B-tree design

NEIL A. EDELMAN

2022-08-05

Abstract

Several design decisions for a 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 $[3, \mathrm{UCHAR\_MAX} + 1]$. 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.

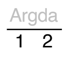
1.2 Minimum and maximum keys

We use fixed-length nodes. In [4], these are (a, b)-trees as $(\mathrm{minimum} + 1, \mathrm{maxumum} + 1)$-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.

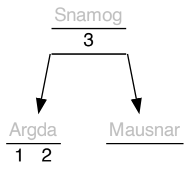
Performance will be $\mathcal{O}(\log_{\mathrm{minimum}+1} \mathrm{size})$. Equation 1 gives the standard B-tree minimum in terms of the maximum.

$$\mathrm{minimum} = \left\lceil \frac{\mathrm{maximum} + 1}{2} \right\rceil - 1 \tag{1}$$

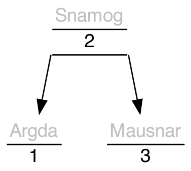
Freeing at empty gives good results in [5]. We compromise with Equation 2. Designed to be less-eager and provides some hysteresis while balancing asymptotic performance.



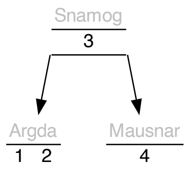
(a) Bulk add 1, 2.



(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. Minimum invariant is violated for `Mausnar`. 1c. finalize would balance all the right nodes below the root. 1d. or continue adding 4 to 1b.

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

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.

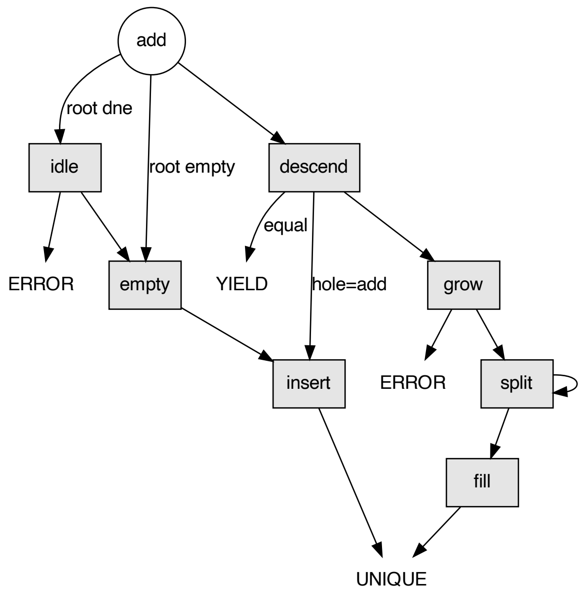


Figure 2: The requirements for `add` are $\mathcal{O}(1)$ space and $\mathcal{O}(\log \text{size})$ time, with no parent pointers. State diagram of adding a key, traversing a maximum twice.

1.4 Adding a key

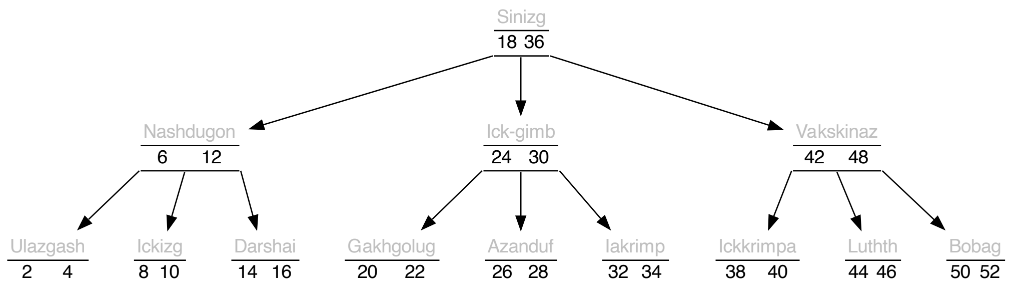
Adding a key is shown schematically in Figure 2. `idle` is no dynamic memory; `empty` contains an unused node. `ERROR` from `idle` and `grow` is a memory error; the state of the tree remains unchanged.

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.

