Left-leaning Red-Black Tree Program

Roman Valiušenko
roman.valiusenko@gmail.com

Intro. Red-black trees are self-balancing trees, also know as half-balanced tree, symmetric binary B-trees. By stipulating an additional requirement, namely that red-links always lean left, we gain simplification on insertion and deletion procedures [1, 3]. This data structure is called left-leaning red-black tree. We will maintain a one-to-one correspondence with 2–3 trees [2]. In those trees we have either 2-way or 3-way branching at each node (such nodes are called 2-node and 3-node respectively), and all external nodes appear on the same level. Essentially a left-leaning red-black tree is a 2–3 tree where red links are used to bind nodes together to form 3-nodes. Experimental results show that in a left-leaning red-black 2–3 tree built from N random keys a random successful search examines $\lg N - 1/2$ nodes [1].

The program will be written in Scheme programming language [10].

1a ⟨*Left-leaning Red-Black Tree implementation* 1a⟩≡
```
  ⟨Tree node definition 1b⟩
  ⟨Helper procedures 2b⟩
  ⟨Insertion procedure 4a⟩
  ⟨Remove minimum key procedure 5a⟩
  ⟨Remove maximum key procedure 7b⟩
  ⟨Remove procedure 8a⟩
```

Tree node. We will define a tree node in a message passing style, that is, a node will be a procedure that takes one argument and dispatch on its value. A node object will have `left` and `right` links, that will point to other nodes, just as in ordinary binary search tree; the node will also have `data` field to store data associated with `key`. Node provides getters and setters for each of these fields. Each node will be marked as either `red` or `black`, denoting the color of the link that goes from its parent to this node. A newly created node will always have `color` set to `black` and both links set to `nil`[1].

A dichromatic framework.

1b ⟨*Tree node definition* 1b⟩≡
```
  (define (make-node data key)
    (define (node data key color left right)
      (lambda (m)
        (cond
              ⟨Getters 1c⟩
              ⟨Setters 2a⟩)))
    (node data key 'red '() '()))
```
Defines:
  `make-node`, used in chunk 4a.

The getters will provide access to `left`, `right`, `key`, and `data`.

1c ⟨*Getters* 1c⟩≡
```
  ((eq? m 'left) left)
  ((eq? m 'right) right)
  ((eq? m 'data) data)
  ((eq? m 'key) key)
  ((eq? m 'color) color)
```

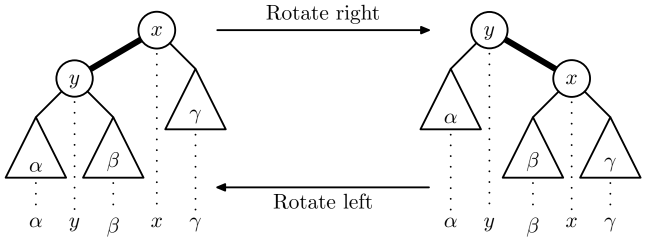
[1]We will use `nil` to reference an empty list in the text below. It plays a role of a sentinel node.

The setters return another procedures, one for each node's field, that take one argument and bind the corresponding node field to the given argument.

2a ⟨*Setters* 2a⟩≡

```
((eq? m 'set-color) (lambda (x) (set! color x)))
((eq? m 'set-data) (lambda (x) (set! data x)))
((eq? m 'set-left) (lambda (x) (set! left x)))
((eq? m 'set-right) (lambda (x) (set! right x)))
```

Helpers. We will employ two simple tree transformations called *rotations* [9, 2, 6]. These transformations preserve symmetric order of the tree nodes (see image below). Such transformations are like applying the associative law to an algebraic formula, replacing $\alpha(\beta\gamma)$ by $(\alpha\beta)\gamma$ [4].



