Implementing Concurrent Red-Black Trees in Java

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

March 10, 2010

Abstract

In the previous assignment, we presented three concurrent implementations of red-black trees. In this assignment, we present their implementation in Java.

1 Introduction

In [1], we presented three different ways to implement red-black trees concurrently. In all three implementations, we considered only the operations Contains and Add. Our first implementation uses monitors. Our second implementation is an adaptation of a solution of the readers-writers problem. In our third implementation, we adapt the concurrent implementation of AVL trees by Ellis [3] to the setting of red-black trees.

In this paper, we discuss the implementations of all three in Java. All three Java implementations are based on a Java implementation of the sequential algorithms for the Contains and Add operations as can be found in [2]. We introduce an interface Set that contains the methods contains and add. To avoid name clashes, each implementation resides in a different package. Each package contains a class RedBlackTree and a class Node. The former implements the interface Set and the latter represents a node of a red-black tree.

The concurrent Java implementation based on monitors is simply obtained from the sequential Java implementation of red-black trees by making the methods contains and add synchronized. This ensures that no method invocation can interfere with another one. This amounts to the execution of the method invocations one at a time.

To implement a variation on a solution to the readers-writers problem is Java, we exploit the class ReentrantReadWriteLock. This class implements the interface ReadWriteLock. According to the documentation of the Java standard library,[1] "a ReadWriteLock maintains a pair of associated locks, one for read-only operations and one for writing. The read lock may be held simultaneously by multiple reader threads, so long as there are no writers. The write lock is exclusive." We use the read lock in the contains method and the write lock in the add method.

A paragraph about the third implementation. A paragraph about testing. A paragraph about performance comparison.

[1] See `http://java.sun.com/javase/6/docs/api/java/util/concurrent/locks/ReadWriteLock.html`.

2 The Set Interface

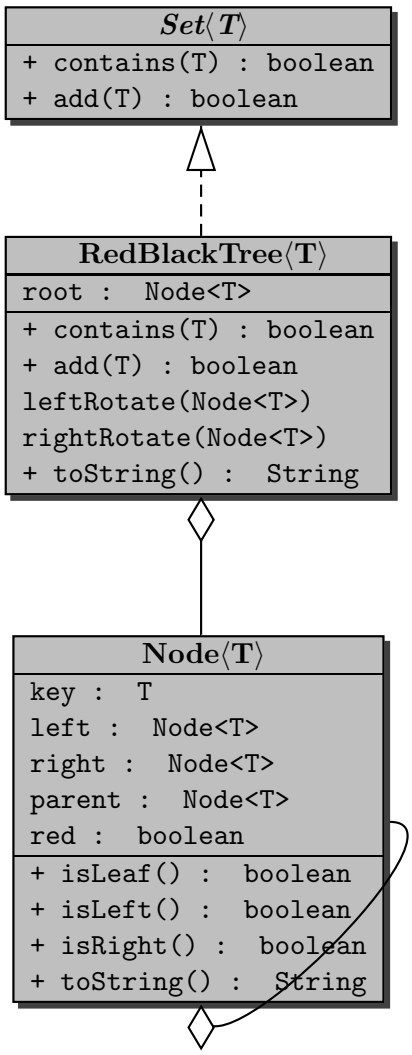
Our interface Set is a simplification of the interface Set which is part of the package java.util. Our interface only contains the methods contains and add, whereas the one in Java's standard library contains several other methods. In our setting, a Set cannot contain **null** whereas a java.util.Set can.

```
1  package collection;
2
3  /**
4   * A set of elements different from null.
5   *
6   * @author Franck van Breugel
7   */
8  public interface Set<T extends Comparable<T>>
9  {
10   /**
11    * Tests whether this tree contains the given element.
12    *
13    * @param element the element for which to search.
14    * @pre. element != null
15    * @return true if this tree contains the given element.
16    */
17   public boolean contains(T element);
18
19   /**
20    * Attempts to add the given element to this tree.
21    * The attempt is successful if this tree does not contain the given element yet.
22    *
23    * @param element the element to be inserted.
24    * @pre. element != null
25    * @return true if the addition is successful, false otherwise.
26    */
27   public boolean add(T element);
28 }
```

3 The RedBlackTree and Node Classes

All three implementations consist of two classes: RedBlackTree and Node. Instances of the class Node represent a node of a red-black tree. An instance of the class RedBlackTree represents a red-black tree.

