C_0 is only 4-balanced. If we were to lay out C_0 in the same way, then C_0's recursive subtree would only be 16-balanced. Instead, we employ the following strategy for laying out recursive subtrees that do not contain leaves of C. Suppose that we want to find recursive subtrees with target size 2^{2^i} above nodes with target weight 2^{2^j}. Then we find nodes with target weight $2^{2^i+2^j}$ to be the roots.

LEMMA 3. *Bottom recursive subtrees have size one to three times their targets size. Internal recursive subtrees have size between one third and three times their target size.*

LEMMA 4. *This nonuniform layout of a nearly balanced binary tree incurs $O(\log_B N)$ block transfers on a root-to-leaf traversal.*

THEOREM 5. *This static COSB-tree represents a set $\mathcal{D}$ of N elements, and supports member, predecessor, successor, and range queries. The operation* MEMBER(κ) *runs in $O(1+\log_B N+\|\kappa\|/B)$ memory transfers w.h.p., and* PRED(κ), *and* SUCC(κ) *run in $O(1+\log_B N+\|\kappa\|/B+\|\kappa'\|/B)$ memory transfers w.h.p., where κ' is the predecessor (resp. successor) of κ. The operation* RANGE-QUERY(κ,κ') *runs in $O(1+\log_B N+(\|\kappa\|+\|\kappa'\|+\|Q\|)/B)$ transfers, where Q is set of keys in the result.*

Proof. There are two cases:

Case 1: $\|\kappa\| = O(M)$, i.e., the key is small compared to memory. Computing the Karp-Rabin fingerprints takes $1+\kappa/B$ memory transfers, and all keys remain in internal memory while we search in the centroid tree.

Case 2: $\|\kappa\| = \Omega(M)$, i.e., the key is large compared to memory. In this case, the Karp-Rabin fingerprints that we compute cannot fit in memory at the same time. Thus, since we query $O(\log N)$ fingerprints, the number of memory transfers is $O(\log N+\log_B N+\|\kappa\|/B+1)$. However, the $O(\log N)$ term is dominated as long as $\log N < \|\kappa\|/B$. Since the CO model is transdichotomous ($B=\Omega(\log N)$) [14] and satisfies the tall-cache assumption ($M=\Omega(B^2)$) [15], $\log N \le M/B \le \|\kappa\|/B$.

The scan bounds are trivially obtained. □

3.2 Locality-preserving front compression

In this subsection we show how to add compression to our static COSB-tree. We develop a new strategy for achieving front compression without high decoding cost. The front-compressed data then replaces the array of keys used in the static COSB-tree above.

Front compression works as follows: Given a sequence of keys $\kappa_1,\kappa_2,\ldots,\kappa_i$ to store, a naive representation requires $\sum_j\|\kappa_j\|$ memory. Instead, we let π_{j+1} be the longest common prefix of κ_j and κ_{j+1}. In this case, we can remove nearly $\sum_j\|\pi_j\|$ memory from the representation by representing the keys as

$$\kappa_1,\|\pi_2\|,\sigma_2,\|\pi_3\|,\ldots,\|\pi_i\|,\sigma_i$$

where σ_j is the suffix of κ_j after removing the first π_j bits. To *decode* κ_j, one concatenates the first π_j bits from κ_{j-1} to σ_j. Finding the first π_j bits of κ_{j-1} may require further decoding, possibly resulting in expensive decoding. This lossless compression scheme has the same space use as the (uncompacted) trie for $\mathcal{D}$ [20]. The total size of a front-compressed set of keys $\mathcal{D}$ is written as $\langle\!\langle\mathcal{D}\rangle\!\rangle$.

Front and rear compression was described in [9, 10]. A more accessible description can be found in [26]. Publication [20] describes front compression in an exercise, but provides less detail. Publication [5] argues that front and rear compression are particularly important for secondary indices. Front compression is relevant for compressing the keys stored at the leaves of a search tree, whereas rear compression is essentially used only in the indices, and is subsumed by the string-B-tree techniques presented here and in [13].

