Practical B-tree design

NEIL A. EDELMAN

2022-08-05

Abstract

Several design decisions for a minimal in-memory B-tree are discussed, covering bulk-loading, insertion, and deletion. Any allocations are fail-fast before modifying the tree so that it remains in a valid state. Concurrency not considered.

1 DESIGN

A tree is used as an ordered set or map. For memory locality, this is implemented B-tree[1]. In an implementation for C, we would expect memory usage to be low, performance to be high, and simplicity over complexity. Practically, this means that B^+-trees and B^*-trees are less attractive, along with an added layer of order statistic tree. The nodes are linked one-way and iteration is very simple. This precludes multi-maps. The use-case has no concurrency and places importance on modification with operations being as lazy as is reasonably possible.

1.1 Branching factor

The branching factor, or order as [2], is a fixed value between $[3, \mathrm{UINT\_MAX} + 1]$. The implementation has no buffering or middle-memory management. Thus, a high-order means greater memory allocation granularity, leading to asymptotically desirable trees. Small values produce much more compact trees. In general, it is left-leaning where convenient because keys on the right side are faster to move because of the array configuration of the nodes.

1.2 Minimum and maximum keys

We use fixed-length nodes. In [3], these are (a, b)-trees as $(\mathrm{minimum} + 1, \mathrm{maxumum} + 1)$-trees. That is, the maximum is the node's key capacity. Since the keys can be thought of as an implicit complete binary tree, necessarily maximum+1 is the order. Unlike [2], we differentiate by branch and leaf; a leaf node has no need for null-children, so we don't include them.

Performance will be $\mathcal{O}(\log \mathrm{size})$. Equation 1 gives the standard B-tree key minimum in terms of the key maximum. (Equivalent to the specifying the minimum children as a function of the order.)

$$\mathrm{minimum} = \left\lfloor \frac{\mathrm{maximum}}{2} \right\rfloor \tag{1}$$

Equation 1 represents the maximum minimum without borrowing from the siblings. However, freeing at empty gives good results in [4]. We compromise with Equation 2.

(a) Bulk add 1, 2. (b) Bulk add 3.

(c) Finalize after 3. (d) Or bulk add 4.

Figure 1: Order-3, maximum keys 2, bulk-addition, with labels for nodes. 1a. Keys 1, 2: full tree. 1b. Adding 3 increases the height, with new nodes (branch) `Snamog` and (leaf) `Mausnar`. Minimum invariant is violated for `Mausnar`. 1c. Finalize would balance all the right nodes below the root. 1d. Or continue adding 4 to 1b.

Designed to be less-eager and provides some hysteresis while balancing asymptotic performance.

$$\text{minimum} = \max\left(1, \left\lfloor \frac{\text{maximum}}{3} \right\rfloor\right) \tag{2}$$

In a sense, it is the opposite of a B*-tree[2, 5], where $\frac{1}{3}$ instead of $\frac{2}{3}$ of the capacity is full. It doesn't affect the shape of the tree unless deletions are performed, where instead of being stricter, it is lazier.

2 OPERATIONS

2.1 Bulk loading

Bulk loading buffers, as described in [6], is too complex for our purposes. Here, the user must supply the keys in order. Such that, a key undergoing bulk-add is packed to the lowest height on the right side, ignoring the rules for splitting. A key recruits zero-key nodes below as needed. When the tree is full, the height increases.

Because the B-tree rule for the minimum number of keys may be violated on bulk-add, it's important to finalize the tree. This balances and restores the B-tree invariants on the right side of the tree, under the root.