We'll start off with rotation to the right procedure. The `new-node` is set to link to the left child of the current node `node` (this `new-node` will eventually replace `node`). In the image above `new-node` points to y, and `node` points to x. Now `new-node`'s right child (i.e. β) becomes `node`'s left child, and `node` itself becomes `new-node`'s right child. Then we set `new-node` color to one that `node` had, and we set `node`'s—which is now right child of `new-node`—color to be `red` (because we are going to rotate 3-nodes only). The procedure returns `new-node` which is the node to be used instead of `node`.

2b ⟨*Helper procedures* 2b⟩≡

```
(define (rotate-right node)
  (let ((new-node (node 'left)))
    ((node 'set-left) (new-node 'right))
    ((new-node 'set-right) node)
    ((new-node 'set-color) (node 'color))
    ((node 'set-color) 'red)
    new-node))
```

Defines:
`rotate-right`, used in chunks 2b, 4b, 6, 7, and 9b.

We will define rotation to the left in a symmetrical way. We could go ahead and merge these two procedures into one procedure and then define `rotate-right` and `rotate-left` in terms of that procedure by passing in appropriate getters and setters. However, I have chosen not to pursue that path.

3a ⟨*Helper procedures* 2b⟩+≡

```
(define (rotate-left node)
  (let ((new-node (node 'right)))
    ((node 'set-right) (new-node 'left))
    ((new-node 'set-left) node)
    ((new-node 'set-color) (node 'color))
    ((node 'set-color) 'red)
    new-node))
```

Defines:
  `rotate-left`, used in chunks 3a, 4b, and 6a.

We will also need an additional predicate procedure, `is-red?`, which tests node's `color`. Color of `nil` is considered to be `black`.

3b ⟨*Helper procedures* 2b⟩+≡

```
(define (is-red? n) (if (null? n) #f (eq? (n 'color) 'red)))
```

Defines:
  `is-red?`, used in chunks 3–7 and 9.

A color flip operation inverts `color` for `node` and its children. This is essentially a *merge* and *split* operation in terms of 2–3 trees.

3c ⟨*Helper procedures* 2b⟩+≡

```
(define (color-flip node)
  (define (flip n) ((n 'set-color) (if (is-red? n) 'black 'red)))
  (flip node)
  (flip (node 'left))
  (flip (node 'right)))
```

Defines:
  `color-flip`, used in chunks 4b, 3, 6a, and 7a.
Uses `is-red?` 3b.

Insertion. Since it's a binary search tree, we search down the tree as in ordinary BST, and once we hit the bottom, we insert the node [6]. Adding a new node may violate our LLRB tree. (We might get unbalanced 4-nodes, i.e. two `red` links in a row, or have right-leaning red links.) The tree must be fixed so that is becomes a valid LLRB tree again. We will do that via `fix-up` procedure. Note that `insert` procedure takes as argument another procedure, `compare`, which in turn takes two key values, m and n, and returns 0, -1, or 1, depending on whether $m = n$, $m < n$, or $m > n$ respectively.

Yes, `fix-up`! Wishful thinking [10]?

4a ⟨*Insertion procedure* 4a⟩≡
```
(define (insert node data key compare)
  (define (insert-node node)
    (cond ((null? node) (make-node data key))
          (else
           (let ((cmp (compare key (node 'key))))
             (cond ((= cmp 0) ((node 'set-data) data))
                   ((< cmp 0) ((node 'set-left) (insert-node (node 'left))))
                   (else ((node 'set-right) (insert-node (node 'right)))))
             (fix-up node)))))
  (insert-node node))
```
Defines:
  `insert`, used in chunk 4.
Uses `fix-up` 4b and `make-node` 1b.