Our goal is to achieve $O(1+\|\kappa\|/B)$ memory transfers to decompress any key in $\mathcal{D}$, but to store $\mathcal{D}$ with $O(\langle\!\langle\mathcal{D}\rangle\!\rangle)$ space. The challenge is that uncompressing a single key may require scanning back through the entire compressed representation. This is a well known problem for front compression. One common strategy is to compress enough keys to fill some predefined block and to start the compression over when that block is full. This idea does not provide any theoretical bounds, however: the compression achieved can be much worse than the best front compression; and a block size may be arbitrarily bigger than $\|Q\|$, so decompression also has no guarantees. Here we show a *locality-preserving front compression* (*LPFC*), which meets our goal.

Our modified compression scheme begins with key κ_1. Suppose we have compressed the first $i-1$ keys and now we want to add key κ_i. We scan back $c\|\kappa_i\|$ characters in the compression to see if we could decode κ_i from just this information. If so, we add π_i,σ_i as before. If not, we add $0,\kappa_i$ to the compression, that is, we do not compress key κ_i at all. Call this sequence the *locality-preserving front compression of* $\mathcal{D}$, denoted LPFC($\mathcal{D}$).

The decoding scheme is just as with standard front compression, and it immediately matches the desired bounds: decoding κ_i touches at most $c\|\kappa_i\|$ contiguous characters, and decoding Q touches $O(\|Q\|)$ contiguous characters. The issue here is to show that the compression is $(1+\varepsilon)\langle\!\langle\mathcal{D}\rangle\!\rangle$, which we do as follows.

LEMMA 6. *The total length of the LPFC($\mathcal{D}$) is at most $(1+\varepsilon)\langle\!\langle\mathcal{D}\rangle\!\rangle$ and every key κ_i can be decoded with $O(\|\kappa_i\|/\varepsilon B)$ block transfers.*

Proof. Call any key κ that has been inserted uncompressed a *copied key*. Denote as *native* any characters in the compression that are not copied. Denote the preceding $c\|\kappa\|$ characters as the *left extent* of κ. Notice that if κ is a copied key, there can be no copied key beginning in the left extent of κ. However, a copied key may end within κ's left extent.

We consider two cases. In the first case, the preceding copied key ends at least $c\|\kappa\|/2$ characters before κ. Then, we say that κ is *uncrowded*. In the second case the preceding copied key κ ends within $c\|\kappa\|/2$ characters of κ. Then, we say that κ is *crowded*.

Partition the sequence of all copied keys just before each uncrowded key. We call each such subsequence a *chain*. Note that each chain begins with an uncrowded key and is followed by a sequence of crowded keys.

Furthermore, the lengths of these crowded keys decrease geometrically. To see this, consider a crowded key κ. Since κ's predecessor in the chain, κ', must begin before κ's left extent, it must have length at least $c\|\kappa\|/2$.

Thus, if κ is uncrowded, the kth crowded key in its chain has length at most $\|\kappa\|(2/c)^k$. The total length of all keys in a chain starting at κ is thus at most $c\|\kappa\|/(c-2)$.

Finally, charge the cost of copying these keys to the $c\|\kappa\|/2$ characters preceeding the uncrowded key at the beginning of the chain. This charge is at most $2/(c-2)$ per character. Finally, set $\varepsilon=2/(c-2)$. □

THEOREM 7. *The static COSB-tree with front compression represents a set $\mathcal{D}$ of N elements, and supports member, predecessor, successor, prefix, and range queries. The operation* MEMBER(κ) *runs in $O(1+\log_B N+\|\kappa\|/B)$ memory transfers w.h.p., and* PRED(κ), *and* SUCC(κ) *run in $O(1+\log_B N+\|\kappa\|/B+\|\kappa'\|/B)$ memory transfers w.h.p., where κ' is the predecessor (resp. successor) of κ. The operation* RANGE-QUERY(κ,κ') *runs*