

B-tree design

NEIL A. EDELMAN

2022-08-05

Abstract

Several design decisions for a practical B-tree are discussed.

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.

1.1 Branching factor

The branching factor, or order as [2], is a fixed value between $[3, \text{UCHAR_MAX} + 1]$. 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. Four produces an isomorphism with left-leaning red-black trees[3]. 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 [4], these are (a, b) -trees as $(\text{minimum} + 1, \text{maximum} + 1)$ -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 $\text{maximum} + 1$ 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 $\mathcal{O}(\log_{\text{minimum}+1} \text{size})$. Equation 1 gives the standard B-tree minimum in terms of the maximum.

$$\text{minimum} = \left\lceil \frac{\text{maximum} + 1}{2} \right\rceil - 1 \quad (1)$$

Freeing at empty gives good results in [5]. We compromise with Equation 2. Designed to be less-eager and provides some hysteresis while balancing asymptotic performance.

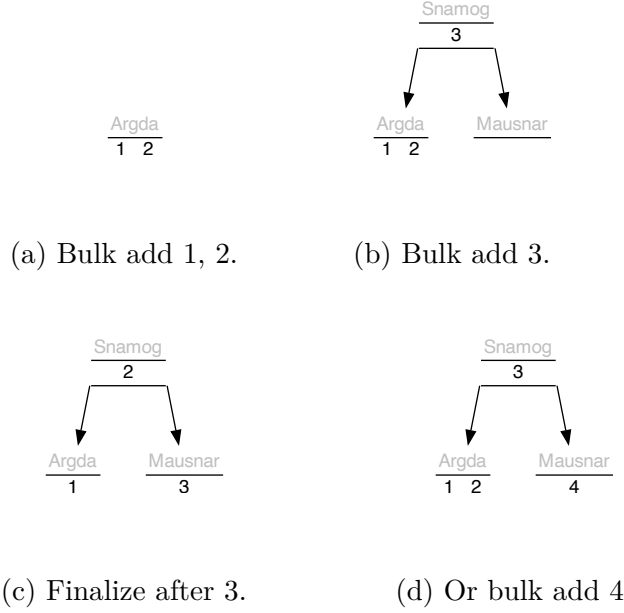


Figure 1: Order-3, maximum keys 2, bulk-addition, with labels for nodes. [1a](#). keys 1, 2: full tree. [1b](#). adding 3 increases the height. Minimum invariant is violated for **Mausnar**. [1c](#). finalize would balance all the right nodes below the root. [1d](#). or continue adding 4 to [1b](#).

$$\text{minimum} = \left\lceil \frac{\text{maximum} + 1}{3} \right\rceil - 1 \quad (2)$$

In a sense, it is the opposite of a B*-tree[2, 6], where $\frac{1}{3}$ instead of $\frac{2}{3}$ of the capacity is full: instead of being stricter, it is lazier.

1.3 Bulk loading

Bulk loading is ordered addition such that the key is always the maximum in the tree so far. Initially, this will 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 on the low side, asymptotically resulting in one-half occupancy.

Bulk-add will add the key to the maximum side at the lowest level where there is space, ignoring the rules for splitting. If this is a non-leaf, additional nodes with no-keys and one child are recruited. If there is no space at all on the maximum side, another level is made. Because the B-tree rules may be violated, it is important to finalize the tree where bulk-addition has occurred; this balances and restores the B-tree invariants on the right side of the tree under the root.

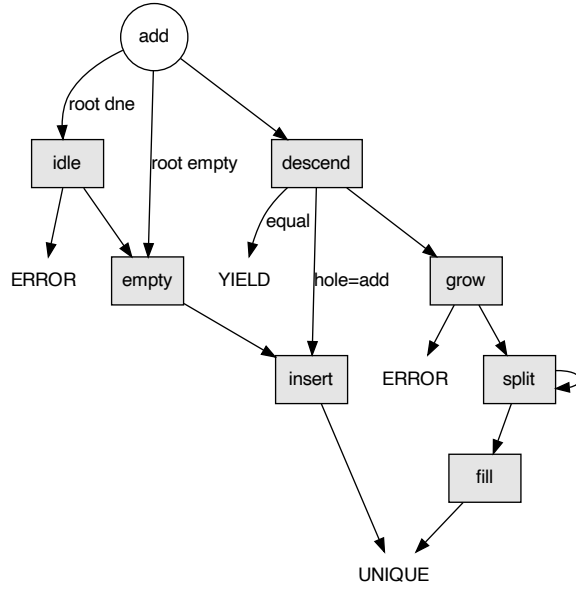


Figure 2: The requirements for `add` are $\mathcal{O}(1)$ space and $\mathcal{O}(\log \text{size})$ time, with no parent pointers. State diagram of adding a key, traversing a maximum twice.

