Push-Down Trees: Optimal Self-Adjusting Complete Trees

Chen Avin, Kaushik Mondal, and Stefan Schmid

***Abstract*—This paper studies a fundamental algorithmic problem related to the design of demand-aware networks: networks whose topologies adjust toward the traffic patterns they serve, in an online manner. The goal is to strike a tradeoff between the benefits of such adjustments (shorter routes) and their costs (reconfigurations). In particular, we consider the problem of designing a self-adjusting tree network which serves single-source, multi-destination communication. The problem is a central building block for more general self-adjusting network designs and has interesting connections to self-adjusting datastructures. We present two constant-competitive online algorithms for this problem, one randomized and one deterministic. Our approach is based on a natural notion of *Most Recently Used (MRU)* tree, maintaining a *working set*. We prove that the working set is a cost lower bound for any online algorithm, and then present a randomized algorithm RANDOM-PUSH which *approximates* such an MRU tree at low cost, by pushing less recently used communication partners down the tree, along a random walk. Our deterministic algorithm MOVE-HALF does not directly maintain an MRU tree, but its cost is still proportional to the cost of an MRU tree, and also matches the working set lower bound.**

***Index Terms*—Reconfigurable networks, Online algorithms, Self-adjusting datastructures, Competitive analysis**

I. INTRODUCTION

While datacenter networks traditionally rely on a *fixed* topology, recent optical technologies enable *reconfigurable* topologies which can adjust to the demand (i.e., traffic pattern) they serve *in an online manner*, e.g. [15], [27], [30], [32], [33], [42]. Indeed, the physical topology is emerging as the next frontier in an ongoing effort to render networked systems more flexible.

In principle, such topological reconfigurations can be used to provide shorter routes between frequently communicating nodes, exploiting structure in traffic patterns [6], [37], [42], and hence to improve performance. However, the design of self-adjusting networks which dynamically optimize themselves toward the demand introduces an algorithmic challenge: an online algorithm needs to be devised which guarantees an efficient tradeoff between the benefits (i.e., shorter route lengths) and costs (in terms of reconfigurations) of topological optimizations.

This paper focuses on the design of a self-adjusting *complete tree (*CT*)* network: a network of nodes (e.g., top-of-rack switches) that forms a complete tree (i.e., the tree is balanced and each internal node has degree 3), and we measure the routing cost in terms of the path length between two nodes in the tree. Trees are not only a most fundamental topological structure of their own merit, but also a crucial building block for more general self-adjusting network designs: it is known that multiple tree networks (optimized individually for a single source node, hence called *ego-trees*) can be combined to build general networks which provide low degree and low distortion [7], [9], [45]. The design of a dynamic single-source multi-destination communication tree, as studied in this paper, is hence a stepping stone. Indeed, reconfigurable networks such as ReNets [12] leverage dynamic ego-trees to efficiently serve sparse traffic matrices which can evolve over time, as it is typically the case in practice [6], [37].

The focus on trees is further motivated by a relationship of our problem to problems arising in self-adjusting datastructures [11]: self-adjusting datastructures such as self-adjusting search trees [53] have the appealing property that they optimize themselves to the workload, leveraging temporal locality, but without knowing the future. Ideally, self-adjusting datastructures store items which will be accessed (frequently) *in the future*, in a way that they can be accessed quickly (e.g., close to the root, in case of a binary search tree), while also accounting for reconfiguration costs. However, in contrast to most datastructures, in a *network*, the search property is not required: the network supports *routing*. Accordingly our model can be seen as a novel flavor of such self-adjusting binary search trees[1] where lookup is supported by a *source routing*: source routing simplifies the control plane of a reconfigurable network, as it avoids complex re-computations of the paths under topological changes. In datastructure terminology, the source maintains a *map* about the locations of possible destinations on the tree, and adds the corresponding path to the packet header. More details will follow.