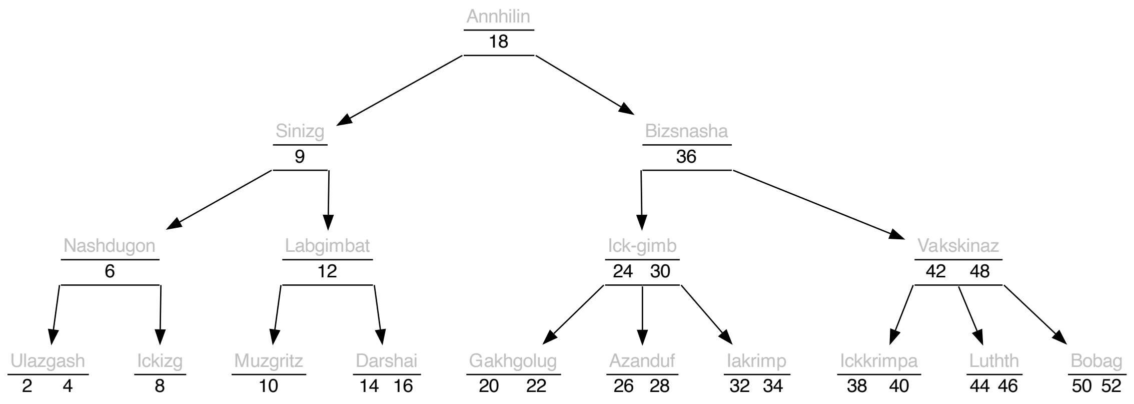
A non-empty leaf node results in the `insert` path, and a single time down the tree; specifying a high-order makes taking this path more likely than having to repeat, but makes the reservation of keys more aggressive.

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, and tree height + 2 nodes. We do pre-allocation so that no exceptions can occur while we modify the tree. The reserved nodes form a stack. We introduce cursor that starts as hole and moves down the tree. (fixme: no! that's not what happens. It is attached more-then with zero keys then balance hole and new-old node.)

`split` takes the full node below `cursor` 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



(a) Full tree of even numbers.



(b) Add 9.

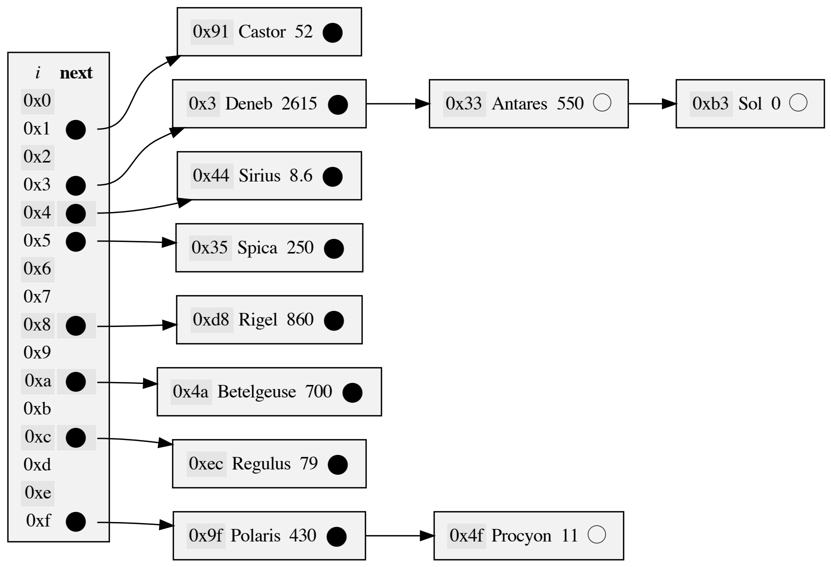
Figure 3: 3a. order-3 full tree. 3b. addition of any number will cause the tree height to increase.

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`.

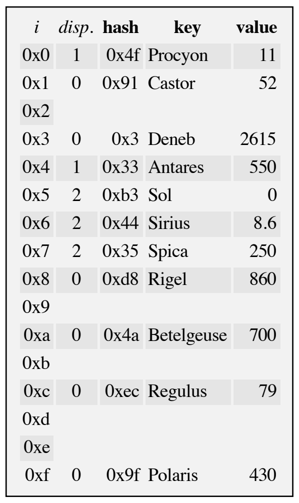
In Figure 3, the path from 3a to 3b to insert 9 on Figure 2 is `descend`, `grow`, `split` four times (the maximum for this size), `fill`, and returns `UNIQUE`. The new nodes are (branches) `Annhilin`, `Bizsnasha`, `Labgimbat`, and (leaf) `Muzgritz`.

