2 Buffer Trees

In the last lecture, we proved that using a B-tree, one can find the rank of an item in a sorted array in $O(\log_B n)$ time, and that this time bound is optimal in the external comparison model. This implies that, unlike in the internal-memory model, we cannot sort an array of N items by inserting them one at a time into a standard balanced search structure and then scanning. The problem is that to insert a new element, we spend one I/O loading the pivot block at the root, finding the rank of *a single element*, and recursing. We can't exploit the inherent parallelism in the external-memory model as well as we'd like, simply because we only have a single element to insert.

However, if we want to use this approach to sort, we *don't* just have one element at a time; we have lots! Intuitively, instead of inserting elements one at a time, we can insert an entire *block* of elements all at once. In a single I/O, we can load the block of pivots at the root and find the ranks of a whole block of new elements. But what does it mean to "recurse" here? Once we've partitioned the block of new elements, we no longer have blocks of elements to pass down to the children.

The solution is to associate a *buffer* of elements with each node in the tree, containing elements to be inserted into that node's subtree. Whenever a buffer becomes full, its contents are pushed down to the node's children using a small number of I/Os. This type of *lazy* data structuring has been used in other contexts before, but it was first developed in the external-memory setting by Lars Arge* in his 1995 PhD thesis.[1]

2.1 (a, b)-Trees

The data structure underlying the buffer tree is a balanced tree where every internal node has degree $\Theta(m)$, not B. Specifically, the underlying tree is an (a, b)*-tree* with $a = m/4$ and $b = m$. Before describing the buffering process, we need to review the standard insertion and deletion algorithms for (a, b)-trees.

Let a and b be constants with $2a < b$. An (a, b)-tree is a rooted search tree where the root has between 2 and b children, every other internal node has between a and b children, and all leaves have the same depth. The actual data is stored in the leaves. Each internal node with degree k stores $k - 1$ *pivot* values. We can determine whether a value x is stored in the tree by comparing x to the pivot values at the root and recursing in the appropriate subtree. Thus, for a tree with N nodes, the search time is $O(\log_a N)$.

```
INSERT($x$):
  add a leaf $\ell$ containing $x$
  $v \leftarrow$ parent($\ell$)
  while deg($v$) $> b$
      ⟨⟨Split $v$⟩⟩
      if $v$ is the root
          create a new root $u$ with $v$ as its only child
      else
          $u \leftarrow$ parent($v$)
      create a new child $v'$ of $u$ immediately to the right of $v$
      move the rightmost $\lfloor (b+1)/2 \rfloor$ children of $v$ to $v'$
      $v \leftarrow u$
```

[1]Lars Arge*. The buffer tree: A technique for designing batched external data structures. *Algorithmica* 37:1-24, 2003. Stars indicates authors who were graduate students when their papers were first written.

A basic insertion just creates a new leaf with containing the desired value. If an insertion creates a node v whose degree is larger than b, that node is split into two nodes, each with half the children of v, and a new pivot is created in the parent of v. The split can propagate all the way to the root.

```
DELETE($x$):
    find the leaf $\ell$ containing $x$
    $v \leftarrow$ parent($\ell$)
    delete $\ell$
    while deg($v$) $< a$ and $v$ is not the root
        $v' \leftarrow$ an adjacent sibling of $v$
        if deg($v'$) $\geq b/2$
            ⟨⟨Share between $v$ and $v'$⟩⟩
            move (deg($v'$) $-$ deg($v$))/2 children of $v'$ to $v$
            ⟨⟨now deg($v$) $\approx$ deg($v'$)⟩⟩
        else
            ⟨⟨Fuse $v$ and $v'$⟩⟩
            move all children of $v'$ to $v$
            delete $v'$
            if $v$ has no siblings
                delete parent($v$) and make $v$ the new root
            else
                $v \leftarrow$ parent($v$)
```

Similarly, a basic deletion just deleted the leaf containing the target value. If a deletion creates a node v with degree less than a, then we either *fuse* or *share* v with one of its immediate siblings v', depending on how many children v' has. Specifically, if v' has more than $b/2$ children, then v steals enough children from v' that their degrees become equal (or off by one). If v' has less than $b/2$ children, then v and v' are fused into a single node with all the children; since this decreases the degree of v's parent, fuses can also propagate up to the root.