We present a formal model for this problem later, but a few observations are easy to make. If we restrict ourselves to the special case of a *line* network (a "linear tree"), the problem of optimally arranging the destinations of a given single communication source is equivalent to the well-known *dynamic list update* problem: for such self-adjusting (unordered) lists, constant-competitive online algorithms (known as *dynamically optimal* [23]) have been known for a long time [52]. In particular, the simple move-to-front algorithm which immediately promotes the accessed item to the front of the list, fulfills the *Most-Recently Used (MRU)* property: the i^{th} furthest away item from the front of the list is the i^{th} most recently used item. In the list (and hence on the line), this property is enough to guarantee optimality. The MRU

Chen Avin is with the Ben Gurion University of the Negev, School of Electrical and Computer Engineering, Israel

Kaushik Mondal is with Indian Institute of Technology Ropar, India

Stefan Schmid is with is with TU Berlin, Germany, and the University of Vienna, Austria.

[1]In binary search trees, each internal node stores a value greater than all the values in the node's left subtree and smaller than all the values in its right subtree, and hence searching a value is easy.

property is related to the so called *working set property*: the cost of accessing item x at time t depends on the number of distinct items accessed since the last access of x prior to time t, including x. Naturally, we wonder whether the *MRU* property is enough to guarantee optimality also in our case. The answer turns out to be non-trivial.

A first contribution of this paper is the observation that if we count only *access* cost (ignoring any rearrangement cost, see Definition 1 for details), the answer is affirmative: the most-recently used tree is what is called *access optimal*. Furthermore, we show that the corresponding access cost is a lower bound for any algorithm which is dynamically optimal. But securing this property, i.e., maintaining the most-recently used items close to the root in the tree, introduces a new challenge: how to achieve this *at low cost*? In particular, assuming that *swapping* the locations of items comes at a *unit cost*, can the property be maintained at cost proportional to the *access* cost? As we show, *strictly* enforcing the most-recently used property in a tree is too costly to achieve optimality. But, as we will show, when turning to an *approximate* most-recently used property, we are able to show two important properties: *i)* such an approximation is good enough to guarantee access optimality; and *ii)* it can be maintained in expectation using a *randomized* algorithm: less recently used communication partners are pushed down the tree along a random walk.

While the most-recently used property is *sufficient*, it is not necessary: we provide a deterministic algorithm which is dynamically optimal but does not even maintain the MRU property approximately. However, its cost is still proportional to the cost of an MRU tree (Definition 6).

Succinctly, we make the following ***contributions***. First we show a working set lower bound for our problem. We do so by proving that an MRU tree is *access optimal*. In the following theorem, let $WS(\sigma)$ denote the working set of σ (a formal definition will follow later).

Theorem 1: Consider a request sequence σ. Any algorithm ALG serving σ using a self-adjusting complete tree, has cost at least $\mathrm{cost}(\textsc{Alg}(\sigma)) \geq WS(\sigma)/4$, where $WS(\sigma)$ is the working set of σ.

Our main contribution is a deterministic online algorithm MOVE-HALF which maintains a constant competitive self-adjusting Complete Tree (CT) network.

Theorem 2: MOVE-HALF algorithm is dynamically optimal.

Interestingly, MOVE-HALF does not require the MRU property and hence does not need to maintain MRU tree. This implies that maintaining a working set on CTs is not a necessary condition for dynamic optimality, although it is a sufficient one.

Furthermore, we present a dynamically optimal, i.e., constant competitive (on expectation) randomized algorithm for self-adjusting CTs called RANDOM-PUSH. RANDOM-PUSH relies on maintaining an approximate MRU tree.

Theorem 3: The RANDOM-PUSH algorithm is dynamically optimal on expectation.

II. MODEL AND PRELIMINARIES

We first present our formal model in an abstract form. Subsequently, we will put the model into perspective with regards to demand-aware reconfigurable network topologies.

A. Abstract Model

Our problem can be formalized using the following simple model. We consider a single *source* that needs to communicate with a set of n nodes $V = \{v_1, \ldots, v_n\}$. The nodes are arranged in a complete binary tree and the source is connected to the root of the tree. While the tree describes a reconfigurable *network*, we will use terminology from datastructures, to highlight this relationship and avoid the need to introduce new terms.