After we have inserted a node we might end up with a tree that is not a LLRB tree. A newly created node is always `red`. Suppose it was added as right child, then we would have a right-leaning red link now. That can be corrected by rotation to the left. Suppose also that the parent node had been `red` as well, which means there would have been two `red` links in a row, making a 4-node. We would have had to split it (in terms of 2–3 trees). That can be achieved by rotation to the right first (i.e. to balance it) and a color flip (i.e. to actually split it). When we correct violations locally we might introduce violations globally, which need further corrections. So we traverse up the tree and do necessary transformations all the way up to the tree root. To recap: if we see a right-leaning `red` link, we rotate to the left. If we see two `red` links in a row, we rotate to the right. If both children are `red`, we do a color flip (i.e. split 4-node).

4b ⟨*Helper procedures* 2b⟩+≡
```
(define (fix-up node)
  (if (not (null? node))
      (begin
       (if (is-red? (node 'right)) (set! node (rotate-left node)))
       (if (and (is-red? (node 'left))
                (is-red? ((node 'left) 'left))) (set! node (rotate-right node)))
       (if (and (is-red? (node 'left))
                (is-red? (node 'right))) (color-flip node))))
  node)
```
Defines:
  `fix-up`, used in chunks 4, 5a, 7b, 4b, 8a, and 4b.
Uses `color-flip` 3c, `is-red?` 3b, `rotate-left` 3a, and `rotate-right` 2b.

Removing the minimum key. Now let us take a look at how to remove the minimum key from the tree. We will use the same approach that is used for 2–3 trees, namely we will merge nodes on the way down the tree, so that current node is not a 2-node (for more on 2–3 trees see [7, 8]). If we maintain this invariant, once we reached the bottom, we can simply remove the key. When going down the left spine of the tree, we go to the left link each time, by calling `remove-min-from` recursively, however, before going to the left, we ensure that the invariant holds, i.e. we merge keys from parent nodes to their 2-node children either by taking a key from current 3-node to the child, or by borrowing key from 3-node sibling, when necessary. Again, we may need to fix the tree, and the same `fix-up` will do. This procedure returns a pair that contains new root of the tree and the removed element.

5a ⟨*Remove minimum key procedure* 5a⟩≡

```
(define (remove-min root)
  (define min-elem '())
  (define (remove-min-from node)
    (if (null? (node 'left)) (begin (set! min-elem node) '())
        (begin
          ⟨Maintain invariant for remove-min 5c⟩
          ((node 'set-left) (remove-min-from (node 'left)))
          (fix-up node))))
  (let ((new-root (remove-min-from root)))
    (cons new-root min-elem)))
```

Defines:
`remove-min`, used in chunks 5 and 9c.
Uses `fix-up` 4b and `remove` 8a.

And this is how we will maintain the invariant: We examine two consecutive left links and if they are both `black`, we move red links to the left. But first, we will define a helper predicate, `is-black?`, so that we could avoid verbose `(not (is-red? ...))` expressions.

5b ⟨*Helper procedures* 2b⟩+≡

```
(define (is-black? n) (not (is-red? n)))
```

Defines:
`is-black?`, used in chunks 5 and 7c.
Uses `is-red?` 3b.

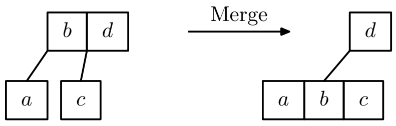
Now maintaining the invariant for `remove-min` is as follows.

5c ⟨*Maintain invariant for* `remove-min` 5c⟩≡

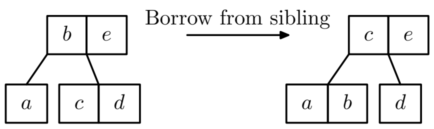
```
(if (and (is-black? (node 'left))
         (is-black? ((node 'left) 'left)))
    ⟨Move red link to the left 6b⟩)
```

Uses `is-black?` 5b.

Moving red link to the left. Let us consider the situation when current node is a 3-node and its first two children are not. In that case we simply merge it, producing a 4-node. Merging is accomplished by `color-flip`. This situation can be distinguished by looking at current node's left child `color`. It should be `red` and its right child's left link should be `black`:



The second case is more difficult, because b's right child is a 3-node, that is b's right child's left link *is* `red`. We rotate b's right child, a 3-node, to the right. Then we rotate b to the left, so that it is substituted with c now. Finally we `color-flip` to split the node. We get the following transformation:



Note that these transformations still preserve symmetric order of the tree.

