Timothy Holland
Seed number: 73

(a,b)-Tree Experimentation Report

Introduction

The objective of this report is to compare the relative performance of (2,3) and (2,4)-trees. The following three experiments are performed. First, an *insert* test, where n elements are inserted into the trees in random order. Second, the *min* test, where n elements are inserted sequentially, then for n iterations, the minimal element is removed and then inserted back. Third, the *random* test, where n elements are inserted sequentially, and then for n iterations, a random element is removed from the tree and a random element is inserted. Note, for the latter test, the removed element is always present, and the inserted element is not. These experiments are run for values of n from 256 to 55,108. All graphs plot n to the average structural changes to the tree, whereby a *structural change* is either a node split (in the case of insert) or the merging of two nodes (in the case of deletion). Through these plots, we will be able to understand the alternative maintenance required for the two (a,b)-trees. Overall, I will show that across all operations, (2,4)-trees requires less structural changes on average (by some small constant). Moreover, I will illustrate two major theorems, the first stating a sequence of m insertions for (a, b)-trees has an amortised big-Θ constant cost. Second, that a sequecne of insertion, and deletions has only an amortised big-Θ constant cost for (a, 2a)-trees. Thus, in the latter theorem, we will see another major difference between these two trees. For each experiment, I will state the results first and then explain their outcomes through theorems proven on (a,b)-trees.

Insert Test

For the first test, *insert*, from an empty tree n elements are inserted in random order. The results are shown in figure 1, mapping n to the average structural changes per operation:

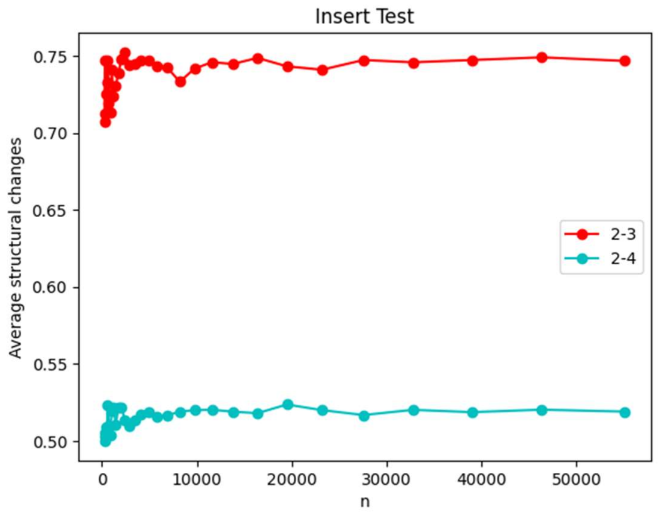

Figure 1: Insert Test

To begin, in terms of time complexity we know from the lectures that search , we know that the worst case time complexity of inserting a node is where we visit at least $\Theta(1)$ nodes on each level of a $O(\log_a n) = O(\frac{\log(n)}{\log(a)})$ height tree, and insert therby spends $\theta(b)$ time on such node, resulting in overal time complexity:

$$\Theta\left(\frac{b\log(n)}{\log(a)}\right)$$

A tight bound exists on searching for a node, which cannot contribute to amortised analysis. However, we can explore how the modifications made in (a, b)-trees amortise, hence our focus on considering average structural changes per operation. Evidently, in the above plot, there is a performance difference between the two trees, with 2-4 on average performing 0.25 less structural changes per n, but their trajectories remain contant across values of n. The beginning cluster of each curve can be attributed to fluctations in the shape of smaller trees, but these smooth out over n, whereby each tree maintains a constant cost in number of structural changes required. This is consistent with the first amortised theorem stated in the lectures, that *a sequence of m Inserts on an initially empty (a,b)-tree performs O(m) node modifications*.

How is this the case? Well, by definition each insert performs a sequence of node splits (which is possibly empty), and then modifies one node. The modification is constant, O(1). A split modifies two nodes, one that already exists, and one created. As we're adding to the tree, an insert can only *add* a new node. Moreover, the number of splits is bounded by the number of nodes, which means it cannot be greater than m. Thus, a sequence of m inserts remains constant, and supports the above evidence.

The question remains as to the performance difference between the two trees? Although no theorem explicitly states this, we can hypothesise that because (2,4) trees have an extra key per node, they therefore have less nodes per n elements, resulting in a smaller upper bound on number of nodes in the tree, thus reducing the number of structural operations by some constant factor. Whilst trees with higher b therbeby have smaller changes on average, their complexity increases in the search of each node, a cost not present in the above evidence.

Min Test

For the second test, *min*, n elements are sequentially inserted into a tree, then for n iterations, the minimal element is removed and then inserted back. The results are shown in figure 2, mapping n onto average structural changes per operation:

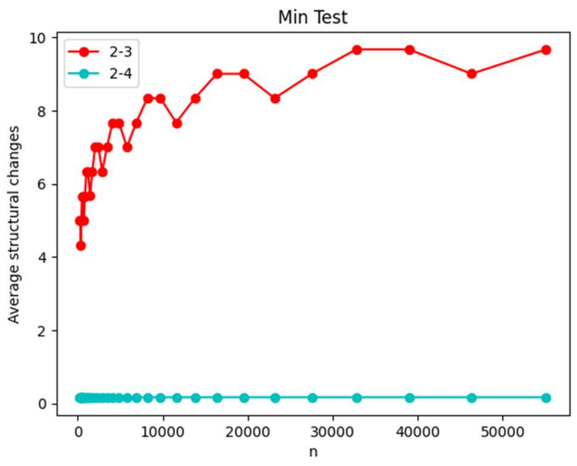

Figure 2: Min Test

In this test we encounter a new method, *deletion* from the tree, with the corollary structural change of mergining nodes. From the theory stated in the lectures, we know that the time complexity of such an operation is the same as insertion $\Theta(\frac{blog(n)}{\log(a)})$. Again, the node must be found through a search operation, then through merges, visiting at most two nodes per level $\Theta(1)$, spending at most $\Theta(b)$ time per node in managing keys.