We easily observe that any node created by a rebalance operation—split, fuse, or share—must lose or gain at least $b/2 - a$ children before another rebalance operation becomes necessary. Thus, as long as we choose $b \geq (2+\varepsilon)a$ for some constant ε, we can amortize the cost of each rebalance operation across past insertions and deletions. The resulting amortized cost of insertions or deletions is dominated by the search time.

2.2 Buffering Insertions

A buffer tree is an (a, b)-tree with $a = m/4$ and $b = m$, built over a set of $\Theta(n)$ leaves, each containing $\Theta(B)$ items, with an additional buffer of size m (blocks) attached to each internal node. Each buffer holds up to M instructions for searching and/or modifying the tree: insertions, deletions, queries, and so forth. We do not require these instructions to be executed immediately, but we do expect the data structure to behave as if they were. In particular, the data structure should *eventually* give the same answer to any buffered queries as a traditional B-tree undergoing the same sequence of operations.

For the moment, let's focus on insertions; that's the only operation required for sorting, and the buffering process for deletions and queries is a little more complicated. The top-level insertion algorithm simply records the new item in a block in internal memory. After B insertions, when this block is full, we sort the block, write it into the buffer at the root, and start a new block. If the root buffer has more than m blocks (after every $M + B$ insertions), we empty it as follows:

- First, we read the first M items of the buffer into main memory, sort them, and merge than with the remainder of the buffer (which is already sorted).
- Next, we distribute items in the sorted buffer to the buffers in the children of the root, maintaining the invariant that each buffer contains at most one non-full block.
- Finally, if any of the child buffers is full (contains more than M items), we empty it recursively.

Ignoring the recursive calls, emptying a buffer containing X items requires $O(m + X/B)$ time. Under normal circumstances, we have $X > M$, so the total buffer-emptying time is $O(X/B)$; if we amortize the cost of emptying a buffer against the X insertions that filled the buffer, the amortized time for a single insertion is $O(1/B)$. Even if $X < M$, we still need $O(m)$ I/Os to distribute the contents of buffer to the m children.

Because the leaves themselves don't have buffers, the parents of leaves have a slightly different buffer-emptying process. Let v be a node with k leaves as children; recall that $k \leq m$.

- First, we read the first M items of the buffer into main memory, sort them, and merge than with the remainder of the buffer (which is already sorted) and the items stored in the leaves.
- Next, we write the first k blocks of items of the expanded buffer into the leaves.
- Finally, we add the remaining blocks of the expanded buffer as new leaves to v one at a time. If at any time v has more than m leaves, split v (and possibly its ancestors) as in the normal (a, b)-tree insertion algorithm.

Note that whenever we split a node, its buffer is empty.

Ignoring the splits, the cost of emptying a leaf-parent buffer containing X items is $O(X/B+k) = O(X/B+m)$. Again, under normal circumstances, we have $X > M$, and the buffer-emptying process adds $O(1/B)$ to the amortized cost of each insertion. Even if $X < M$, we still need $O(k) = O(m)$ I/Os to read the contents of v's leaves.

Our earlier analysis of (a, b)-trees implies that a series of N insertions into an initially empty buffer tree causes at most $O(n/m)$ splits, even if all the buffers are cleared. Each split requires $O(m)$ I/Os, so the total cost of keeping the tree balanced is $O(n)$; equivalently, the balancing adds $O(1/B)$ to the cost of each insertion.

The total amortized cost for an insertion is $O((\log_m n)/B)$—each node on the path from the root to a leaf in the *final* tree charges $O(1/B)$ I/Os for maintaining its buffer, and the whole tree charges $O(1/B)$ for keeping itself balanced.

2.3 Buffered Insertion Sort

To get an I/O-optimal sorting algorithm, we can now simply insert the unsorted elements one at a time into an initially empty buffer tree, clear all the buffers, and traverse the tree. The total time for N insertions is $O(n \log_m n)$, and we can clear the buffers and traverse the tree in $O(n)$ additional I/Os.