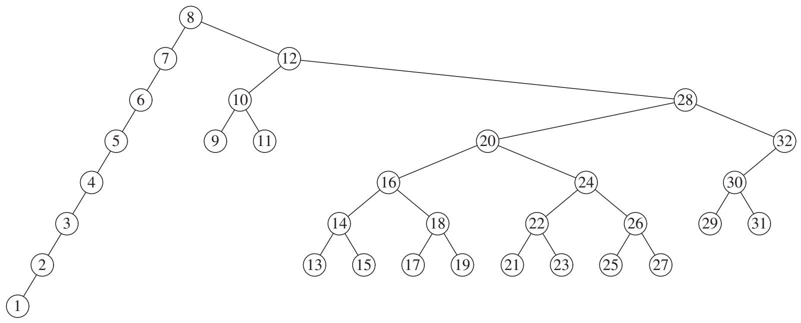


Figure 4.58 Result of splaying the previous tree at node 8

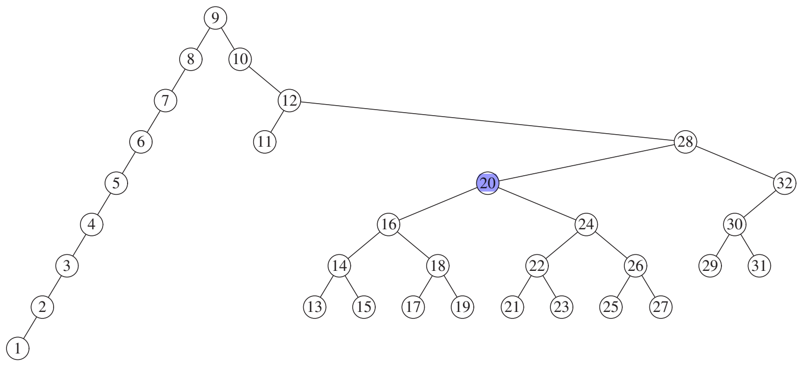


Figure 4.59 Result of splaying the previous tree at node 9

The analysis of splay trees is difficult, because it must take into account the ever-changing structure of the tree. On the other hand, splay trees are much simpler to program than most balanced search trees, since there are fewer cases to consider and no balance information to maintain. Some empirical evidence suggests that this translates into faster code in practice, although the case for this is far from complete. Finally, we point out that there are several variations of splay trees that can perform even better in practice. One variation is completely coded in Chapter 12.

~~4.6 Tree Traversals (Revisited)~~

~~Because of the ordering information in a binary search tree, it is simple to list all the items in sorted order. The recursive function in Figure 4.60 does the real work.~~

~~Convince yourself that this function works. As we have seen before, this kind of routine when applied to trees is known as an **inorder traversal** (which makes sense, since it lists the items in order). The general strategy of an inorder traversal is to process the left subtree first, then perform processing at the current node, and finally process the right subtree. The interesting part about this algorithm, aside from its simplicity, is that the total running time is $O(N)$. This is because there is constant work being performed at every node in the tree. Each node is visited once, and the work performed at each node is testing against `nullptr`, setting up two function calls, and doing an output statement. Since there is constant work per node and N nodes, the running time is $O(N)$.~~

~~Sometimes we need to process both subtrees first before we can process a node. For instance, to compute the height of a node, we need to know the height of the subtrees first. The code in Figure 4.61 computes this. Since it is always a good idea to check the special cases—and crucial when recursion is involved—notice that the routine will declare the height of a leaf to be zero, which is correct. This general order of traversal, which we have also seen before, is known as a **postorder traversal**. Again, the total running time is $O(N)$, because constant work is performed at each node.~~

CHAPTER 12

Advanced Data Structures and Implementation

In this chapter, we discuss six data structures with an emphasis on practicality. We begin by examining alternatives to the AVL tree discussed in Chapter 4. These include an optimized version of the splay tree, the red-black tree, and the treap. We also examine the *suffix tree*, which allows searching for a pattern in a large text.

We then examine a data structure that can be used for multidimensional data. In this case, each item may have several keys. The *k*-d tree allows searching relative to any key.

Finally, we examine the pairing heap, which seems to be the most practical alternative to the Fibonacci heap.

Recurring themes include . . .

- Nonrecursive, top-down (instead of bottom-up) search tree implementations when appropriate.
- Implementations that make use of, among other things, sentinel nodes.

12.1 Top-Down Splay Trees

In Chapter 4, we discussed the basic splay tree operation. When an item, X, is inserted as a leaf, a series of tree rotations, known as a *splay*, makes X the new root of the tree. A splay is also performed during searches, and if an item is not found, a splay is performed on the last node on the access path. In Chapter 11, we showed that the *amortized* cost of a splay tree operation is $O(\log N)$.

A direct implementation of this strategy requires a traversal from the root down the tree, and then a bottom-up traversal to implement the splaying step. This can be done either by maintaining parent links, or by storing the access path on a stack. Unfortunately, both methods require a substantial amount of overhead, and both must handle many special cases. In this section, we show how to perform rotations on the initial access path. The result is a procedure that is faster in practice, uses only $O(1)$ extra space, but retains the $O(\log N)$ amortized time bound.

Figure 12.1 shows the rotations for the *zig*, *zig-zig*, and *zig-zag* cases. (As is customary, three symmetric rotations are omitted.) At any point in the access, we have a current node,

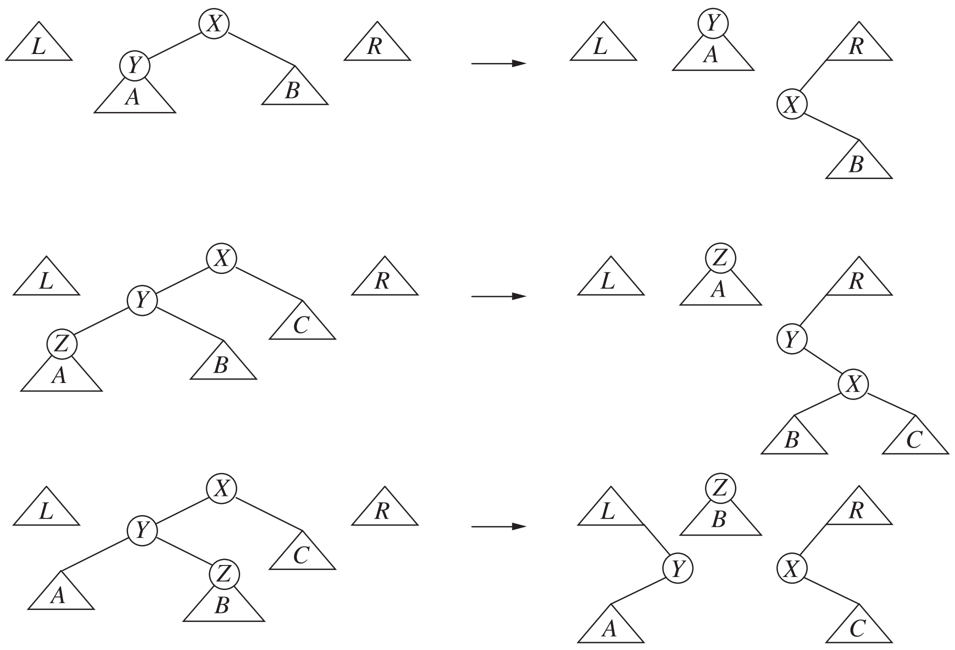


Figure 12.1 Top-down splay rotations: *zig*, *zig-zig*, and *zig-zag*

X, that is the root of its subtree; this is represented in our diagrams as the "middle" tree.[1] Tree *L* stores nodes in the tree *T* that are less than *X*, but not in *X*'s subtree; similarly tree *R* stores nodes in the tree *T* that are larger than *X*, but not in *X*'s subtree. Initially, *X* is the root of *T*, and *L* and *R* are empty.

If the rotation should be a *zig*, then the tree rooted at *Y* becomes the new root of the middle tree. *X* and subtree *B* are attached as a left child of the smallest item in *R*; *X*'s left child is logically made `nullptr`.[2] As a result, *X* is the new smallest item in *R*. Note carefully that *Y* does not have to be a leaf for the *zig* case to apply. If we are searching for an item that is smaller than *Y*, and *Y* has no left child (but does have a right child), then the *zig* case will apply.

For the *zig-zig* case, we have a similar dissection. The crucial point is that a rotation between *X* and *Y* is performed. The *zig-zag* case brings the bottom node *Z* to the top in the middle tree, and attaches subtrees *X* and *Y* to *R* and *L*, respectively. Note that *Y* is attached to, and then becomes, the largest item in *L*.

The *zig-zag* step can be simplified somewhat because no rotations are performed. Instead of making *Z* the root of the middle tree, we make *Y* the root. This is shown in Figure 12.2. This simplifies the coding because the action for the *zig-zag* case becomes

[1] For simplicity we don't distinguish between a "node" and the item in the node.

[2] In the code, the smallest node in *R* does not have a `nullptr` left link because there is no need for it. This means that `printTree(r)` will include some items that logically are not in *R*.

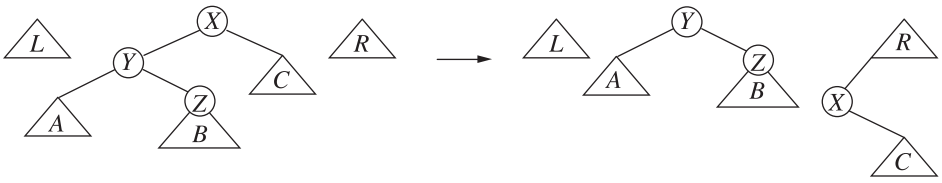


Figure 12.2 Simplified top-down *zig-zag*

identical to the *zig* case. This would seem advantageous because testing for a host of cases is time-consuming. The disadvantage is that by descending only one level, we have more iterations in the splaying procedure.

Once we have performed the final splaying step, Figure 12.3 shows how *L*, *R*, and the middle tree are arranged to form a single tree. Note carefully that the result is different from bottom-up splaying. The crucial fact is that the $O(\log N)$ amortized bound is preserved (Exercise 12.1).

An example of the top-down splaying algorithm is shown in Figure 12.4. We attempt to access 19 in the tree. The first step is a *zig-zag*. In accordance with (a symmetric version of) Figure 12.2, we bring the subtree rooted at 25 to the root of the middle tree and attach 12 and its left subtree to *L*.