The UML diagram below contains those attributes and methods that are common to all three implementations. The classes also contain accessors and mutators for all its attributes.



Note that both the RedBlackTree class and the Node class contain a toString method. This method is useful for debugging and testing. The methods leftRotate and rightRotate are auxiliary methods to the add method.

4 The Monitors Approach

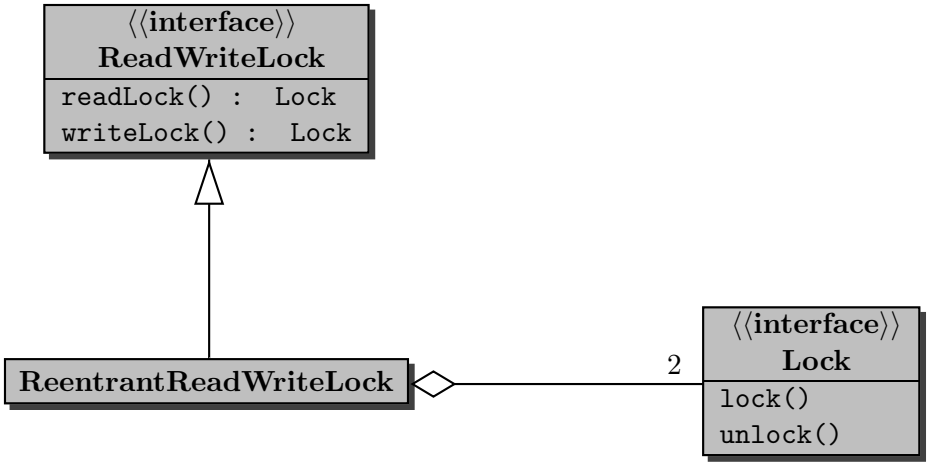
The monitors approach presented in [1] can be mapped to Java in a straightforward way. The only thing that needs to be done to turn a sequential implementation of a red-black tree into a concurrent one is make the methods contains and add synchronized.

```
1 public class RedBlackTree<T extends Comparable<T>> implements Set<T>
2 {
3   ...
4   public synchronized boolean contains(T key)
5   ...
6   public synchronized boolean add(T key)
```

```
7     ...
8  }
```

5 The Readers-Writers Approach

The key ingredient of this implementation is the the class ReentrantReadWriteLock which implements the interface ReadWriteLock. A ReadWriteLock has two Locks: a read-lock and a write-lock. The relevant interefaces and classes and their relationships are given in the UML diagram below.



The read-lock is used in the contains method and the write-lock is used in the add method as follows.

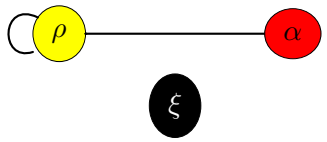
```
1  public class RedBlackTree<T extends Comparable<T>> implements Set<T>
2  {
3    private ReadWriteLock lock;
4    ...
5
6    public RedBlackTree()
7    {
8      this.lock = new ReentrantReadWriteLock();
9      ...
10   }
11
12   public boolean contains(T element)
13   {
14     this.lock.getReadLock().lock();
15     ...
16     this.lock.getReadLock().unlock();
```

```
17    }
18
19    public boolean add(T element)
20    {
21      this.lock.getWriteLock().lock();
22      ...
23      this.lock.getWriteLock().unlock();
24    }
25
26    ...
27  }
```

6 The Fine-Grained Locking Approach

Next, we discuss the implementation in Java of our adaptation of the concurrent AVL trees algorithms proposed by Ellis [3] to red-black trees.

