Concurrent Red-Black Trees

Franck van Breugel
Department of Computer Science and Engineering, York University
4700 Keele Street, Toronto, Ontario, Canada, M3J 1P3

January 1, 2010

Abstract

Three concurrent implementations of red-black trees are presented. Only the operations Contains and Add are considered. The first implementation exploits monitors. The second implementation is based on a solution of the readers-writers problem, where the readers are threads that perform the Contains operation and the writers are the threads that perform the Add operation. The third implementation is an adaptation of the concurrent implementation of AVL trees by Ellis to the setting of red-black trees. In this implementation, the threads lock nodes of the tree. A node can be locked in three different ways and different threads can have a lock on a node simultaneously.

1 Introduction

Data structures such as sets can be efficiently implemented by means of red-black trees. For example, the class TreeSet of the package java.util of the Java standard library has been implemented by means of a red-black tree. A red-black tree is a special type of binary search tree. Such a tree is approximately balanced by colouring the nodes of the tree and placing certain restrictions on the way the nodes can be coloured.

With the arrival of multicore machines, there is a need for concurrent implementations of fundamental data structures such as sets. In this paper, we present a sequential implementation and three different concurrent implementations of red-black trees. We first present the sequential implementation as can be found in [2]. The first concurrent implementation is a simple modification of the sequential implementation by representing the red-black tree as a monitor. The second concurrent implementation allows for more concurrency by modifying a solution to the readers-writers problem [3]. The third and final concurrent implementation uses fine grain locking to allow even more concurrency. In this implementation, we adapt the approach proposed by Ellis [4] for concurrent AVL trees to red-black trees.

2 Red-Black Trees

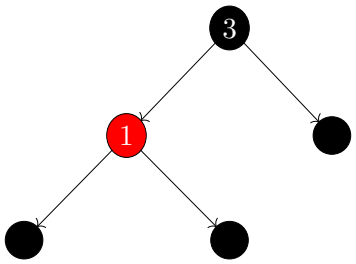
We assume that the reader is familiar with binary search trees (their definition can be found in, for example, [2, Section 13.1]). A red-black tree is a binary search tree where each node has a colour. A node is either coloured red or black. By restricting the way nodes can be coloured, the tree becomes approximately balanced. These trees were first introduced as symmetric binary B-trees by

Bayer [1]. Guibas and Sedgewick [5] characterized these trees by colouring the nodes red or black, leading to the following definition.

Definition 1 *A red-black tree is a binary search tree where each node is either coloured red or black and*

- *the root is black,*
- *each leaf is black,*
- *if a node is red, then both its children are black, and*
- *for every node, every path from that node to a leaf contains the same number of black nodes.*

For example, the binary search tree



is a red-black tree. Note that only the internal nodes contain elements. Red-black trees have the following key property.

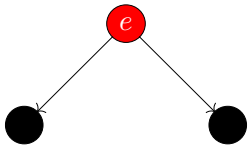
Theorem 2 *A red-black tree with n internal nodes has height at most $2\log_2(n+1)$.*

A proof of this result can be found in, for example, [2, Section 14.1]. Since a red-black tree is approximately balanced, the operations Contains and Add can be implemented efficiently. More precisely, both Contains and Add can be implemented in $O(\log_2(n))$, where n is the number of internal nodes of the red-black tree.

3 Sequential Implementation

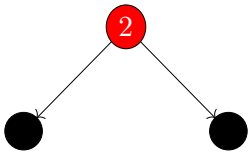
The Contains operation for red-black trees can be implemented in exactly the same way as for binary search trees. Its pseudocode can be found in Appendix A.