Next we have a *zig-zig*: 15 is elevated to the root of the middle tree, and a rotation between 20 and 25 is performed, with the resulting subtree being attached to *R*. The search for 19 then results in a terminal *zig*. The middle tree's new root is 18, and 15 and its left subtree are attached as a right child of *L*'s largest node. The reassembly, in accordance with Figure 12.3, terminates the splay step.

We will use a header with left and right links to eventually contain the roots of the left and right trees. Since these trees are initially empty, a header is used to correspond to the min or max node of the right or left tree, respectively, in this initial state. This way the code can avoid checking for empty trees. The first time the left tree becomes nonempty, the right pointer will get initialized and will not change in the future; thus it will contain the root of the left tree at the end of the top-down search. Similarly, the left pointer will eventually contain the root of the right tree.

The `SplayTree` class interface, along with its constructor and destructor, are shown in Figure 12.5. The constructor allocates the `nullNode` sentinel. We use the sentinel `nullNode` to represent logically a `nullptr` pointer; the destructor `delete`s it after calling `makeEmpty`.

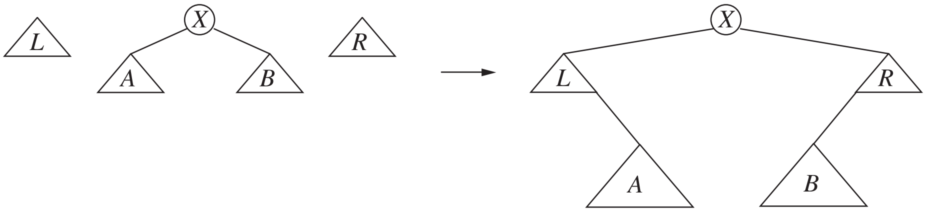


Figure 12.3 Final arrangement for top-down splaying

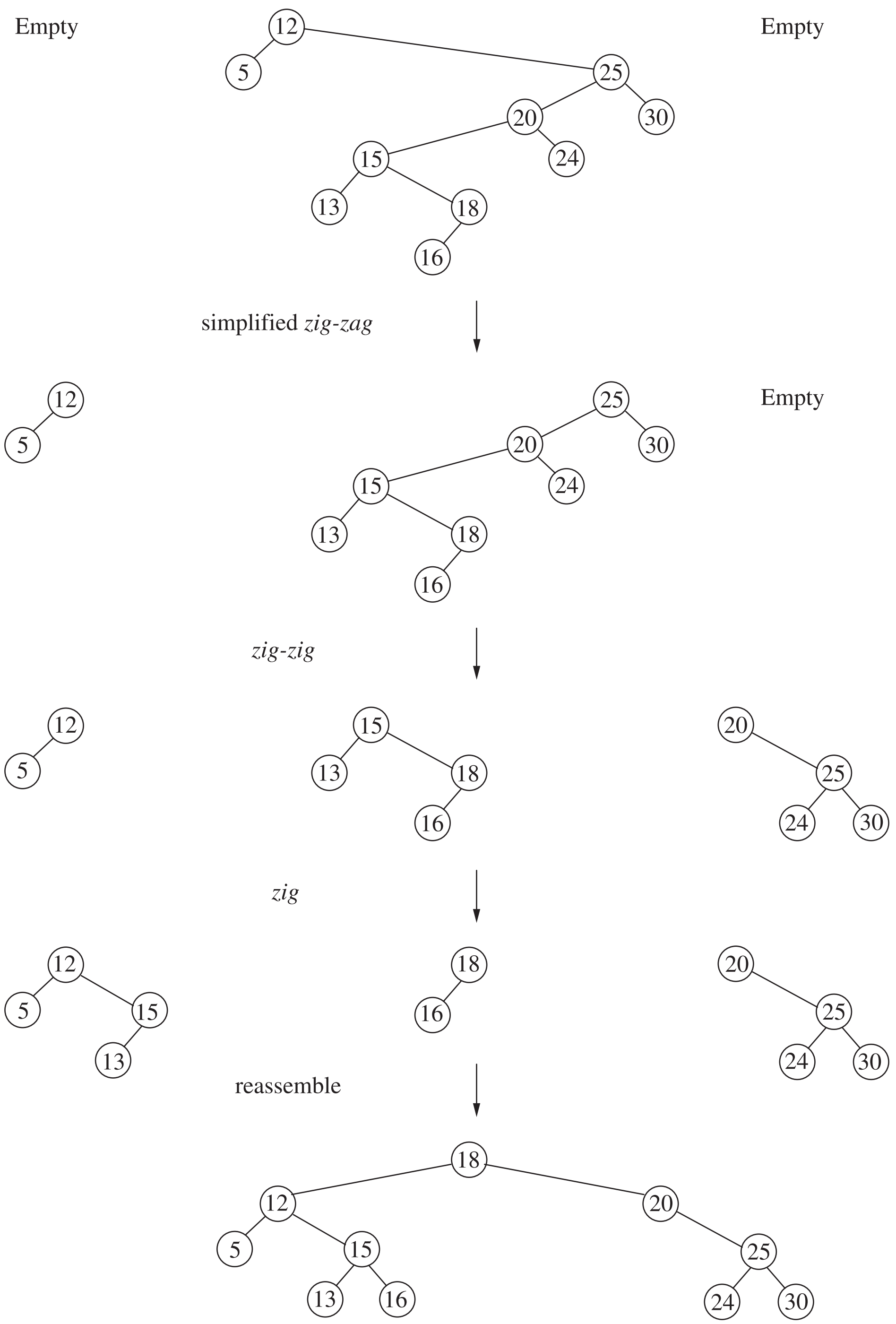


Figure 12.4 Steps in top-down splay (access 19 in top tree)

```
template <typename Comparable>
class SplayTree
{
  public:
    SplayTree( )
    {
        nullNode = new BinaryNode;
        nullNode->left = nullNode->right = nullNode;
        root = nullNode;
    }

    ~SplayTree( )
    {
        makeEmpty( );
        delete nullNode;
    }

    // Same methods as for BinarySearchTree (omitted)
    SplayTree( const SplayTree & rhs );
    SplayTree( SplayTree && rhs );
    SplayTree & operator=( const SplayTree & rhs );
    SplayTree & operator=( SplayTree && rhs )

  private:
    struct BinaryNode
      { /* Usual code for binary search tree nodes */ };

    BinaryNode *root;
    BinaryNode *nullNode;

    // Same methods as for BinarySearchTree (omitted)

        // Tree manipulations
    void rotateWithLeftChild( BinaryNode * & k2 );
    void rotateWithRightChild( BinaryNode * & k1 );
    void splay( const Comparable & x, BinaryNode * & t );
};
```

Figure 12.5 Splay trees: class interface, constructor, and destructor

We will repeatedly use this technique to simplify the code (and consequently make the code somewhat faster). Figure 12.6 gives the code for the splaying procedure. The `header` node allows us to be certain that we can attach *X* to the largest node in *R* without having to worry that *R* might be empty (and similarly for the symmetric case dealing with *L*).

```
/**
 * Internal method to perform a top-down splay.
 * The last accessed node becomes the new root.
 * This method may be overridden to use a different
 * splaying algorithm, however, the splay tree code
 * depends on the accessed item going to the root.
 * x is the target item to splay around.
 * t is the root of the subtree to splay.
 */
void splay( const Comparable & x, BinaryNode * & t )
{
    BinaryNode *leftTreeMax, *rightTreeMin;
    static BinaryNode header;

    header.left = header.right = nullNode;
    leftTreeMax = rightTreeMin = &header;

    nullNode->element = x;   // Guarantee a match

    for( ; ; )
        if( x < t->element )
        {
            if( x < t->left->element )
                rotateWithLeftChild( t );
            if( t->left == nullNode )
                break;
            // Link Right
            rightTreeMin->left = t;
            rightTreeMin = t;
            t = t->left;
        }
        else if( t->element < x )
        {
            if( t->right->element < x )
                rotateWithRightChild( t );
            if( t->right == nullNode )
                break;
            // Link Left
            leftTreeMax->right = t;
            leftTreeMax = t;
            t = t->right;
        }
        else
            break;

    leftTreeMax->right = t->left;
    rightTreeMin->left = t->right;
    t->left = header.right;
    t->right = header.left;
}
```

Figure 12.6 Top-down splaying method

As we mentioned above, before the reassembly at the end of the splay, `header.left` and `header.right` point to the roots of *R* and *L*, respectively (this is not a typo—follow the links). Except for this detail, the code is relatively straightforward.

Figure 12.7 shows the method to insert an item into a tree. A new node is allocated (if necessary), and if the tree is empty, a one-node tree is created. Otherwise, we splay `root` around the inserted value `x`. If the data in the new root is equal to `x`, we have a

```
void insert( const Comparable & x )
{
    static BinaryNode *newNode = nullptr;

    if( newNode == nullptr )
        newNode = new BinaryNode;
    newNode->element = x;

    if( root == nullNode )
    {
        newNode->left = newNode->right = nullNode;
        root = newNode;
    }
    else
    {
        splay( x, root );
        if( x < root->element )
        {
            newNode->left = root->left;
            newNode->right = root;
            root->left = nullNode;
            root = newNode;
        }
        else
        if( root->element < x )
        {
            newNode->right = root->right;
            newNode->left = root;
            root->right = nullNode;
            root = newNode;
        }
        else
            return;
    }
    newNode = nullptr;   // So next insert will call new
}
```

Figure 12.7 Top-down splay tree `insert`

```
void remove( const Comparable & x )
{
    if( !contains( x ) )
        return;   // Item not found; do nothing

      // If x is found, it will be splayed to the root by contains
    BinaryNode *newTree;

    if( root->left == nullNode )
        newTree = root->right;
    else
    {
        // Find the maximum in the left subtree
        // Splay it to the root; and then attach right child
        newTree = root->left;
        splay( x, newTree );
        newTree->right = root->right;
    }
    delete root;
    root = newTree;
}
```

Figure 12.8 Top-down deletion procedure and `makeEmpty`

duplicate. Instead of reinserting `x`, we preserve `newNode` for a future insertion and return immediately. If the new root contains a value larger than `x`, then the new root and its right subtree become a right subtree of `newNode`, and `root`'s left subtree becomes the left subtree of `newNode`. Similar logic applies if `root`'s new root contains a value smaller than `x`. In either case, `newNode` becomes the new root.

In Chapter 4, we showed that deletion in splay trees is easy, because a splay will place the target of the deletion at the root. We close by showing the deletion routine in Figure 12.8. It is indeed rare that a deletion procedure is shorter than the corresponding insertion procedure. Figure 12.8 also shows `makeEmpty`. A simple recursive postorder traversal to reclaim the tree nodes is unsafe because a splay tree may well be unbalanced, even while giving good performance. In that case, the recursion could run out of stack space. We use a simple alternative that is still $O(N)$ (though that is far from obvious). Similar considerations are required for `operator=`.