We consider a complete tree T connecting n servers $S = \{s_1, \ldots, s_n\}$. We will denote by $s_1(T)$ the root of the tree T, or s_1 when T is clear from the context, and by $s_i.\mathrm{left}$ (resp. $s_i.\mathrm{right}$) the left (resp. right) child of server s_i. We assume that the n servers store n items (nodes) $V = \{v_1, \ldots, v_n\}$, one item per server. For any $i \in [1, n]$ and any time t, we will denote by $s_i.\mathrm{guest}^{(t)} \in V$ the item mapped to s_i at time t. Similarly, $v_i.\mathrm{host}^{(t)} \in S$ denotes the server hosting item v_i. Note that if $v_i.\mathrm{host}^{(t)} = s_j$ then $s_j.\mathrm{guest}^{(t)} = v_i$.

The *depth* of a server s_i is fixed and describes the distance from the root; it is denoted by $s_i.\mathrm{dep}$, and $s_1.\mathrm{dep} = 0$. The depth of an item v_i *at time* t is denoted by $v_i.\mathrm{dep}^{(t)}$, and is given by the depth of the server to which v_i is mapped at time t. Note that $v_i.\mathrm{dep}^{(t)} = v_i.\mathrm{host}.\mathrm{dep}^{(t)}$.

To this end, we interpret communication requests from the source as *accesses* to *items* stored in the (unordered) tree. All access requests (resp. communication requests) to items (resp. nodes) originate from the root s_1. If an item (resp. node) is frequently requested, it can make sense to move this item (node) closer to the root of T: this is achieved by *swapping* items which are neighboring in the tree (resp. by performing local topological swaps).

Access requests occur over time, forming a (finite or infinite) sequence $\sigma = (\sigma^{(1)}, \sigma^{(2)}, \ldots)$, where $\sigma^{(t)} = v_i \in V$ denotes that item v_i is requested, and needs to be accessed at time t. The sequence σ (henceforth also called the *workload*) is revealed one-by-one to an online algorithm ON. The *working set* of an item v_i at time t is the set of distinct items accessed since the last access of v_i prior to time t, including v_i. We define the *rank* of item v_i at time t to be the size of the working set of v_i at time t and denote it as $v_i.\mathrm{rank}^{(t)}$. When t is clear of context, we simply write $v_i.\mathrm{rank}$. The working set bound of sequence σ of m requests is defined as $WS(\sigma) = \sum_{t=1}^{m} \log(\sigma^{(t)}.\mathrm{rank})$.

Both serving (i.e., *routing*) the request and adjusting the configuration comes at a cost. We will discuss the two cost components in turn. Upon a request, i.e., whenever the source wants to communicate to a partner, it routes to it via the tree T. To this end, a message passed between nodes can include, for each node it passes, a bit indicating which child to forward the message next (requires $O(\log n)$ bits). Such a *source routing* header can be built based on a dynamic global *map* of the tree that is maintained at the source node. As mentioned, the source node is a direct neighbor of the root of the tree, aware of all requests, and therefore it can maintain the map. The *access cost* is hence given by the distance between the root and the