The path of `hole` is...

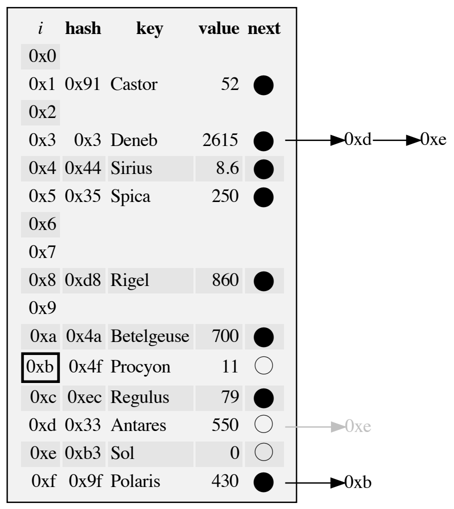
1.5 Deleting a key



(a) Separate-chaining.



i	*disp.*	**hash**	**key**	**value**
0x0	1	0x4f	Procyon	11
0x1	0	0x91	Castor	52
0x2				
0x3	0	0x3	Deneb	2615
0x4	1	0x33	Antares	550
0x5	2	0xb3	Sol	0
0x6	2	0x44	Sirius	8.6
0x7	2	0x35	Spica	250
0x8	0	0xd8	Rigel	860
0x9				
0xa	0	0x4a	Betelgeuse	700
0xb				
0xc	0	0xec	Regulus	79
0xd				
0xe				
0xf	0	0x9f	Polaris	430



i	**hash**	**key**	**value**	**next**
0x0				
0x1	0x91	Castor	52	●
0x2				
0x3	0x3	Deneb	2615	●
0x4	0x44	Sirius	8.6	●
0x5	0x35	Spica	250	●
0x6				
0x7				
0x8	0xd8	Rigel	860	●
0x9				
0xa	0x4a	Betelgeuse	700	●
0xb	0x4f	Procyon	11	○
0xc	0xec	Regulus	79	●
0xd	0x33	Antares	550	○
0xe	0xb3	Sol	0	○
0xf	0x9f	Polaris	430	●

(b) Open-addressing.

(c) Inline-chaining.

Figure 4: A map from some star names to light-year distance using different collision-resolution schemes. Load factor $^{11}\!/_{16} = 0.69$. Expected value: chained number of queries, 1.4(7); open probe-length, 1.6(9), (standard deviation.) Order of insertion: Sol, Sirius, Rigel, Procyon, Betelgeuse, Antares, Spica, Deneb, Regulus, Castor, Polaris.

2 EXAMPLE COMPARISON

Figure 4 shows a comparison of some standard hash-table types. It uses D.J. Bernstein's *djb2* to hash a string to 8-bit unsigned integer. All tables use a most-recently-used heuristic as probe-order; experimentally, this was found to make little difference in the run-time, and is advantageous when the access pattern is non-uniform[7, 8].

Separate-chaining is seen in Figure 4a; this is more like T.D. Hanson's *uthash*: only being in one hash-table at a time. A similar hash to C^{++}'s `std::unordered_map`, *Lua*'s `table`, and many others, would have another dereference between the linked-list and the entry. This style of hash-table allows unconstrained load factors. With ordered data, keeping a self-balancing tree cuts down the worst case to $\mathcal{O}(\log n)$[9], as in *Java*. The expected number of dereferences is a constant added to the number of queries. The nodes are allocated separately from hash-table.

Open-addressing[10] as seen in Figure 4b, is another, more compact, and generally more cache-coherent table design. Robin Hood hashing[11] has been used to keep the variation in the query length to a minimum; here, with the condition on whether to evict strengthened because of the most-recently-used heuristic. It has lower maximum load-factor, because clustering decreases performance as the load-factor reaches saturation.[12] Although they have less data *per* entry, on average they have more entries. Practically, 0.69 is high; *Python*'s `dict` [9] uses a maximum of $^2/_3$. One can calculate the displacement from the hash, but we have to have a general way of telling if it's null. The lack of symmetry presents a difficulty removing entries.