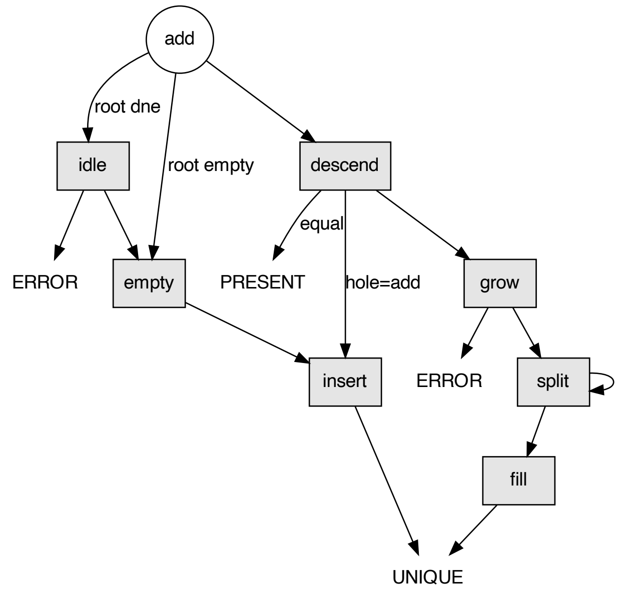


Figure 2: The requirements for `add` are $\mathcal{O}(1)$ space and $\mathcal{O}(\log \text{size})$ time, with no parent pointers. State diagram of adding a key, traversing a maximum twice.

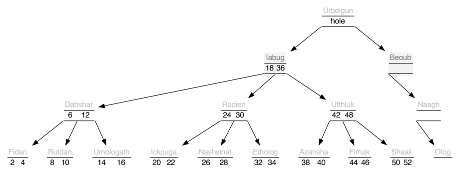
Initially, this will efficiently produce a more compact tree. For example, Figure 1 shows bulk addition of natural numbers in order. Compare adding them normally: after a split, there's not any more keys to be added on the low side; this asymptotically results in one-half occupancy.

2.2 Adding a key

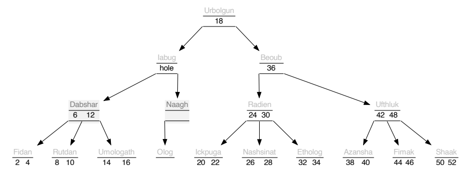
In [1, 2, 5], insertion is a $\mathcal{O}(\log_{\text{minimum}+1} \text{size})$ operation. In theory, the key is inserted in a leaf node. If this operation causes the leaf to be overfull, the leaf is split into two, with the median element promoted. This repeats until all the nodes are within the specified key maximum. If implemented directly, this requires an extra temporary element to be added to each node and double-linking to access the parent node.

It is advantageous, then, to flip the insertion and keep track of the hole, getting rid of the temporary size overload and moving the intermediate keys only once. A potential state machine is shown schematically in Figure 2. `idle` is no dynamic memory; `empty` contains an unused node. `ERROR` from `idle` and `grow` is a memory error; the state of the tree remains unchanged.

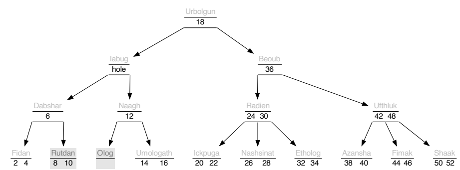
The `descend` path is taken by any non-empty tree, and descends to find the space in the leaf that it will go. It stops with `PRESENT` if it finds an already present match. When it completes finding a new key in a leaf node, the `hole` will be the lowest-height node that has free space in the path from the root to that leaf.



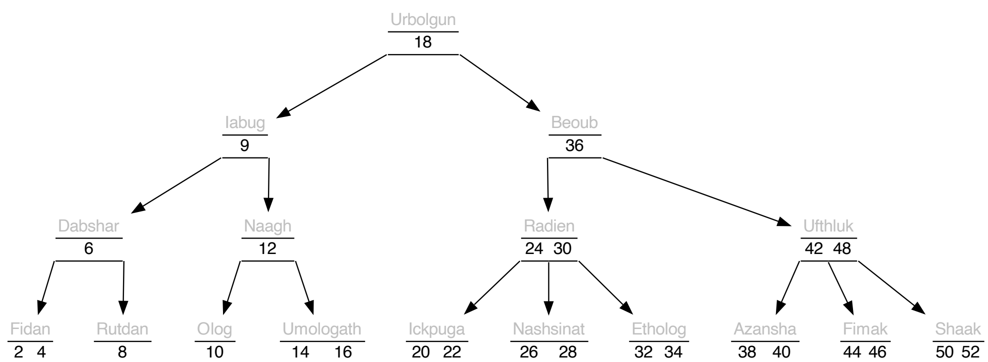
(a) Add 9; four new nodes, the first split.



