Persistent Binary Search Trees

Datastructures, UvA. *May 30, 2008*
0440949, Andreas van Cranenburgh

Abstract

A persistent binary tree allows access to all previous versions of the tree. This paper presents implementations of such trees that allow operations to be undone (*rollback*) as well. Two ways of attaining persistence are implemented and compared. Finally, a balanced version (AA tree) is compared to an unbalanced version.

1 Introduction

According to the NIST dictionary of algorithms and datastructures, a persistent datastructure is "a data structure that preserves its old versions, that is, previous versions may be queried in addition to the latest version."[1] There exist at least four types of persistence (Driscoll et al., 1989):

Naive persistence Examples are making a copy of the whole tree after each update, or storing a history of update operations while only retaining the last version of the tree. In general these methods either waste space (first example), or have slow access time to older versions (second example).

Partial persistence Each version of the tree is retained, but only update operations on the last (live) version of the tree are allowed. The versions form a linear sequence.

Full persistence Older version are kept and can be updated as well. The versions form a Directed Acyclic Graph (DAG).

Confluent persistence Trees can be melded with older versions. This is analogous to version control systems such as `cvs` and `svn`, where revisions can be *merged*.

This paper concerns partially persistent trees, with the added operation of `rollback`, which can promote an arbitrary earlier version of the tree to live version, allowing it to be updated. Note that this `rollback` operation itself cannot be undone, to simplify matters. This behaviour is analogous to the "unlimited undo" of text editors: although each older version is accessible, a modification destroys the versions after that and makes redo, the opposite of undo, impossible.

2 Achieving partial persistence

2.1 Path copying

A simple though naive way of achieving partial persistence is path copying. Instead of copying the whole tree, only the nodes on the path to a modified node can

[1] Algorithms and Theory of Computation Handbook, CRC Press LLC, 1999, "persistent data structure", in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 4 January 2005. (accessed May 30, 2008) Available from: `http://www.nist.gov/dads/HTML/persistentDataStructure.html`

be copied. This means at least $\log n$ nodes have to be copied, for a perfectly balanced tree; but in the worst case, in a degenerate tree all nodes have to be copied. Despite these shortcomings, the advantage of path copying is that it is very easy to implement. All destructive assignments are replaced with updated copies of nodes; afterwards the root is added to a version list.

2.2 Node copying

An optimally efficient way of implementing partial persistence is node copying (Driscoll et al., 1989). By adding an extra field to nodes, each node can contain two different versions of itself. When a node already contains a modification a copy of it is made. This means that instead of copying the whole path it is possible to retain the path to a node, while only modifying the node pointing to the new node. This scheme uses $O(1)$ worst case space, and $O(1)$ extra time, compared to an ephemeral (non-persistent) datastructure.

Hidden behind this Big-oh notation there hides a constant slowdown, since traversing the tree requires an extra comparison on each node, to see if the node contains a modification which corresponds to the version being requested. This slowdown on accesses is not present with path copying; on the other hand, the unnecessary copying of nodes on updates slows down path copying by a logarithmic factor.

3 A simple balanced tree

The most popular balanced binary search tree seems to be the Red-Black tree and the AVL tree. Both trees have complicated balancing schemes, having a multitude of rebalancing cases after each update.

The AA tree (Anderssen, 1993) is a constrained version of the Red-Black tree. Only right nodes are allowed to contain "red" nodes. Also, instead of storing balance information as a bit in each node, nodes contain a level field describing the height of their biggest subtree. This results in only two balance operations: skew (conditional right rotation) and split (conditional left rotation). Insertion requires one skew and one split operation, whereas deletion (typically the most complicated operation in a balanced tree) requires only three skews and two splits.

4 Implementation

The implementation is in Java, with a focus on simplicity, not speed. Hence the use of recursion instead of the more involved iterative traversal using an explicit stack. Also accessing or updating a pointer field, a frequently occurring operation, is done with auxiliary functions, which can be expensive; but inlining them would case tedious code duplication.

Access to previous versions is provided with a `search` method taking an optional version argument, as returned from earlier update operations. On top of that previous versions can be promoted to current (ie., mutable) version with the `rollback` operation.

In order for `rollback` to work on all previous versions some additional bookkeeping has to be done. Namely two lists are maintained; the first containing the

root corresponding to each version while the second contains references to each node that has been mutated. Upon rolling back this last list is unwinded, removing each mutation as it goes.