Also the Add operation for red-black trees is similar to that for binary search trees. If the element e, which is to be added, is not already part of the red-black tree, then the appropriate leaf is replaced with the tree

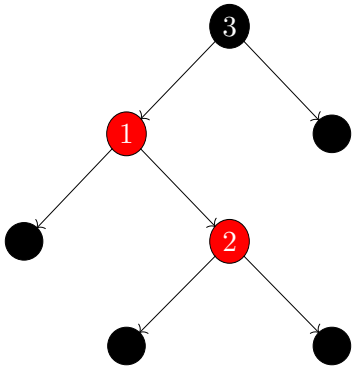


This modification does not violate condition 1, 2 and 4 of Definition 1, but it may violate condition 3. To reestablish this condition, the colour of some of the nodes may have to be changed and the structure of the tree may have to be modified. The details can be found in Appendix A. For

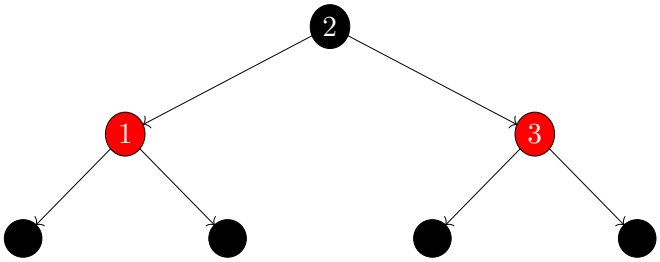
example, if we add the element 2 to the red-black tree depicted in Section 2, then we first replace the right child of the node containing the element 1 with the tree



obtaining the tree



Note that this is not a red-black tree, since the red node labelled 1 has a red child. After restructuring the tree, we obtain the following red-black tree.



We believe that multiple threads manipulating a red-black tree concurrently using the operations Contains and Add may lead to counter-intuitive results. Consider the following concurrent program.

```
1 Add(3)
2 Add(1)
3 (Add(2) || Contains(1))
```

Starting from an empty red-black tree, we first add the elements 3 and 1. This results in the red-black tree depicted in the previous section. Subsequently, one threads adds the element 2 whereas the other thread checks if the tree contains the element 1. One would expect the Contains operation to always return true. However, we believe that by interleaving the elementary operations of the operations Add and Contains in a particular way, the operation Contains may return false. When the Add operation modifies the structure of the tree, the Contains operation may not be able to find the element 1. We plan to confirm this conjecture by our implementation or our verification effort. In the following sections, we present three ways to rule out this undesirable behaviour.

4 The Monitors Approach

A simple way to ensure that the Add operation does not interfere with the Contains operation is to implement the red-black tree as a monitor. Monitors were introduced by Brinch Hansen and Hoare in [7, 8]. Below we use the syntax as used in [8].

```
1  RedBlackTree : monitor
2  begin
3    root : node
4    procedure contains (element : int, result contains : boolean)
5    begin
6      ...
7    end
8    procedure add (element : int, result added : boolean)
9    begin
10     ...
11   end
12   root := black node
13 end
```

Within the body of the procedures Contains and Add we place the code presented in Appendix A. Since monitor procedures are always mutually exclusive, the Add procedure never interferes with the Contains procedure.

5 The Readers-Writers Approach

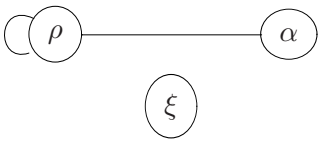
The solution presented in the previous section ensures that one operation at a time is performed on the red-black tree. However, multiple Contains operations can be performed concurrently without giving rise to undesirable results. To accomplish this, we can easily modify a solution to the readers-writers problem [3]. In our setting, the threads that perform the Contains operation are the readers and the threads that perform the Add operation are the writers. For a solution to the readers-writers problem in which no reader waits because a writer is waiting for other readers to finish, we refer the reader to [3].

6 Locking Nodes

In an attempt to increase concurrency even more, we adapt the approach proposed by Ellis [4] for concurrent AVL trees to red-black trees.

The key idea of this implementation is that individual nodes are locked. A node can be locked in three different ways. A thread that searches for an element (by performing the Contains operation) ρ-locks a node of the tree to ensure that the locked node is not part of a restructuring of the tree. A thread that searches for a leaf to add an element (as part of the Add operation) α-locks a node of the tree to prevent another thread, which also wants to add an element, access to the subtree rooted at the locked node. Just before a thread restructures the tree (as part of the Add operation), it ξ-locks the nodes that are part of the restructuring. This prevents other threads from accessing these nodes.