Recall that the key idea of this implementation is that individual nodes can be locked in three different ways: ρ-locked, α-locked and ξ-locked. Although threads can hold a lock on the same node, there are some restrictions. The following graph [3] 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.

Most of the pseudocode presented in [1] can be translated into Java in a straightforward way. The most challenging part of the implementation is the locking and unlocking of the nodes. For that purpose, we add the following methods to the Node class.

```
1  public synchronized void readLock() { ... }
2  public synchronized void readUnlock() { ... }
3  public synchronized void writeLock() { ... }
4  public synchronized void writeUnock() { ... }
5  public synchronized void exclusiveLock() { ... }
6  public synchronized void exclusiveUnlock() { ... }
```

The methods readLock and readUnlock correspond to ρ-lock and ρ-unlock, respectively. Furthermore, the methods writeLock and writeUnlock correspond to α-lock and α-unlock, respectively. Finally, the methods exclusiveLock and exclusiveUnlock correspond to ξ-lock and ξ-unlock, respectively. All these methods are synchronized since, as we will see, they manipulate shared data.

In order to implement the locking and unlocking, we keep track of the following data:

- the number of threads that have ρ-locked this node. In order to ξ-lock the node, we need to know that no thread has ρ-locked it. Hence, we introduce the attribute

```
private int readers;
```

which is initialized to zero.

- whether a thread has α-locked this node. This allows us to ensure that at most one thread α-locks a node. For that purpose we introduce the attribute

```
private boolean write;
```

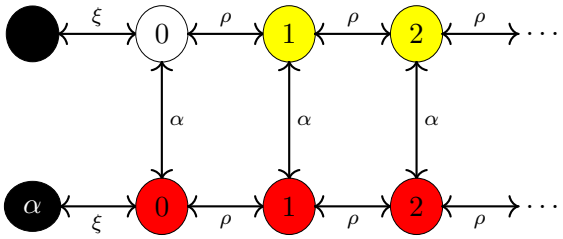
which is initialized to false.

- whether a thread has ξ-locked this node. To ensure that a ξ-lock is exclusive we introduce the attribute

```
private boolean exclusive;
```

which is initialized to false.

The above three attributes capture the state of a node. In the diagram below, we depict how the state of a node changes by performing locking and unlocking. The numbers correspond to the value of the attribute readers. In the red states, the attribute write has the value true. In the black states, the attribute exclusive has the value true. Note that if a thread has α-locked a node, that thread can change it into a ξ-lock. Once the thread ξ-unlocks the node, the node will still be α-locked. To distinguish between a node whose α-lock was changed into a ξ-lock and a node that was unlocked before it was ξ-locked, we label the former with an α.



The lock and unlock methods are all implemented similarly. Let us only look at the most interesting ones: exclusiveLock and exclusiveUnlock.

```
public synchronized void exclusiveLock()
{
    while (this.readers != 0)
    {
      try
      {
        this.wait();
      }
      catch (InterruptedException e)
      {
         System.out.println("wait within exclusiveLock was interrupted");
      }
    }
    this.exclusive = true;
}
```

```
1 public synchronized void exclusiveUnlock()
2 {
3   this.exclusive = false;
4   this.notifyAll();
5 }
```

7 Testing

A brief description of the testing strategy (which found several bugs in the code).

8 Performance Comparison

An overview of the performance numbers.

9

In [1], we conjectured 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))
```

We conjectured that by interleaving the elementary operations of the operations Add and Contains in a particular way, the operation Contains may return false. We ran the Java counterpart of the above code many times, but the contains method never returned false. We hope to confirm this conjecture by our verification effort.

10 Conclusion

Comparison of the three implementations. Discuss what may be improved.

References

[1] Franck van Breugel. Concurrent red-black trees, January 2010.

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

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