Now the transformation procedure is as follows. We do `color-flip` to merge nodes. Then we check if b's right child is a 2-node by inspecting it's left link. If its `red`, then this is a 3-node and we do two rotations and flip colors as described above.

6a ⟨*Helper procedures* 2b⟩+≡
```
(define (move-red-left node)
  (color-flip node)
  (if (is-red? ((node 'right) 'left))
        ((node 'set-right (rotate-right (node 'right)))
         (set! node (rotate-left node))
         (color-flip node)))
  node)
```
Defines:
  `move-red-left`, used in chunks 6 and 9a.
Uses `color-flip` 3c, `is-red?` 3b, `rotate-left` 3a, and `rotate-right` 2b.

The procedure above, `move-red-left`, may return a different root after transformation. So we need to reset the node after we have applied the procedure. This completes `remove-min` procedure.

6b ⟨*Move red link to the left* 6b⟩≡
```
(set! node (move-red-left node))
```
Uses `move-red-left` 6a.

Moving red link to the right. We, again, maintain the invariant that current node is not a 2-node by merging and borrowing from sibling. Since the red links lean left `move-red-right` is even simpler than `move-red-left`.

7a ⟨*Helper procedures* 2b⟩+≡

```
(define (move-red-right node)
  (color-flip node)
  (if (is-red? ((node 'left) 'left))
      ((set! node (rotate-right node))
       (color-flip node)))
  node)
```

Defines:
`move-red-right`, used in chunks 7 and 9b.
Uses `color-flip` 3c, `is-red?` 3b, and `rotate-right` 2b.

Removing the maximum key. This procedure is very like `remove-min`, except that it goes down the right spine of a tree, and it rotates left-leaning red links to the right.

7b ⟨*Remove maximum key procedure* 7b⟩≡

```
(define (remove-max node)
  (define max-elem '())
  (define (remove-max-from node)
    (if (is-red? (node 'left)) (set! node (rotate-right node)))
    (if (null? (node 'right)) (begin (set! max-elem node) '())
        (begin
          ⟨Maintain invariant for remove-max 7c⟩
          ((node 'set-right) (remove-max-from (node 'right)))
          (fix-up node))))
  (let ((new-root (remove-max-from root)))
    (cons new-root max-elem)))
```

Defines:
`remove-max`, used in chunk 7b.
Uses `fix-up` 4b, `is-red?` 3b, `remove` 8a, and `rotate-right` 2b.

We can detect the situation when we need to move red link to the right by looking at the right link of the current node, and, as opposite to the situation analyzed in `remove-min` procedure, to the left link of the right child, since the tree leans left. If these are both `black`, we need to do the `move-red-right` transformation.

7c ⟨*Maintain invariant for* `remove-max` 7c⟩≡

```
(if (and (is-black? (node 'right))
         (is-black? ((node 'right) 'left)))
    (set! node (move-red-right node)))
```

Uses `is-black?` 5b and `move-red-right` 7a.

Removing arbitrary key. Removing an arbitrary key is easy once we understood `remove-max` and `remove-min` procedures. On searching we do simple BST search except that when we are moving to the right, we move `red` link to the right when necessary, and when we are moving to the left, we move `red` link to the left when necessary. We detect these necessities by examining two consecutive links: Two left links in a row when moving to the left, and right and right child's left links when moving to the right. We also save parent's appropriate setter in `transplant-proc` so that we can *transplant* the node. That is, if we remove internal node, we substitute it with its *successor*, i.e. minimum key of the right subtree, via `transplant-proc`. The `remove` procedure is quite large, so we will break it into a few modules that will be described separately. Of course, we use `fix-up` to fix the tree after we have performed removal. Argument `compare` is the same procedure that we used in `insert` procedure.

