Left-leaning Red-Black Trees

Robert Sedgewick
Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract

The red-black tree model for implementing balanced search trees, introduced by Guibas and Sedgewick thirty years ago, is now found throughout our computational infrastructure. Red-black trees are described in standard textbooks and are the underlying data structure for symbol-table implementations within C++, Java, Python, BSD Unix, and many other modern systems. However, many of these implementations have sacrificed some of the original design goals (primarily in order to develop an effective implementation of the delete operation, which was incompletely specified in the original paper), so a new look is worthwhile. In this paper, we describe a new variant of red-black trees that meets many of the original design goals and leads to substantially simpler code for insert/delete, less than one-fourth as much code as in implementations in common use.

All red-black trees are based on implementing 2-3 or 2-3-4 trees within a binary tree, using red links to bind together internal nodes into 3-nodes or 4-nodes. The new code is based on combining three ideas:

Use a recursive implementation.

Require that all 3-nodes lean left.

Perform rotations on the way up the tree (after the recursive calls).

Not only do these ideas lead to simple code, but they also unify the algorithms: for example, the left-leaning versions of 2-3 trees and top-down 2-3-4 trees differ in the position of one line of code.

All of the red-black tree algorithms that have been proposed are characterized by a worst-case search time bounded by a small constant multiple of lg N in a tree of N keys, and the behavior observed in practice is typically that same multiple faster than the worst-case bound, close the to optimal lg N nodes examined that would be observed in a perfectly balanced tree. This performance is also conjectured (but not yet proved) for trees built from random keys, for all the major variants of red-black trees. Can we analyze average-case performance with random keys for this new, simpler version? This paper describes experimental results that shed light on the fascinating dynamic behavior of the growth of these trees. Specifically, in a left-leaning red-black 2-3 tree built from N random keys:

A random successful search examines lg N 0.5 nodes.

The average tree height is about 2 ln N (!)

The average size of left subtree exhibits log-oscillating behavior.

The development of a mathematical model explaining this behavior for random keys remains one of the outstanding problems in the analysis of algorithms.

From a practical standpoint, left-leaning red-black trees (LLRB trees) have a number of attractive characteristics:

Experimental studies have not been able to distinguish these algorithms from optimal.

They can be implemented by adding just a few lines of code to standard BST algorithms.

Unlike hashing, they support ordered operations such as *select*, *rank*, and *range search*.

Thus, LLRB trees are useful for a broad variety of symbol-table applications and are prime candidates to serve as the basis for symbol tables in software libraries in the future.

Introduction

We focus in this paper on the goal of providing efficient implementations of the following operations on in a symbol table containing generic keys and associated values.

Search for the value associated with a given key.

Insert a key-value pair into the symbol table.

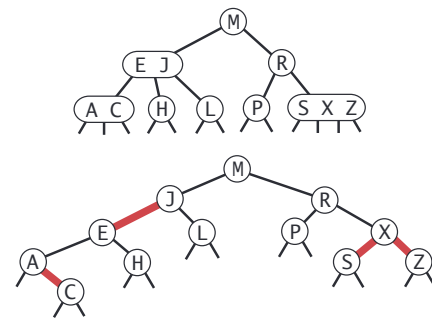*Delete* the key-value pair with a given key from the symbol table.

When an insert operation involves a key that is already in the table, we associate that key with the new value, as specified. Thus we do not have duplicate keys in the table and are implementing the *associative array* abstraction. We further assume that keys are *comparable*: we have available a *compare* operation that can determine whether one given key is less than, equal to, or greater than another given key, so that we preserve the potential to implement the *ordered associative array abstraction*, where we can support *rank*, *select*, *range search*, and similar operations that are of critical importance in many applications.

Developing data structures and efficient algorithms for these operations is an old and well-studied problem. The starting point for this paper is the *balanced tree* data structures that were developed in the 1960s and 1970s, which provide a guaranteed worst-case running time that is proportional to log N for both operations. These algorithms are based on modifying the elementary binary search tree (BST) data structure to guarantee that the length of every path to an external node is proportional to log N. Examples of such algorithms are 2-3 trees, 2-3-4 trees, AVL trees, and B trees. This paper is largely self-contained for people familiar with balanced-tree algorithms; others can find basic definitions and examples in a standard textbook such as [6], [9], or [13].