Once more, a tight bound exists on the efficiency of search, which dominates this complexity. The question once more arises as to how the combination of insertion and deletion amortises in terms of operations? Considering the above evidence, as per the insertion test (2,4) trees remain constant in number of operations needed, whereas the (2,3) tree becomes logarithmic. This remains consistent with the next amortization theorem stated in the lectures, *a sequence of m Inserts and Deletes on an initially empty (a, 2a)-tree performs O(m) node modifications*.

Indeed, it just so happens that for (2,3) trees, the worst case sequence of actions is repeated insertion and deletions of certain elements (in this case the minimum). This sequence breaks from the first theorem, as the tree fails to gain potential by distributing the costs between operations. Indeed, in such an instance, the insertion, then deletion results in a splitting of nodes all the way up to the root, then a merging of nodes all the way up to the root, making the worst case logarithmic instead of constant.

Why, then, does the (2,4)-tree avoid this fate under the above theorem? Well, this structure adds ‘breathing-room’ to the above test, avoiding the potential for oscillations between two states. Consider this by induction, through an example of an insert, followed by a delete, followed by an insert of the minimum. Assuming the node already has four leaves, insertion creates an overflow condition, whereby it must split into two pairs of 2, each with three leaves. When the delete operation occurs, it removes the minimum element, leaving two nodes of one and two keys respectively. Upon the next insertion, the element simply returns to its previous position, and the next deletion would remove it once again. Thus, despite an initial restructuring of the tree, the loss of potential is recompensated for by a sequence of constant modifications, resulting in constant structural changes in the number of operations.

Random Test

For the last test, *random*, n elements are inserted sequentially, then for n iterations, a random element is removed from the tree and a random element is inserted. The experiment can be visualised in Figure 3, mapping n to the average number of structural changes per operation:

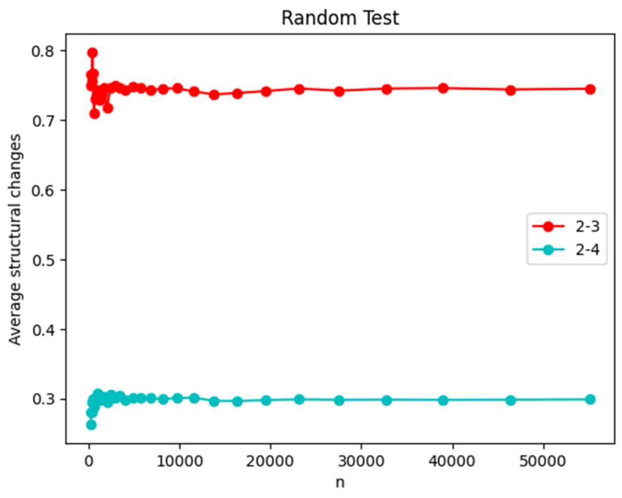

Figure 3: Random Test

From the previous two experiments, I have already stated proven why an insertion of n elements has a constant big-Θ amortised cost across structural changes for all (a, b)-trees, and why a sequence of insert-delete operations only maintains a constant big-Θ amortised cost in (a, 2a)-trees. In this test, we are once again considering a sequence of insertion and deletion operations. Thus, the central question is why does the performance of both trees appear constant? In particular, why does the (2, 3)-tree return to its performance as stated in the first theorem?

The second theorem allowed us to state a tight bound on the constancy of operations for (a, 2a) trees, and the *min* experiment illustrated that for (2, 3)-trees the upper bound big-0 is actually logarithmic. In this experiment, the tree returns to constant performance across all iterations. We can ascribe this to the normalised distribution of insertions and deletions in a sequence. In the previous experiment, the *worst-case* was repeated, thus generating a logarithmic scale. Within this experiment however, different elements are inserted and different elements are deleted. Thus, the worst-case of repeated oscillation over contraction and expansion conditions of the tree are averaged out over the normalised sequence of operations. We can thus conclude that despite the upper bound, the (2, 3)-tree is constant in the sequence of insertion-deletion operations, *on average*.

One last thing stands out in this graph, which is actually present in the min test but wasn't evident because of scale. For the (2, 4)-tree, the average number of operations from a sequence of inserts decreases from around 0.53 to 0.3 when the sequence becomes a combination of insertions and deletions. We can hypothesise as to the answer. For the first *insert* test, the average structural changes is a representation of a tree that is constantly expanding. Whereas, for the min and random test, the average structural changes is a representation of a tree that remains the same size. Thus, for the former test, the upper threshold of overflow invariably occurs when inserting a sequence of elements. However, for a sequence of insert-delete operations, this overflow condition isn't necessarily occurring across the n operations (although it is probabilistically *likely*). Thereby, we see a constant factor improvement of a sequence of insert-delete operations compared to a sequence of insert operations.