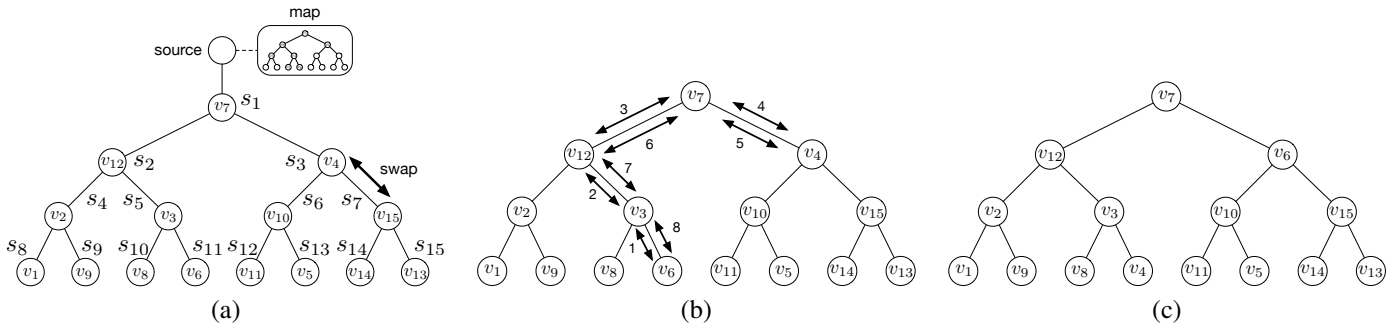

Fig. 1. (a) Our *complete tree* model: a source with a map, a tree of servers that host items (nodes) and a *swap* operation between neigboring items. (b) The node's tree network implied by the tree T from (a) and the set of swaps needed to interchange the location of v_6 and v_4. (c) The tree network after the interchange and swap operations of (b).

requested item, which is basically the depth of the item in the tree.

The *reconfiguration cost* is due to the adjustments that an algorithm performs on the tree. We define the unit cost of reconfiguration as a *swap*: a swap means changing position of an item with its parent. Note that, any two items u, v in the tree can be *interchanged* using a number of swaps equal to twice the distance between them. This can be achieved by u first swapping along the path to v and then v swapping along the same path to initial location of u. This interchange operation results in the tree staying the same, but only u and v changing locations. We assume that to interchange items, we first need to access one of them. See Figure 1 for an example of our model and interchange operation.

Definition 1 (Cost): The cost incurred by an algorithm ALG to serve a request $\sigma^{(t)} = v_i$ is denoted by $\mathrm{cost}(\textsc{Alg}(\sigma^{(t)}))$, short $\mathrm{cost}^{(t)}$. It consists of two parts, *access* cost, denoted $\text{acc-cost}^{(t)}$, and *adjustment* cost, denoted $\text{adj-cost}^{(t)}$. We define access cost simply as $\text{acc-cost}^{(t)} = v_i.\mathrm{dep}^{(t)}$ since ALG can maintain a global *map* and access v_i via the shortest path. Adjustment cost, $\text{adj-cost}^{(t)}$, is the total number of swaps, where a single swap means changing position of an item with its parent or a child. The total cost, incurred by ALG is then

$$\begin{aligned}\mathrm{cost}(\textsc{Alg}(\sigma)) &= \sum_t \mathrm{cost}(\textsc{Alg}(\sigma^{(t)})) \\ &= \sum_t \mathrm{cost}^{(t)} = \sum_t (\text{acc-cost}^{(t)} + \text{adj-cost}^{(t)})\end{aligned}$$

Our main objective is to design online algorithms that perform almost as well as optimal offline algorithms (which know σ ahead of time), even in the worst-case. In other words, we want to devise online algorithms which minimize the competitive ratio:

Definition 2 (Competitive Ratio ρ): We consider the standard definition of (strict) competitive ratio ρ, i.e.,
$\rho = \max_\sigma \mathrm{cost}(\textsc{On})/\mathrm{cost}(\textsc{Opt})$ where σ is any input sequence and where OPT denotes the optimal offline algorithm.
If an online algorithm is constant competitive, independently of the problem input, it is called *dynamically optimal*.

Definition 3 (Dynamic Optimality): An (online) algorithm ON achieves *dynamic optimality* if it asymptotically matches the offline optimum on every access sequence. In other words, the algorithm ON is $O(1)$-competitive.

We also consider a weaker form of competitivity (similarly to the notion of *search-optimality* in related work [14]), and say that ON is *access-competitive* if we consider only the access cost of ON (and ignore any adjustment cost) when comparing it to OPT (which needs to pay both for access and adjustment). For a randomized algorithm, we consider an oblivious online adversary which does not know the random bits of the online algorithm a priori.

The ***Self-adjusting Complete Tree Problem*** considered in this paper can then be formulated as follows: Find an online algorithm which serves any (finite or infinite) online request sequence σ with minimum cost (including both access and rearrangement costs), on a self-adjusting complete binary tree.

