B-trees

Let’s invent B(+)-trees.

Say we have some things, and we want to be able to store and look them up in order. We could just put them in a big sorted array:

```
10,20,30,40,50,60,70
```

This is great, but it’s hard to mutate – to insert or delete an element, we might have to move all the others!

So, instead, let’s split our array into fixed-size blocks – which can be in any order – and keep references to the blocks in sorted order:

```
[10,20,30,40] [50,60,70,--]
```

Each block is allowed to have some empty space, which we can use for insertions.

If we inserted 55 above, we’d get:

```
[10,20,30,40] [50,55,60,70]
```

So we only need to move up to a blocksworth of elements to insert into a block. What if the block we want to insert into is full? We can split a block into two halves. Say we want to insert 15 above:

```
[10,20,30,40] [50,55,60,70]
[10,20,--,--] [30,40,--,--] [50,55,60,70]
[10,15,20,--] [30,40,--,--] [50,55,60,70]
```

When we split a full block, it becomes half-empty. In order not to have too many blocks, we don’t permit blocks to be any emptier than that (which means at most 50% of block space is wasted).

(Deletion is mostly like insertion: If, after deleting an element, a block is less than half-full, we either merge it with its neighbor if they’re both half-full, or we move an element over from its neighbor. If it has no neighbors, we just leave it as it is.)

The next question is: How do we store the sorted list of blocks, which there might be a lot of? We can just use the same structure: Blocks indexing blocks indexing values (until the top-level list of blocks fits in a single block). This is just a tree structure.

The above is skipping over an important detail: We don't just want blocks to be ordered, we want to be able to look our things up by key. So for each block we want to know the range of keys it can contain. If a block contains keys, that's easy – the range is [first,last]. If a block contains blocks, we could store the full range of each subblock –

```
    (l1,r1)   (l2,r2)   (l3,r3)   (l4,r4)
[    b1,         b2,         b3,         b4    ]
```

– but we really just care about finding the right block, so l1 and r4 are irrelevant, and r1/l2,r2/l3,r3/l4 are redundant – any value in that range would work, since it just tells us which side to descend into. So we just need to store n-1 keys:

```
          l2         l3      l4
[   b1,      b2,      b3,      b4    ]
```