A New Memory Layout for Self-Rebalancing Trees
Paul Iannetta

To cite this version:
Paul Iannetta. A New Memory Layout for Self-Rebalancing Trees. CGO’21, Feb 2021, Séoul, South Korea. hal-03123491
A New Memory Layout for Self-Rebalancing Trees

Paul Iannetta∗

Univ Lyon 1 UCBL
CNRS, ENS de Lyon, Inria,
LIP, F-69342, LYON Cedex 07, France
paul.iannetta@ens-lyon.fr

Abstract
In this paper, we show that trees implemented as a collection
of pointers suffer from a lack of parallelism opportunities.
We propose an alternative implementation based on arrays.
Both implementations appear to be equivalently efficient
time-wise. However, this new layout exposes new parallelism
opportunities which can be then exploited by an optimizing
compiler.

ACM Reference Format:
Paul Iannetta. 2021. A New Memory Layout for Self-Rebalancing
Trees. In Proceedings of xxxx (CGO’21). ACM, New York, NY, USA,
3 pages.

1 Introduction
Trees are pervasive in systems which need to be frequently
queried such as databases. They can take the form of T-Trees
[11] (a kind of balanced tree built on AVL trees [1, 9]) or
B-Trees [9]. Improving the processing of trees is a necessary
endeavor. A first step in this direction has been made by
Blelloch et al. [2, 12, 13] who investigated the benefits of
bulk operations to increase parallelism. However, we claim
that there is still room for optimizations in other directions,
such as cache locality. Here, we show that traditional imple-
mentations which store trees as collection of pointers can
be enhanced by using an array with a layout designed to
improve data locality and performance. This new memory
layout induces deep changes to the underlying mechanisms
(see Section 3). Our goal is that the underlying changes in
complexity gets amortized by better data locality and comp-
iler optimizations. In particular, not only we exhibit better
opportunities for vectorization, but also we plan to reuse
the ideas behind the polyhedral-model [5, 6, 8] (a framework
which aims at increasing the code locality of affine loops by
rescheduling their instructions using various methods such
as tiling and pipelining). As far as we know, the polyhedral
model has never been used to address programs using com-
plex data structures relying on pointers such as trees, apart
from the work of Feautrier and Cohen [3, 4, 7] which uses
algebraic languages to describe the iteration space over trees.
Our approach does not extend directly the polyhedral model
to tree-like data structures. Rather, it fits trees into arrays
and see to what extent the operations like insertion, find
or deletion can be written so as to fall within the reach of
the polyhedral model or how ideas behind the polyhedral
model can be reused in this context. Here we focus our pre-
sentation on the insertion operation and its parallelization
opportunities. Bulk operations is left for future work.

2 Breadth-first arrays
There are two natural categories of tree traversals: breadth-
first and depth-first traversals. Both induce an ordering on
the nodes which can be used to store the elements in an
array. On one hand, the numbering based on the depth-first
traversal provides no cheap way to recompute the structure
of the tree which is needed to perform insertions, deletions
and searches. On the other hand, the numbering induced by
the breadth-first traversal can easily store the structure of the
tree by keeping holes for unused nodes. This way, the num-
bering induces layers where the i-th layer is 2\(^i\)-wide.

3 Rotations on breadth-first arrays
Traditionally, a tree-rotation is a cheap operation which:
1. moves around two pointers and 2. updates the informa-
tion about the heights and the balance ratio of the nodes.
However, when trees are internally represented as arrays,
rotations become much more expensive because, now, part
of the array has to be actually moved from one memory loca-
tion to another. Hence, the cost of a rotation in the worst case
becomes O(n). This section describes each operation (left
and right rotation, but also left-right and right-left rotations)
as a sequence of low-level operations on breadth-first array,
namely shifts and pulls as well as their performance against
a traditional implementation.

3.1 Low-level operations on breadth-first
arrays
As presented in Section 2, breadth-first arrays provide a
convenient index scheme which allows to view the array as
a collection of layers.

∗This work was partially funded by the French National Agency of Research
in the CODAS Project (ANR-17-CE23-0004-01)
Left and right shifts (Figure 1a). A shift moves a subtree at a certain depth to the left or to the right. The tree is moved such that it is still on the same depth.

Pull up (Figure 1c). A pull up takes a subtree and graft it in place of its father.

Left and right pulls down (Figure 1b). A pull down takes a subtree and graft it at the place of its right (in the case of a right pull down) or left (in the case of a left pull down) children.

Those low-level operations can be used to implement the rotation by following the steps in the following table.

<table>
<thead>
<tr>
<th>Right</th>
<th>Left</th>
<th>Right-left</th>
<th>Left-right</th>
</tr>
</thead>
<tbody>
<tr>
<td>pull down $T_4$</td>
<td>pull down $T_1$</td>
<td>pull down $T_1$</td>
<td>pull down $T_1$</td>
</tr>
<tr>
<td>shift right $T_1$</td>
<td>shift left $T_1$</td>
<td>pull up $T_1$</td>
<td>pull up $T_1$</td>
</tr>
<tr>
<td>pull up $z$</td>
<td>pull up $z$</td>
<td>relabel $x, y, z$</td>
<td>relabel $x, y, z$</td>
</tr>
<tr>
<td>relabel $x, y, z$</td>
<td>relabel $x, y, z$</td>
<td>relabel $x, y, z$</td>
<td>relabel $x, y, z$</td>
</tr>
</tbody>
</table>

3.2 Performance analysis

The following table presents the results of the insertion of size random elements into a breadth-first-array-based AVL tree. Each experiment has been conducted 10 times and the table presents the average. The machine is an Intel® Core™ i5-5300U CPU @ 2.30GHz with 3072KB of cache.

Both (unoptimized) implementations have similar running times. The number of cache-misses with avl-tree (tree based implementation) is not constant despite what the measure notes tend to show in the table, avl-bf (array based implementation) cache-misses increase steadily as the density (d) (the number of occupied nodes divided by the total number of cells including holes) decrease. However, avl-bf can be compacted to mitigate this problem. The compression process transforms a sparse array into an array representing the densest binary search tree with the same elements, this can be done because there exists an equivalent, in this case avl tree, which is close to a perfect binary search tree. This process cost is $O(n)$ and can be performed each time the density reach a fixed threshold. The best value for this threshold is still unknown.

4 Parallelization opportunities

The implementation of the low-level operations can be optimized by applying standard loop transformations. The implementation of the shift operation is a parallel for, which means that it can be distributed over many cores easily. Pulls operations can also be optimized through the use of pipelining and tiling. More details about the internals of the optimization can be found in our research report [10].

5 Conclusion

We proposed a new memory layout for rotation-based balanced binary search trees and we have showed that despite the change in complexity of the underlying mechanisms, the
naive implementation of both strategies have similar running times and there are strategies to parallelize and further optimize the array-based implementation.

References