12.2 Red-Black Trees

A historically popular alternative to the AVL tree is the **red-black tree**. Operations on red-black trees take $O(\log N)$ time in the worst case, and, as we will see, a careful nonrecursive implementation (for insertion) can be done relatively effortlessly (compared with AVL trees).

A red-black tree is a binary search tree with the following coloring properties:

1. Every node is colored either red or black.
2. The root is black.
3. If a node is red, its children must be black.
4. Every path from a node to a null pointer must contain the same number of black nodes.

A consequence of the coloring rules is that the height of a red-black tree is at most $2\log(N + 1)$. Consequently, searching is guaranteed to be a logarithmic operation. Figure 12.9 shows a red-black tree. Red nodes are shown with double circles.

The difficulty, as usual, is inserting a new item into the tree. The new item, as usual, is placed as a leaf in the tree. If we color this item black, then we are certain to violate condition 4, because we will create a longer path of black nodes. Thus, the item must be colored red. If the parent is black, we are done. If the parent is already red, then we will violate condition 3 by having consecutive red nodes. In this case, we have to adjust the tree to ensure that condition 3 is enforced (without introducing a violation of condition 4). The basic operations that are used to do this are color changes and tree rotations.

12.2.1 Bottom-Up Insertion

As we have already mentioned, if the parent of the newly inserted item is black, we are done. Thus insertion of 25 into the tree in Figure 12.9 is trivial.

There are several cases (each with a mirror image symmetry) to consider if the parent is red. First, suppose that the sibling of the parent is black (we adopt the convention that null nodes are black). This would apply for an insertion of 3 or 8, but not for the insertion of 99. Let X be the newly added leaf, P be its parent, S be the sibling of the parent (if it exists), and G be the grandparent. Only X and P are red in this case; G is black, because otherwise there would be two consecutive red nodes *prior* to the insertion, in violation of red-black rules. Adopting the splay tree terminology, X, P, and G can form either a *zig-zig*

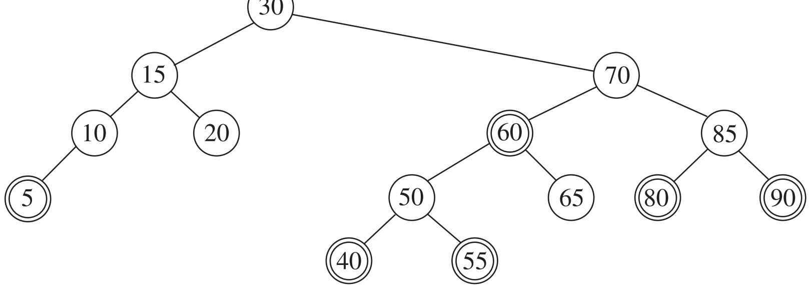


Figure 12.9 Example of a red-black tree (insertion sequence is: 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55)

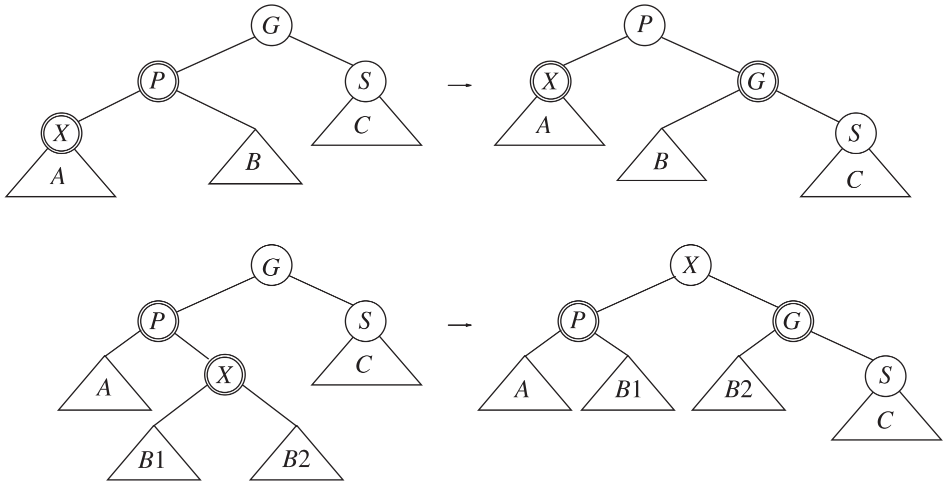


Figure 12.10 *Zig* rotation and *zig-zag* rotation work if *S* is black

chain or a *zig-zag* chain (in either of two directions). Figure 12.10 shows how we can rotate the tree for the case where *P* is a left child (note there is a symmetric case). Even though *X* is a leaf, we have drawn a more general case that allows *X* to be in the middle of the tree. We will use this more general rotation later.

The first case corresponds to a single rotation between *P* and *G*, and the second case corresponds to a double rotation, first between *X* and *P* and then between *X* and *G*. When we write the code, we have to keep track of the parent, the grandparent, and, for reattachment purposes, the great-grandparent.

In both cases, the subtree's new root is colored black, and so even if the original great-grandparent was red, we removed the possibility of two consecutive red nodes. Equally important, the number of black nodes on the paths into *A*, *B*, and *C* has remained unchanged as a result of the rotations.

So far so good. But what happens if *S* is red, as is the case when we attempt to insert 79 in the tree in Figure 12.9? In that case, initially there is one black node on the path from the subtree's root to *C*. After the rotation, there must still be only one black node. But in both cases, there are three nodes (the new root, *G*, and *S*) on the path to *C*. Since only one may be black, and since we cannot have consecutive red nodes, it follows that we'd have to color both *S* and the subtree's new root red, and *G* (and our fourth node) black. That's great, but what happens if the great-grandparent is also red? In that case, we could percolate this procedure up toward the root as is done for B-trees and binary heaps, until we no longer have two consecutive red nodes, or we reach the root (which will be recolored black).

12.2.2 Top-Down Red-Black Trees

Implementing the percolation would require maintaining the path using a stack or parent links. We saw that splay trees are more efficient if we use a top-down procedure, and it

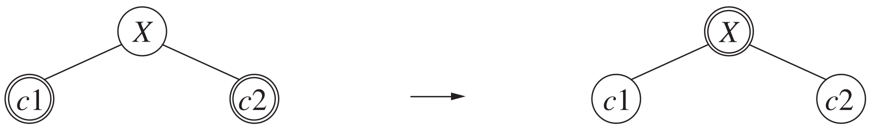


Figure 12.11 Color flip: only if *X*'s parent is red do we continue with a rotation

turns out that we can apply a top-down procedure to red-black trees that guarantees that *S* won't be red.

The procedure is conceptually easy. On the way down, when we see a node *X* that has two red children, we make *X* red and the two children black. (If *X* is the root, after the color flip it will be red but can be made black immediately to restore property 2.) Figure 12.11 shows this color flip. This will induce a red-black violation only if *X*'s parent *P* is also red. But in that case, we can apply the appropriate rotations in Figure 12.10. What if *X*'s parent's sibling is red? This possibility has been removed by our actions on the way down, and so *X*'s parent's sibling can't be red! Specifically, if on the way down the tree we see a node *Y* that has two red children, we know that *Y*'s grandchildren must be black, and that since *Y*'s children are made black too, even after the rotation that may occur, we won't see another red node for two levels. Thus when we see *X*, if *X*'s parent is red, it is not possible for *X*'s parent's sibling to be red also.

As an example, suppose we want to insert 45 into the tree in Figure 12.9. On the way down the tree, we see node 50, which has two red children. Thus, we perform a color flip, making 50 red, and 40 and 55 black. Now 50 and 60 are both red. We perform the single rotation between 60 and 70, making 60 the black root of 30's right subtree, and 70 and 50 both red. We then continue, performing an identical action if we see other nodes on the path that contain two red children. When we get to the leaf, we insert 45 as a red node, and since the parent is black, we are done. The resulting tree is shown in Figure 12.12.

As Figure 12.12 shows, the red-black tree that results is frequently very well balanced. Experiments suggest that the average red-black tree is about as deep as an average AVL tree and that, consequently, the searching times are typically near optimal. The advantage

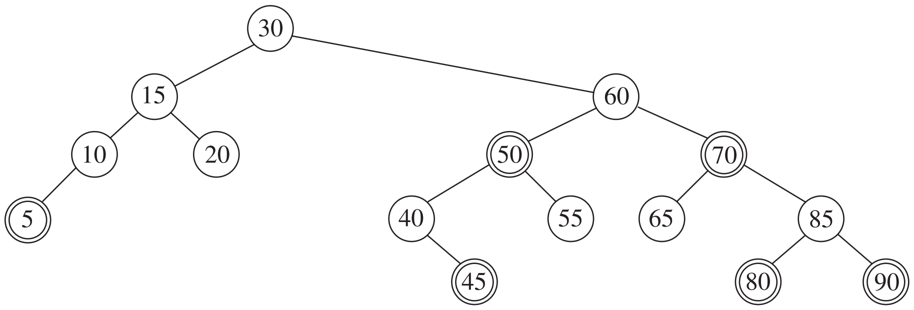


Figure 12.12 Insertion of 45 into Figure 12.9

of red-black trees is the relatively low overhead required to perform insertion, and the fact that, in practice, rotations occur relatively infrequently.

An actual implementation is complicated not only by the host of possible rotations but also by the possibility that some subtrees (such as 10's right subtree) might be empty, and the special case of dealing with the root (which among other things, has no parent). Thus, we use two sentinel nodes: one for the root, and `nullNode`, which indicates a `nullptr` pointer as it did for splay trees. The root sentinel will store the key $-\infty$ and a right link to the real root. Because of this, the searching and printing procedures need to be adjusted. The recursive routines are trickiest. Figure 12.13 shows how the inorder traversal is rewritten. The `printTree` routines are straightforward. The test `t!=t->left` could be written as `t!=nullNode`. However, there is a trap in a similar routine that performs the deep copy. This is also shown in Figure 12.13. The copy constructor calls `clone` after other initialization is complete. But in `clone`, the test `t==nullNode` does not work, because `nullNode` is the target's `nullNode`, not the source's (that is, not `rhs`'s). Thus we use a trickier test.

Figure 12.14 shows the `RedBlackTree` skeleton, along with the constructor.

