ZIP TREES IN CONTEXT: A COMPARATIVE APPRAISAL

CHRIS MCNALLY, VICTOR QUACH, BEATRICE NASH

Abstract. The *zip tree*, recently presented by Tarjan, Levy, and Timmel [1], is a type of randomized binary search tree (BST) designed for concurrency that is both simpler than its closest relatives and theoretically more efficient in time and space under certain conditions. We conduct a comparative study of a selection of BST-like dictionaries in order to assess the viability of zip trees in practice. We conclude that zip trees indeed boast performance comparable to common BST dictionaries, and identify regimes in which they outperform and underperform relative to their peer data structures. Such empirical comparisons are particularly useful for evaluating these randomized data structures, as theoretical bounds do not always fully capture their range of behavior. Finally, we present data on the advantages and limitations of four zip tree variants.

1. Introduction

Concurrent and parallel systems[1] are a major object of study in both theoretical computer science and contemporary computer engineering [2]. One of the chief obstacles in concurrent programming lies in casting algorithms in an concurrence-amenable form. This means that in some sense, distant pieces of data on which an algorithm acts should be decoupled, so that separate actors or threads of execution can operate without interference. This is the problem of *mutual exclusion*, which pervades the theory and practice of concurrent, parallel and distributed systems. In general, it is also a supremely challenging problem, the difficulty of which has been appreciated for more than half a century [3]. However, there are certain common design motifs that support good concurrent hygiene.

One such motif applies to pointer-based data structures and is an intuitive corollary of the proscription against couplings between distant data elements. For any pointer-based implementation of an abstract data type (ADT), all its methods should involve only *local* operations. We take this in an informal sense to mean that, morally, each method is allowed to move $O(1)$ pointers within a ball of constant radius of a query node. This condition—while imprecise and neither necessary nor sufficient for concurrent data structures—ensures that effects cannot propagate arbitrarily far, and generally promote liberal locking strategies. One would therefore hope to find implementations of common ADTs that satisfy a similar locality condition.

The *dictionary*, supporting `insert`, `search`, and `delete`, is a foundational ADT on which countless data structures and algorithms are built. We say that a data structure "solves the dictionary problem" if it acts as a key-value store supporting these operations. Binary search trees (BSTs) are a ubiquitous implementation choice when the keys are drawn from a totally-ordered set. Provided that they are balanced, BSTs solve the dictionary problem with $O(\log n)$ runtime on all operations. Countless balancing schemes have been devised over the decades to maintain this crucial invariant [7, 8, 9], yet—perhaps surprisingly—theoretical progress marches on. A recent preprint of Tarjan *et al.* introduces the *zip tree*, a randomized BST closely related to treaps and skip lists, which purports to offer attractive locality properties for concurrent applications. These claims are beyond the scope of an implementation project, and we do not evaluate them here.

In this report, we investigate whether any such conjectural benefit of zip trees comes at a measurable, substantial cost in access time, memory, depth, or other resources. We have written

[1]These concepts are closely related, but are distinguished by their respective emphases on *logical* and *temporal* simultaneity.

a C++ zip tree implementation, and written or adopted four other related data structure implementations to provide a diverse set of relative benchmarks of zip tree performance. We compare the performance of these data structures—in terms of a variety of figures of merit—on randomly-generated and real-world data sets. Finally, in order to overcome the possible limitations of zip trees, we explore variations on the data structure and stress-test them in turn.

2. Data structures background

We provide a brief overview of each of the data structures that we implement for comparison against the zip tree. Namely, we discuss three additional binary search trees–the *treap*, the *splay tree*, and the *red-black tree*–as well as the *skip list*, a randomized data structure descended from linked lists but spiritually aligned with randomized BSTs. We assume that each of these key-value stores only treats data with distinct keys. Extending to identical-key nodes is possible in all cases, but inessential for stress-testing the data structures. All keys are assumed to be integral.

2.1. **Red-black trees.**
Red-black trees are a deterministic balanced BST and a staple of undergraduate curricula [5, 7]. They enforce balance by maintaining flag bits at every vertex, obeying a local compatibility law. A simple local-to-global correspondence then guarantees that no root-leaf path is longer than any other by more than a constant factor. This kind of scheme is characteristic of other elementary balanced BSTs, such as AVL trees [8]. Maintaining this property allows for search, insert, and delete operations to have time complexity $O(\log n)$ in the worst-case, but are the most complicated operations to implement of the data structures compared here.

2.2. **Splay trees.**
Splay trees are a BST that seeks to keep frequently-accessed nodes close to the root. They forego explicit balancing in favor of a strongly nonlocal operation called `splay` that keeps dictionary operations fast in an amortized sense.

Our implementation is a top-down splay tree adapted from a C language implementation of D. Sleator [16].

2.3. **Treaps.**
The *treap*—also known as a *Cartesian tree*[2]—is the classic randomized BST on which the zip tree is based [10, 11, 1]. It arises from the observation that binary trees generated by random node attachment are balanced in expectation, and seeks to extend this trait even to *on-line* operations [5] by incorporating a randomized constraint that must be relieved by tree rotations. Structurally, treaps are nothing but the intersection of trees and (max-)heaps. Each node x in a treap T has members $x.\mathtt{key}$ and $x.\mathtt{rank}$, in addition to its children $x.\mathtt{left}$ and $x.\mathtt{right}$. The treap invariant asserts that T is symmetrically-ordered with respect to keys, and heap-ordered with respect to ranks; that is, for every node x in T,

$$x.\mathtt{left.key} < x.\mathtt{key} < x.\mathtt{right.key}$$
$$x.\{\mathtt{left},\mathtt{right}\}.\mathtt{rank} < x.\mathtt{rank}.$$

This is just the structure that results from inserting keys in a standard BST in rank-order.

The only treap operations that differ from those of a general BST are the dynamic ones: during `insert` and `delete` operations, node keys and ranks uniquely determine the tree structure. In the classic treap, the rank of each node is a uniformly[3] random real number[4] sampled on `insert`, so the rank of each node is almost surely[5] unique.

[2]Today, this term usually refers to a different, more primitive data structure than the treap.

[3]It is worth noting that, since the treap structure only depends on the relative ranks of the nodes, the exact distribution used is irrelevant, modulo implementation considerations.

[4]In practice, we can simply choose random integers from a large range, like `[0; MAXINT]`.