B. Putting the Model into Perspective

Our model above revolves around a single tree, where requests originate from a root. The practical motivation behind this model is threefold. First, (dynamic) trees can easily be realized with optical switches that support (dynamic) matchings, for example optical spine switches that connect Top-of-Rack (ToR) switches in dynamic topologies [30]. Second and most importantly, it has been shown by Avin et al. [7] that efficient more general demand-aware networks can be built from such trees which are optimized toward an individual source: given the individual trees of the different sources in a demand matrix (called *ego-trees* in the literature), rooted at the source, it is possible to construct a constant-degree topology which interconnects all sources with their destinations, without significantly distorting the route lengths. Essentially, the demand-aware network is simply the union of these ego-trees, where the degree is reduced in a postprocessing step. Since the original work by Avin et al., this network design principle has been successfully employed to design a variety of static and dynamic demand-aware networks, e.g., [7], [9], [10], [12], [45]. For example, it has been shown that the approach cannot only be used to minimize route lengths, but also to optimize other network metrics such as congestion [9]. However, so far, dynamic demand-aware networks such as ReNets [12] do not provide any non-trivial competitive guarantees against offline algorithms, only against static algorithms.

The third motivation behind our model arises in the context of datastructures. In particular, self-adjusting binary search trees are rooted trees whose topology dynamically adjusts to

optimally serve requests originating at the tree's root. A key difference is that while datastructures need to be *searchable*, networks generally do not have this constraint due to the presence of routing protocols. That said, it is important to ensure that routing protocols are implemented efficiently, as in dynamic topologies there are frequent updates. Accordingly, we in this paper employ *source routing*, which we model with a *map* at each source node: the map allows us to trivially access a node (or item) at distance k from the front at a cost k. Interestingly, while the quest for constant competitive online algorithms for binary search trees remains a major open problem [53], we will show in this paper that, under our cost model, dynamically optimal algorithms for tree networks exist.[2]

Finally, for simplicity of terminology and due to the connection of our model to datastructure literature, we presented our abstract model above in terms of servers and items.

III. Warm Up: Optimal Fixed Trees

The key difference between binary search trees and binary trees is that the latter provides more flexibilities in how items can be arranged on the tree. Accordingly, one may wonder whether more flexibilities will render the optimal data structure design problem algorithmically simpler or harder.

In this section, we consider the static problem variant, and investigate offline algorithms to compute optimal trees for a *fixed* frequency distribution over the items. To this end, we assume that for each item v, we are given a frequency $v.\mathrm{freq}$, where $\sum_{v \in V} v.\mathrm{freq} = 1$.

Definition 4 (Optimal Fixed Tree): We call a tree *optimal static tree* if it minimizes the expected path length $\sum_{i \in [1,n]} (v_i.\mathrm{freq} \cdot v_i.\mathrm{dep})$.

Our objective is to design an optimal static tree according to Definition 4. Now, let us define the following notion of *Most Frequently Used (MFU)* tree which keeps items of larger empirical frequencies closer to the root:

Definition 5 (MFU Tree): A tree in which for every pair of items $v_i, v_j \in V$, it holds that if $v_i.\mathrm{freq} \geq v_j.\mathrm{freq}$ then $v_i.\mathrm{dep} \leq v_j.\mathrm{dep}$, is called *MFU tree*.

Observe that MFU trees are not unique but rather, there are many MFU trees. In particular, the positions of items at the same depth can be changed arbitrarily without violating the MFU properties.

Theorem 4 (Optimal Fixed Trees): Any MFU tree is an optimal fixed tree.