Next, Figure 12.15 (page 574) shows the routine to perform a single rotation. Because the resultant tree must be attached to a parent, `rotate` takes the parent node as a parameter. Rather than keeping track of the type of rotation as we descend the tree, we pass `item` as a parameter. Since we expect very few rotations during the insertion procedure, it turns out that it is not only simpler, but actually faster, to do it this way. `rotate` simply returns the result of performing an appropriate single rotation.

Finally, we provide the insertion procedure in Figure 12.16 (on page 574). The routine `handleReorient` is called when we encounter a node with two red children, and also when we insert a leaf. The trickiest part is the observation that a double rotation is really two single rotations, and is done only when branching to X (represented in the `insert` method by `current`) takes opposite directions. As we mentioned in the earlier discussion, `insert` must keep track of the parent, grandparent, and great-grandparent as the tree is descended. Since these are shared with `handleReorient`, we make these class members. Note that after a rotation, the values stored in the grandparent and great-grandparent are no longer correct. However, we are assured that they will be restored by the time they are next needed.

12.2.3 Top-Down Deletion

Deletion in red-black trees can also be performed top-down. Everything boils down to being able to delete a leaf. This is because to delete a node that has two children, we replace it with the smallest node in the right subtree; that node, which must have at most one child, is then deleted. Nodes with only a right child can be deleted in the same manner, while nodes with only a left child can be deleted by replacement with the largest node in the left subtree, and subsequent deletion of that node. Note that for red-black trees, we don't want to use the strategy of bypassing for the case of a node with one child because that may connect two red nodes in the middle of the tree, making enforcement of the red-black condition difficult.

```
void printTree( ) const
{
    if( header->right == nullNode )
        cout << "Empty tree" << endl;
    else
        printTree( header->right );
}

void printTree( RedBlackNode *t ) const
{
    if( t != t->left )
    {
        printTree( t->left );
        cout << t->element << endl;
        printTree( t->right );
    }
}

RedBlackTree( const RedBlackTree & rhs )
{
    nullNode       = new RedBlackNode;
    nullNode->left = nullNode->right = nullNode;

    header         = new RedBlackNode{ rhs.header->element };
    header->left   = nullNode;
    header->right  = clone( rhs.header->right );
}

RedBlackNode * clone( RedBlackNode * t ) const
{
    if( t == t->left )  // Cannot test against nullNode!!!
        return nullNode;
    else
        return new RedBlackNode{ t->element, clone( t->left ),
                                 clone( t->right ), t->color };
}
```

Figure 12.13 Tree traversals with two sentinels: `printTree` and copy constructor

Deletion of a red leaf is, of course, trivial. If a leaf is black, however, the deletion is more complicated because removal of a black node will violate condition 4. The solution is to ensure during the top-down pass that the leaf is red.

Throughout this discussion, let *X* be the current node, *T* be its sibling, and *P* be their parent. We begin by coloring the root sentinel red. As we traverse down the tree, we attempt to ensure that *X* is red. When we arrive at a new node, we are certain

```
template <typename Comparable>
class RedBlackTree
{
  public:
    explicit RedBlackTree( const Comparable & negInf );
    RedBlackTree( const RedBlackTree & rhs );
    RedBlackTree( RedBlackTree && rhs );
    ~RedBlackTree( );

    const Comparable & findMin( ) const;
    const Comparable & findMax( ) const;
    bool contains( const Comparable & x ) const;
    bool isEmpty( ) const;
    void printTree( ) const;

    void makeEmpty( );
    void insert( const Comparable & x );
    void remove( const Comparable & x );

    enum { RED, BLACK };

    RedBlackTree & operator=( const RedBlackTree & rhs );
    RedBlackTree & operator=( RedBlackTree && rhs );

  private:
    struct RedBlackNode
    {
        Comparable    element;
        RedBlackNode *left;
        RedBlackNode *right;
        int           color;

        RedBlackNode( const Comparable & theElement = Comparable{ },
                            RedBlackNode *lt = nullptr, RedBlackNode *rt = nullptr,
                            int c = BLACK )
          : element{ theElement }, left{ lt }, right{ rt }, color{ c } { }

        RedBlackNode( Comparable && theElement, RedBlackNode *lt = nullptr,
                      RedBlackNode *rt = nullptr, int c = BLACK )
          : element{ std::move( theElement ) }, left{ lt }, right{ rt }, color{ c } { }
    };

    RedBlackNode *header;   // The tree header (contains negInf)
    RedBlackNode *nullNode;

        // Used in insert routine and its helpers (logically static)
    RedBlackNode *current;
```

Figure 12.14 Class interface and constructor

```
    RedBlackNode *parent;
    RedBlackNode *grand;
    RedBlackNode *great;

        // Usual recursive stuff
    void reclaimMemory( RedBlackNode *t );
    void printTree( RedBlackNode *t ) const;

    RedBlackNode * clone( RedBlackNode * t ) const;

        // Red-black tree manipulations
    void handleReorient( const Comparable & item );
    RedBlackNode * rotate( const Comparable & item, RedBlackNode *theParent );
    void rotateWithLeftChild( RedBlackNode * & k2 );
    void rotateWithRightChild( RedBlackNode * & k1 );
};

    /**
     * Construct the tree.
     * negInf is a value less than or equal to all others.
     */
    explicit RedBlackTree( const Comparable & negInf )
    {
        nullNode = new RedBlackNode;
        nullNode->left = nullNode->right = nullNode;

        header = new RedBlackNode{ negInf };
        header->left = header->right = nullNode;
    }
```

Figure 12.14 *(continued)*

that P is red (inductively, by the invariant we are trying to maintain), and that X and T are black (because we can't have two consecutive red nodes). There are two main cases.

First, suppose X has two black children. Then there are three subcases, which are shown in Figure 12.17. If T also has two black children, we can flip the colors of X, T, and P to maintain the invariant. Otherwise, one of T's children is red. Depending on which one it is,[3] we can apply the rotation shown in the second and third cases of Figure 12.17. Note carefully that this case will apply for the leaf, because `nullNode` is considered to be black.

[3] If both children are red, we can apply either rotation. As usual, there are symmetric rotations for the case when X is a right child that are not shown.

```
/**
 * Internal routine that performs a single or double rotation.
 * Because the result is attached to the parent, there are four cases.
 * Called by handleReorient.
 * item is the item in handleReorient.
 * theParent is the parent of the root of the rotated subtree.
 * Return the root of the rotated subtree.
 */
RedBlackNode * rotate( const Comparable & item, RedBlackNode *theParent )
{
    if( item < theParent->element )
    {
        item < theParent->left->element ?
            rotateWithLeftChild( theParent->left )  :  // LL
            rotateWithRightChild( theParent->left ) ;  // LR
        return theParent->left;
    }
    else
    {
        item < theParent->right->element ?
            rotateWithLeftChild( theParent->right ) :  // RL
            rotateWithRightChild( theParent->right );  // RR
        return theParent->right;
    }
}
```

Figure 12.15 `rotate` method

```
/**
 * Internal routine that is called during an insertion if a node has two red
 * children. Performs flip and rotations. item is the item being inserted.
 */
void handleReorient( const Comparable & item )
{
        // Do the color flip
    current->color = RED;
    current->left->color = BLACK;
    current->right->color = BLACK;

    if( parent->color == RED )   // Have to rotate
```

Figure 12.16 Insertion procedure

```
    {
        grand->color = RED;
        if( item < grand->element != item < parent->element )
            parent = rotate( item, grand );  // Start dbl rotate
        current = rotate( item, great );
        current->color = BLACK;
    }
    header->right->color = BLACK; // Make root black
}

void insert( const Comparable & x )
{
    current = parent = grand = header;
    nullNode->element = x;

    while( current->element != x )
    {
        great = grand; grand = parent; parent = current;
        current = x < current->element ? current->left : current->right;

            // Check if two red children; fix if so
        if( current->left->color == RED && current->right->color == RED )
            handleReorient( x );
    }

        // Insertion fails if already present
    if( current != nullNode )
        return;
    current = new RedBlackNode{ x, nullNode, nullNode };

        // Attach to parent
    if( x < parent->element )
        parent->left = current;
    else
        parent->right = current;
    handleReorient( x );
}
```

Figure 12.16 *(continued)*

Otherwise one of X's children is red. In this case, we fall through to the next level, obtaining new X, T, and P. If we're lucky, X will land on the red child, and we can continue onward. If not, we know that T will be red, and X and P will be black. We can rotate T and P, making X's new parent red; X and its grandparent will, of course, be black. At this point, we can go back to the first main case.

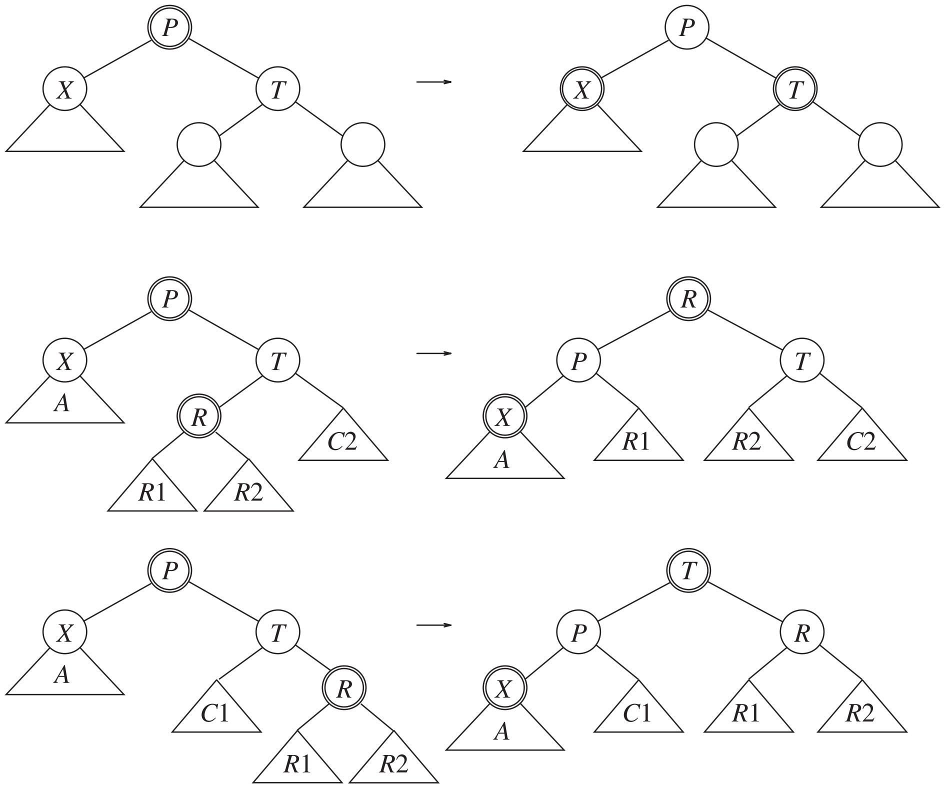


Figure 12.17 Three cases when X is a left child and has two black children

12.3 Treaps

Our last type of binary search tree, known as the **treap**, is probably the simplest of all. Like the skip list, it uses random numbers and gives $O(\log N)$ expected time behavior for any input. Searching time is identical to an unbalanced binary search tree (and thus slower than balanced search trees), while insertion time is only slightly slower than a recursive unbalanced binary search tree implementation. Although deletion is much slower, it is still $O(\log N)$ expected time.