[5]i.e. with probability 1

Box 1: Treap operations.

- T.`search`(k): standard BST search.
- T.`insert`(k, v): Create a new treap node x with key k and value v, and assign it a rank x.`rank` $\leftarrow$ `pickrank`(). If T is empty, insert x as its root. Otherwise, insert the key as in an ordinary BST. The resulting structure is symmetrically-ordered, but not necessarily heap-ordered. We correct this by performing rotations at x until the heap property is restored. Because the BST property is invariant under tree rotations, the structure remains a BST throughout.
- T.`delete`(k): First let $x \leftarrow$ `search`(k). If x is null, return. Otherwise, perform the insertion operation in reverse. Repeatedly rotate x with its higher-rank child until it is a leaf, at which time it is the only node that can be in violation of the heap property. Then remove it from the tree.
- `pickrank`: Sample r from $\mathcal{U}([0,1])$ and return r.

The proof of correctness of these operations is outside the scope of an implementation project, but in fact both `insert` and `delete` always result in a rankwise heap. Treaps offer $O(\log n)$ expected runtime for all dictionary operations. For proof, we refer the reader to the literature [11, 5].

2.4. **Skip lists.** The *skip list* is a randomized, linked-list based key-value store designed for fast access and closely related to BSTs [13, 14]. In a skip list L, key-value pairs are stored in a collection $\mathcal{N}$ of nodes ordered by key. Each a node x has a *level* (which need not be stored as a separate data member) $1 \leq \texttt{level}(x) \leq \textsc{MaxLevel} = O(\log N)$, where N is an upper bound on the number of elements. Each node also contains an array x.`forward` of size `level`(x) of pointers to subsequent nodes such that x.`forward`[i] points to the minimum-key node y such that x.`key` < y.`key` and `level`(y) $\geq i$. Finally, there is an entry node L.`header` of level $\textsc{MaxLevel}$ with key $-\infty$, and a terminal node L.`nil` with key $+\infty$ and maximum level.

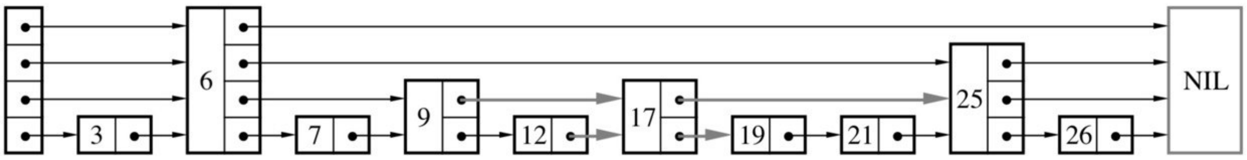

Figure 1. Schematic skip list structure. Between the entry node and exit node, nodes are inserted with levels‘ approximately geometrically distributed with $p = 1/2$. Reproduced from [13].

The principle behind skip lists' fast access comes from the determination of the node levels. In the classic skip list, a node is assigned level l with probability $(1-p)^{l-1}p^l$ for some $0 < p < 1$ when it is inserted, and retains that level throughout it lifetime. In practice, we do not allow arbitrarily high levels. In our implementation, any level assignment is capped by $\max(l, \textsc{MaxLevel})$, so the level distribution is not truly geometric, but this is of little consequence for $n \ll 2^{\textsc{MaxLevel}}$. Dictionary operations in skip lists are performed by traversing `forward` pointers greedily to the location of the search key and descending level-sets when no further progress can be made, then adjusting up to $O(\textsc{MaxLevel})$ pointers when editing a node. The need to explicitly step down through the level sets burdens access times in skip lists. We will see that zip trees alleviate this burden.

Box 2: Skip list operations.

- L.`search`(k): To find the node with key k, enter the data structure, letting $x \leftarrow L$.`header` and keep a counter i initialized to MaxLevel. Decrement i until x.`forward`$[i] \neq L$.`nil` and x.`forward`$[i]$.`key` $\leq k$, and then **skip** forward, letting $x \leftarrow x$.`forward`$[i]$. If the skip condition is never met; that is, if $i = 1$ and x.`forward`$[i] > k$, return with an exception. If $x.\text{key} = k$, return x. Otherwise, repeat. This skipping procedure is also used by both of the dynamic methods to traverse the skip list.
- L.`insert`(k, v): to insert a key, we need to pick a level for the new node. Let $l \leftarrow$ `picklevel`(p) be the new level. We also need to perform a little surgery to reroute list pointers through the inserted node. To this end, instantiate an array `update` of size MaxLevel to keep track of the upstream nodes pointing through the future position of the new node. As in `search`, let $i \leftarrow$ MaxLevel be a level index, and enter at $x \leftarrow L$.`header`. Skip forward until x.`forward`$[i]$.`key` $\geq k$, and if a node is found with key k, simply overwrite its value with v and return. Then let `update`$[i] = x$, decrement an index i from MaxLevel, and repeat until $i = 1$, at which point the new node should be inserted between x and x.`forward`$[1]$. To do this, we have to intercept the forward-pointers up to level l by routing them through the new node. Create a new node y of level l. Then for all $1 \leq j \leq l$, let y.`forward`$[j] \leftarrow$ `update`$[j]$ and `update`$[j] \leftarrow y$.
- L.`delete`(k): deletion is essentially the inverse operation of `insert`. An `update` array tracks nodes pointing to the target node x with key k. When all `level`(x) of the upstream nodes pointing to x are found, forward them all through x by letting `update[i].forward[i]` $\leftarrow x$.`forward[i]` and free x.
- `picklevel`(p): Let D_p be the geometric distribution with mean $1/p$ over $\mathbb{N}^+$. Sample an $l \sim D_p$ and return l.

Like randomized BSTs, skip lists offer $O(\log n)$ expected runtime for all dictionary operations. For proof, we refer the reader to the literature [13]. A key advantage of skip lists in comparison with randomized BSTs is that they have a simple implementation, whereas data structures like treaps can be complicated to implement in practice.

3. Zip trees

The zip tree, the focus of this report, is a fusion of the skip list and the treap. Like the treap, the zip tree consists of nodes x with fields x.`key` and x.`rank` (called *priority* by some authors), and are symetrically-ordered with respect to `key` and heap-ordered with respect to `rank`. However, it differs from the treap by using a different rank distribution, by allowing rank ties, and by using a different insertion routine known as `zipping` to enforce heap-ordering. By zipping nodes into a tree, the data structure eschews the need for long chains of rotations to carry nodes to their proper place. Besides its "simplified treap" identity, the zip tree can also be understood as a skip list "exploded" into a tree, lowering memory footprint and access times.