8a ⟨*Remove procedure* 8a⟩≡
```
(define (remove root key compare)
  ⟨Internal remove procedure definitions 8b⟩
  ⟨Internal remove procedure helper methods 8c⟩
  ⟨Definition of go-to-left-subtree 9a⟩
  ⟨Definition of go-to-right-subtree 9b⟩
  (define (remove-node node)
      (fix-up
       (if (< (compare key (node 'key)) 0)
           (go-to-left-subtree node)
           (go-to-right-subtree node))))
  (let ((new-root (remove-node root)))
    (cons new-root removed-node)))
```
Defines:
`remove`, used in chunks 8a, 5a, 8, and 7–9.
Uses `fix-up` 4b.

We will need a couple of variables. One that will be used to reference removed node, and the second to point to the procedure that will set appropriate child links in the parent of a node.

8b ⟨*Internal remove procedure definitions* 8b⟩≡
```
(define removed-node '())
(define transplant-proc '())
```

We better define a local helper procedure for tree traversal. This procedure will do two things: Set `transplant-proc` appropriately, and go the given link of a node. Argument `link` must be either `'right` or `'left`.

8c ⟨*Internal remove procedure helper methods* 8c⟩≡
```
(define (proceed-with node link)
  (set! transplant-proc (node link))
  ((node (if (eq? link 'left) 'set-left 'set-right)) (remove-node (node link))) node)
```
Uses `remove` 8a.

Now we will define two separate procedures for left and right subtrees of a node. When we go to the left subtree, we look for two left `black` links in a row, if there are any, we do `move-red-left`, in the same way as in `remove-min` procedure.

9a ⟨*Definition of* `go-to-left-subtree` 9a⟩≡

```
(define (go-to-left-subtree node)
  (if (and (not (is-red? (node 'left)))
           (not (is-red? ((node 'left) 'left))))
      (set! node (move-red-left node)))
  (proceed-with node 'left))
```

Uses `is-red?` 3b and `move-red-left` 6a.

If we go to the right subtree of a tree though, we first check, if we are at a 3-node and whether it leans left. If it is a 3-node and it does lean left, we need to rotate it to the right. Next, we check for two `red`s links in a row. However, now it's not a straight line situation as in `go-to-left-subtree`, but is a "zig-zag", because our `red` links lean left. So we check `color` of right and right child left links. If they are both `black`, we do `move-red-right` to ensure that our next node won't be a 2-node. But there's more: We compare `key`s again. If they match, we remove the node by substituting it with the successor node, that is the minimum node of current node's right subtree. Otherwise we search further, i.e. go the right subtree.

9b ⟨*Definition of* `go-to-right-subtree` 9b⟩≡

```
(define (go-to-right-subtree node)
  (if (is-red? (node 'left)) (set! node (rotate-right node)))
  (if (and (= (compare key (node 'key)) 0) (null? (node 'right))) '()
      (begin
        (if (and (not (is-red? (node 'right)))
                 (not (is-red? ((node 'right) 'left))))
            (set! node (move-red-right node)))
        (if (= (compare key (node 'key)) 0)
            ⟨Substitute with the successor node 9c⟩
            (proceed-with node 'right))
        node)))
```

Uses `is-red?` 3b, `move-red-right` 7a, and `rotate-right` 2b.

We use `remove-min` procedure to get the successor node. It will be removed from the right subtree of the current node. Then we use `transplant-proc` to correct parent's appropriate link so that it points to the successor. We also set `removed-node` to the removed node. Then we return, and `fix-up` fixes the tree. This completes the removal procedure.