In [7], Guibas and Sedgewick showed that all of these algorithms can be implemented with *red-black trees*, where each link in a BST is assigned a color (red or black) that can be used to control the balance, and that this framework can simplify the implementation of the various algorithms. In particular, the paper describes a way to maintain a correspondence between red-black trees and 2-3-4 trees, by interpreting red links as internal links in 3-nodes and 4-nodes. Since red links can lean either way in 3-nodes (and, for some implementations in 4-nodes), the correspondence is not necessarily 1-1. For clarity in our code, we use a boolean variable (a single bit) to encode the color

Red-black representation of a 2-3-4 tree

of a link in the node it points to, though Brown [5] has pointed out that we can mark nodes as red by switching their pointers, so that we can implement red-black trees without any extra space.

One of the most important feature of red-black trees is that they add no overhead for *search*, the most commonly used operation. Accordingly, red-black trees are the underlying data structure for symbol-table implementations within C++, Java, Python, BSD Unix, and many other systems.

Why revisit such a successful data structure? The actual code found in many implementations is difficult to maintain and to reuse in new systems because it is lengthy, running 100-200 lines of code in typical implementations. Full implementations are rarely found in textbooks, with numerous "symmetric" cases left for the student to implement. In this paper, we present an approach that can dramatically reduce the amount of code required. To prove the point, we present a full Java implementation, comprising three short utility methods, adding 3 lines of code to standard BST code for *insert*, a 5-line method for *delete the maximum*, and 30 additional lines of code for *delete*.

Rotations and color flips

One way to view red-black BST algorithms is as maintaining the following invariant properties under insertion and deletion:

No path from the root to the bottom contains two consecutive red links.

The number of black links on every such path is the same.

These invariants imply that the length of every path in a red-black tree with N nodes is no longer than $2 \lg N$. This worst case is realized, for example, in a tree whose nodes are all black except for those along a single path of alternating red and black nodes.

The basic operations that balanced-tree algorithms use to maintain balance under insertion and deletion are known as *rotations*. In the context of red-black trees, these operations are easily understood as the transformations needed to transform a 3-node whose red link leans to the left to a 3-node whose red link leans to the right and vice-versa. The Java code for these operations (for a `Node` type that we will consider late that contains a left link, a right link, and a color field that can be set to the value `RED` to represent the color of the incoming link) is given to the left and to the right on this page. Rotations obviously preserve the two invariants stated above.