Unlike the treap, the zip tree allows for rank ties to occur, which leads to an increase in expected depth, but reduces the cost required to randomly generate a unique rank for each element inserted [1].

3.1. **Zip tree dictionary operations.** While treap operations require rotations to maintain the max-heap property of its priorities upon insertion and deletion of elements, the zip tree uses an idiosyncratic pair subroutines, referred to as *zipping* and—its inverse—*unzipping* (figure 2). Like the other randomized BSTs we study, it seeks to (probably) keep the heights of most nodes low by randomized insertion. In a full binary tree, a fraction $1/2^{h+1}$ of all nodes have height h. Upon insertion in a zip tree, each node x is assigned a rank r with probability $\frac{1}{2^r}$, and shuttled down to the appropriate corresponding depth where it can be zipped in with only

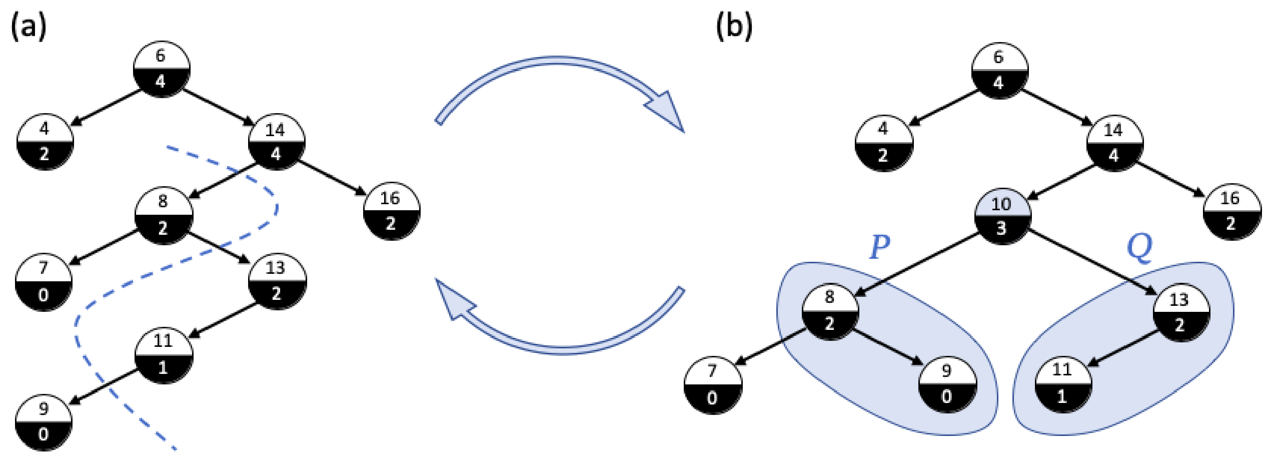

Figure 2. Schematic zip tree structure. The upper semicircle of each node contains that node's key, and the lower semisircle its rank. **(a)** Z before insertion (after deletion) of a node with key 10. The dashed curve denotes the boundary along which Z is unzipped. Together, the nodes with incident pointers cut by this curve form the inferior and superior paths P and Q. **(b)** Z after insertion (before deletion) of the node with key 10.

very few pointer operations, close to depth r. Rather than choosing ranks solely as a source of randomness to trigger rotations and indirectly balance the tree, they are, as in skip lists, chosen as a proxy for height.

Box 3: Zip tree operations.

- Z.`search`(k): standard BST search.
- Z.`insert`(k, v): Create a new zip node x with key k and value v and assign it a rank x.`rank` $\leftarrow$ `pickrank`(). Follow the BST search path to k until a node y is encountered with rank y.`rank` $:= r \leq x$.`rank`, allowing ties if y.`key` $\geq x$.`key`. x is inserted as a new child of the parent of y, and the remainder of the search path to k is traced out and *unzipped* from y down. The search path's nodes partitioned into subsequences P of keys $< k$ and Q of keys $> k$, which are fused together to form the new right spine of the left subtree (resp. left spine of the right subtree) of x.
- Z.`delete`(k): Search for node of rank k. If there is none, return. If there is such a node, zip together the right spin of its left subtree and the right spin of its left subtree into heap order, in the inverse operation to the above.
- Z.`pickrank`(): Let D_p be the geometric distribution with mean $1/p$ over $\mathbb{N}^+$. Sample an $l \sim D_p$ and return l.

For both zipping and unzipping, a constant expected number of pointers need to be changed. It is shown by Tarjan *et al.* that the expected number of nodes in $P \cup Q$, for a node of rank r, is at most $(3/2)r + (1/2)$. As rank r is assigned with probability $1/2^r$, the net expected path length is $O(1)$. Since the maximum number of pointers that need to be changed is equal to 1 plus the length of the two paths , it follows that $O(1)$ pointers need to be changed in expectation [1].

A relative weakness of the zip tree is its depth. It achieves expected $O(\log n)$ run time for each of its dictionary operations with expected depth $1.5 \log n$. The treap, by comparison, achieves expected depth $\ln n = 1.44 \log n$, or 4% less than that of the zip tree. This difference is due to the zip tree's tolerance of rank ties, which cause additional, which the treap does not.

Although the zip tree is quite similar to the treap, it offers two notable advantages: first, the zipping and unzipping methods are simpler to implement than bottom-up insertion and tree rotations, and second, only $O(\log \log n)$ bits are required in expectation to represent the ranks.

This follows from the fact that the root node, r, has rank $\leq \log n + c$ with probability:

$$\begin{aligned}
\mathbb{P}[\, r.\mathtt{rank} \leq \log n + c\,] &= \sum_{k=0}^{\log n + c} \frac{1}{2^{k+1}} \\
&= \left(1 - \frac{1}{2^{\log n + c + 1}}\right)^n \\
&\geq 1 - \frac{n}{2^{\log n + c + 1}} \\
&= \frac{1}{2^{c+1}}
\end{aligned}$$

and consequently that the expected value of the root node is $\leq \log n + \sum_{c=0}^{\infty} c/2^{c+1} \leq \log n + 3$. Therefore, each rank requires $O(\log\log n)$ bits in their representation, as opposed to $O(\log n)$ required for treaps.