1.4 Adding a key

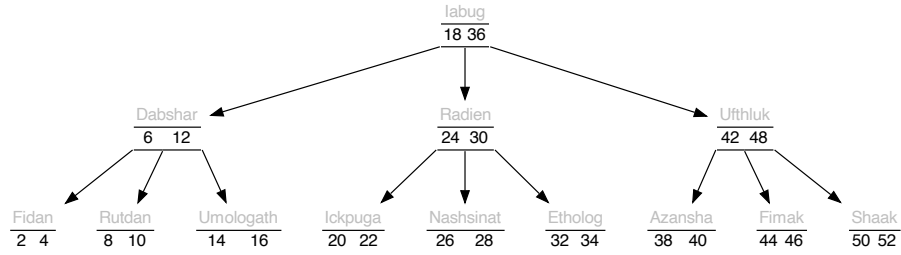
Adding a key is shown schematically in Figure 2. `idle` is no dynamic memory; `empty` contains an unused node. `ERROR` from `idle` and `grow` is a memory error; the state of the tree remains unchanged.

The `descend` path is taken by any non-empty tree, and descends the tree to find the space in the leaf that it will go. It stops with `YIELD` 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.

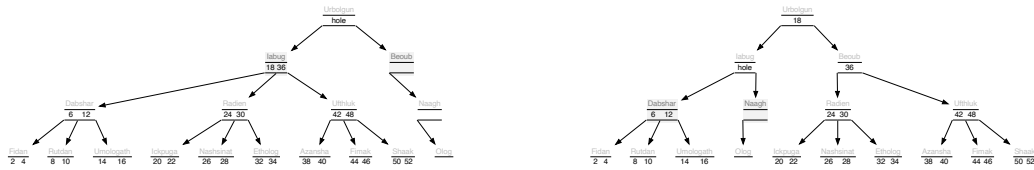
A non-empty leaf node results in the `insert` path, and a single time down the tree; specifying a high-order makes taking this path more likely than having to repeat, but makes the reservation of keys more aggressive.

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 nodes. These new nodes are strung together and added to the tree.

`split` introduces `cursor`, along the path to the leaf, and `sibling`, the new zero-key node. in the path to the key and expands it to include `hole`, thus splits it with the popped item from the stack. `hole` can either go to the left, right, or exactly in between. Now `cursor` does not have a full node, so it descends height on the path. This repeats while the stack is not empty. In fill, the added key then goes in `hole`.

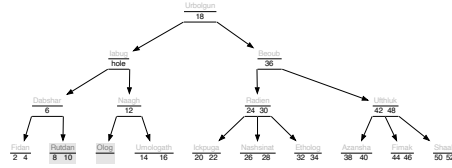


(a) Full tree of even numbers.

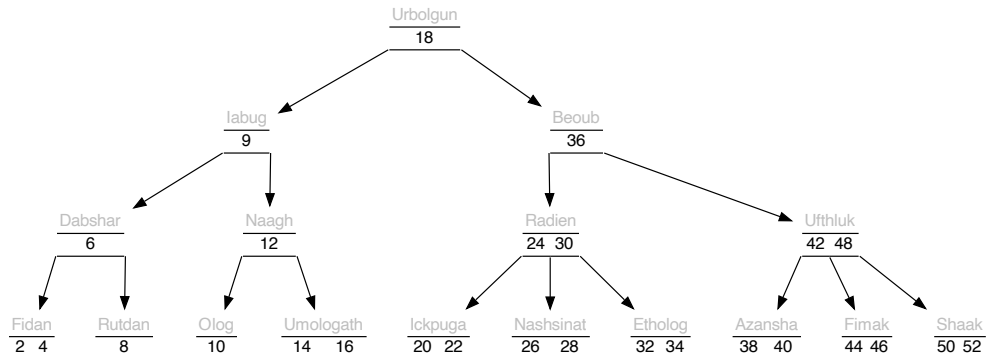


(b) Add 9; four new nodes, the first split.

(c) After one split.



(d) Further split.



(e) After fill.

Figure 3: 3a: order-3 full tree. 3b – 3d: working down the tree. 3e: addition of any number will cause the tree height to increase.

In Figure 3, the path from 3a to 3e 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.

1.5 Deleting a key

2 PERFORMANCE

3 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] R. Sedgewick, “Left-leaning red-black trees,” in *Dagstuhl Workshop on Data Structures*, vol. 17, 2008.
- [4] M. T. Goodrich, R. Tamassia, and D. M. Mount, *Data structures and algorithms in C++*. John Wiley & Sons, 2011.
- [5] 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.
- [6] C. Douglas, “Ubiquitous b-tree,” *ACM Computing Surveys*, vol. 11, no. 2, pp. 121–137, 1979.