In red-black trees, we also use a simple operation known as a *color flip* (shown at the bottom of this page). In terms of 2-3-4 trees, a color flip is the essential operation: it corresponds to splitting a 4-node and passing the middle node up to the parent. A color flip obviously does not change the number of black links on any path from the root to the bottom, but it may introduce two consecutive red links higher in the tree, which must be corrected.
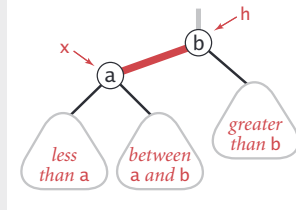
Red-black BST algorithms differ on whether and when they do rotations and color flips, in order to maintain the global invariants stated at the top of this page.
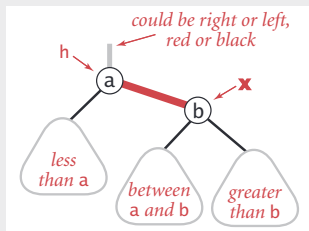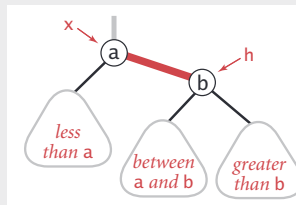


```
Node rotateLeft(Node h)
{
    x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```
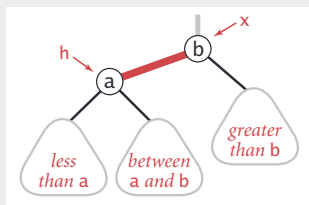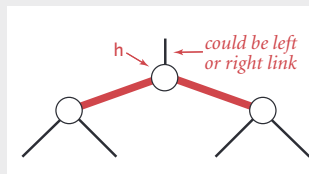

Left rotate right link of)


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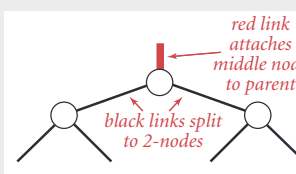

Right rotate left link of)


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

Flipping colors to split a 4-node

Left-leaning red-black trees

Our starting point is the Java implementation of standard BSTs shown in the gray code on the next page. Java aficionados will see that the code uses generics to support, in a type-safe manner, arbitrary types for client keys and values. Otherwise, the code is standard and easily translated to other languages or adapted to specific applications where generic types may not be needed.

In the present context, an important feature of the implementation is that the implementation of `insert()` is *recursive*: each recursive call takes a link as argument and returns a link, which is used to reset the field from which the link was taken. For standard BSTs, the argument and return value are the same except at the bottom of the tree, where this code serves to insert the new node. For red-black trees, this recursive implementation helps simplify the code, as we will see. We could also use a recursive implementation for `search()` but we do not do so because this operation falls within the inner loop in typical applications.

The basis of algorithms for implementing red-black trees is to add *rotate* and *color flip* operations to this code, in order to maintain the invariants that dictate balance in the tree. Most published implementations involve code laden with cases that are nearly identical for right and left. In the code in this paper, we show that the number of cases can be substantially reduced by:

requiring that 3-nodes always lean to the left (and that 4-nodes are balanced)

doing rotations after the recursive calls, on the way up the tree.

The lean-to-the-left requirement gives a 1-1 correspondence between red-black and 2-3-4 trees and reduces the number of cases to consider. The rotate-on-the-way up strategy simplifies the code (and our understanding of it) by combining various cases in a natural way. Neither idea is new (the first was used by Andersson [2] and the second is used in [9]) but in combination they surprisingly effective in reducing the amount of code required for several versions of the data structure. The code in black on the next page derives two classic algorithms by adding 3 lines of code to `insert()`.

Top-down 2-3-4 trees To insert a new node, we flip colors to split any 4-node encountered on the way *down* the tree and do rotations to balance 4-nodes (eliminate occurrences of consecutive red links on the way *up* the tree). This approach is natural because splitting 4-nodes to ensure that the search does not terminate on a 4-node means that a new node can be added by attaching it with a red link, and balancing a 4-node amounts to handling the three possible ways a red link could be attached to a 3-node, as shown in the diagram at right. If the red link that is passed up happens to lean to the right in a 3-node, we correct that condition when we encounter it.
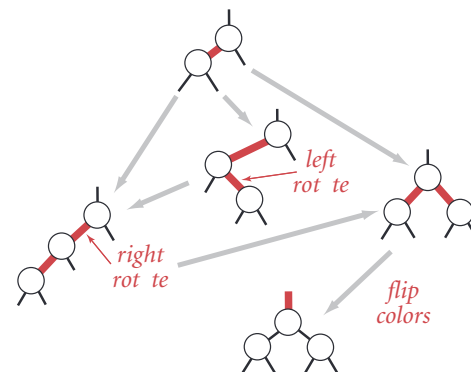
Passing a red link up in a LLRB tree

2-3 trees Remarkably, moving the color flip to the end in the top-down 2-3-4 tree implementation just described yields an implementation for 2-3 trees. We split any 4-node that is created by doing a color flip, passing a red link up the tree, and dealing with the effects of doing so in precisely the same way as we move up the tree.

These ideas are also effective for simplifying other variations of red-black trees that have been studied, which we cannot consider in this short abstract for lack of space. These include handling equal keys, completing the insertion in a single top-down pass, and completing the insertion with at most one rotation in 2-3-4 trees.

```java
public class LLRB<Key extends Comparable<Key>, Value>
{
   private static final boolean RED   = true;
   private static final boolean BLACK = false;

   private Node root;

   private class Node
   {
      private Key key;
      private Value val;
      private Node left, right;
      private boolean color;          ◄——— represent color with a 1-bit field

      Node(Key key, Value val)
      {
         this.key = key;
         this.val = val;
         this.color = RED;            ◄——— new nodes are always red
      }
   }

   public Value search(Key key)
   {
      Node x = root;
      while (x != null)
      {
         int cmp = key.compareTo(x.key);
         if (cmp == 0) return x.val;
         else if (cmp < 0) x = x.left;
         else if (cmp > 0) x = x.right;
      }
      return null;
   }

   public void insert(Key key, Value value)
   {
      root = insert(root, key, value);
      root.color = BLACK;
   }
   private Node insert(Node h, Key key, Value value)
   {
      if (h == null)       return new Node(key, value);

      if (isRed(h.left) && isRed(h.right)) colorFlip(h);

