Concurrent Red-Black Trees

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

January 23, 2015

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 class 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 [3]. 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 [4]. 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 [6] for concurrent AVL trees to red-black trees.

2 Red-Black Trees

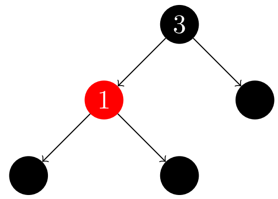
We assume that the reader is familiar with binary search trees (their definition can be found in, for example, [3, Section 13.1]). A red-black tree is a binary search tree where each node has a colour. A node is either coloured red or black. By restricting the way nodes can be coloured, the tree becomes approximately balanced. These trees were first introduced as symmetric binary B-trees by Bayer [1]. Guibas and Sedgewick [7] 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



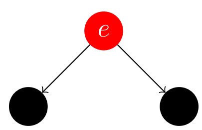
is a red-black tree. Note that only the internal nodes contain elements. Red-black trees have the following key property.

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

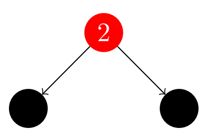
A proof of this result can be found in, for example, [3, Section 14.1]. Since a red-black tree is approximately balanced, the operations Contains and Add can be implemented efficiently. More precisely, both Contains and Add can be implemented in $O(\log_2(n))$, where n is the number of internal nodes of the red-black tree.

3 Sequential Implementation

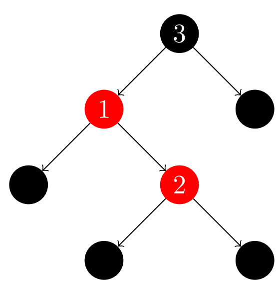
The Contains operation for red-black trees can be implemented in exactly the same way as for binary search trees. Its pseudocode can be found in Appendix A. Also the Add operation for red-black trees is similar to that for binary search trees. If the element e, which is to be added, is not already part of the red-black tree, then the appropriate leaf is replaced with the tree



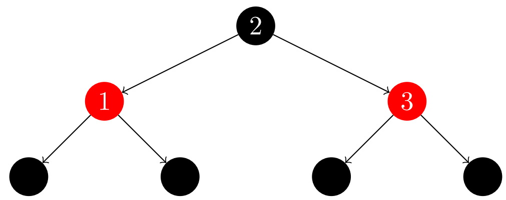
This modification does not violate condition 1, 2 and 4 of Definition 1, but it may violate condition 3. To reestablish this condition, the colour of some of the nodes may have to be changed and the structure of the tree may have to be modified. The details can be found in Appendix A. For example, if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

```
Add(3)
Add(1)
(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. 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 [9, 10]. Below we use the syntax as used in [10].

```
RedBlackTree : monitor
begin
  root : node
  procedure contains (element : int, result contains : boolean)
  begin
    ...
  end
  procedure add (element : int, result added : boolean)
  begin
    ...
  end
  root := black node
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.

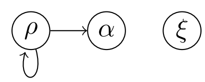
5 The Readers-Writers Approach

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 [4]. 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 [4].

6 Locking Nodes

In an attempt to increase concurrency even more, we adapt the approach proposed by Ellis [6] 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 [6] 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 appropriate child of that node. This lock coupling should avoid deadlock. We will try to confirm this with our implementation. 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 the 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, [11] 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 [6]. Hanke [8] also mentions that the implementation of Ellis can be adapted to red-black trees. Nurmi and Soisalon-Soininen [12] present a slightly different concurrent implementation of red-black trees. Also their work is based on the original work of Ellis. Although the work of Ellis is more than thirty years old, the quest for efficient concurrent implementations of balanced binary search trees is still ongoing (see, for example, [2, 5]).

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] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In R. Govindarajan, David A. Padua, and Mary W. Hall, editors, *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 257–268, Bangalore, India, January 2010. ACM.

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

[4] 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.

[5] Dana Drachsler, Martin Vechev, and Eran Yahav. Practical concurrent binary search trees via logical ordering. In José Moreira and James R. Larus, editors, *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 343–356, Orlando, FL, USA, February 2014. ACM.

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

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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 [3, Section 13.2].

```
Contains(e)
  found ← false
  node ← root
  while node is not a leaf ∧ ¬ found do
    if e = element of node then
      found ← true
    else if e < element of node then
      node ← left child of node
    else
      node ← right child of node
  return found
```

A.2 Pseudocode for the Add Operation

The following pseudocode is based on the pseudocode found in [3, 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.

```
Left−Child(p, c)
  parent of c ← p
  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.