The treap is so simple that we can describe it without a picture. Each node in the tree stores an item, a left and right pointer, and a priority that is randomly assigned when the node is created. A treap is a binary search tree with the property that the node priorities satisfy heap order: Any node's priority must be at least as large as its parent's.

A collection of distinct items each of which has a distinct priority can only be represented by one treap. This is easily deduced by induction, since the node with the lowest priority must be the root. Consequently, the tree is formed on the basis of the $N!$ possible arrangements of priority instead of the $N!$ item orderings. The node declarations are straightforward, requiring only the addition of the `priority` data member. The sentinel `nullNode` will have priority of ∞, as shown in Figure 12.18.

```
template <typename Comparable>
class Treap
{
  public:
    Treap( )
    {
        nullNode = new TreapNode;
        nullNode->left = nullNode->right = nullNode;
        nullNode->priority = INT_MAX;
        root = nullNode;
    }

    Treap( const Treap & rhs );
    Treap( Treap && rhs );
    ~Treap( );
    Treap & operator=( const Treap & rhs );
    Treap & operator=( Treap && rhs );

      // Additional public member functions (not shown)

  private:
    struct TreapNode
    {
        Comparable element;
        TreapNode *left;
        TreapNode *right;
        int        priority;

        TreapNode( ) : left{ nullptr }, right{ nullptr }, priority{ INT_MAX }
          { }

        TreapNode( const Comparable & e, TreapNode *lt, TreapNode *rt, int pr )
          : element{ e }, left{ lt }, right{ rt }, priority{ pr }
          { }

        TreapNode( Comparable && e, TreapNode *lt, TreapNode *rt, int pr )
          : element{ std::move( e ) }, left{ lt }, right{ rt }, priority{ pr }
          { }
    };

    TreapNode *root;
    TreapNode *nullNode;
    UniformRandom randomNums;

      // Additional private member functions (not shown)
};
```

Figure 12.18 Treap class interface and constructor

Insertion into the treap is simple: After an item is added as a leaf, we rotate it up the treap until its priority satisfies heap order. It can be shown that the expected number of rotations is less than 2. After the item to be deleted has been found, it can be deleted by increasing its priority to ∞ and rotating it down through the path of low-priority children. Once it is a leaf, it can be removed. The routines in Figure 12.19 and Figure 12.20 implement these strategies using recursion. A nonrecursive implementation is left for the reader (Exercise 12.14). For deletion, note that when the node is logically a leaf, it still has `nullNode` as both its left and right children. Consequently, it is rotated with the right child. After the rotation, `t` is `nullNode`, and the left child, which now stores the item to be deleted, can be freed. Note also that our implementation assumes that there are no duplicates; if this is not true, then the `remove` could fail (why?).

The treap implementation never has to worry about adjusting the `priority` data member. One of the difficulties of the balanced tree approaches is that it is difficult to track down errors that result from failing to update balance information in the course of an operation. In terms of total lines for a reasonable insertion and deletion package, the treap, especially a nonrecursive implementation, seems like the hands-down winner.

```
/**
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the tree.
 * Set the new root of the subtree.
 * (randomNums is a UniformRandom object that is a data member of Treap.)
 */
void insert( const Comparable & x, TreapNode* & t )
{
    if( t == nullNode )
        t = new TreapNode{ x, nullNode, nullNode, randomNums.nextInt( ) };
    else if( x < t->element )
    {
        insert( x, t->left );
        if( t->left->priority < t->priority )
            rotateWithLeftChild( t );
    }
    else if( t->element < x )
    {
        insert( x, t->right );
        if( t->right->priority < t->priority )
            rotateWithRightChild( t );
    }
    // else duplicate; do nothing
}
```

Figure 12.19 Treaps: insertion routine

```
/**
 * Internal method to remove from a subtree.
 * x is the item to remove.
 * t is the node that roots the tree.
 * Set the new root of the subtree.
 */
void remove( const Comparable & x, TreapNode * & t )
{
    if( t != nullNode )
    {
        if( x < t->element )
            remove( x, t->left );
        else if( t->element < x )
            remove( x, t->right );
        else
        {
                // Match found
            if( t->left->priority < t->right->priority )
                rotateWithLeftChild( t );
            else
                rotateWithRightChild( t );

            if( t != nullNode )     // Continue on down
                remove( x, t );
            else
            {
                delete t->left;
                t->left = nullNode;  // At a leaf
            }
        }
    }
}
```

Figure 12.20 Treaps: deletion procedure

12.4 Suffix Arrays and Suffix Trees

One of the most fundamental problems in data processing is to find the location of a pattern, P, in a text, T. For instance, we may be interested in answering questions such as

- Is there a substring of T matching P?
- How many times does P appear in T?
- Where are all occurrences of P in T?

Assuming that the size of P is less than T (and usually it is significantly less), then we would reasonably expect that the time to solve this problem for a given P and T would be at least linear in the length of T, and in fact there are several $O(\,|\,T\,|\,)$ algorithms.

However, we are interested in a more common problem, in which T is fixed, and queries with different P occur frequently. For instance, T could be a huge archive of email messages, and we are interested in repeatedly searching the email messages for different patterns. In this case, we are willing to preprocess T into a nice form that would make each individual search much more efficient, taking time significantly less than linear in the size of T—either logarithmic in the size of T, or even better, independent of T and dependent only on the length of P.

One such data structure is the *suffix array* and *suffix tree* (that sounds like two data structures, but as we will see, they are basically equivalent, and trade time for space).

12.4.1 Suffix Arrays

A suffix array for a text, T, is simply an array of all suffixes of T arranged in sorted order. For instance, suppose our text string is `banana`. Then the suffix array for `banana` is shown in Figure 12.21.

A suffix array that stores the suffixes explicitly would seem to require quadratic space, since it stores one string of each length 1 to N (where N is the length of T). In C++, this is not exactly true, since in C++ we can use the primitive null-terminated array of character representation of strings, and in that case, a suffix is specified by a `char *` that points at the first character of the substring. Thus, the same array of characters is shared, and the additional memory requirement is only the `char *` pointer for the new substring. Nonetheless, using a `char *` is highly C or C++ dependent; thus it is common for a practical implementation to store only the starting indices of the suffixes in the suffix array, which is much more language independent. Figure 12.22 shows the indices that would be stored.

The suffix array by itself is extremely powerful. For instance, if a pattern, P, occurs in the text, then it must be a prefix of some suffix. A binary search of the suffix array would be enough to determine if the pattern P is in the text: The binary search either lands on P, or P would be between two values, one smaller than P and one larger than P. If P is a prefix of some substring, it is a prefix of the larger value found at the end of the binary search. Immediately, this reduces the query time to $O(\,|\,P\,|\log|\,T\,|\,)$, where the $\log|\,T\,|$ is the binary search, and the $|\,P\,|$ is the cost of the comparison at each step.

0	`a`
1	`ana`
2	`anana`
3	`banana`
4	`na`
5	`nana`

Figure 12.21 Suffixes for "banana"

	Index	*Substring Being Represented*
0	5	`a`
1	3	`ana`
2	1	`anana`
3	0	`banana`
4	4	`na`
5	2	`nana`

Figure 12.22 Suffix array that stores only indices (full substrings shown for reference)

We can also use the suffix array to find the number of occurrences of P: They will be stored sequentially in the suffix array, thus two binary searches suffice to find a range of suffixes that will be guaranteed to begin with P. One way to speed this search is to compute the *longest common prefix* (LCP) for each consecutive pair of substrings; if this computation is done as the suffix array is built, then each query to find the number of occurrences of P can be sped up to $O(\,|\,P\,|+\log|\,T\,|\,)$ although this is not obvious. Figure 12.23 shows the LCP computed for each substring, relative to the preceding substring.

The longest common prefix also provides information about the longest pattern that occurs twice in the text: Look for the largest LCP value, and take that many characters of the corresponding substring. In Figure 12.23, this is 3, and the longest repeated pattern is `ana`.

Figure 12.24 shows simple code to compute the suffix array and longest common prefix information for any string. Line 26 obtains a primitive (`char *`) string from `str`, and lines 28 to 31 obtain the suffixes by computing and storing these pointers using pointer arithmetic (line 31). At lines 33 and 34, the suffixes are sorted; the code at line 34 represents the **C++11 lambda feature**, in which the "less than" function that is needed for two `char *` types is provided as the third parameter to `sort`, without the need to write a named function. Lines 36 and 37 compute the suffixes' starting indices using pointer arithmetic, and lines 39 to 41 compute the longest common prefixes for adjacent entries by calling the `computeLCP` routine written at lines 4 to 12.

