A Java implementation of Non-blocking Binary Search Trees

ALESSIO BOGON

University of Trento - Master Degree in Computer Science

Concurrency

alessio.bogon@studenti.unitn.it

August 10, 2015

1 Introduction

This project is a non-blocking concurrent Java implementation of the binary search tree structure, and aims to reproduce the work of Ellen et al[1] (from now on, "the paper"). The interface targeted is `Set<T>`, which defines the *find*, *insert* and *remove* operations. The class that implements it is `Non-BlockingBinarySearchTree<T>` and can be found inside the `it.unitn.studenti.a-lessiobogon.concurrency.nbbst` package.

All the interface methods implemented are total and lock-free. Moreover, the `find` method is wait-free. This is due to the fact that insertions and deletions might need to *help* other ongoing operations in order to complete, while `find` can just look for the item without caring since the tree is always in a consistent state.

The project also provides the following features:

- **Java "generics"** When useful, classes are declared using generic types.
- **Javadoc** Code documentation.
- **OCaml-like output syntax** Tree representation in OCaml-like style.
- **Dot output syntax** Tree representation in the "dot" format.
- **Logging facility** The class logs all the important events. You can choose to print just method calls/returns or to go deeper and show also the result of CAS operations.

2 Usage

The program can be executed by running the provided script on UNIX platforms:

```
./run.sh
```

It basically compiles all the sources needed by the `Main` class and then run it. Please ensure that you are using Java 7 or above and that the `java` and `javac` executables are available in your `PATH`. You can also configure the verbosity level of the output via the VM parameters (e.g. `java -DlogLevel=FINEST ...`), which allows to trace the various compare-and-swap (CAS) operations. The default level is *FINE*, which shows the methods invocations and return values of the interface methods. The main class will also save the dot representation of the resulting tree inside the `graph.dot` file. Please refer to `README.txt` for further information.

Thread#10	**Thread#11**	**Thread#12**
`bst.insert(20)`	`bst.insert(15)`	`bst.insert(30)`
`bst.insert(30)`	`bst.delete(20)`	`bst.find(20)`
	`bst.find(15)`	`bst.insert(15)`
		`bst.find(20)`

Table 1: Three threads and the operations they are willing to execute

The provided *Main* uses five threads and executes many operations, but for the sake of brevity we report this simple history, where three threads try to perform the operations described in Table 1.

One possible execution is the following:

```
 1  [02:03:55.648] Thread#10:    insert: ENTRY 20
 2  [02:03:55.648] Thread#12:        insert: ENTRY 30
 3  [02:03:55.648] Thread#11:      insert: ENTRY 15
 4  [02:03:55.652] Thread#10:    insert: RETURN true
 5  [02:03:55.653] Thread#10:    insert: ENTRY 30
 6  [02:03:55.653] Thread#10:    insert: RETURN true
 7  [02:03:55.654] Thread#11:      insert: RETURN true
 8  [02:03:55.654] Thread#11:      delete: ENTRY 20
 9  [02:03:55.655] Thread#12:        insert: RETURN false
10  [02:03:55.655] Thread#12:        find: ENTRY 20
11  [02:03:55.655] Thread#12:        find: RETURN true
12  [02:03:55.655] Thread#12:        insert: ENTRY 15
13  [02:03:55.656] Thread#12:        insert: RETURN false
14  [02:03:55.656] Thread#11:      delete: RETURN true
15  [02:03:55.656] Thread#12:        find: ENTRY 20
16  [02:03:55.656] Thread#11:      find: ENTRY 15
17  [02:03:55.657] Thread#12:        find: RETURN false
18  [02:03:55.657] Thread#11:      find: RETURN true
19  The resulting tree is: Node(2147483647, Node(2147483646,
20      Node(30, Leaf(15), Leaf(30)), Leaf(2147483646)),
21      Leaf(2147483647))
22  [02:03:56.023] Thread#1:  main: Graph written into graph.dot
23  [...]
```

As you can verify, this is a valid execution: *Thread#10* and *Thread#11* successfully complete all the operations. *Thread#12* fails to insert 30 (line 9) because it already exists, then it finds 20 because *Thread#10* inserted it before (line 4) and *Thread#11* didn't delete it yet. Then it fails again trying to insert 15 (line 13) and then it looks for 20 and can't find it (line 17) since *Thread#11* managed to delete it (line 14). You can see the OCaml representation of the resulting tree in the output and the visual representation in Fig. 1, which is stored in the file `graph.dot`.

3 Implementation

The provided implementation is basically a rewriting of the work done by Ellen et al in Java. Some trickery is needed when dealing with the CAS objects.

The implemented tree is leaf-oriented: every node has a key that needs to route the visit, and the real items are stored in the leafs (although it is not required). It is initialized with two sentinels to avoid dealing situations where the tree has less than three nodes. This class is called `NonBlocking-BinarySearchTree<T>`.