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

      if (isRed(h.right) && !isRed(h.left))    h = rotateLeft(h);
      if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);

      return h;
   }
}
```

move this line to the end to get 2-3 trees

Java code to implement LLRB trees (standard BST code in gray)

Deletion

Efficient implementation of the delete operation is a challenge in many symbol-table implementations, and red-black trees are no exception. Industrial-strength implementations run to over 100 lines of code, and text books generally describe the operation in terms of detailed case studies, eschewing full implementations. Guibas and Sedgewick presented a delete implementation in [7], but it is not fully specified and depends on a call-by-reference approach not commonly found in modern code. The most popular method in common use is based on a parent pointers (see [6]), which adds substantial overhead and does not reduce the number of cases to be handled.

The code on the next page is a full implementation of `delete()` for LLRB 2-3 trees. It is based on the reverse of the approach used for *insert* in top-down 2-3-4 trees: we perform rotations and color flips on the way down the search path to ensure that the search does not end on a 2-node, so that we can just delete the node at the bottom. We use the method `fixUp()` to share the code for the color flip and rotations following the recursive calls in the `insert()` code. With `fixUp()`, we can leave right-leaning red links and unbalanced 4-nodes along the search path, secure that these conditions will be fixed on the way up the tree. (The approach is also effective 2-3-4 trees, but requires an extra rotation when the right node off the search path is a 4-node.)

As a warmup, consider the *delete-the-minimum* operation, where the goal is to delete the bottom node on the left spine while maintaining balance. To do so, we maintain the invariant that the current node or its left child is red. We can do so by moving to the left unless the current node is red and its left child and left grandchild are both black. In that case, we can do a color flip, which restores the invariant but may introduce successive reds on the right. In that case, we can correct the condition with two rotations and a color flip. These operations are implemented in the `moveRedLeft()` method on the next page. With `moveRedLeft()`, the recursive implementation of `deleteMin()` above is straightforward.

```
public void deleteMin()
{
    root = deleteMin(root);
    root.color = BLACK;
}

private Node deleteMin(Node h)
{
    if (h.left == null) return null;

    if (!isRed(h.left) && !isRed(h.left.left))
        h = moveRedLeft(h);

    h.left = deleteMin(h.left);

    return fixUp(h);
}
```

Delete-the-minimum code for LLRB 2-3 trees

For general deletion, we also need `moveRedRight()`, which is similar, but simpler, and we need to rotate left-leaning red links to the right on the search path to maintain the invariant. If the node to be deleted is an internal node, we replace its key and value fields with those in the minimum node in its right subtree and then delete the minimum in the right subtree (or we could rearrange pointers to use the node instead of copying fields). The full implementation of `delete()` that dervies from this discussion is given on the facing page. It uses one-third to one-quarter the amount of code found in typical implementations. It has been demonstrated before [2, 11, 13] that maintaining a field in each node containing its height can lead to code for *delete* that is similarly concise, but that extra space is a high price to pay in a practical implementation. With LLRB trees, we can arrange for concise code having a logarithmic performance guarantee and using no extra space.

```
private Node moveRedLeft(Node h)
{
   colorFlip(h);
   if (isRed(h.right.left))
   {
      h.right = rotateRight(h.right);
      h = rotateLeft(h);
      colorFlip(h);
   }
   return h;
}

private Node moveRedRight(Node h)
{
    colorFlip(h);
    if (isRed(h.left.left))
    {
       h = rotateRight(h);
       colorFlip(h);
    }
    return h;
}

public void delete(Key key)
{
   root = delete(root, key);
   root.color = BLACK;
}

private Node delete(Node h, Key key)
{
    if (key.compareTo(h.key) < 0)
        {
            if (!isRed(h.left) && !isRed(h.left.left))
                h = moveRedLeft(h);
            h.left =  delete(h.left, key);
        }
    else
        {
            if (isRed(h.left))
                h = rotateRight(h);
            if (key.compareTo(h.key) == 0 && (h.right == null))
                return null;
            if (!isRed(h.right) && !isRed(h.right.left))
                h = moveRedRight(h);
            if (key.compareTo(h.key) == 0)
                {
                    h.val = get(h.right, min(h.right).key);
                    h.key = min(h.right).key;
                    h.right = deleteMin(h.right);
                }
            else h.right = delete(h.right, key);
        }