	Index	*LCP*	*Substring Being Represented*
0	5	-	`a`
1	3	1	`ana`
2	1	3	`anana`
3	0	0	`banana`
4	4	0	`na`
5	2	2	`nana`

Figure 12.23 Suffix array for "banana"; includes longest common prefix (LCP)

```cpp
/*
 * Returns the LCP for any two strings
 */
int computeLCP( const string & s1, const string & s2 )
{
    int i = 0;

    while( i < s1.length( ) && i < s2.length( ) && s1[ i ] == s2[ i ] )
        ++i;

    return i;
}

/*
 * Fill in the suffix array and LCP information for String str
 * str is the input String
 * SA is an existing array to place the suffix array
 * LCP is an existing array to place the LCP information
 */
void createSuffixArraySlow( const string & str, vector<int> & SA, vector<int> & LCP )
{
    if( SA.size( ) != str.length( ) || LCP.size( ) != str.length( ) )
        throw invalid_argument{ "Mismatched vector sizes" };

    size_t N = str.length( );
    const char *cstr = str.c_str( );

    vector<const char *> suffixes( N );

    for( int i = 0; i < N; ++i )
        suffixes[ i ] = cstr + i;

    std::sort( begin( suffixes ), end( suffixes ),
      [] ( const char *s1, const char *s2 ) { return strcmp( s1, s2 ) < 0; } );

    for( int i = 0; i < N; ++i )
        SA[ i ] = suffixes[ i ] - cstr;

    LCP[ 0 ] = 0;
    for( int i = 1; i < N; ++i )
        LCP[ i ] = computeLCP( suffixes[ i - 1 ], suffixes[ i ] );
}
```

Figure 12.24 Simple algorithm to create suffix array and LCP array

The running time of the suffix array computation is dominated by the sorting step, which uses $O(N \log N)$ comparisons. In many circumstances this can be reasonably acceptable performance. For instance, a suffix array for a 3,000,000-character English-language novel can be built in just a few seconds. However, the $O(N \log N)$ cost, based on the number of comparisons, hides the fact that a `String` comparison between *s1* and *s2* takes time that depends on *LCP(s1, s2)*, So while it is true that almost all these comparisons end quickly when run on the suffixes found in natural language processing, the comparisons will be expensive in applications where there are many long common substrings. One such example occurs in pattern searching of DNA, whose alphabet consists of four characters (A, C, G, T) and whose strings can be huge. For instance, the DNA string for human chromosome 22 has roughly 35 million characters, with a maximum LCP of approximately 200,000 and an average LCP of nearly 2,000. And even the HTML/Java distribution for JDK 1.3 (much smaller than the current distribution) is nearly 70 million characters, with a maximum LCP of roughly 37,000 and an average LCP of roughly 14,000. In the degenerate case of a `String` that contains only one character, repeated N times, it is easy to see that each comparison takes $O(N)$ time, and the total cost is $O(N^2 \log N)$.

In Section 12.4.3, we will show a linear-time algorithm to construct the suffix array.

12.4.2 Suffix Trees

Suffix arrays are easily searchable by binary search, but the binary search itself automatically implies $\log T$ cost. What we would like to do is find a matching suffix even more efficiently. One idea is to store the suffixes in a *trie*. A binary trie was seen in our discussion of Huffman codes in Section 10.1.2.

The basic idea of the trie is to store the suffixes in a tree. At the root, instead of having two branches, we would have one branch for each possible first character. Then at the next level, we would have one branch for the next character, and so on. At each level we are doing multiway branching, much like radix sort, and thus we can find a match in time that would depend only on the length of the match.

In Figure 12.25, we see on the left a basic trie to store the suffixes of the string *deed*. These suffixes are *d, deed, ed*, and *eed*. In this trie, internal branching nodes are drawn in circles, and the suffixes that are reached are drawn in rectangles. Each branch is labeled with the character that is chosen, but the branch prior to a completed suffix has no label.

This representation could waste significant space if there are many nodes that have only one child. Thus in Figure 12.25, we see an equivalent representation on the right, known as a *compressed trie*. Here, single-branch nodes are collapsed into a single node. Notice that although the branches now have multicharacter labels, all the labels for the branches of any given node must have unique first characters. Thus, it is still just as easy as before to choose which branch to take. Thus we can see that a search for a pattern, P, depends only on the length of the pattern P, as desired. (We assume that the letters of the alphabet are represented by numbers 1, 2, ... Then each node stores an array representing each possible branch and we can locate the appropriate branch in constant time. The empty edge label can be represented by 0.)

If the original string has length N, the total number of branches is less than $2N$. However, this by itself does not mean that the compressed trie uses linear space: The labels on the edges take up space. The total length of all the labels on the compressed

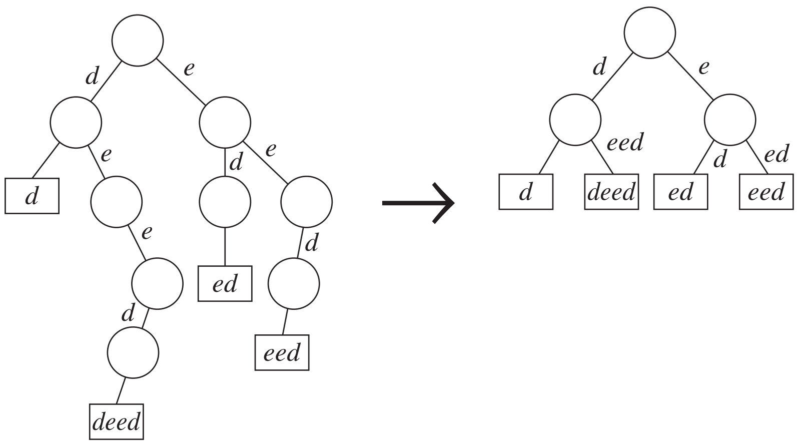


Figure 12.25 Left: trie representing the suffixes for *deed*: *{d, deed, ed, eed}*; right: compressed trie that collapses single-node branches

trie in Figure 12.25 is exactly one less than the number of internal branching nodes in the original trie in Figure 12.25. And of course writing all the suffixes in the leaves could take quadratic space. So if the original used quadratic space, so does the compressed trie. Fortunately, we can get by with linear space as follows:

1. In the leaves, we use the index where the suffix begins (as in the suffix array).
2. In the internal nodes, we store the number of common characters matched from the root until the internal node; this number represents the *letter depth*.

Figure 12.26 shows how the compressed trie is stored for the suffixes of *banana*. The leaves are simply the indices of the starting points for each suffix. The internal node with a letter depth of 1 is representing the common string "a" in all nodes that are below it. The internal

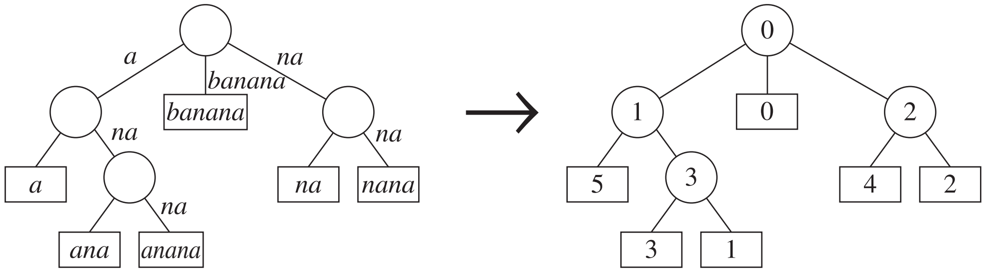


Figure 12.26 Compressed trie representing the suffixes for *banana*: {*a, ana, anana, banana, na, nana*}. Left: the explicit representation; right: the implicit representation that stores only one integer (plus branches) per node.

node with a letter depth of 3 is representing the common string "ana" in all nodes that are below it. And the internal node with a letter depth of 2 is representing the common string "na" in all nodes that are below it. In fact, this analysis makes clear that a suffix tree is equivalent to a suffix array plus an LCP array.

If we have a suffix tree, we can compute the suffix array and the LCP array by performing an inorder traversal of the tree (compare Figure 12.23 with the suffix tree in Figure 12.26). At that time we can compute the LCP as follows: If the suffix node value PLUS the letter depth of the parent is equal to N, then use the letter depth of the grandparent as the LCP; otherwise use the parent's letter depth as the LCP. In Figure 12.26, if we proceed inorder, we obtain for our suffixes and LCP values

Suffix $= 5$, with LCP $= 0$ (the grandparent) because $5 + 1$ equals 6
Suffix $= 3$, with LCP $= 1$ (the grandparent) because $3 + 3$ equals 6
Suffix $= 1$, with LCP $= 3$ (the parent) because $1 + 3$ does not equal 6
Suffix $= 0$, with LCP $= 0$ (the parent) because $0 + 0$ does not equal 6
Suffix $= 4$, with LCP $= 0$ (the grandparent) because $4 + 2$ equals 6
Suffix $= 2$, with LCP $= 2$ (the parent) because $2 + 2$ does not equal 6

This transformation can clearly be done in linear time.

The suffix array and LCP array also uniquely define the suffix tree. First, create a root with letter depth 0. Then search the LCP array (ignoring position 0, for which LCP is not really defined) for all occurrences of the minimum (which at this phase will be the zeros). Once these minimums are found, they will partition the array (view the LCP as residing *between* adjacent elements). For instance, in our example, there are two zeros in the LCP array, which partitions the suffix array into three portions: one portion containing the suffixes {5, 3, 1}, another portion containing the suffix {0}, and the third portion containing the suffixes {4, 2}. The internal nodes for these portions can be built recursively, and then the suffix leaves can be attached with an inorder traversal. Although it is not obvious, with care the suffix tree can be generated in linear time from the suffix array and LCP array.

The suffix tree solves many problems efficiently, especially if we augment each internal node to also maintain the number of suffixes stored below it. A small sampling of suffix tree applications includes the following:

1. *Find the longest repeated substring in T*: Traverse the tree, finding the internal node with the largest number letter depth; this represents the maximum LCP. The running time is $O(\mid T \mid)$. This generalizes to the longest substring repeated at least k times.
2. *Find the longest common substring in two strings T_1 and T_2*: Form a string $T_1\#T_2$ where # is a character that is not in either string. Then build a suffix tree for the resulting string and find the deepest internal node that has at least one suffix that starts prior to the #, and one that starts after the #. This can be done in time proportional to the total size of the strings and generalizes to an $O(kN)$ algorithm for k strings of total length N.
3. *Find the number of occurrences of the pattern P*: Assuming that the suffix tree is augmented so that each node keeps track of the number of suffixes below it, simply follow the path down the tree; the first internal node that is a prefix of P provides the answer;

if there is no such node, the answer is either zero or one and is found by checking the suffix at which the search terminates. This takes time, proportional to the length of the pattern P, and is independent of the size of $|T|$.

4. *Find the most common substring of a specified length* $L > 1$: Return the internal node with largest size amongst those with letter depth at least L. This takes time $O(\ |\ T\ |\ )$.

12.4.3 Linear-Time Construction of Suffix Arrays and Suffix Trees

In Section 12.4.1 we showed the simplest algorithm to construct a suffix array and an LCP array, but this algorithm has $O(\ N^2 \log N\ )$ worst-case running time for an N-character string and can occur if the string has suffixes with long common prefixes. In this section we describe an $O(\ N\ )$ worst-case time algorithm to compute the suffix array. This algorithm can also be enhanced to compute the LCP array in linear time, but there is also a very simple linear-time algorithm to compute the LCP array from the suffix array (see Exercise 12.9 and complete code in Fig. 12.49). Either way, we can thus also build a suffix tree in linear time.