(b) After one split.



(c) Further split.



(d) After fill.

Figure 3: Full order-3 tree: addition of any number will cause the tree height to increase. 3a – 3c: working down the tree in `split`. 3d after `fill`.

A non-empty leaf node results in the `insert` path, and a single time down the tree. Specifying a high-order makes taking this path more likely than having to repeat the `grow` path.

An add must `grow` if it has the maximum keys in the leaf node. The height of the `hole` (zero-based) is the number of nodes extra that need to be reserved. A null `hole` means all of the path is full of keys; this requires increased tree height: tree height + 2 new nodes. These new empty nodes are then strung together and added to the tree.

`split` introduces `cursor`, along the path to the leaf, and `sibling`, the new node ancestry. The `cursor` is initially the `hole`; it descends a second time, balancing with `sibling`, thus splits it with the new node. `hole` can either go to the left, right, or exactly in between. `cursor` descends on the path, and `sibling` descends on the new

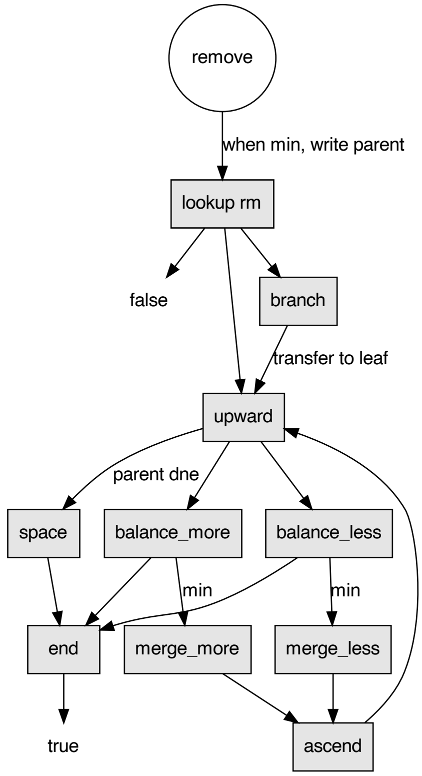


Figure 4: `remove` is the opposite of `add` in Figure 2. `rm` starts at the bottom; works up to find any node or siblings that have excess.

nodes. In `fill`, the added key then goes in `hole`.

In Figure 3, the path from before 3a to 3d to insert 9 on Figure 2 is `descend`, `grow`, `split` four times (the maximum for this size), `fill`, and returns `UNIQUE`. The new nodes are (branches) `Urbolgun`, `Beoub`, `Naagh`, and (leaf) `Olog`. This is the most complicated path, when the height increases.

2.3 Deleting a key

Figure 4 shows a schematic diagram of removal of one key . . .

intrim? provisional!

3 CHARACTERISTICS

3.1 Map

When the tree set has an extra value *per* key that is not associated with the tree operations, it becomes a map. To facilitate cache performance, these values are not