Different threads can hold a lock on the same node at the same time. However, there are some restrictions. The following graph [4] captures those restrictions.



If there is an edge between two lock types, then two threads can have a lock of the given type on a particular node at the same time. For example, multiple threads can ρ-lock a node and a single thread can α-lock that node all at the same time.

6.1 The Contains Operation

While searching for an element in the tree, we ρ-lock nodes on the path from the root of the tree to either a leaf (if the element is not stored in the tree) or the node containing the element. We start to ρ-lock the root of the red-black tree. Assume that, during the search, we have ρ-locked a particular node. Before releasing the lock, we first ρ-lock the left or right child of that node. This avoid should deadlock. We will try to confirm this with our implementation or verification effort. The details of the implementation of the Contains operation can be found in Section B.1.

6.2 The Add Operation

The Add operation consists of two parts. The first part is very similar to the Contains operation. In the first part, we locate the leaf where the element is to be inserted. While searching for that leaf, we α-lock nodes on the path from the root to the leaf. An α-lock on a node prevents other threads, which also want to add elements to the tree, access to the subtree rooted at that node, so that this subtree can be modified. We want to keep this subtree as small as possible to allow as much concurrency as possible. Initially, we α-lock the root. If we encounter two consecutive black nodes on the path from the root to the leaf, we know that the potential restructuring will be limited to the subtree rooted at first black node (the one closest to the root). Hence, we α-lock this node and we release the lock on the previously α-locked node. In this way, the subtree rooted at the α-locked node becomes smaller, therefore, allowing more concurrency.

After we have inserted the element at a leaf of the tree, we may have to modify the structure of the tree and change the colour of some of the nodes. These changes will be limited to the subtree rooted at the α-locked node. Whenever, we change the structure of the tree, we ξ-lock all the nodes involved in the restructuring. We lock them in a top-down fashion to avoid deadlock.

The details can be found in Section B.2.

7 Conclusion

A lot of work has been done on the concurrent implementation of data structures. We refer the reader to, for example, [9] for an overview. The concurrent red-black tree implementation described in Section 6 is an adaptation of the concurrent implementation of AVL trees as introduced by Ellis in [4]. Hanke [6] also mentions that the implementation of Ellis can be adapted to red-black trees. Nurmi and Soisalon-Soininen [10] present a slightly different concurrent implementation of red-black trees. Also their work is based on the original work of Ellis.

We have presented three concurrent implementations of red-black trees. The implementation in Section 5 allows for more concurrency than the one in Section 4. The implementation in Section 6 gives rise to even more concurrency. However, as the concurrency increases, so does the complexity of the implementation.

There seem to be opportunities to increase the amount of concurrency of the implementation in Section 6. First of all, rather than locking nodes, we could lock only "half a node." For example, instead of locking a node, we can lock only its left part. In this way, its right child is still available. Secondly, there seem opportunities to decrease the lock granularity. For example, line 73–87 of Section B.2 can be modified as follows.

```
ξ−lock grandparent
ξ−lock parent
ξ−lock node
Left−Child(node, parent)
Right−Child(node, grandparent)
root ← node
ξ−unlock node
Left−Child(grandparent, right)
ξ−unlock grandparent
Right−Child(parent, left)
ξ−unlock parent
```

Note that left and right are not locked at all. Also notice that grandparent and node are locked for a "smaller amount of time." Thirdly, we may attempt to avoid using locks completely by using atomic operations such as "compare and swap."

References

[1] Rudolf Bayer. Symmetric binary B-trees: data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, USA, 1990.

[3] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with "reader" and "writers". *Communications of the ACM*, 14(10):667–668, October 1971.

[4] Carla Schlatter Ellis. Concurrent search and insertion in AVL trees. *IEEE Transactions on Computers*, 29(9):811–817, September 1980.