9c ⟨*Substitute with the successor node* 9c⟩≡

```
(let* ((minimum (remove-min (node 'right)))
       (sub-tree (car minimum))
       (min-node (cdr minimum)))
 (begin
   ((min-node 'set-left) (node 'left))
   ((min-node 'set-right) sub-tree)
   ((min-node 'set-color) (node 'color))
   (if (procedure? transplant-proc) (transplant-proc min-node))
   (set! removed-node node)
   (set! node min-node)))
```

Uses `remove` 8a and `remove-min` 5a.

Tree drawing. Drawing a binary tree is almost trivial, but I have decided to include this little informal description of the algorithm that I came up with for drawing. I didn't need putting `key` values in the nodes, I just wanted to see the overall structure of a tree. The idea is to generate auxiliary data for METAPOST using which it will be easy to write a loop that draws a tree.

Of course, I could have used a graph visualizations tool like graphviz, but in this particular case I didn't need that heavy machinery. Besides, I didn't like results it produced. The algorithm that I came up with is easy, it's just a few lines of code, and I'm more than happy with the results I got.

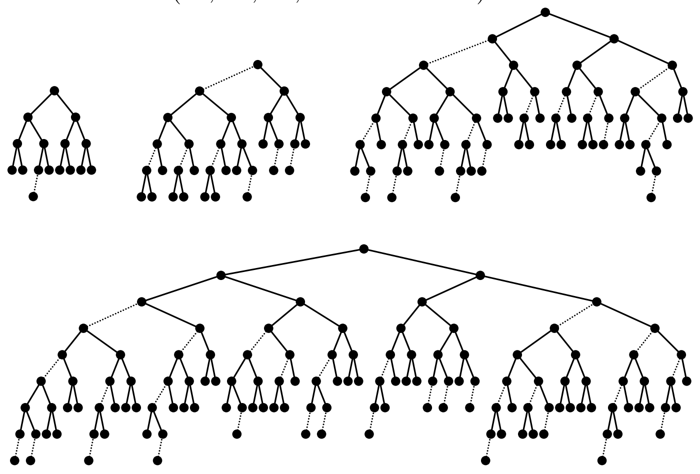
We initialize three numeric arrays of N sizes, where N is the number of nodes in a tree. One of the arrays, d, holds *depth* values for each node. In other words, d maps a sequence of nodes $n_0, n_1, \ldots, n_N$, $0 \le k \le N-1$, gotten from traversing the tree in a symmetrical order, to the depths of those nodes. (Depth of the root node is 0, depths of root's children is 1 etc.) Let d_k be the depth of the kth node n_k in that sequence. Using this data we can easily determine location (x, y) of a node n_k. To determine the x component we simply multiply k by the "width" parameter, w, since k means that there are k nodes to the left (all nodes, not just subtree nodes). The parameter w determines how far nodes are placed from each other. The larger this value the more the tree "breathes". Generally, it should be slightly larger than the width of a node. The y component can be computed easily as well: It's d_k times parameter h. The parameter h determines how far tree levels are from each other. The larger this value the higher the tree. Initialization of array d is trivial.

Location of any node n_k, $0 \le k \le N-1$, is $(k \cdot w, d_k \cdot h)$, if we have d initialized as described above. But we also need to draw edges. In order to do that we need to know location of node's parent. We introduce another auxiliary array p. It maps node n_k to its parent node. So we can calculate location of a parent node as $(p_k \cdot w, d_{p_k} \cdot h)_{p_k}$; and we can draw an edge as a line between two points $(k \cdot w, d_k \cdot h)$ and $(p_k \cdot w, d_{p_k} \cdot h)$.

We can initialize p at the same time when traversing the tree and initializing array d. Calculating p_k for the given n_k depends on whether n_k is left or right child of n_{p_k}. If n_k is the left child, then we set $p_k \leftarrow k + (c_r + 1)$, where c_r is the number of nodes in the right subtree of n_k. If, however, n_k is the right child, then $p_k \leftarrow k - (c_l + 1)$, where c_l is the number of nodes in the left subtree of node n_k.

