UNIVERSITY OF TRENTO
Department of Information Engineering and Computer Science

Master of Science in Computer Science

Concurrency

Non-blocking Binary Search Trees: Project C

Academic Year 2014 - 2015

I. Introduction

The Java standard library has several non-blocking data structures, but no search trees. Non-blocking data structures means that always some operation makes progress. Concerning binary search trees (BST), many proposals for a concurrent implementation of them came out in the last 25 years, but most implementations where lock-based or too complex as in [1], where 8-word compare-and-swap (CAS) are used. F. Ellen published in 2010 a new implementation of BST [2], with many advantages compared to all prior results:

- Non blocking
- Use single word CAS
- Concurrent updates to different parts of tree do not conflict
- Conceptually simple and good performance

This report describes how, at the end of the *Concurrency course* held at the Computer Science department of the University of Trento, a concrete Java implementation of F. Ellen's BST has been developed.

II. Student task

The provided Java implementation of the BST is based on the pseudo-code and data model presented in [2]. The BST supports FIND, INSERT and DELETE operation even if invoked by concurrent threads. Given the type definitions and initialization rule shown in figure 1 and the pseudo-code taken always from [2], in a few days the Java code that strictly follows F. Ellen's guidelines has been developed.

The actually implemented tree is a Leaf oriented tree and the defined concurrent operations makes intensive use of compare-and-swap (CAS), in order to make "non-blocking" the data structure. Data types that require to be updated atomically with CAS operation have been implemented using *AtomicReference* or *AtomicStampedReference* class from the *java.util.concurrent.atomic* package.

A. algorithmic problems

Concurrent interferences between different threads could arise if operations conflict on same nodes. Coordination is therefore required and introduced by F. Ellen defining status of nodes as either CLEAN, FLAGGED or MARKED, and changing status using CAS. A flag indicates that an update is changing a child pointer and a mark means that an internal node has been (or soon will be) removed from the tree. Every thread to complete its operation should flag interested node and restart if that node is already flagged or marked, this mechanism is required to gain correctness. Can think of flag or mark as a lock on the child pointers of a node. Flag corresponds to temporary ownership of lock while a mark corresponds to permanent ownership of lock. Actually, whenever "locking" a node, some info are left under the doormat. A flag or mark is in truth a pointer to a small record that tells a process how to help the original operation. If an operation fails to acquire a lock, it helps complete the update that holds the lock before retrying. At this stage, starting from a node with CLEAN state, the INSERT and DELETE operations could be represented as cycles of subsequent CAS as in figure figure 2.

```
1  type Update {                ▷ stored in one CAS word
2      {CLEAN, DFLAG, IFLAG, MARK} state
3      Info *info
4  }
5  type Internal {              ▷ subtype of Node
6      Key ∪ {∞1, ∞2} key
7      Update update
8      Node *left, *right
9  }
10 type Leaf {                  ▷ subtype of Node
11     Key ∪ {∞1, ∞2} key
12 }
13 type IInfo {                 ▷ subtype of Info
14     Internal *p, *newInternal
15     Leaf *l
16 }
17 type DInfo {                 ▷ subtype of Info
18     Internal *gp, *p
19     Leaf *l
20     Update pupdate
21 }
   ▷ Initialization:
22 shared Internal *Root := pointer to new Internal node
       with key field ∞2, update field ⟨CLEAN, ⊥⟩, and
       pointers to new Leaf nodes with keys ∞1 and
       ∞2, respectively, as left and right fields.
```

Figure 1. Type definitions and inizialitation.

B. correctness

Based on the CAS cycles we can state that concurrent operation always complete successfully or either restart. We want to show that the described data strucutre is non-blocking, meaning that always some operation completes. Concerning FIND operations they just traverse edges and they can ignore flags

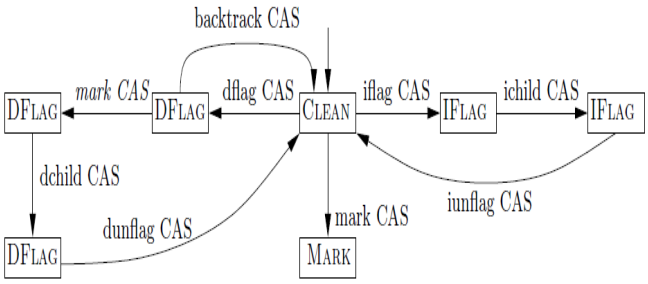

Figure 2. Type definitions and inizialitation.

and marks, no problem with them. We are left with INSERT and DELETE but:

- If an INSERT successfully flags, it finishes.
- If a DELETE successfully flags and marks, it finishes.
- If updates stop happening, searches must finish.

Given that one CAS fails only if another succeeds this means that progress is granted. This is why F. Ellen's BST claims algorithmic correctness and non-blocking property.

Given that the provided Java implementation reflects faithfully the Ellen's proposal, the same kind of correctness is claimed for this implementation.