3.2. **Zip Tree Variants.** Besides the conventional zip tree, we implemented four modified zip tree data structures[6], each intended to challenge its competitors in a different way.

(1) p-zip tree. In the mildest variation on the zip tree data structure, we straightforwardly adjust the rank distribution. Like skip lists, zip trees have a single degree of freedom in the mean $1/p$ of the geometric distribution that determines their ranks, and there is no *a priori* reason to constrain the data stucture to $p = 1/2$.
(2) Fractional-rank zip tree. By allowing for fractional ranks and eliminating rank ties, zip trees can in principle reduce their expected depth from $1.5 \log n$ to match that of treaps, $1.44 \log n$, while still only requiring $O(\log\log n)$ bits to represent the ranks. This is done by maintaining a *secondary rank* chosen uniformly at random from the unit interval. Tarjan *et al.* probe that that $O(\log\log n)$ bits of precision for f are sufficient for rank ties to be avoided with high probability.
(3) Self-adjusting zip tree. Designed as an answer to splay trees for strongly biased access distributions, this modification reduces the access time to frequency-accessed nodes. However, it comes at the cost of introducing nonlocal rotations to the data structure's repertoire, which somewhat nullifies the spirit of the zip tree and ultimately renders it more treapish. The self-adjustment works as follows: when a node x is accessed, a new rank r is computed by a call to `pickrank`(), and we assign $x \leftarrow \max(x.\mathtt{key}, r)$. In general this violates heap order, so x is rotated towards the root until heap order is restored. After many accesses, the most frequently-queried keys will tend to have the highest ranks, and reside near the root. However, the ranks will also tend to *saturate*, and we therefore conjecture that self-adjusting zip trees will be less "agile," or accommodating of nonstationary access distributions, than classic splay trees.
(4) Find-before-insert zip tree. In the original implementation of `insert` from [1], the authors require that no element with the same key be present in the tree. Contrary to other variants of BST, this constraint can not be easily lifted by during the traversal step because we may never reach the original element if its rank is too low. Deleting the original element during the unzipping step would then require to nest an additional zipping operation. In practice, we can not always assume that `insert` operations are only called on different keys. We resolve this technical challenge by calling `search` before every `insert` to decide whether to insert a new element or update it.

4. Implementation details

We wrote C++ implementations of the canonical zip tree, treap and skip list, as well as the modified zip trees. We also used the C++ standard library `std::map` red-black tree implementation for comparison with a non-randomized balanced BST, and adapted a C language

[6]Three of which were suggested by the paper of Tarjan *et al.*

top-down splay tree due to D. Sleator [16] for comparison with a self-adjusting BST. Our implementations, together with tests, data, and related utilities, can be found online [7]. This code was compiled with clang (Apple LLVM version 10.0.0) under `-O2` optimization[8] and run on an Intel i7-7700HQ processor to produce the data below.

Some compromises in the implementation are inevitable. Tarjan *et al.* suggest that zip trees should be more memory-efficient than treaps, on account of the smaller rank space. However, in practice it appears unnecessary to use high-precision floating-point ranks to achieve good treap depth for the instance sizes we tested, so we instead used uniformly random `int` ranks, bringing the treap memory footprint in line with that of zip trees. We therefore omit plots of memory usage, but do note our finding that skip lists use about $\sim 40\%$ more space than treaps and zip trees.

The dynamic zip tree methods were implemented iteratively rather than recursively in order to minimize the number of pointer operations; however, we do not report pointer operation counts in this work.

All runtimes are the wall-clock time reported by `<sys/time.h>`. We always report average runtimes over many runs, in part to mitigate the jitter imparted by this measurement technique.

5. Experiments and Results

5.1. **Dependency on parameter p.** Tarjan *et al.* posit that performance metrics for p-zip trees should be essentially independent of p due to the tradeoff between rank-tie frequency (low p) and the maximum rank (high p). Our first experiment sought to substantiate or refute this claim in order to set the optimal p for future experiments, and draw the curtain on zip tree variant (1). To do so, we measured the time required to insert 1024 elements into a zip tree of size 2^{14} for $p = (n/100)$, with n ranging from 1 to 99. The results of this test are shown in Fig. 3.

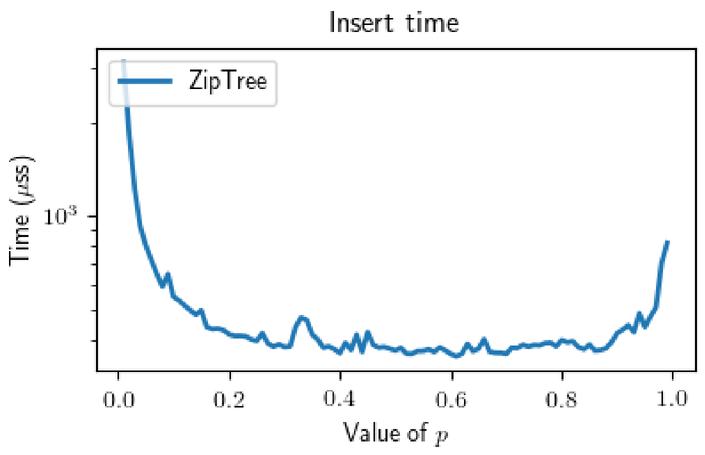

Figure 3. Zip tree inserts as a function of p.

These results support the claim that zip tree performance is largely independent of p over a range of values centered around 1/2. Having seen that p is a non-critical parameter, we henceforth set $p = 1/2$ for the remaining tests.

5.2. **Basic operations.** Our first battery of tests aims to evaluate the performance of the three dictionary operations (`insert`, `delete`, and `search`) for each of the data structures. In these tests, we populated a dictionary with a fixed number N of key-value pairs for each of the data structures. We then performed k dictionary operations each of `insert`, `delete`,

[7]`https://github.com/varal7/random_bst`

[8]Despite initial concerns that this might dangerously alter the data structures' relative runtimes under certain conditions, we found no evidence of such an effect, and retained the compiler optimizations for faster data.

and `search` starting from the same initialized dictionaries. We measure the running time and report the average over 300 iterations. Furthermore, the `search` operations were performed with multiple distributions over the accesses keys. A uniform distribution was used as a control case. To model increasingly biased sample distributions, we sampled keys according to a *Zipf distribution*: a family of power-law distributions famously pervasive in lexical data [6], indexed by a concentration parameter α characterizing the degree of nonuniformity or bias. Although these tests do not exhaust all possible loads under which the data structures could operate, they serve to illustrate the cost of each individual operation for a given dictionary under controlled conditions.