Finally, the array c holds colors of the nodes, so that we can draw edges differently, depending on node's color. If n_k is red then $c_k = 1$, and $c_k = 0$ if it's not.

A few random LLRB trees (16, 32, 64, and 128 nodes):



References

[1] Robert Sedgewick, Left-leaning Red-Black Trees, 2008

[2] Robert Sedgewick, Algorithms, Addison-Wesley, 1983

[3] Leo J. Guibas, Robert Sedgewick, A dichromatic framework for balanced trees, Foundations of Computer Science, 1978, pp 8–21

[4] Donald E. Knuth, The Art of Computer Programming vol. 3, 2008

[5] Donald E. Knuth, Literate Programming, The Computer Journal, 1984, pp 97–111

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, 2009

[7] Alfred V. Aho, J.E. Hopcroft, Jeffrey D. Ullman, Data Structures and Algorithms, Addison-Wesley Series in Computer Science and Information Processing, 1983

[8] Alfred V. Aho, J.E. Hopcroft, Jeffrey D. Ullman, The Design And Analysis of Computer Algorithms, Addison-Wesley, 1974

[9] Adelson-Velskii, G., E. M. Landis, An algorithm for the organization of information. Proceedings of the USSR Academy of Sciences 146, 1962, pp 263-266

[10] Harold Abelson, Gerald Jay Sussman, Julie Sussman, Structure and Interpretation of Computer Programs, 1996

[11] John D. Hobby, METAPOST, a user's manual, 2010

Definitions

⟨*Definition of* `go-to-left-subtree` 9a⟩ 8a, 9a
⟨*Definition of* `go-to-right-subtree` 9b⟩ 8a, 9b
⟨*Getters* 1c⟩ 1b, 1c
⟨*Helper procedures* 2b⟩ 1a, 2b, 3a, 3b, 3c, 4b, 5b, 6a, 7a
⟨*Insertion procedure* 4a⟩ 1a, 4a
⟨*Internal remove procedure definitions* 8b⟩ 8a, 8b
⟨*Internal remove procedure helper methods* 8c⟩ 8a, 8c
⟨*Left-leaning Red-Black Tree implementation* 1a⟩ 1a
⟨*Maintain invariant for* `remove-max` 7c⟩ 7b, 7c
⟨*Maintain invariant for* `remove-min` 5c⟩ 5a, 5c
⟨*Move red link to the left* 6b⟩ 5c, 6b
⟨*Remove maximum key procedure* 7b⟩ 1a, 7b
⟨*Remove minimum key procedure* 5a⟩ 1a, 5a
⟨*Remove procedure* 8a⟩ 1a, 8a
⟨*Setters* 2a⟩ 1b, 2a
⟨*Substitute with the successor node* 9c⟩ 9b, 9c
⟨*Tree node definition* 1b⟩ 1a, 1b

Index

`color-flip`: 3c, 4b, 3c, 3c, 3c, 6a, 7a
`fix-up`: 4b, 4b, 4a, 4b, 4b, 5a, 7b, 4b, 8a, 4b
`insert`: 4a, 4a, 4a
`is-black?`: 5b, 5b, 5c, 7c
`is-red?`: 3b, 3b, 3c, 4b, 3b, 5b, 6a, 7a, 7b, 9a, 9b
`make-node`: 1b, 4a
`move-red-left`: 6a, 6a, 6b, 6a, 6a, 9a
`move-red-right`: 7a, 7a, 7a, 7c, 7a, 9b
`remove`: 8a, 5a, 8a, 8a, 8a, 7b, 8a, 8a, 8a, 8a, 8a, 8c, 8a, 8a, 9c
`remove-max`: 7b, 7b
`remove-min`: 5a, 5a, 5a, 5a, 5a, 5a, 5a, 5a, 5a, 5a, 9c
`rotate-left`: 3a, 3a, 4b, 6a
`rotate-right`: 2b, 2b, 4b, 6a, 7a, 7b, 9b