A concurrent implementation of a binary search tree dictionary

Diem Hoang Nguyen - hong@itu.dk
Mustapha Malik Bekkouche - mube@itu.dk
Supervisor: Peter Sestoft

January 2018

1 Abstract - please ignore

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam commodo neque a urna rutrum, quis viverra libero vehicula. Vivamus aliquam ligula nec magna venenatis dictum. Curabitur vel suscipit ligula, in lacinia ligula. Morbi sed sem sollicitudin, pulvinar erat non, vulputate mauris. Integer pharetra, ligula quis mattis aliquet, erat nibh cursus magna, maximus ullamcorper mauris odio et nunc. Morbi et risus interdum, volutpat enim posuere, pulvinar quam. Mauris vestibulum, lectus quis accumsan posuere, ligula urna tincidunt quam, at cursus lectus leo at dolor. Nulla sit amet nisi purus. Nunc non diam tristique, feugiat libero at, viverra tellus. Sed et lacus odio. Maecenas non tellus nec ante porta molestie in et ligula. Aliquam urna ante, sagittis vitae fermentum ac, lacinia in dolor. Ut id pretium massa, a pharetra lorem. Vestibulum sit amet blandit nisi. Sed volutpat in arcu et euismod. Suspendisse in sem at nulla blandit lobortis sed eu odio.

Sed porta et sapien vitae placerat. Suspendisse at cursus elit. Donec nec pellentesque sem. Donec vitae metus sit amet turpis scelerisque aliquet. Nunc viverra nec velit ac interdum. Nullam hendrerit sapien ligula, quis pulvinar lorem hendrerit quis. Curabitur et fringilla tortor. Quisque scelerisque interdum ligula, quis mattis sapien vehicula at. Sed fermentum erat ut tristique volutpat. Ut vitae interdum magna.

Contents

2 Introduction - please ignore

3 Background

For years, processor manufacturers delivered increases in clock rates, so that single-threaded code executed faster on newer processors with no modification. As we are increasingly approaching the limits of Moore's law with transistors getting as small as 5 nm each, it has become harder to scale up clock rates without negative side effects. Constructors have thus turned to multi-core architectures for processors, this architecture allows parallelization of execution. With this parallel paradigm comes benefits and drawbacks, while it can allow a huge increase in performance and throughput it also increases the possibility for developers of making mistakes because of the unpredictable execution flow. It is thus necessary to use specially designed classes for this paradigm called thread safe classes. The Java framework has done this with J2SE 5.0 developped under JSR166 and released on the 30th of September 2004 (ref :https://www.jcp.org/en/jsr/detail?id=166) which contains high-level thread constructs, including executors, which are a thread task framework, thread safe queues, Timers, locks (including atomic ones), and other synchronization primitives. (ref : http://www.oracle.com/technetwork/articles/javase/j2se15-141062.html) The concurrency utilities have then been updated with each new release. In this paper we will implement

3.1 Tree dictionaries

A tree dictionary is a data structure composed of nodes, where each node might contain a key-value pair ¡k,v¿ in addition to pointers to children nodes, in this project we will focus on binary search trees where each node will have at most two children called Left and Right. For the rest of this paper we will refer to the children of a node n as LeftChild(n) and RightChild(n)
A BST can be Internal/External and tree : an internal tree is a tree where each node contains a key-value pair, while an external tree only has it in the leaf nodes (nodes that do not have children) and the internal nodes are used solely as routing nodes, this will create tree with greater height but also simplifies operations like remove(Key) where you only have to delete the link between the node containing Key and it's parent. balanced/unbalanced : a balanced tree is a tree that has the minimum possible maximum height, it uses balancing operations to re-arrange the nodes when needed (after put or remove operations), this will guarantee O(log n) running time for both look up and insertion operations The binary search tree has three important operations : get, put and remove. get(k) will start from the root node r and compare k with the key stored in the node, if they are equal we return the value stored in the node, if k is less than or greater than the stored key we re do the same operation on LeftChild(r) or RightChild(r) respectively. put(k,v) will go through the tree to find the correct place to insert the new node, we use the same routing algorithm to find that place this operation might trigger a rebalancing operation in case of a balanced

tree. If the key was already in the tree this operation will replace the value in the node with v. remove(k) will also use the same routing algorithm, if a node containing k is found the node will be deleted and the value it contained will be returned, a null otherwise. B.inary tree is a sespecial case of a K-ary tree where k=2, in a binary search tree it is guaranteed that for each node n : Key(n) ¿ Key(LeftChild(n)) and Key(n) ¡= Key(RightChild(n))

3.2 Concurrent tree dictionaries

3.3 Brown's work (working title)

4 Current concurrency tree dictionary implementations

5 Tests

5.1 Tests for correctness

5.2 Tests for scalability

6 Test results

7 Concurrent tree dictionary

7.1 Mark I

7.2 Mark II

7.3 Mark III

7.4 ... etc

7.5 further improvements

8 Discussion

9 Conclusion

10 References