In **Table 1**, we present our results for $N = 2^{18}$ and $r = 2^{14}$. In this experiment, it is clear that zip trees, while generally slower than treaps, splay trees, and deterministic balanced BSTs, nevertheless outperform their sister data structure the skip list. The table also shows that, although self-adjusting binary trees see the *greatest* improvement in their access times under strongly nonuniform access distributions, *all* of the data structures measured some degree of faster access under high-skewed Zipf distributions. We suspect that this is a caching effect, since few enough accesses are performed that in the case of uniform access, *no* node is likely to be accessed twice. Only highly nonuniform access is likely to lead to cache hits on nodes. Incidentally, self-adjusting zip trees see the greatest performance gains under nonuniform access of all zip tree variants, but this is in part because they are slower in the first place.

	`insert`	`delete`	uniform	$\text{Zipf}_{0.50}$	$\text{Zipf}_{0.90}$	$\text{Zipf}_{0.99}$
zip tree	14.6±1.4	14.4±2.2	11.3±1.0	11.3±1.1	9.8±1.1	8.9±1.1
fractional-rank	13.2±1.4	13.4±1.6	11.1±1.0	10.9±1.2	9.5±1.1	8.4±0.8
find-before-insert	17.4±1.5	13.4±1.2	11.2±1.3	11.1±1.2	9.8±1.1	8.6±0.8
self-adjusting	14.6±1.5	13.7±1.3	12.6±1.4	12.2±1.2	10.1±1.0	8.5±0.8
skip list	21.6±1.6	21.7±1.4	15.2±1.2	15.3±0.9	13.2±1.1	11.8±1.0
treap	14.8±1.6	13.5±1.7	8.6±1.0	8.6±0.8	7.7±1.3	6.8±1.0
splay tree	10.5±1.3	10.8±1.8	8.0±1.2	7.9±0.9	5.9±1.1	4.6±0.7
red-black tree	9.4±0.8	11.4±1.3	8.8±1.0	8.5±1.0	7.1±0.9	6.2±0.7

Table 1. Runtimes in ms of a sequence of 2^{14} `insert`, `delete` or `search` operations on each data structure. `search` operations were performed either with a uniform distribution over the 2^{18} key-value pairs, or with a Zipf distribution with bias parameter $\alpha = 0.50, 0.90$ or 0.99.

In **Figure 4** and **Figure 5**, we demonstrate the impact of varying N, the initial size of the dictionary. We let N vary from 2^4 to 2^{20} with multiplicative increments of 2. We measure the runtime for k operations in the same experimental conditions as previously. For inserts and deletes, we chose $k = 1024$, while we needed to use $k = 2^{22}$ for accesses.

In **Figure 4**, we compare three variants of zip trees to the original version. The results we obtain are consistent with our expectations. The variant (4) find-before-insert differs from the original zip tree only for the insertion task. For uniform access, the self-adjusting variant is slower than the vanilla version because it performs useless tree rotations on each access. When the accessed keys are distributed according to Zipf's distribution, the self-adjusting zip tree still underperforms from small instances but performs significantly better on instances greater than 10^4 elements.

In **Figure 5**, we compare the original zip tree to well-known baselines. For sake of clarity, we do not report the standard deviation of each data point. Across the board, skip lists are the slowest data structures. The only exception are for red-black on small instances. We suspect that this is due to a constant overhead of the `std::map` implementation. In fact, red-black trees seem to scale better and additional experiments[9] on larger instance sizes reinforce this view. It is also worth noting that splay trees perform particularly well when the access key distribution is skewed, which is what we expected.

[9]that we do not report for lack of iterations

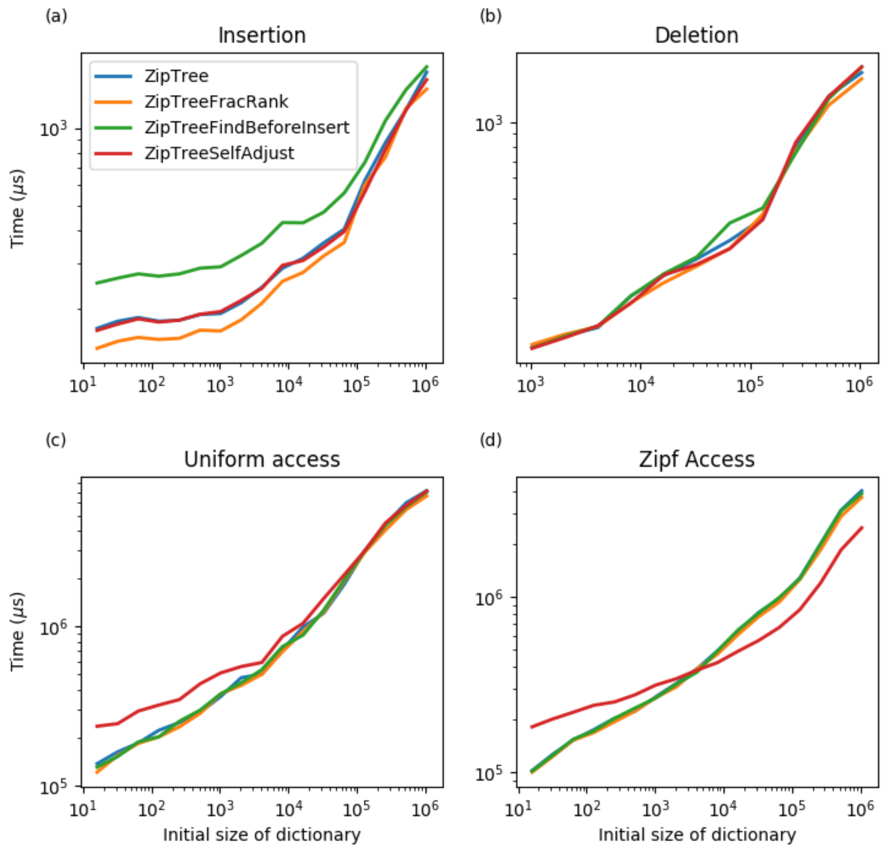

Figure 4. Cost of basic operations for varying initial size N of dictionary. In (a) and (b), we use $k = 1024$ operations. In (c) and (d), we use $k = 2^{22}$ accesses (n.b. the difference in scale). Zipf-distributed accesses are made with $\alpha = 0.99$ In both cases, we plot the mean of 100 iterations.