    return fixUp(h);
}
```
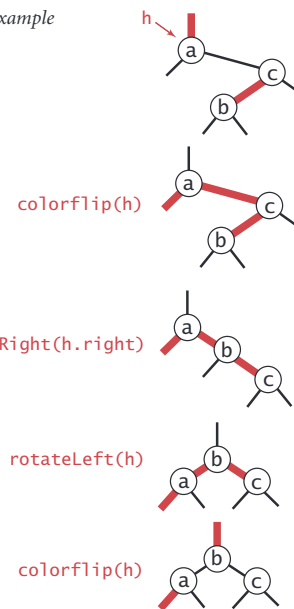

moveRedLeft(h) *example*

colorflip(h)

h.right = rotateRight(h.right)

h = rotateLeft(h)

colorflip(h)

Delete code for LLRB 2-3 trees

Properties of LLRB trees built from random keys

By design, the worst-case cost of a search in an LLRB tree with N nodes is 2 lg N. In practical applications, however, the cost of a typical search is half that value, not perceptibly different from the cost of a search in a perfectly balanced tree. Since searches are far more common than inserts in typical symbol-table applications, the usual first step in studying a symbol-table algorithm is to assume that a table is built from random keys (precisely, a random permutation of distinct keys) and then study the cost of searchers. For standard BSTs and other methods, mathematical models based on this assumption have been developed and validated with experimental results and practical experience. The development of a corresponding mathematical model for balanced trees is one of the outstanding problems in the analysis of algorithms.

In this paper, we present experimental results that may help guide the development of such a model, using a modified form of a plot format suggested by Tufte [12]. Specifically, we use
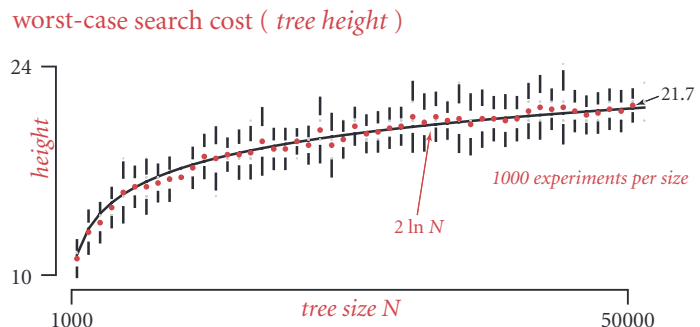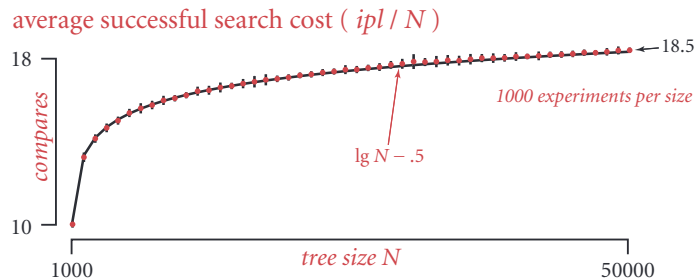
a *gray dot* to depict the result of each experiment

a *red dot* to depict the average value of the experiments for each tree size

black line segments to depict the standard deviation   of the experiments for each tree size, of length   and spaced   above and below the red dots

While sometimes difficult to distinguish individually, the gray dots help illustrate the extent and the dispersion of the experimental results. The plots at right each represent the results of 50,000 experiments, each involving building a 2-3 tree from a random permutation of distinct keys.
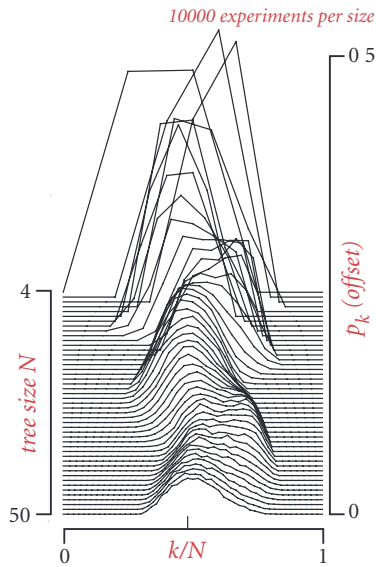
Average path length. What is the cost of a typical search? That is the question of most interest in practice. In typical large-scale applications, most searches are successful and bias towards specific keys is relatively insignificant, so the measuring the average length to a node in a tree constructed from random keys is a reasonable estimate. As shown in our first plot, this measure is extremely close to the optimal value lg N − .5 that would be found in a fully balanced tree. The plots for top-down 2-3-4 trees and other types of red-black trees are indistinguishable from this one.
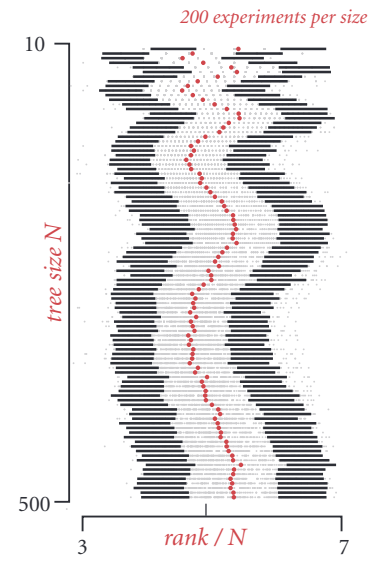