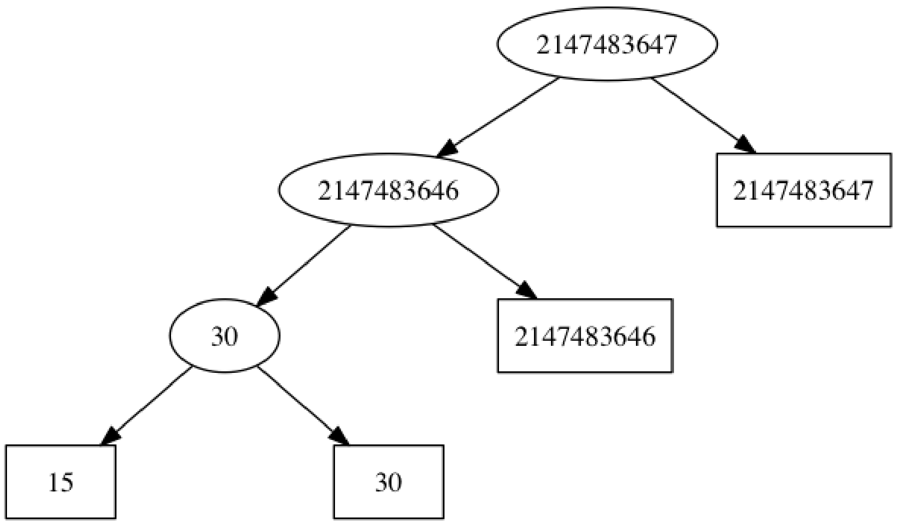

Figure 1: Visualization of the output tree for the example given in Section 2. Internal nodes are represented by ellipses, while leafs are represented by boxes. The top two nodes are ∞_2 and ∞_1 respectively.

The types described in the paper (*Update*, *Internal*, *Info*, etc.) are implemented using classes, according to the standard object-oriented paradigms (subclassing, abstract classes, etc.). Classes that are to be accessed with compare-and-set operation are wrapped by an `AtomicReference`. Section 4 shows some of the challenges involved. All the instance variables that are not going to be changed are declared using the *final* modifier to ensure this property. Basically all of them are *final* because threads cannot modify object fields directly, but they must get through the `AtomicReference` methods.

The following is a brief overview of the most interesting methods of the tree class:

- `find(T item)` descends the tree looking for the element with key given by the hash code of the item. It returns true if and only if it has been found. This function is wait-free, since it does not help ongoing operations.

- `insert(T item)` looks for the parent of the new node to insert, after checking that it is not already present. Then it creates an `UpdateInfo` object with all the information required to complete this operation. It tries to *iflag* the parent, and if the operation was successful it physically adds the new node (*ichild*) and marks the parent as clean (*iunchild*). If it was not possible to *iflag* the parent, then it starts the procedure again. This function is lock-free, because it might encounter some unfinished operations that prevents the insertion, and in that case it would help them to complete and start the procedure again.

- `remove(T item)` looks for the parent and the grandparent of the node to remove, if any. Just like the previous method, it creates a `DeleteInfo` object with enough information for any thread to complete the deletion. Then it tries to *dflag* the grandparent node; if it succeeds it tries to *mark* the parent node. If any of these operations fail, the procedure is restarted (it may also need to *backtrack*). The operation is completed with the *dchild* (physical node removal) and *dunflag* (grandparent state cleaneded) steps. Also this method is lock-free, because it might need to *help* other operations.

- `help(), helpDelete(), helpMarked(), etc.`: these methods encapsulate the behavior that helps a specific operation to complete. They are to be called by insertions and deletions that encounter nodes state that prevent them to complete.

To get a deeper view of the class, please refer to the *javadoc*.

4 Challenges

In order to achieve the *non-blocking* requirements we use the Java variant of the compare-and-swap (CAS) operation, which is the compare-and-set. The key difference is that compare-and-swap returns the value the object had before the operation, while compare-and-set returns *true* if and only if the operation has been completed successfully.

In the paper, the result of a compare-and-swap operation is checked by comparing the value returned with the one we are expecting. If they are equal, this means that the operation was successful, otherwise the operation failed. With the compare-and-set semantics, it is sufficient to check the return value to determine the outcome.

The main difficulty comes when we need to check the reason why an operation was unsuccessful. This happens in the `helpDelete()` method: it tries to *MARK* the parent node of the given operation, but the compare-and-set might fail because some other thread already marked it. In this case we cannot blindly return false, because it would mean that the operation cannot be completed and needs to backtrack. To solve this, the result of the compare-and-set is checked. If it failed, the current state of the parent is obtained and checked if it has been *MARK*ed by another node. This works fine because, intuitively, once a node has been marked it cannot change its state (from this point, the `delete()` operation can only succeed) and will eventually be physically deleted by unlinking it from its parent (*dchild*).

Another problem comes with the implementation of the `Node.update` field, which is a *word* that points to an `Info` record. This same word is also used to store a `State` record, by stealing the first two bits. Since in Java we do not have access to raw pointers, we cannot steal the first two bits of the reference. "All problems in computer science can be solved by another level of indirection [...]" (David Wheeler): indeed the provided solution just wraps the `Info` and `State` records into an `Update` object. To conclude, the `Node.update` field is an `AtomicReference` that points to an `Update` object.

Alternatively, one could have used the `AtomicStampedReference`, but this class does exactly what is described above[1]. However this would make the code less clear, so the former solution is used.

Worth of note is the fact that the paper initializes the tree with two special values, ∞_2 and ∞_1. To manage this issue this implementation replaces them with `Integer.MAX_VALUE` and `Integer.MAX_VALUE - 1` respectively. The only limitation is that it is not possible to insert nodes with such values, but it is an acceptable compromise in my opinion. Alternatively, one could write an `ExtendedInteger` class that implements the `Comparable` interface and that deals with these special values. This would result in a more messy code because comparisons like `this.key < other.key` would become `this.key.compareTo(other.key) == -1`, since Java does not allow to define custom operator overloads.

References

[1] Ellen, F., Fatourou, P., Ruppert, E., and van Breugel, F. *Non-blocking Binary Search Trees*. Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing (PODC), 131140, 2010.

[1]See the OpenJDK implementation at `https://github.com/openjdk-mirror/jdk7u-jdk/blob/jdk7u6-b08/src/share/classes/java/util/concurrent/atomic/AtomicStampedReference.java`