This algorithm makes use of divide and conquer. The basic idea is as follows:

1. Choose a sample, A, of suffixes.
2. Sort the sample A by recursion.
3. Sort the remaining suffixes, B, by using the now-sorted sample of suffixes A.
4. Merge A and B.

To get an intuition of how step 3 might work, suppose the sample A of suffixes are all suffixes that start at an odd index. Then the remaining suffixes, B, are those suffixes that start at an even index. So suppose we have computed the sorted set of suffixes A. To compute the sorted set of suffixes B, we would in effect need to sort all the suffixes that start at even indices. But these suffixes each consist of a single first character in an even position, followed by a string that starts with the second character, which must be in an odd position. Thus the string that starts in the second character is exactly a string that is in A. So to sort all the suffixes B, we can do something similar to a radix sort: First sort the strings in B starting from the second character. This should take linear time, since the sorted order of A is already known. Then stably sort on the first character of the strings in B. Thus B could be sorted in linear time, after A is sorted recursively. If A and B could then be merged in linear time, we would have a linear-time algorithm. The algorithm we present uses a different sampling step, that admits a simple linear-time merging step.

As we describe the algorithm, we will also show how it computes the suffix array for the string ABRACADABRA. We adopt the following conventions:

$S[i]$	represents the ith character of string S
$S[i \rightarrow]$	represents the suffix of S starting at index i
$<>$	represents an array

Step 1: Sort the characters in the string, assigning them numbers sequentially starting at 1. Then use those numbers for the remainder of the algorithm. Note that the numbers that are assigned depend on the text. So, if the text contains DNA characters A, C, G, and T only, then there will be only four numbers. Then pad the array with three 0s to avoid boundary cases. If we assume that the alphabet is a fixed size, then the sort takes some constant amount of time.

Example:
In our example, the mapping is A = 1, B = 2, C = 3, D = 4, and R = 5; the transformation can be visualized in Figure 12.27.

Step 2: Divide the text into three groups:

$$\begin{aligned}
S_0 &= < S[3i]S[3i+1]S[3i+2] && \text{for } i = 0, 1, 2, \ldots > \\
S_1 &= < S[3i+1]S[3i+2]S[3i+3] && \text{for } i = 0, 1, 2, \ldots > \\
S_2 &= < S[3i+2]S[3i+3]S[3i+4] && \text{for } i = 0, 1, 2, \ldots >
\end{aligned}$$

The idea is that each of S_0, S_1, S_2 consists of roughly $N/3$ symbols, but the symbols are no longer the original alphabet, but instead each new symbol is some group of three symbols from the original alphabet. We will call these *tri-characters*. Most importantly, the suffixes of S_0, S_1, and S_2 combine to form the suffixes of S. Thus one idea would be to recursively compute the suffixes of S_0, S_1, and S_2 (which by definition implicitly represent sorted strings) and then merge the results in linear time. However, since this would be three recursive calls on problems 1/3 the original size, that would result in an $O(N \log N)$ algorithm. So the idea is going to be to avoid one of the three recursive calls, by computing two of the suffix groups recursively and using that information to compute the third suffix group.

Example:
In our example, if we look at the original character set and use \$ to represent the padded character, we get

$$\begin{aligned}
S_0 &= [\text{ABR}], [\text{ACA}], [\text{DAB}], [\text{RA\$}] \\
S_1 &= [\text{BRA}], [\text{CAD}], [\text{ABR}], [\text{A\$\$}] \\
S_2 &= [\text{RAC}], [\text{ADA}], [\text{BRA}]
\end{aligned}$$

We can see that in S_0, S_1, and S_2, each tri-character is now a trio of characters from the original alphabet. Using that alphabet, S_0 and S_1 are arrays of length four and S_2 is an

Input String, *S*	A	B	R	A	C	A	D	A	B	R	A			
New Problem	1	2	5	1	3	1	4	1	2	5	1	0	0	0
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Figure 12.27 Mapping of character in string to an array of integers

array of length three. S_0, S_1, and S_2 thus have four, four, and three suffixes, respectively. S_0's suffixes are [ABR][ACA][DAB][RA$], [ACA][DAB][RA$], [DAB][RA$], [RA$], which clearly correspond to the suffixes ABRACADABRA, ACADABRA, DABRA, and RA in the original string S. In the original string S, these suffixes are located at indices 0, 3, 6, and 9, respectively, so looking at all three of S_0, S_1, and S_2, we can see that each S_i represents the suffixes that are located at indices i mod 3 in S.

Step 3: Concatenate S_1 and S_2 and recursively compute the suffix array. In order to compute this suffix array, we will need to sort the new alphabet of tri-characters. This can be done in linear time by three passes of radix sort, since the old characters were already sorted in step 1. If in fact all the tri-characters in the new alphabet are unique, then we do not even need to bother with a recursive call. Making three passes of radix sort takes linear time. If $T(N)$ is the running time of the suffix array construction algorithm, then the recursive call takes $T(2N/3)$ time.

Example:
In our example

$$S_1S_2 = \text{[BRA], [CAD], [ABR], [A\$\$], [RAC], [ADA], [BRA]}$$

The sorted suffixes that will be computed recursively will represent tri-character strings as shown in Figure 12.28.

Notice that these are not exactly the same as the corresponding suffixes in S; however, if we strip out characters starting at the first $, we do have a match of suffixes. Also note that the indices returned by the recursive call do not correspond directly to the indices in S, though it is a simple matter to map them back. So to see how the algorithm actually forms the recursive call, observe that three passes of radix sort will assign the following alphabet: [A$$] = 1, [ABR] = 2, [ADA] = 3, [BRA] = 4, [CAD] = 5, [RAC] = 6. Figure 12.29 shows the mapping of tri-characters, the resulting array that is formed for S_1, S_2, and the resulting suffix array that is computed recursively.

	Index	*Substring Being Represented*
0	3	`[A$$] [RAC] [ADA] [BRA]`
1	2	`[ABR] [A$$] [RAC] [ADA] [BRA]`
2	5	`[ADA] [BRA]`
3	6	`[BRA]`
4	0	`[BRA] [CAD] [ABR] [A$$] [RAC] [ADA] [BRA]`
5	1	`[CAD] [ABR] [A$$] [RAC] [ADA] [BRA]`
6	4	`[RAC] [ADA] [BRA]`

Figure 12.28 Suffix array for S_1 S_2 in tri-character set

S_1S_2	[BRA]	[CAD]	[ABR]	[A$$]	[RAC]	[ADA]	[BRA]			
Integers	4	5	2	1	6	3	4	0	0	0
SA[S_1S_2]	3	2	5	6	0	1	4	0	0	0
Index	0	1	2	3	4	5	6	7	8	9

Figure 12.29 Mapping of tri-characters, the resulting array that is formed for S_1, S_2, and the resulting suffix array that is computed recursively

Step 4: Compute the suffix array for S_0. This is easy to do because

$$\begin{aligned} S_0[i \rightarrow] &= S[3i \rightarrow] \\ &= S[3i]\, S[3i+1 \rightarrow] \\ &= S[3i] S_1[i \rightarrow] \\ &= S_0[i] S_1[i \rightarrow] \end{aligned}$$

Since our recursive call has already sorted all $S_1[i \rightarrow]$, we can do step 4 with a simple two-pass radix sort: The first pass is on $S_1[i \rightarrow]$, and the second pass is on $S_0[i]$.

Example:
In our example

$$S_0 = \text{[ABR], [ACA], [DAB], [RA\$]}$$

From the recursive call in step 3, we can rank the suffixes in S_1 and S_2. Figure 12.30 shows how the indices in the original string can be referenced from the recursively computed suffix array and shows how the suffix array from Figure 12.29 leads to a ranking of suffixes among $S_1 + S_2$. Entries in the next-to-last row are easily obtained from the prior two rows. In the last row, the ith entry is given by the location of i in the row labelled $SA[S_1, S_2]$.

The ranking established in S_1 can be used directly for the first radix sort pass on S_0. Then we do a second pass on the single characters from S, using the prior radix sort to break ties. Notice that it is convenient if S_1 has exactly as many elements as S_0. Figure 12.31 shows how we can compute the suffix array for S_0.

At this point, we now have the suffix array for S_0 and for the combined group S_1 and S_2. Since this is a two-pass radix sort, this step takes $O(N)$.

	S_1				S_2		
	[BRA]	[CAD]	[ABR]	[A$$]	[RAC]	[ADA]	[BRA]
Index in S	1	4	7	10	2	5	8
SA[S_1S_2]	3	2	5	6	0	1	4
SA using S's indices	10	7	5	8	1	4	2
Rank in group	5	6	2	1	7	3	4

Figure 12.30 Ranking of suffixes based on suffix array shown in Figure 12.29

	S_0				
	[ABR]	[ACA]	[DAB]	[RA$]	
Index	0	3	6	9	
Index of second element	1	4	7	10	*add one to above*
Radix Pass 1 ordering	5	6	2	1	*last line of Figure 12.30*
Radix Pass 2 ordering	1	2	3	4	*stably radix sort by first char*
Rank in group	1	2	3	4	*using results of previous line*
SA, using S's indices	0	3	6	9	*using results of previous line*

Figure 12.31 Computing suffix array for S_0

Step 5: Merge the two suffix arrays using the standard algorithm to merge two sorted lists. The only issue is that we must be able to compare each suffix pair in constant time. There are two cases.

Case 1: Comparing an S_0 element with an S_1 element: Compare the first letter; if they do not match, we are done; otherwise, compare the remainder of S_0 (which is an S_1 suffix) with the remainder of S_1 (which is an S_2 suffix); those are already ordered, so we are done.

Case 2: Comparing an S_0 element with an S_2 element: Compare at most the first two letters; if we still have a match, then at that point compare the remainder of S_0 (which after skipping the two letters becomes an S_2 suffix) with the remainder of S_2 (which after skipping two letters becomes an S_1 suffix); as in case 1, those suffixes are already ordered by SA12 so we are done.

Example:
In our example, we have to merge

	A	A	D	R
SA for S_0	0	3	6	9
	↑			

with

	A	A	A	B	B	C	R
SA for S_1 and S_2	10	7	5	8	1	4	2
	↑						

The first comparison is between index 0 (an A), which is an S_0 element and index 10 (also an A) which is an S_1 element. Since that is a tie, we now have to compare index 1 with index 11. Normally this would have already been computed, since index 1 is S_1, while index 11 is in S_2. However, this is special because index 11 is past the end of the string; consequently it always represents the earlier suffix lexicographically, and the first element in the final suffix array is 10. We advance in the second group and now we have.

