Master Degree in Computer Science

Concurrency

AA 2014-2015

An Implementation of a non-blocking Binary Search Tree in Java

Project report

Davide Spadini (id 164393)

1 Introduction

A non-blocking Binary Search Tree (BST) is a binary search tree in which the three operations Find(key), Insert(key), and Delete(key) maybe invoked by concurrent threads.

The goal of this project is to implement a BST in Java using the algorithm described in [1]. This algorithm is *non-blocking* (starting from any configuration of any infinite asynchronous execution, with any number of crash failures, some operation always completes) and *linearizable*. The full (and lengthy) proof of correctness appears in [2].

2 Usage

In order to compile the project, open the terminal in the root directory and digit:

```
javac -d bin src/*.java
```

To run it, go in the *bin* directory and digit:

```
java Main
```

In the configuration that is used there are **5 threads**, concurrently performing 15 insert, 10 delete and 10 find.

3 Output

When the program starts, each thread prints on the terminal when it starts and ends a new operation (i.e. Thread-1 trying to insert 8, Thread-1 inserted correctly key 8). In this way, we can see the order in which the operation has been performed. When all the threads have finished to execute their operations, the program prints the resulting tree in two different formats:

- **OCaml syntax** Printed on the terminal
- **Dot format**[1] Saved on an external file *tree.dot*

To build the graph starting from the *dot* file, open the terminal in the root directory and launch:

```
dot tree.dot | gvpr -c -ftree.gv | neato -n -Tpng -o binarytree.png
```

and then open *binarytree.png*.

In order to obtain a better result, in the command above I used an external script *tree.gv* that enforces the horizontal node ordering.[2]

4 Implementation

As explained before, the aim of this project is to give an implementation of a non-blocking BST using the algorithm described in [1], so I will not discuss the algorithm but only the implementation of it. It follows step by step the algorithm, it has the same classes and methods. The only differences or extra method that I implemented are explained in Sec.[4.1].

The main classes are the following:

- **Leaf** The node with the key
- **Internal** The node used to traverse the tree

[1] see `http://www.graphviz.org`

[2] All credits of this script goes to *Emden R. Gansner*, script available at `https://mailman.research.att.com/pipermail/graphviz-interest/2010q2/007101.html`

- **Update** Contains the state of the node and an *Info* record.
- **IInfo** When an insert is performed, the thread create an IInfo record in order to save enough information so that another thread that encounters the flagged node can complete the operation
- **DInfo** The same as the IInfo, just for the delete function
- **BinarySearchTree** The real program, contains all the functions of the BST
- **Main** The main of the program. In this class are instantiated 5 threads and 1 BST

Furthermore there are minor class, but for more information I remind the reader to Sec.[5]

4.1 Implementation Details

- **Compare-And-Set (CAS)** The non-blocking algorithm uses CAS[3] in order to update atomically some information (i.e. change the left child pointer of a node from one node to another one). Another point in which CAS is used is when we need to change the *Update* field of a node, and here is where the problem comes out: in the algorithm they CAS a record composed by two field (state and info), but in Java there is not this possibility. So instead of using a simple reference I use an *AtomicReference<Update>*. At the beginning of the functions that use the CAS on that field (i.e. insert or delete) I copy the reference on a local variable (a sort of *snapshot*, in this way I'm sure that no-one can change it during the execution of the function), and then I CAS the current value of the field with the *snapshot* that I took before.

- **TreeThread class** I use this class in order to give a sequence of operations to the threads. The list of operations are passed by parameters in form of a tuple <action, value> (i.e. <insert, 4 >). When a thread completes all the operations, it exits.

- **Output in OCaml syntax** In order to print the tree with this syntax, I use the *toString()* function recursively from the *Root* until the leafs. The implementation is simply:

  ```
  Node(key , left.toString(), right.toString())
  ```

- **Dot format** There is a function *toDot()* which writes the tree according to the dot format in an external file *tree.dot*. This function is recursive, and it is called starting from the root. The result is a directed graph, in which the internal nodes are represented by black circles, and the leafs are represented by blue squares.

All the other functions are the identical representation in Java of the algorithm desribed in [1]. In the next sections we will briefly see them, for a complete overview see the Javadoc in Sec.[5].

4.2 Find

This function takes in input a key, and it uses the *search* method to traverse the tree searching for the leaf with that key. If this leaf exists return it, otherwise return *null*.

4.3 Insert

This function takes in input a key and it inserts a new leaf in the tree with that key. The main steps are the following:

- First of all it checks if the key already exists (duplicated keys are not admitted in the search tree)
- It checks if there are other pending operations on the same nodes that he needs to operate on, in this case it tries to complete them, otherwise it builds a new structure with the key.
- After that it tries to change the state of its parent to "IFLAG" (inform other processes that an insert is being performed)
  - If it fails, it helps the operation that has flagged or marked the parent, if any, and begins a new attempt.
  - If it succeeds, it completes the insertion.

[3] for more details see `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicReference.html`

4.4 Delete

This function takes in input a key and it delete the corresponding leaf from the tree. The main steps are the following:

- First calls the *search* method to find the leaf to be deleted, if it fails returns False.
- If some other operation has already flagged or marked the parent or grandparent, it helps that operation complete and then begins over with a new attempt.
- Otherwise, it attempts to set the flag of the grandparent to "DFLAG"
  - If this fails, it helps the operation that has flagged or marked the grandparent, if any, and then begins a new attempt
  - If the "DFLAG" is successful, then it attempts to set the flag of the parent to "MARK"
    - If it is successful, it completes the deletion
    - Otherwise backtrack and restart the procedure

In the delete function we have to "DFLAG" the grandparent of the node, and "MARK" the parent of the node. Due to the fact that this 2 steps may be not being executed sequentially, what could happens is that the first step is completed correctly and second one fails (because for example another process changes the state of the parent). So, in this case, there is the possibility to reset the state of the grandparent from "DFLAG" to "CLEAN" and restart again all the procedure (backtrack).

5 Javadoc

In the root of the project there is a directory *Javadoc*[4] containing the API documentation in *HTML* format. Each class and method's working principles are described in detail, so for all the functions that I don't explain in this paper, or to see them in more details, refers to it.

6 Test

References

[1] Faith Ellen, Panagiota Fatourou, Eric Ruppert, Franck van Breugel *Non-blocking binary search trees* Available at `http://dl.acm.org/citation.cfm?doid=1835698.1835736`

[2] Faith Ellen, Panagiota Fatourou, Eric Ruppert, Franck van Breugel *Non-blocking binary search trees* Technical Report CSE-2010-04, York University, 2010.

[4] For the documentation and more information see `http://www.oracle.com/technetwork/articles/java/index-137868.html`