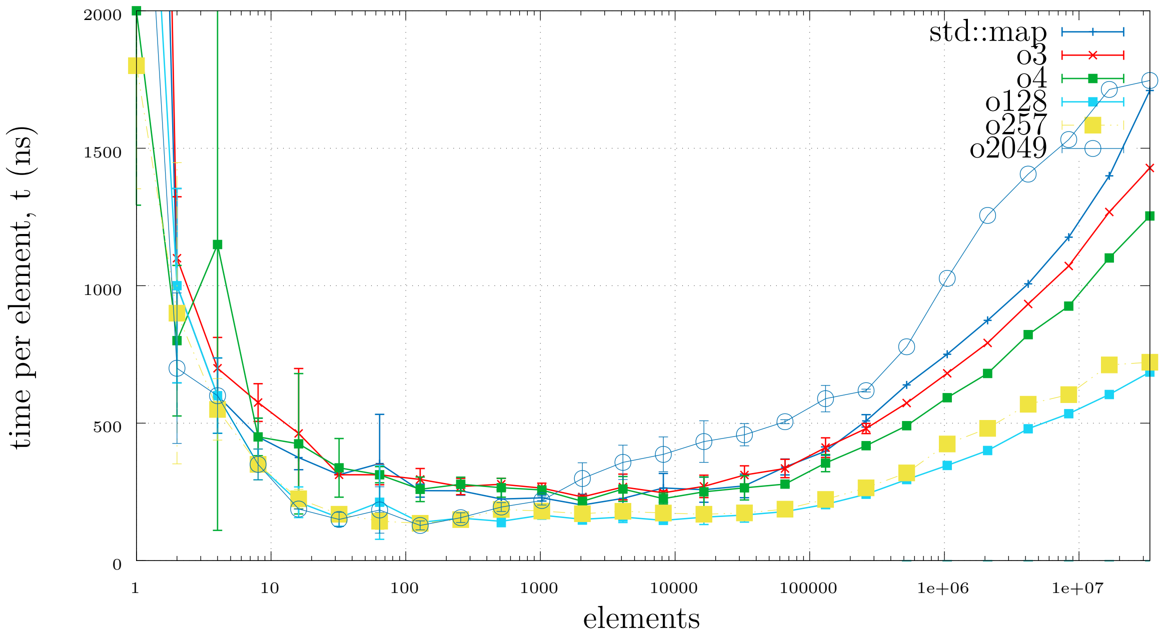


Figure 5: A comparison different orders and `std::map` in C^{++} benchmark, with a log-x scale.

interspersed with the keys. On lookup, only the key's value is ever accessed or iterated over.

3.2 Performance

Figure 5 shows the common case where one has an ordered map of `int` to pointer. It is of straight insertion of random elements, (not in any order.)

`std::map` is a red-black tree in `C++`. Though order 4 produces an isomorphism with left-leaning red-black trees[7], it is very close in performance to an order-3 tree. This is probably because of the importance of caching: locality-of-reference in order-3 trees, which have up to 2 elements grouped together is very similar to having single elements.

Higher orders are generally more cache-friendly, but it gets to the point where all the data is in one node. This node is essentially an ordered array. Order 2049 is an example; with more data to be transferred every insertion, the performance goes down. The more the size in bytes of the individual keys (and values, if applicable,) the more large the nodes become, and this negative effect is exacerbated.

When choosing a default fixed order, it is useful to pick a fixed number of elements from Figure 5 . . .

4 CONCLUSION

REFERENCES

[1] R. Bayer and E. McCreight, "Organization and maintainence of large ordered indices," *Acta Informatica*, vol. 1, no. 3, p. 1, 1972.

[2] D. E. Knuth, *The Art of Computer Programming, Sorting and Searching*, vol. 3. Addison-Wesley, Reading, Massachusetts, 2 ed., 1998.

[3] M. T. Goodrich, R. Tamassia, and D. M. Mount, *Data structures and algorithms in C++*. John Wiley & Sons, 2011.

[4] T. Johnson and D. Shasha, "B-trees with inserts and deletes: Why free-at-empty is better than merge-at-half," *Journal of Computer and System Sciences*, vol. 47, no. 1, pp. 45–76, 1993.

[5] D. Comer, "Ubiquitous b-tree," *ACM Computing Surveys (CSUR)*, vol. 11, no. 2, pp. 121–137, 1979.

[6] G. Graefe *et al.*, "Modern b-tree techniques," *Foundations and Trends® in Databases*, vol. 3, no. 4, 2011.

[7] R. Sedgewick, "Left-leaning red-black trees," in *Dagstuhl Workshop on Data Structures*, vol. 17, 2008.