III. Major problems

Defining in Java the data types suggested in figure 1 and translating the pseudo-code into real Java code has been not difficult. The onliest problem encounterd has been the proper use of CAS as described by F. Ellen by using the *comapreAndSwap* provided by Java 8. In particular the implementation of *unflag* (both after insertion or deletion) and *backtrack* CAS has been problematic. The lack of a copy by reference of the Update field to be CASed led, in a first attempt to implement the above cited operation, to this error: compareAndSet has been called passing as "expected" parameter not a snapshot-pointer, but the actual pointer to the Update field.

update.compareAndSet(update,new Update())

This kind of «self-CAS» (this self-CAS always succeeds), make it possible to unflag operations every time such a self-CAS is invoked, maybe interrupting operation that are not completed. Moreover during unflag/backtrack CAS two objects are expected to be checked, and the simple compareAndSet offered by *AtomicReference* Java class was unsuitable.

These problems has been solved with the use of a more powerful class, i.e. *AtomicStampedReference* that provides this kind of CAS:

boolean compareAndSet(V expectedReference, V newReference, int expectedStamp, int newStamp)
Atomically sets the value of both the reference and stamp to the given update values if the current reference is == to the expected reference and the current stamp is equal to the expected stamp.

The idea of using *AtomicStampedReference* to solve above described problems has been inspired by the code of a student-colleague[1]

A. testing

Given the multi-thread environment of testing, it has been very difficult to trace the execution of any test. However timestamps with nanosecond precision reporting start and end of execution of any operation on the tree are provided. Is granted that every operation can be linearized to an instant of time between the *start* and *end* timestamps related to the operation, but this time-bounds are loose: this means that a lot of sequential consistent linearizations can be found analyzing the log, so unfortunately the real trace of execution can not be univocally identified.

IV. Usage

The full code and required libraries are published and available on github.

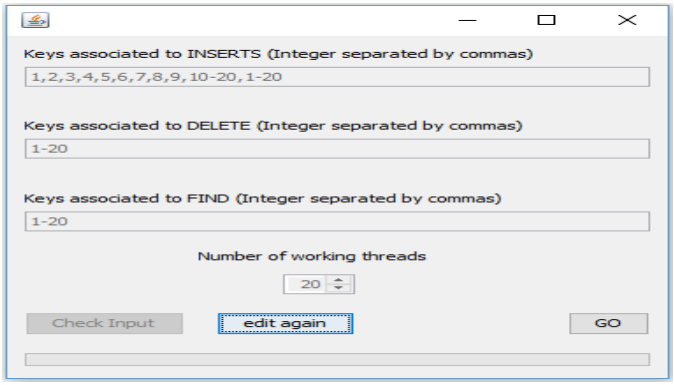

Figure 3. Screenshot from a run.

[1]code of D. Bertolini, provided as a hint by professor

To run a test with GUI (a screenshot is shown in figure 3), open a console and move to *"Conc2015BST/"*, then simply type:

ant start

Code will be compiled and Test class executed. Outputs of a run are a log file and a tree representation respectively in .csv and .dot format, that will be automatically shown at the end of the execution (an example in figures 4 and 5)

```
Description,nanoStart,nanoEnd
INSERT(16):true,348539641146397,348539646216035
DELETE(4):false,348539641168638,348539646515431
INSERT(16):false,348539640006980,348539647032104
INSERT(5):true,348539640507827,348539646214751
DELETE(1):false,348539640561291,348539646370010
INSERT(11):true,348539640585670,348539646975646
FIND(15):true,348539640739646,348539647076586
```

Figure 4. log of some operations

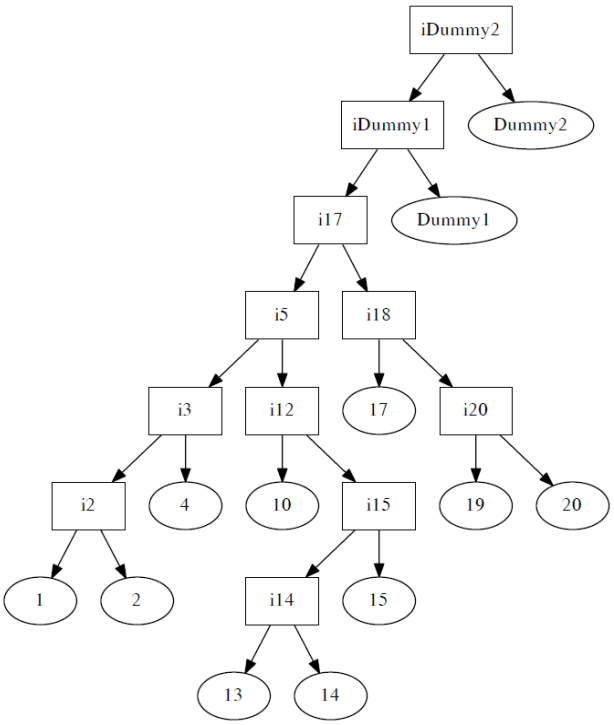

Figure 5. tree representation

REFERENCES

[1] K. Fraser, "Practical lock-freedom," Ph.D. dissertation, University of Cambridge, 2004.

[2] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, "Non-blocking binary search trees," in *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. ACM, 2010, pp. 131–140.