Height. What is the expected worst-case search cost? This question is primarily of academic interest, but may shed some light on the structure of the trees. Though the dispersion is much higher than the average, our second plot shows that the height is close to 2 ln N, the same value as the *average* cost of a search in a BST (!). However, this precise value is pure conjecture: for example, experiments for standard BSTs would suggest the average height 3 lg N , but the actual value of the coefficient is known to be slightly less than 3.

Experimental results for LLRB trees built from random keys

Distribution. The first step to developing a mathematical model that explains these results is to understand the distribution of the probability p_k that the root is of rank k, when a LLRB (2-3) tree is built from random keys. We know this probability 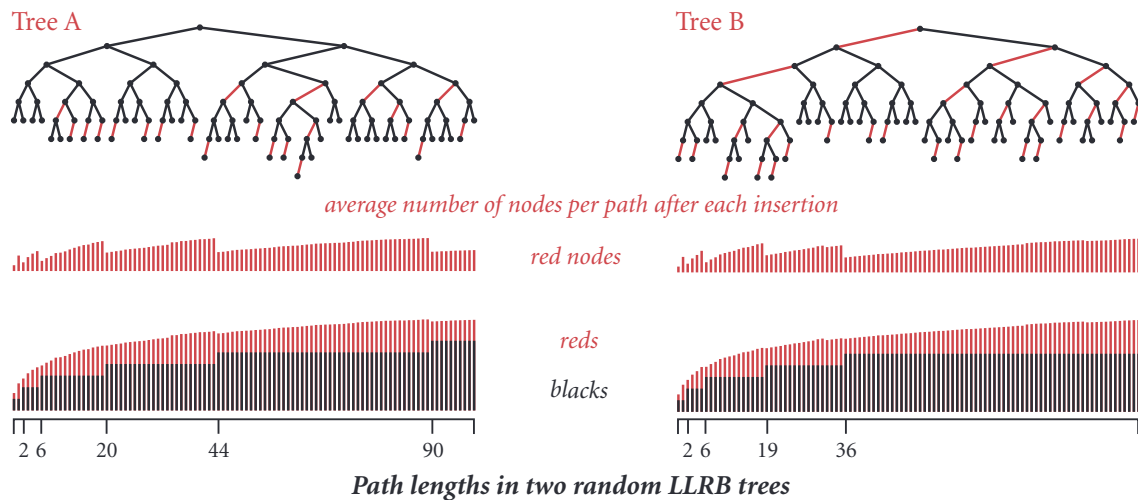to be 0 for small k and for large k, and we expect it to be high when k is near $N/2$. The figure at left shows the result of computing the distribution exactly for small N and estimating its shape for intermediate values of N. Following the format introduced in [10], the curves are normalized on the x axis and slightly separated on the y axis, so that convergence to a distribution can be identified. The irregularities in the curves are primarily (but not completely) due to expected variations in the experimental results. (These curves are the result of building 10000 trees for each size, and are smoother than the curves based on a smaller number of experiments). Ideally, we would like to see convergence at the bottom to some distribution (whose properties we can analyze) for large N. Though it suggests the possibility of eventual convergence to a distribution that can be suitable approximated, this figure also exhibits an oscillation that may complicate such analysis. At right is shown a Tufte plot of the average for this distribution for a large number of experiments. This figure clearly illustrates a log-oscillatory behavior that is often found in the analysis of algorithms, and also shows that the dispersion is significant and does not seem to be decreasing.

10000 experiments per size

LLRB tree root rank distribution

200 experiments per size

Average root rank in LLRB trees

Red path length. How many red nodes are on the search path, on the average? This question would seem to be the key to understanding LLRB trees. The figure below shows that this varies (even

Tree A

Tree B

average number of nodes per path after each insertion

red nodes

reds

blacks

Path lengths in two random LLRB trees