5.3. **Inserting from scratch.** In this subsection, we investigate the specific scenario of inserting elements into an empty dictionary. We explore three cases: the keys to insert are given in sorted order, reversed order and random order.

In **Figure 6**, we plot the average time for an insert as the time taken for the complete insertion procedure divided by the number of elements inserted. Note that in this respect, **Figure 6b** is different from **Figure 5a**: the former computes the average cost while the latter computes the marginal cost.

5.4. **Height and average depth.** The preceding experiments have shown that treaps generally outperform zip trees in standard benchmarks. One reason they might be faster is that they balance more effectively by disallowing rank ties. This should be detectable by inserting[10] sequences of elements in the two data structures and measuring their height, or other proxies for tree balance. **Figure 7** reports our results for this experiment. N keys are inserted in each data structure To confirm the proposition that fractional-rank zip trees achieve the same depth as treaps, they are included in the same figure as well. By maintaining a secondary tiebreaker rank, they do in fact remain as shallow as treaps—even with few bits of precision.

5.5. **Nonstationary access distributions and high locality of reference.** In the analysis of data structures, one often assumes that data or method calls are served according to a fixed probability distribution. This is such a fundamental assumption that one can lose sight of

[10]in any order: this does not limit the generality of our results, because the trees are completely structurally determined by the keys and ranks of their nodes.

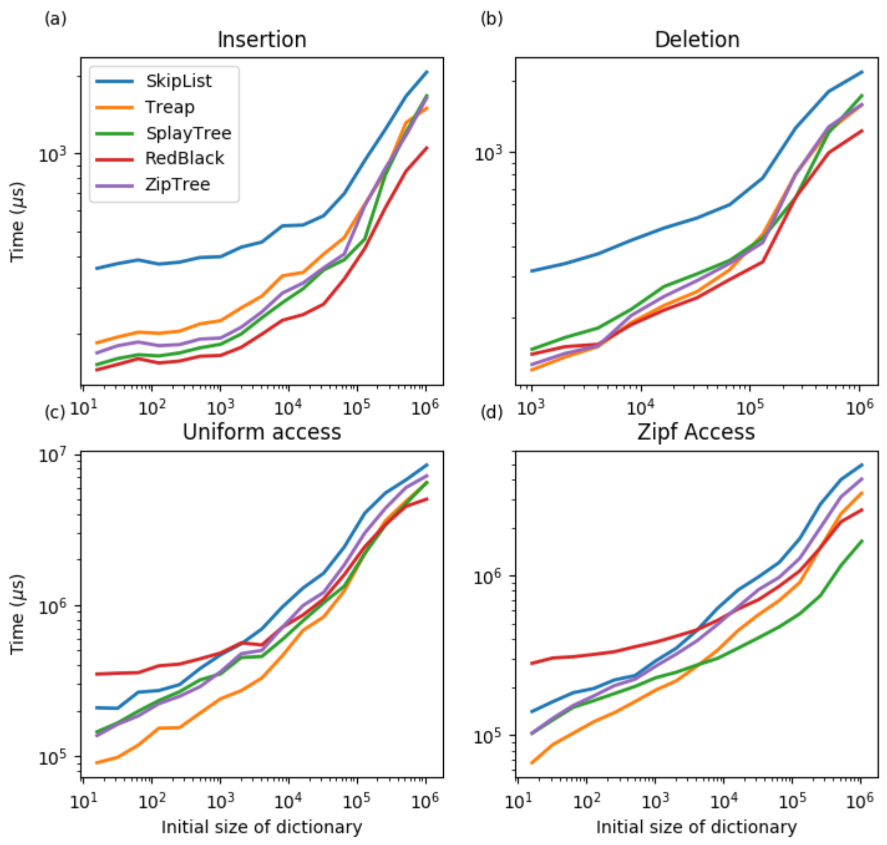

Figure 5. Cost of basic operations for varying initial size N of dictionary. In a) and b), we use $k = 1024$ operations. In c) and d), we use $k = 2^{22}$ accesses. In both cases, we plot the mean of 100 iterations

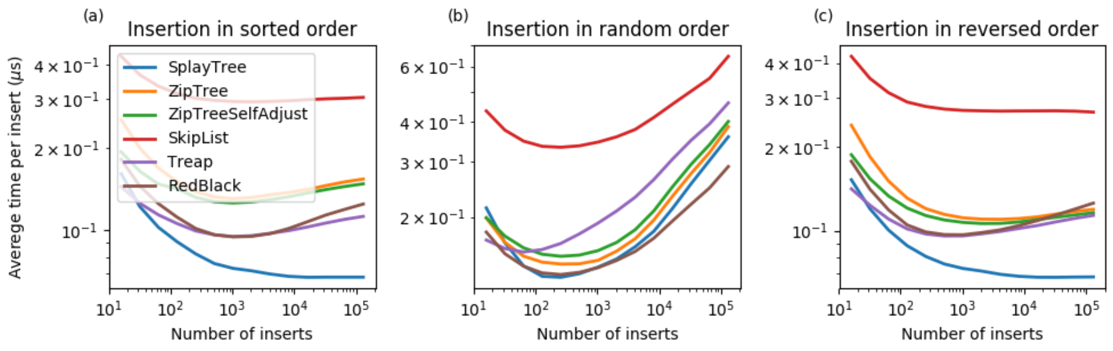

Figure 6. Average cost of insertion of an element when inserting into an empty dictionary

the complexities of the real world. Non-stationary distributions complicated by all manner of hopping and "popcorn noise" are a ubiquitous phenomenon in real-world systems, notably including network traffic [18]. The effects of these phenomena are difficult to analyze, and in many cases can only be understood through modelling.