```
Right-Child(p, c)
  parent of c ← p
  right child of p ← c
```

```
Add(e)
  found ← false
  node ← root
  while node is not a leaf ∧ ¬ found do
    if e = element of node then
      found ← true
    else if e < element of node then
      node ← left child of node
    else
      node ← right child of node
  if ¬ found then
    colour of node ← red
    element of node ← e
    left ← black node
    right ← black node
    Left-Child(node, left)
    Right-Child(node, right)
    while node ≠ root ∧ parent of node is red do
      parent ← parent of node
      grandparent ← parent of parent
      if parent is left child of grandparent then
        aunt ← right child of grandparent
        if aunt is red then
          colour of aunt ← black
          colour of parent ← black
          colour of grandparent ← red
          node ← grandparent
        else if node is left child of parent then
          colour of parent ← black
          colour of grandparent ← red
          sister ← right child of parent
          Right-Child(parent, grandparent)
          Left-Child(grandparent, sister)
          if grandparent = root then
            root ← parent
          else
            grandgrandparent ← parent of grandparent
            if grandparent is a left child of grandgrandparent then
              Left-Child(grandgrandparent, parent)
            else
              Right-Child(grandgrandparent, parent)
        else (node is right child of parent)
          colour of node ← black
          colour of grandparent ← red
          left ← left child of node
          right ← right child of node
          Left-Child(node, parent)
          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.

```
Add(e)
  found ← false
  node ← root
  α–lock node
  locked ← node
  while node is not a leaf ∧ ¬ found do
    parent ← node
    if e = element of node then
      found ← true
    else if e < element of node then
      node ← left child of node
    else
      node ← right child of node
    if node and parent are black and parent ≠ locked then
      α–lock parent
      α–unlock locked
      locked ← parent
  if ¬ found then
    ξ–lock node
    colour of node ← red
    element of node ← e
    left ← black node
    right ← black node
```

```
    Left-Child(node, left)
    Right-Child(node, right)
    ξ-unlock node
    while node ≠ root ∧ parent of node is red do
      parent ← parent of node
      grandparent ← parent of parent
      if parent is left child of grandparent then
        aunt ← right child of grandparent
        if aunt is red then
          colour of aunt ← black
          colour of parent ← black
          colour of grandparent ← red
          node ← grandparent
        else if node is left child of parent then
          colour of parent ← black
          colour of grandparent ← red
          sister ← right child of parent
          if grandparent = root then
            ξ-lock grandparent
            ξ-lock parent
            ξ-lock sister
            root ← parent
            Right-Child(parent, grandparent)
            Left-Child(grandparent, sister)
            ξ-unlock sister
            ξ-unlock parent
            ξ-unlock grandparent
          else
            grandgrandparent ← parent of grandparent
            ξ-lock grandgrandparent
            ξ-lock grandparent
            ξ-lock parent
            ξ-lock sister
            if grandparent is a left child of grandgrandparent then
              Left-Child(grandgrandparent, parent)
            else
              Right-Child(grandgrandparent, parent)
            Right-Child(parent, grandparent)
            Left-Child(grandparent, sister)
            ξ-unlock sister
            ξ-unlock parent
            ξ-unlock grandparent
            ξ-unlock grandgrandparent
        else (node is right child of parent)
          colour of node ← black
          colour of grandparent ← red
          left ← left child of node
          right ← right child of node
          if grandparent = root then
            ξ-lock grandparent
            ξ-lock parent
```

```
          ξ-lock node
          ξ-lock left
          ξ-lock right
          root ← node
          Left-Child(node, parent)
          Right-Child(node, grandparent)
          Right-Child(parent, left)
          Left-Child(grandparent, right)
          ξ-unlock right
          ξ-unlock left
          ξ-unlock node
          ξ-unlock parent
          ξ-unlock grandparent
        else
          grandgrandparent ← parent of grandparent
          ξ-lock grandgrandparent
          ξ-lock grandparent
          ξ-lock parent
          ξ-lock node
          ξ-lock left
          ξ-lock right
          if grandparent is a left child of grandgrandparent then
            Left-Child(grandgrandparent, node)
          else
            Right-Child(grandgrandparent, node)
          Left-Child(node, parent)
          Right-Child(node, grandparent)
          Right-Child(parent, left)
          Left-Child(grandparent, right)
          ξ-unlock right
          ξ-unlock left
          ξ-unlock node
          ξ-unlock parent
          ξ-unlock grandparent
    else (parent is right child of grandparent)
      aunt ← left child of grandparent
      if aunt is red then
        colour of aunt ← black
        colour of parent ← black
        colour of grandparent ← red
        node ← grandparent
      else if node is right child of parent then
        colour of parent ← black
        colour of grandparent ← red
        sister ← left child of parent
        if grandparent = root then
          ξ-lock grandparent
          ξ-lock parent
          ξ-lock sister
          root ← parent
          Left-Child(parent, grandparent)
```

```
        Right−Child(grandparent, sister)
        ξ−unlock sister
        ξ−unlock parent
        ξ−unlock grandparent
      else
        grandgrandparent ← parent of grandparent
        ξ−lock grandgrandparent
        ξ−lock grandparent
        ξ−lock parent
        ξ−lock sister
        if grandparent is a left child of grandgrandparent then
          Left−Child(grandgrandparent, parent)
        else
          Right−Child(grandgrandparent, parent)
        Left−Child(parent, grandparent)
        Right−Child(grandparent, sister)
        ξ−unlock sister
        ξ−unlock parent
        ξ−unlock grandparent
        ξ−unlock grandgrandparent
    else (node is left child of parent)
      colour of node ← black
      colour of grandparent ← red
      left ← left child of node
      right ← right child of node
      if grandparent = root then
        ξ−lock grandparent
        ξ−lock parent
        ξ−lock node
        ξ−lock left
        ξ−lock right
        root ← node
        Right−Child(node, parent)
        Left−Child(node, grandparent)
        Left−Child(parent, right)
        Right−Child(grandparent, left)
        ξ−unlock right
        ξ−unlock left
        ξ−unlock node
        ξ−unlock parent
        ξ−unlock grandparent
      else
        grandgrandparent ← parent of grandparent
        ξ−lock grandgrandparent
        ξ−lock grandparent
        ξ−lock parent
        ξ−lock node
        ξ−lock left
        ξ−lock right
        if grandparent is a left child of grandgrandparent then
          Left−Child(grandgrandparent, node)
```

```
          else
            Right−Child(grandgrandparent, node)
          Right−Child(node, parent)
          Left−Child(node, grandparent)
          Left−Child(parent, right)
          Right−Child(grandparent, left)
          ξ−unlock right
          ξ−unlock left
          ξ−unlock node
          ξ−unlock parent
          ξ−unlock grandparent
          ξ−unlock grandgrandparent
  colour of root ← black
  α−unlocked locked
  return ¬ found
```