[5] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, Ann Arbor, MI, USA, October 1978. IEEE Computer Society.

[6] Sabine Hanke. The performance of concurrent red-black tree algorithms. In Jeffrey Scott Vitter and Christos D. Zaroliagis, editors, *Proceedings of the 3rd International Workshop on Algorithm Engineering*, volume 1668 of *Lecture Notes in Computer Science*, pages 286–300, London, UK, July 1999. Springer-Verlag.

[7] Per Brinch Hansen. *Operating System Principles*. Prentice Hall, Englewood Cliffs, NJ, USA, 1973.

[8] C.A.R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[9] Mark Moir and Nir Shavit. Concurrent data structures. In Dinesh P. Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*, chapter 47, pages 47–1–47–23. Chapman & Hall/CRC Press, Boca Raton, FL, USA, 2005.

[10] Otto Nurmi and Eljas Soisalon-Soininen. Chromatic binary search trees: a structure for concurrent rebalancing. *Acta Informatica*, 31(6):547–557, September 1996.

A Pseudocode for the Sequential Implementation

A.1 Pseudocode for the Contains Operation

The following pseudocode is based on the pseudocode found in [2, Section 13.2].

```
1  Contains(e)
2    found ← false
3    node ← root
4    while node is not a leaf ∧ ¬ found do
5      if e = element of node then
6        found ← true
7      else if e < element of node then
8        node ← left child of node
9      else
10       node ← right child of node
11   return found
```

A.2 Pseudocode for the Add Operation

The following pseudocode is based on the pseudocode found in [2, Section 14.3]. Before presenting the pseudocode for Add, we first present some simple operations that will be used in the pseudocode for Add.

The operation Left−Child(p, c) ensures that the node c becomes the left child of the node p.

```
1  Left−Child(p, c)
2    parent of c ← p
3    left child of p ← c
```

Similarly, the operation Right−Child(p, c) ensures that the node c becomes the right child of the node p.

```
1  Right−Child(p, c)
2    parent of c ← p
3    right child of p ← c
```

```
1  Add(e)
2    found ← false
3    node ← root
4    while node is not a leaf ∧ ¬ found do
5      if e = element of node then
6        found ← true
7      else if e < element of node then
8        node ← left child of node
9      else
10       node ← right child of node
11   if ¬ found then
12     colour of node ← red
13     element of node ← e
14     left ← black node
15     right ← black node
16     Left−Child(node, left)
17     Right−Child(node, right)
18     while node ≠ root ∧ parent of node is red do
19       parent ← parent of node
20       grandparent ← parent of parent
21       if parent is left child of grandparent then
22         aunt ← right child of grandparent
23         if aunt is red then
24           colour of aunt ← black
25           colour of parent ← black
26           colour of grandparent ← red
27           node ← grandparent
28         else if node is left child of parent then
29           colour of parent ← black
30           colour of grandparent ← red
31           sister ← right child of parent
32           Right−Child(parent, grandparent)
33           Left−Child(grandparent, sister)
34           if grandparent = root then
35             root ← parent
36           else
37             grandgrandparent ← parent of grandparent
38             if grandparent is a left child of grandgrandparent then
39               Left−Child(grandgrandparent, parent)
40             else
41               Right−Child(grandgrandparent, parent)
42         else (node is right child of parent)
43           colour of node ← black
44           colour of grandparent ← red
45           left ← left child of node
```

```
46            right ← right child of node
47            Left-Child(node, parent)
48            Right-Child(node, grandparent)
49            Right-Child(parent, left)
50            Left-Child(grandparent, right)
51            if grandparent = root then
52              root ← node
53            else
54              grandgrandparent ← parent of grandparent
55              if grandparent is a left child of grandgrandparent then
56                Left-Child(grandgrandparent, node)
57              else
58                Right-Child(grandgrandparent, node)
59          else (parent is right child of grandparent)
60            aunt ← left child of grandparent
61            if aunt is red then
62              colour of aunt ← black
63              colour of parent ← black
64              colour of grandparent ← red
65              node ← grandparent
66            else if node is right child of parent then
67              colour of parent ← black
68              colour of grandparent ← red
69              sister ← left child of parent
70              Left-Child(parent, grandparent)
71              Right-Child(grandparent, sister)
72              if grandparent = root then
73                root ← parent
74              else
75                grandgrandparent ← parent of grandparent
76                if grandparent is a left child of grandgrandparent then
77                  Left-Child(grandgrandparent, parent)
78                else
79                  Right-Child(grandgrandparent, parent)
80            else (node is left child of parent)
81              colour of node ← black
82              colour of grandparent ← red
83              left ← left child of node
84              right ← right child of node
85              Right-Child(node, parent)
86              Left-Child(node, grandparent)
87              Left-Child(parent, right)
88              Right-Child(grandparent, left)
89              if grandparent = root then
90                root ← node
```

```
91            else
92              grandgrandparent ← parent of grandparent
93              if grandparent is a left child of grandgrandparent then
94                Left−Child(grandgrandparent, node)
95              else
96                Right−Child(grandgrandparent, node)
97    colour of root ← black
98    return ¬ found
```