As we have already seen, the zip tree can make a Faustian bargain and weaken its locality properties by promoting frequently-accessed keys to higher ranks. Knowing that self-adjusting zip trees enjoy some of the same advantages as splay trees when access calls are drawn from a power-law distribution, we next thought to ask how zip-tree self-adjustment progresses in the time domain. In the process of adjusting its structure to a concentrated access distribution, the

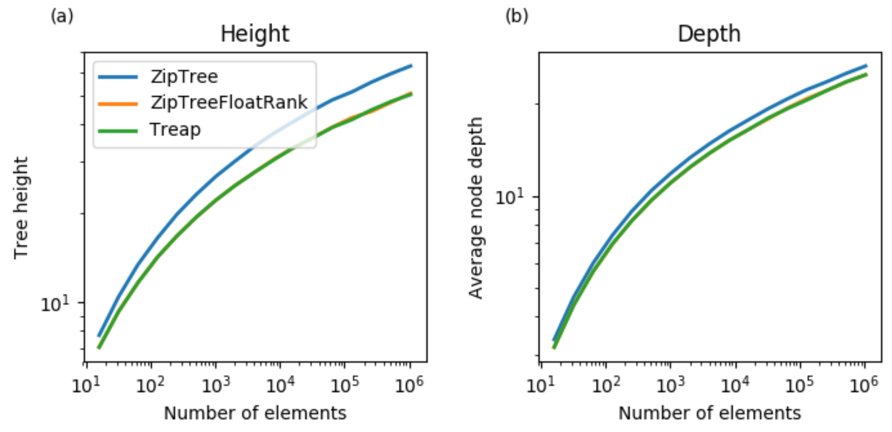

Figure 7. Average height/ depth of nodes in a tree-like dictionary, as a function of the number of its elements. Note the orange fractional-rank zip tree curve is perfectly occluded by the treap curve.

self-adjusting zip tree increases the ranks of the preferred keys without bound, while seldom-visited keys still have low rank. Should the distribution ever *change*, the tree can be expected to take many more calls before equilibrating to a structure optimized for the new distribution. Each time the distribution changes, readjustment will take longer and longer as ever greater ranks are “claimed,” and the plasticity of the data structure will gradually diminish.

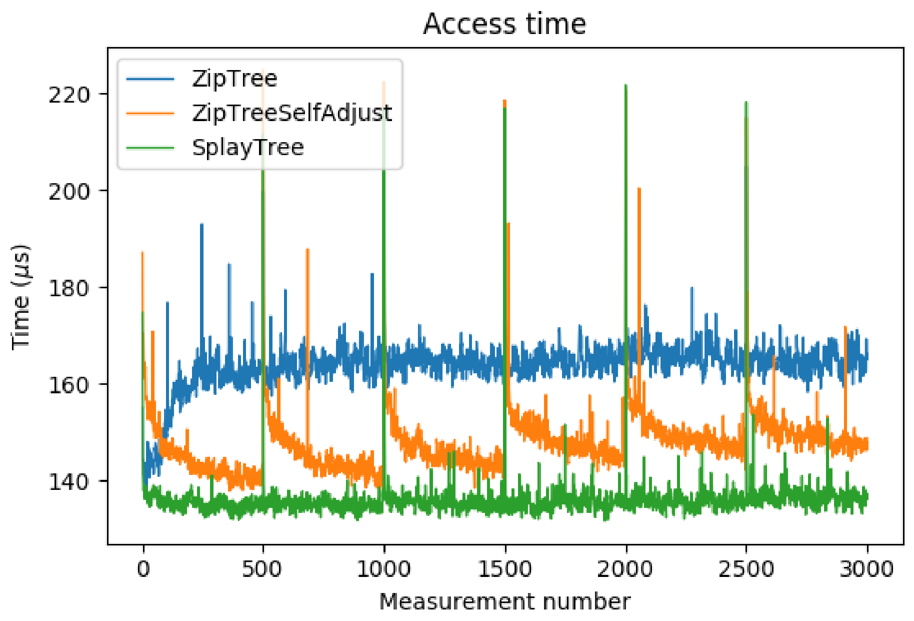

Figure 8. Access times for splay trees, zip trees and self-adjusting zip trees over six hopping epochs.

We put this hypothesis to the test by timing `search` calls issued to three data structures: self-adjusting zip trees, splay trees, and zip trees (a control group). Each data structure was initialized with 2^{15} nodes. Over 6 epochs, `search` calls were issued to each data structure uniformly accessing keys from a randomly-chosen $M = 128$-key subset of the key space. Each epoch consisted of 500 timing measurements of 100 accesses per measurement. At the end of each epoch, a new 128-key subset was nominated for access. This procedure was repeated and averaged over 300 sequential runs to boost the signal-to-noise ratio. The result of this protocol was a highly concentrated and highly time-dependent access distribution, meant to expose whether old self-adjusting zip trees cannot in fact learn new tricks.

The result of this experiment, shown in **Figure 8**, suggests that this hypothesis is plausible. Not only do self-adjusting zip trees take much longer to adjust to a new distribution—a

dataset	inserts	deletes	accesses	distinct keys
bll	13069	12807	77148	324
br-actif	32522	31706	54071	633
br-support	20190	19794	16028	423
br	16664	15987	58739	375
faerix	20576	19992	14591	287
federez	49746	48953	89430	1320
pasloin	13203	13061	84669	395
platal	47887	47092	159297	1401
ragal	17656	17120	5190	236
root-br	22834	22238	120695	401
x	51492	50812	1878	1308
x2013	10508	10262	22111	241
x2014	17644	17282	77114	397