though the total is relatively smooth. Close examination reveals that the average number of reds per path increases slowly, then drops each time the root splits. One important challenge is to characterize the root split events. The remarkable figure at right shows that variability in the time of root splits creates a significant challenge in developing a detailed characterization of the average number of red nodes per path. It is a modified Tufte plot showing that this quantity oscillates between periods of low and high variance and increases very slowly, if at all. This behavior is the result of averaging the sawtooth plots with different root split times like the ones at the bottom of the previous page. It is quite remarkable that the quantity of primary practical interest (the average path length) should be so stable (as shown in our first plot and in the sum of the black and red path lengths in the plot at the bottom of the previous), but the underlying process should exhibit such wildly oscillatory behavior.
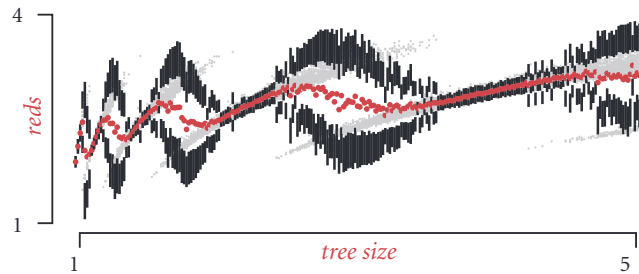
Average number of reds per path in random LLRB trees

Acknowledgement

The author wishes to thank Kevin Wayne for many productive discussions and for rekindling interest in this topic by encouraging work on the *delete* implementation.

References

1. G. M. Adelson-Velskii and E. M. Landis, *An algorithm for the organization of information,* Soviet Math. Doklady **3** (1962), 1259–1263.

2. A. Andersson, *Balanced search trees made simple,* Proceedings of the 3rd Workshop on Algorithms and Data Structures (1993), 290–306.

3. R. Baeza-Yates, *Fringe analysis revisited,* ACM Computing Surveys **27** (1995), 109–119.

4. R. Bayer, *Symmetric binary B-Trees: data structure and maintenance algorithms,* Acta Informatica **1** (1972), 290–306.

5. M. Brown, *Some observations on 2-3 trees,* Information Processing Letters **9** (1979), 57–59.

6. T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, MIT Press.

7. L. Guibas and R. Sedgewick, *A dichromatic framework for balanced trees,* Proceedings of the 19th Annual Conference on Foundations of Computer Science, Ann Arbor, MI (1978). (Also in *A Decade of Research — Xerox Palo Alto Research Center 1970–1980,* ed. G. Laverdel and E. R. Barker).

8. D. E. Knuth, The Art of Computer Programming, Vol. 3, Sorting and Searching, Addison–Wesley.

9. R. Sedgewick, Algorithms in Java, Parts 1–4: Fundamentals, Data Structures, Sorting, and Searching, Addison–Wesley.

10. R. Sedgewick and P. Flajolet, Introduction to the Analysis of Algorithms, Addison–Wesley, 1996.

11. R. Seidel, *personal communication.*

12. E. Tufte, Envisioning Information, Graphics Press, Chesire, CT, 1990.

13. M. Weiss, Data Structures and Problem Solving using Java, Addison-Wesley, 2002.

14. A. Yao, *On random 2-3 trees,* Acta Informatica **9** (1978), 159–170.