Inline-chaining, as seen in Figure 4c, is, in many ways, a hybrid between the two. The dark circles represent the closed heads, and the outline the open stack, with a highlight to indicate the stack position. The expected probe-length is number of chained queries. Because of the `next` field, the space taken is one index *per* entry more then open-addressing. Being chained offers a higher load-factor, but it is not possible to exceed one: like open-addressing, the hash-table is contained in one block of memory.

3 PERFORMANCE

Because the closed and open entries are orthogonal for inline-chaining, the limiting factor in the worst-case is the same as for separate-chaining, and it will have identical behaviour as long as the load-factor doesn't exceed one. In the average case, we do at least as much work by a constant factor; in addition to the steps required for chaining, we also have to sometimes also have to manage the stack. Moving the top of the stack involves finding the closed head of the top entry and iterating until the top. However, modification requires only copying one entry; we aren't concerned with the order of the stack, only the order of the next indices. However, inline-chaining does have the advantage of cache-locality.

Figure 5 benchmarks straight insertion on a closed separately-chained set like Figure 4a, an inline-chained set like Figure 4c, and a `std::unordered_set`. The data is a pointer to a randomly generated set of non-sense names, formed out of syllables, Poisson-distributed up to 15-letters in length; a `char[16]` with a null-terminator.

Figure 5: A comparison of chained techniques in a C^{++} benchmark with a log-x scale.

Hashed by *djb2* to a 64-bit `size_t`.

The same data is used for each replica across different methods, thus the same number of duplicates were ignored. The hash-tables were then destroyed for the next replica. The data is in a memory pool. Pre-allocation of the nodes of the separately-chained is done in an array; this is not counted towards the run-time; for the `unordered_set`, however, this is transparent, and is timed. For the inline-chaining, it's behaviour with respect to allocation is like open-addressing: it is all contained it in one array.

4 IMPLEMENTATION

This section talks about the specific implementation of inline-chained hash-table whose results are shown in Figure 5.

4.1 Next Entry

The `next` field in offers a convenient place to store out-of-band information without imposing restrictions on the key; specifically, we do not assume that it is non-zero. There are two special values that must be differentiated from the indices: there is no closed value associated with this address, called `NULL`, and there is no next value in the bucket, called `END`.

Since the `next` field must store the range of addressable buckets, minus one for itself, this is one short of representing the whole range. Since the implementation uses power-of-two resizes, it causes the addressable space to be one-bit less; we waste nearly a bit, half the size, or the equivalent of a signed integer. The values `NULL` and `END` were chosen to minimize average power requirements while leaving a natural $[0, 011..11]$ for addressing. That is, 100..00, and 100..01. In Figure 4c, where there are 8 bits, the addressable space is $[0, 127]$, and `NULL`, `END` are 128, 129. The `next` theoretically takes up one-byte, but alignment issues will cause it to take more.

Since the `top` is an address, and the addresses only go to half the available space, we have a bit to spare. This has been used for a lazy stack. In this way, repeatedly adding and deleting to the open stack just sets the lazy bit. Only when one deletes twice does it break hysteresis loop, a move is forced.

5 CONCLUSION

This specific data for the average use-case shows the difference between separate-chaining and inline-chaining is not great enough to matter to performance, and even helps in some cases. For situations where one needs a new hash-table, the simplicity of inline-chaining is appealing.

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. Knuth, “Sorting and searching. third edn. volume 3 of the art of computer programming,” 1997.

[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.

[7] R. P. Brent, “Reducing the retrieval time of scatter storage techniques,” *Communications of the ACM*, vol. 16, pp. 105–109, Feb 1973.

[8] D. D. Sleator and R. E. Tarjan, “Self-adjusting binary search trees,” *Journal of the ACM (JACM)*, vol. 32, no. 3, pp. 652–686, 1985.

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

[10] W. W. Peterson, “Addressing for random-access storage,” *IBM Journal of Research and Development*, vol. 1, pp. 130–146, April 1957.

[11] P. Celis, P.-A. Larson, and J. I. Munro, “Robin hood hashing,” in *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pp. 281–288, IEEE, 1985.

[12] S. S. Skiena, *The algorithm design manual*, vol. 2. Springer, 2008.