5 Performance

Benchmarking has been done with two different data sets. To measure average case performance an array of random integers is used; whereas worst-case performance is measured with a sorted, monotonically increasing sequence (ie., the counter variable of the for-loop). Each benchmark run performs a series of insertions, lookups, rollbacks and finally deletions.

Testing worst-case performance proved to be too much for the unbalanced trees; the node copying implementation crashed due to an unknown bug (traversal enters an endless series of null pointers), while the path copying implementation simply runs out of memory. Since the unbalanced versions are presented for comparison only and not for actual usage with possibly worst-case data, I have neglected to fix these shortcomings.

Time complexity has been measured by the `System.currentTimeMillis` method of the standard library, while memory usage has been approximated by subtracting the amount of free heap space from the total heap size, after repeatedly summoning the garbage collector to free all available space.[2]

5.1 Average case perfomance

	n	min	mean	std dev	max
Red-Black tree (baseline)	1000	47904	47904	0	47904
	3000	143856	143856	0	143856
	5000	239904	239904	0	239904
AA-Tree, Node Copying	1000	313120	369978	85231	632712
	3000	920920	1058569	248125	1914152
	5000	1516800	1805062	559645	3752264
BST, Node Copying	1000	292680	331883	79688	645928
	3000	880952	971937	151794	1432792
	5000	1470008	1666562	403379	3258848
BST, Path Copying	1000	1180808	1198809	40500	1357320
	3000	4255824	4315701	136016	4851560
	5000	7389712	7480430	204906	8283320

Table 1: Memory usage (bytes)

As can be seen in table 1 and 2, path copying requires significantly more memory, linearily increasing, whereas the node copying implementations grow logarithmically. Note that the balanced AA tree does require more space, due to the

[2]Inspired by:
`http://www.roseindia.net/javatutorials/determining_memory_usage_in_java.shtml`

rotations increasing the amount of modified nodes per operation. But in this case the difference is a constant factor, apparantly a factor of 2 looking at the data.

	n	min	mean	std dev	max
Red-Black tree (baseline)	1000	24.00	26.17	1.05	28.00
	3000	87.00	89.20	1.52	92.00
	5000	151.00	157.10	3.04	163.00
AA-Tree, Node Copying	1000	99.00	113.23	3.87	124.00
	3000	351.0	582.6	49.5	633.0
	5000	596.0	703.1	24.6	752.0
BST, Node Copying	1000	39.00	42.23	2.34	53.00
	3000	140.00	146.17	5.19	159.00
	5000	237.0	251.3	10.1	268.0
BST, Path-Copying	1000	42.00	47.03	4.13	57.00
	3000	150.0	168.0	24.3	263.0
	5000	266.0	301.2	48.3	467.0

Table 2: Cputime (ms)

The cpu timings of the AA tree are longer than those of their unbalanced counterparts. This could be because of the extra overhead associated with the getters and setters employed in the AA tree implementation. Either way, the difference is a constant factor and does not affect its time complexity.

5.2 Worst case performance

	n	min	mean	std dev	max
Baseline Red-Black tree	1000	29336	47236	3469	48000
	3000	141872	143929	389	144000
	5000	240000	240000	0	240000
AA-Tree, Node-Copying	1000	370832	505154	71429	731696
	3000	1134672	1507562	223002	2289208
	5000	1869464	2559440	364014	3706656

Table 3: Memory usage (bytes)

In table 3 and 4, as with the average case results, node copying appears to require extra space by a constant factor of 10. The fact that the unbalanced search trees could not be benchmarked for this test speaks volumes as to the necessity of the node copying scheme for persistence.

	n	min	mean	std dev	max
Baseline Red-Black tree	1000	34.00	36.63	1.85	44.00
	3000	120.00	122.97	2.46	130.00
	5000	207.00	213.07	3.04	221.00
AA-Tree, Node-Copying	1000	84.00	86.63	2.34	92.00
	3000	269.0	281.6	7.1	293.0
	5000	483.0	501.3	16.2	551.0

Table 4: Cputime (ms)

6 References

Driscoll et al., 1989, "Making Data Structures Persistent", journal of computer and system sciences, vol. 38, no. 1, february 1989. `http://www.cs.cmu.edu/~sleator/papers/Persistence.htm`

Andersson, 1993, "Balanced Search Trees Made Simple" In Proc. Workshop on Algorithms and Data Structures, pages 60–71. Springer Verlag. `http://user.it.uu.se/~arnea/abs/simp.html`