Note that line 59–96 is the mirror image of line 21–58.

B Pseudocode for the Concurrent Implementation

We augment the pseudocode of Appendix A with the locking of nodes.

B.1 Pseudocode for the Contains Operation

We modify the implementation of the Contains operation as follows.

```
1  Contains(e)
2    found ← false
3    node ← root
4    ρ−lock node
5    while node is not a leaf ∧ ¬ found do
6      parent ← node
7      if e = element of node then
8        found ← true
9      else if e < element of node then
10       node ← left child of node
11     else
12       node ← right child of node
13     ρ−lock node
14     ρ−unlock parent
15   ρ−unlock node
16   return found
```

Note that line 4, 6, 13, 14 and 15 are new.

B.2 Pseudocode for the Add Operation

We modify the implementation of the Add operation as follows.

```
1  Add(e)
2    found ← false
3    node ← root
4    α−lock node
5    locked ← node
6    while node is not a leaf ∧ ¬ found do
```

```
7       parent ← node
8       if e = element of node then
9         found ← true
10      else if e < element of node then
11        node ← left child of node
12      else
13        node ← right child of node
14      if node and parent are black and parent ≠ locked then
15        α–lock parent
16        α–unlock locked
17        locked ← parent
18    if ¬ found then
19      ξ–lock node
20      colour of node ← red
21      element of node ← e
22      left ← black node
23      right ← black node
24      Left–Child(node, left)
25      Right–Child(node, right)
26      ξ–unlock node
27      while node ≠ root ∧ parent of node is red do
28        parent ← parent of node
29        grandparent ← parent of parent
30        if parent is left child of grandparent then
31          aunt ← right child of grandparent
32          if aunt is red then
33            colour of aunt ← black
34            colour of parent ← black
35            colour of grandparent ← red
36            node ← grandparent
37          else if node is left child of parent then
38            colour of parent ← black
39            colour of grandparent ← red
40            sister ← right child of parent
41            if grandparent = root then
42              ξ–lock grandparent
43              ξ–lock parent
44              ξ–lock sister
45              root ← parent
46              Right–Child(parent, grandparent)
47              Left–Child(grandparent, sister)
48              ξ–unlock sister
49              ξ–unlock parent
50              ξ–unlock grandparent
51            else
```

```
52              grandgrandparent ← parent of grandparent
53              ξ–lock grandgrandparent
54              ξ–lock grandparent
55              ξ–lock parent
56              ξ–lock sister
57              if grandparent is a left child of grandgrandparent then
58                Left–Child(grandgrandparent, parent)
59              else
60                Right–Child(grandgrandparent, parent)
61              Right–Child(parent, grandparent)
62              Left–Child(grandparent, sister)
63              ξ–unlock sister
64              ξ–unlock parent
65              ξ–unlock grandparent
66              ξ–unlock grandgrandparent
67            else (node is right child of parent)
68              colour of node ← black
69              colour of grandparent ← red
70              left ← left child of node
71              right ← right child of node
72              if grandparent = root then
73                ξ–lock grandparent
74                ξ–lock parent
75                ξ–lock node
76                ξ–lock left
77                ξ–lock right
78                root ← node
79                Left–Child(node, parent)
80                Right–Child(node, grandparent)
81                Right–Child(parent, left)
82                Left–Child(grandparent, right)
83                ξ–unlock right
84                ξ–unlock left
85                ξ–unlock node
86                ξ–unlock parent
87                ξ–unlock grandparent
88              else