	A	A	D	R
SA for S_0	0	3	6	9
	↑			

	A	A	A	B	B	C	R
SA for S_1 and S_2	10	7	5	8	1	4	2
		↑					

Final SA	10										
Input S	A	B	R	A	C	A	D	A	B	R	A
Index	0	1	2	3	4	5	6	7	8	9	10

Again the first characters match, so we compare indices 1 and 8, and this is already computed, with index 8 having the smaller string. So that means that now 7 goes into the final suffix array, and we advance the second group, obtaining

	A	A	D	R
SA for S_0	0	3	6	9
	↑			

	A	A	A	B	B	C	R
SA for S_1 and S_2	10	7	5	8	1	4	2
			↑				

Final SA	10	7									
Input S	A	B	R	A	C	A	D	A	B	R	A
Index	0	1	2	3	4	5	6	7	8	9	10

Once again, the first characters match, so now we have to compare indices 1 and 6. Since this is a comparison between an S_1 element and an S_0 element, we cannot look up the result. Thus we have to compare characters directly. Index 1 contains a B and index 6 contains a D, so index 1 wins. Thus 0 goes into the final suffix array and we advance the first group.

	A	A	D	R
SA for S_0	0	3	6	9
		↑		

	A	A	A	B	B	C	R
SA for S_1 and S_2	10	7	5	8	1	4	2
			↑				

Final SA	10	7	0								
Input S	A	B	R	A	C	A	D	A	B	R	A
Index	0	1	2	3	4	5	6	7	8	9	10

The same situation occurs on the next comparison between a pair of A's; the second comparison is between index 4 (a C) and index 6 (a D), so the element from the first group advances.

	A	A	D	R
SA for S_0	0	3	6	9
			↑	

	A	A	A	B	B	C	R
SA for S_1 and S_2	10	7	5	8	1	4	2
			↑				

Final SA	10	7	0	3							
Input S	A	B	R	A	C	A	D	A	B	R	A
Index	0	1	2	3	4	5	6	7	8	9	10

At this point, there are no ties for a while, so we quickly advance to the last characters of each group:

	A	A	D	R
SA for S_0	0	3	6	9
				↑

	A	A	A	B	B	C	R
SA for S_1 and S_2	10	7	5	8	1	4	2
							↑

Final SA	10	7	0	3	5	8	1	4	6		
Input S	A	B	R	A	C	A	D	A	B	R	A
Index	0	1	2	3	4	5	6	7	8	9	10

Finally, we get to the end. The comparison between two R's requires that we compare the next characters, which are at indices 10 and 3. Since this comparison is between an S_1 element and an S_0 element, as we saw before, we cannot look up the result and must compare directly. But those are also the same, so now we have to compare indices 11 and 4, which is an automatic winner for index 11 (since it is past the end of the string). Thus the R in index 9 advances, and then we can finish the merge. Notice that had we not been at the end of the string, we could have used the fact that the comparison is between an S_2 element and an S_1 element, which means the ordering would have been obtainable from the suffix array for $S_1 + S_2$.

	A	A	D	R
SA for S_0	0	3	6	9

↑

	A	A	A	B	B	C	R
SA for S_1 and S_2	10	7	5	8	1	4	2

↑

Final SA	10	7	0	3	5	8	1	4	6	9	2
Input S	A	B	R	A	C	A	D	A	B	R	A
Index	0	1	2	3	4	5	6	7	8	9	10

```
/*
 * Fill in the suffix array information for String str
 * str is the input String
 * sa is an existing array to place the suffix array
 * LCP is an existing array to place the LCP information
 */
void createSuffixArray( const string & str, vector<int> & sa, vector<int> & LCP )
{
    if( sa.size( ) != str.length( ) || LCP.size( ) != str.length( ) )
        throw invalid_argument{ "Mismatched vector sizes" };

    int N = str.length( );

    vector<int> s( N + 3 );
    vector<int> SA( N + 3 );

    for( int i = 0; i < N; ++i )
        s[ i ] = str[ i ];

    makeSuffixArray( s, SA, N, 250 );

    for( int i = 0; i < N; ++i )
        sa[ i ] = SA[ i ];

    makeLCPArray( s, sa, LCP );
}
```

Figure 12.32 Code to set up the first call to `makeSuffixArray`; create appropriate size arrays, and to keep things simple; just use the 256 ASCII character codes

Since this is a standard merge, with at most two comparisons per suffix pair, this step takes linear time. The entire algorithm thus satisfies $T(N) = T(2N/3) + O(\ N\ )$ and takes linear time. Although we have only computed the suffix array, the LCP information can also be computed as the algorithm runs, but there are some tricky details that are involved, and often the LCP information is computed by a separate linear-time algorithm.

We close by providing a working implementation to compute suffix arrays; rather than fully implementing step 1 to sort the original characters, we'll assume only a small set of ASCII characters (residing in values 1–255) are present in the string. In Figure 12.32, we allocate the arrays that have three extra slots for padding and call `makeSuffixArray`, which is the basic linear-time algorithm.

Figure 12.33 shows `makeSuffixArray`. At lines 11 to 15, it allocates all the needed arrays and makes sure that S_0 and S_1 have the same number of elements (lines 16 to 21); it then delegates work to `assignNames, computeS12, computeS0`, and `merge`.

```
 1  // find the suffix array SA of s[0..n-1] in {1..K}^n
 2  // requires s[n]=s[n+1]=s[n+2]=0, n>=2
 3  void makeSuffixArray( const vector<int> & s, vector<int> & SA, int n, int K )
 4  {
 5      int n0 = ( n + 2 ) / 3;
 6      int n1 = ( n + 1 ) / 3;
 7      int n2 = n / 3;
 8      int t = n0 - n1;  // 1 iff n%3 == 1
 9      int n12 = n1 + n2 + t;
10
11      vector<int> s12( n12 + 3 );
12      vector<int> SA12( n12 + 3 );
13      vector<int> s0( n0 );
14      vector<int> SA0( n0 );
15
16      // generate positions in s for items in s12
17      // the "+t" adds a dummy mod 1 suffix if n%3 == 1
18      // at that point, the size of s12 is n12
19      for( int i = 0, j = 0; i < n + t; ++i )
20          if( i % 3 != 0 )
21              s12[ j++ ] = i;
22
23      int K12 = assignNames( s, s12, SA12, n0, n12, K );
24
25      computeS12( s12, SA12, n12, K12 );
26      computeS0( s, s0, SA0, SA12, n0, n12, K );
27      merge( s, s12, SA, SA0, SA12, n, n0, n12, t );
28  }
```

Figure 12.33 The main routine for linear-time suffix array construction

```
 1  // Assigns the new supercharacter names.
 2  // At end of routine, SA will have indices into s, in sorted order
 3  // and s12 will have new character names
 4  // Returns the number of names assigned; note that if
 5  // this value is the same as n12, then SA is a suffix array for s12.
 6  int assignNames( const vector<int> & s, vector<int> & s12, vector<int> & SA12,
 7                   int n0, int n12, int K )
 8  {
 9        // radix sort the new character trios
10      radixPass( s12 , SA12, s, 2, n12, K );
11      radixPass( SA12, s12 , s, 1, n12, K );
12      radixPass( s12 , SA12, s, 0, n12, K );
13
14        // find lexicographic names of triples
15      int name = 0;
16      int c0 = -1, c1 = -1, c2 = -1;
17
18      for( int i = 0; i < n12; ++i )
19      {
20          if( s[ SA12[ i ] ] != c0 || s[ SA12[ i ] + 1 ] != c1
21                                   || s[ SA12[ i ] + 2 ] != c2 )
22          {
23              ++name;
24              c0 = s[ SA12[ i ] ];
25              c1 = s[ SA12[ i ] + 1 ];
26              c2 = s[ SA12[ i ] + 2 ];
27          }
28
29          if( SA12[ i ] % 3 == 1 )
30              s12[ SA12[ i ] / 3 ]      = name;   // S1
31          else
32              s12[ SA12[ i ] / 3 + n0 ] = name;   // S2
33      }
34
35      return name;
36  }
```

Figure 12.34 Routine to compute and assign the tri-character names

assignNames, shown in Figure 12.34, begins by performing three passes of radix sort. Then, it assigns names (i.e., numbers), sequentially using the next available number if the current item has a different trio of characters than the prior item (recall that the tri-characters have already been sorted by the three passes of radix sort, and also recall that S_0 and S_1 have the same size, so at line 32, adding n0 adds the number of elements in S_1). We can use the basic counting radix sort from Chapter 7 to obtain a linear-time sort. This code is shown in Figure 12.35. The array in represents the indexes into s; the result of the

```
// stably sort in[0..n-1] with indices into s that has keys in 0..K
// into out[0..n-1]; sort is relative to offset into s
// uses counting radix sort
void radixPass( const vector<int> & in, vector<int> & out,
                const vector<int> & s, int offset, int n, int K )
{
    vector<int> count( K + 2 );                 // counter array

    for( int i = 0; i < n; ++i )
        ++count[ s[ in[ i ] + offset ] + 1 ];   // count occurrences

    for( int i = 1; i <= K + 1; ++i )           // compute exclusive sums
        count[ i ] += count[ i - 1 ];

    for( int i = 0; i < n; ++i )
        out[ count[ s[ in[ i ] + offset ] ]++ ] = in[ i ];      // sort
}

// stably sort in[0..n-1] with indices into s that has keys in 0..K
// into out[0..n-1]
// uses counting radix sort
void radixPass( const vector<int> & in, vector<int> & out,
                const vector<int> & s, int n, int K )
{
    radixPass( in, out, s, 0, n, K );
}
```

Figure 12.35 A counting radix sort for the suffix array

radix sort is that the indices are sorted so that the characters in `s` are sorted at those indices (where the indices are `offset` as specified).

Figure 12.36 contains the routines to compute the suffix arrays for `s12`, and then `s0`.

Finally, the `merge` routine is shown in Figure 12.37, with some supporting routines in Figure 12.38. The merge routine has the same basic look and feel as the standard merging algorithm seen in Figure 7.12.

12.5 *k*-d Trees

Suppose that an advertising company maintains a database and needs to generate mailing labels for certain constituencies. A typical request might require sending out a mailing to people who are between the ages of 34 and 49 and whose annual income is between \$100,000 and \$150,000. This problem is known as a two-dimensional range query. In one dimension, the problem can be solved by a simple recursive algorithm in $O(M + \log N)$ average time, by traversing a preconstructed binary search tree. Here M is the number