Table 2. Number of operations in each dataset

dataset	zip tree	frac-rank	self-adjust	skip list	treap	splay tree	red-black
bll	7.4±1.5	7.0±1.3	8.6±1.6	13.4±2.7	6.2±1.2	5.1±1.2	11.0±2.3
br-actif	10.4±3.7	9.8±3.4	11.3±3.5	20.1±7.7	9.2±3.5	7.3±2.6	12.9±3.1
br-support	9.4±3.4	8.9±3.2	10.1±3.4	18.7±6.8	8.4±3.1	6.7±2.4	11.1±3.6
br	9.1±3.2	8.7±3.0	9.8±3.2	17.7±6.4	8.0±2.9	6.4±2.2	10.7±3.3
faerix	8.7±3.1	8.3±2.9	9.2±3.1	17.2±6.0	7.7±2.7	6.1±2.1	10.0±3.4
federez	11.2±7.0	10.6±6.3	11.8±6.9	22.0±13.0	10.0±6.1	7.7±4.6	13.0±15.9
pasloin	10.8±6.6	10.2±5.9	11.5±6.5	21.0±12.3	9.5±5.7	7.5±4.3	12.8±14.8
platal	12.7±8.3	12.0±7.5	13.5±8.4	24.5±15.5	11.2±7.2	8.8±5.6	15.3±16.1
ragal	12.0±8.2	11.3±7.3	12.7±8.3	23.3±15.0	10.6±7.1	8.4±5.4	14.3±15.4
root-br	12.1±7.8	11.4±7.0	12.8±7.9	23.3±14.4	10.6±6.7	8.3±5.2	14.7±14.7
x	12.7±7.8	12.0±7.0	13.3±7.8	24.8±14.7	11.3±6.8	8.8±5.2	14.9±14.1
x2013	12.0±7.8	11.3±7.1	12.6±7.9	23.5±14.8	10.7±6.9	8.3±5.3	14.1±13.8
x2014	11.8±7.6	11.1±6.9	12.4±7.6	22.9±14.3	10.4±6.7	8.2±5.1	13.9±13.3

Table 3. Runtimes in ms of each sequence of operations for each dataset on each data structure. We report the mean over 1000 runs, along with the standard deviation

slow asymptote to optimality, as opposed to the splay tree's $O(M)$-time adjustment period, is expected—the rate of convergence also visibly decreases from epoch to epoch. We consider this evidence that self-adjusting zip trees are fundamentally less agile than splay trees, and would be a poor choice to replace them in time-dependent environments. On a slightly deeper level, it also indicates that not just any casually contrived self-adjustment scheme can reap the same benefits as splaying.

5.6. **Evaluation on real-world data.** Finally, we evaluate the performance of the data structures on a real-world instance of the dictionary problem. The data was collected using three years of IRC[11] logs across 13 public channels. We considered the situation from the perspective of the IRC server that has to associate each nickname to its corresponding IP address. When a user joins the chat room, we insert an item into the dictionary and we delete it when she leaves. Every message or action is treated as an access to the dictionary.

We report the statistics of our datasets in **Table 2** and our results in **Table 3**. We find that zip trees, treaps and splay trees perform similarly, all within a standard deviation of one another. Splay trees consistently performed the best, followed by treaps and then frac-rank zip trees. The varying sizes and number of operations in the different data sets did not appear to

[11]https://www.rfc-editor.org/info/rfc1459

have an affect on the relative performance of frac-rank zip tree and splay trees, with frac-rank zip trees consistently performing around 35% slower.

This is consistent with our results on synthetic data. The slight advantage of treaps over zip trees is attenuated when using the fractional ranks variant. The self-adjusting variant of zip trees lags behind: its efforts to rotate up the last accessed element are not rewarded by frequent enough accesses. On such a small dictionary size, red-black trees underperform, due to the huge overhead of the `std::map` implementation. Skip lists consistently have the worst performance.

6. Conclusion

In comparison with general BST dictionaries, zip trees are unlikely to revolutionize the undergraduate data structures curriculum or to supplant classic deterministic balanced BSTs in major language libraries. However, it genuinely excels within its limited niche. Among its sister randomized BSTs, the zip tree has a unique advantage. The treap, though usually faster than the zip tree, is relatively ill-suited to applications demanding only local pointer modifications. The skip list, on the other hand, *does* occupy this niche, but is slower on all dictionary operations—in all test cases we examined—than the zip tree, and uses substantially more memory. The zip tree is also relatively simple to implement, maintaining an advantage over red-black trees, especially for small data sets.

In the situations where zip trees lag behind the other data structures, we demonstrated how zip trees variants could help narrow the performance gaps with its main competitors (using fractional ranks to act more like a treap, or using a self-adjusting strategy to take advantage of skewed distributions). However, we pointed out that even the self-adjusting strategy as described by [1] suffers from both theoretical and practical drawbacks when compared to actual splay trees.

Although we considered a not-insignificant variety of data structures, augmentations, test cases and figures of merit, this study was far from exhaustive. We initially wanted to include a few more standard performance measures which did not make the final cut. Among these are figures of merit like number of pointer operations per dictionary method call, and number of comparisons per call. A more meticulous analysis of memory layout might reveal worthwhile insights on minimizing the memory footprint of these data structures. Finally, the argument that self-adjusting zip trees are not as "agile" as splay trees, however convincingly backed by our experiments it is, would benefit from a more rigorous and quantitative analysis.

References

[1] R. E. Tarjan, C. E. Levy, S. Timmel. Zip trees. `arXiv:1806.06726v2`.
[2] L. Lamport. Turing lecture: The computer science of concurrency: the early years. *Communications of the ACM*, 58(6):71-76, 2013.
[3] E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.
[4] B. Pfaff. Performance analysis of BSTs in system software. *Proceedings of the joint international conference on measurement and modeling of computer systems*, 410-411, 2004.
[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms. 3rd edition. The MIT press, 2009.
[6] D.E. Knuth. The Art of Computer Programming, volume 3: Sorting and Searching. Addison Wesley, 1973.
[7] L. J. Guibas, R. Sedgewick. A dichromatic framework for balanced trees. *19^th annual symposium on the foundations of computer science (FOCS)*, 8-21, 1978.
[8] G. M. Adel'son-Velskii and Y.M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady* 3:1259-1262, 1962.
[9] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *Proceedings of the fourth annual ACM symposium on theory of computing*, 137-142, 1972.
[10] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4-5):464-497, 1996.
[11] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229-239, 1980.
[12] C. Martínez and S. Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):228-323, 1998.

[13] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668-676, 1990.

[14] B. C. Dean and Z. H. Jones. Exploring the duality between skip lists and binary search trees. *Proceedings of the 45th Annual Southeast Regional Conference*, 395-400, 2007.

[15] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32 (1985) 652-686.

[16] https://www.link.cs.cmu.edu/link/ftp-site/splaying/top-down-splay.c

[17] G. Blelloch and M. Reid-Miller. Fast set operations using treaps. *ACM symposium on parallel algorithms and architectures (SPAA)*, 16-26, 1998.

[18] J. Cao, W. S. Cleveland, D. Lin, and D. Sun. On the nonstationarity of internet traffic. *Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 102-112.