89                grandgrandparent ← parent of grandparent
90                ξ–lock grandgrandparent
91                ξ–lock grandparent
92                ξ–lock parent
93                ξ–lock node
94                ξ–lock left
95                ξ–lock right
96                if grandparent is a left child of grandgrandparent then
```

```
97              Left-Child(grandgrandparent, node)
98            else
99              Right-Child(grandgrandparent, node)
100           Left-Child(node, parent)
101           Right-Child(node, grandparent)
102           Right-Child(parent, left)
103           Left-Child(grandparent, right)
104           ξ-unlock right
105           ξ-unlock left
106           ξ-unlock node
107           ξ-unlock parent
108           ξ-unlock grandparent
109       else (parent is right child of grandparent)
110         aunt ← left child of grandparent
111         if aunt is red then
112           colour of aunt ← black
113           colour of parent ← black
114           colour of grandparent ← red
115           node ← grandparent
116         else if node is right child of parent then
117           colour of parent ← black
118           colour of grandparent ← red
119           sister ← left child of parent
120           if grandparent = root then
121             ξ-lock grandparent
122             ξ-lock parent
123             ξ-lock sister
124             root ← parent
125             Left-Child(parent, grandparent)
126             Right-Child(grandparent, sister)
127             ξ-unlock sister
128             ξ-unlock parent
129             ξ-unlock grandparent
130           else
131             grandgrandparent ← parent of grandparent
132             ξ-lock grandgrandparent
133             ξ-lock grandparent
134             ξ-lock parent
135             ξ-lock sister
136             if grandparent is a left child of grandgrandparent then
137               Left-Child(grandgrandparent, parent)
138             else
139               Right-Child(grandgrandparent, parent)
140             Left-Child(parent, grandparent)
141             Right-Child(grandparent, sister)
```

```
142             ξ−unlock sister
143             ξ−unlock parent
144             ξ−unlock grandparent
145             ξ−unlock grandgrandparent
146           else (node is left child of parent)
147             colour of node ← black
148             colour of grandparent ← red
149             left ← left child of node
150             right ← right child of node
151             if grandparent = root then
152               ξ−lock grandparent
153               ξ−lock parent
154               ξ−lock node
155               ξ−lock left
156               ξ−lock right
157               root ← node
158               Right−Child(node, parent)
159               Left−Child(node, grandparent)
160               Left−Child(parent, right)
161               Right−Child(grandparent, left)
162               ξ−unlock right
163               ξ−unlock left
164               ξ−unlock node
165               ξ−unlock parent
166               ξ−unlock grandparent
167             else
168               grandgrandparent ← parent of grandparent
169               ξ−lock grandgrandparent
170               ξ−lock grandparent
171               ξ−lock parent
172               ξ−lock node
173               ξ−lock left
174               ξ−lock right
175               if grandparent is a left child of grandgrandparent then
176                 Left−Child(grandgrandparent, node)
177               else
178                 Right−Child(grandgrandparent, node)
179               Right−Child(node, parent)
180               Left−Child(node, grandparent)
181               Left−Child(parent, right)
182               Right−Child(grandparent, left)
183               ξ−unlock right
184               ξ−unlock left
185               ξ−unlock node
186               ξ−unlock parent
```

```
187             ξ−unlock grandparent
188             ξ−unlock grandgrandparent
189     colour of root ← black
190     α−unlocked locked
191     return ¬ found
```