Proof: Recall that by definition, MFU trees have the property that for all node pairs v_i, v_j: $v_i.\mathrm{freq} > v_j.\mathrm{freq} \Rightarrow v_i.\mathrm{dep} \leq v_j.\mathrm{dep}$. For the sake of contradiction, assume that there is a tree T which achieves the minimum expected path length but for which there exists at least one item pair v_i, v_j which violates our assumption, i.e., it holds that $v_i.\mathrm{freq} > v_j.\mathrm{freq}$ but $v_i.\mathrm{dep} > v_j.\mathrm{dep}$. From this, we can derive a contradiction to the minimum expected path length: by swapping the positions of items v_i and v_j, we obtain a tree T' with an expected path length which is shorter by

$$\begin{aligned}\mathrm{cost}(T,\sigma) - \mathrm{cost}(T',\sigma) &= (v_i.\mathrm{freq} \cdot v_i.\mathrm{dep} + v_j.\mathrm{freq} \cdot v_j.\mathrm{dep}) \\ &\quad - (v_i.\mathrm{freq} \cdot v_j.\mathrm{dep} + v_j.\mathrm{freq} \cdot v_i.\mathrm{dep}) \\ &> 0\end{aligned}$$

□

MFU trees can also be constructed very efficiently, e.g., by performing the following *ordered insertion*: we insert the items into the tree T in a top-down, left-to-right manner, in descending order of their frequencies (i.e., item v_i is inserted before item v_j if $v_i.\mathrm{freq} > v_j.\mathrm{freq}$).

IV. Access Optimality: A Working Set Lower Bound

For *fixed* trees, it is easy to see that keeping frequent items close to the root, i.e., using a *Most-Frequently Used* (MFU) policy, is optimal (cf. Appendix). The design of online algorithms for *adjusting* trees is more involved. In particular, it is known that MFU is not optimal for lists [52]. A natural strategy could be to try and keep items close to the root which have been frequent "recently". However, this raises the question over which time interval to compute the frequencies. Moreover, changing from one MFU tree to another one may entail high adjustment costs.

This section introduces a natural *pendant* to the MFU tree for a dynamic setting: the *Most Recently Used (MRU) tree*. Intuitively, the MRU tree tries to keep the "working set" resp. *recently* accessed items close to the root. In this section we consider online request sequences and show a working set lower bound for any self-adjusting complete binary tree.

While the move-to-front algorithm, known to be dynamically optimal for self-adjusting lists [52], naturally provides such a "most recently used" property, generalizing move-to-front to the tree is non-trivial. We therefore first show that any algorithm that maintains an MRU tree is *access-competitive*. With this in mind, let us first formally define the MRU tree.

Definition 6 (MRU Tree): For a given time t, a tree T is an *MRU* tree if and only if,

$$v_i.\mathrm{dep} = \lfloor \log v_i.\mathrm{rank} \rfloor \tag{1}$$

Accordingly the root of the tree (level zero) will always host an item of rank one. More generally, servers in level i will host items that have a rank between $(2^i, 2^{i+1} - 1)$. Upon a request of an item, say v_j with rank r, the rank of v_j is updated to one, and only the ranks of items with rank smaller than r are increased, each by 1. Therefore, the rank of items with rank higher than r do not change and their level (i.e., depth) in the MRU tree remains the same (but they may switch location within the same level).

Definition 7 (MRU algorithm): An online algorithm On has the *MRU* property (or the working set property) if for each time t, the tree $T^{(t)}$ that On maintains, is an *MRU* tree.

The working set lower bound will follow from the following theorem (Theorem 5) which states that any algorithm that has the *MRU* property is *access competitive*. Recall that an analogous statement of Theorem 5 is known to be true for a *list* [52]. As such, one would hope to find a simple proof that

[2]There are self-adjusting binary search trees that are known to be *access optimal* [14], but their rearrangement cost it too high.

holds for complete trees, but it turns out that this is not trivial, since OPT has more freedom in trees. We therefore present a direct proof based on a potential function, similar in spirit to the list case.

Theorem 5: Any online algorithm ON that has the *MRU* property is 4 access-competitive.

Proof: Consider the two algorithms ON and OPT. We employ a potential function argument which is based on the difference in the items' locations between ON's tree and OPT's tree. For any server s_i, we define a pair (s_i, s_j) as *bad* on a tree of some algorithm A if $s_i.\text{dep} < s_j.\text{dep}$ but $s_i.\text{guest.rank}(A) > s_j.\text{guest.rank}(A)$, i.e., s_i is at a lower level although $s_j.\text{guest}(A)$ has been accessed more recently. We observe that any bad pair (s_i, s_j) for s_i is an ordered pair, i.e., this pair is not bad for s_j. Also note that, for any server s_i, $s_i.\text{dep}$ is same on any tree for any algorithm, what may differ is $s_i.\text{guest}$ resp. $s_i.\text{guest.rank}$. Since ON has the *MRU* property it follows by definition that none of its pairs are bad. Hence bad pairs appear only on OPT's tree. Let, for any algorithm A, $\alpha_i(A)$ denote the number of *bad* pairs for s_i in A's tree. Let $B_i(A)$ be equal to one plus $\alpha_i(A)$ divided by the number of items at level $s_i.\text{dep}$. More formally,

$$B_i(A) = 1 + \frac{\alpha_i(A)}{2^{s_i.\text{dep}}}$$

Define $B(A) = \prod_{i=1}^{n} B_i(A)$. Now we define the potential function $\Phi = \log B(\text{OPT}) - \log B(\text{ON})$ which is based on the difference in the number of bad pairs between ON's tree and OPT's tree. According to our definition, $B(\text{ON}) = 1$ and hence $\Phi = \log B(\text{OPT})$. Therefore, from now onwards, we use B resp. $\log B$ instead of $B(\text{OPT})$ resp. $\log B(\text{OPT})$. We consider the occurrence of events in the following order. Upon a request, ON adjusts its tree, then OPT performs the rearrangements it requires.

Let the potential at time t be Φ (i.e., before ON's adjustment, after serving request $\sigma^{(t)}$, and before OPT's rearrangements between requests t and $t+1$) and the potential after ON adjusted to its tree be Φ'. Then the potential change due to ON's adjustment is

$$\Delta\Phi_1 = \Phi' - \Phi = \log B' - \log B = \log \frac{B'}{B}$$

We assume that the initial potential is 0 (i.e., no item was accessed). Since the potential is always positive by definition, we can use it to bound the *amortized* cost of ON, $\text{amortized}(\text{ON})$. Consider a request at time t to an item at depth k in the tree of ON. The access-cost is $\text{cost}^{(t)}(\text{ON}) = k$ and we would like to have the following bound: $\text{amortized}^{(t)}(\text{ON}) \leq \text{cost}^{(t)}(\text{ON}) + \Delta\Phi$. Assume that the requested item $\sigma^{(t)}$ is at server s_r at depth j in OPT's tree, so OPT must pay at least an access cost of j. Let k be the depth of $\sigma^{(t)}$ in ON tree. First we assume that $j < k$.

Let us compute the potential after ON updated its MRU tree. For any server s_i at depth lower than j i.e., for which $s_i.\text{dep} < j$, it holds that

$$B_i' = 1 + \alpha_i'/2^{s_i.\text{dep}} = 1 + (\alpha_i + 1)/2^{s_i.\text{dep}} = B_i + 1/2^{s_i.\text{dep}}$$

This is true since the rank of the guest of the last accessed server, s_r, changed (to 1) and hence α_i increases by 1 for each server server s_i s.t. $s_i.\text{dep} < j$. Additionally, for all servers for which $s_i.\text{dep} \geq j$ (excluding s_r), $B_i' = B_i$. The potential of the accessed server, s_r, will be $B_r' = 1$, since its guest's rank becomes 1. Although due to the access, the rank of some other items increase by 1, that does not affect the number of bad pairs. Let the rank of the requested item $\sigma^{(t)}$ before it was accessed be $\sigma^{(t)}.\text{rank}$. After the access at time t, the rank of all the items with rank lesser than $\sigma^{(t)}.\text{rank}$ will increase by 1. Consider any pair (s_i, s_j) before the access of $\sigma^{(t)}$. We have already seen what happens if either s_i or s_j is s_r. Otherwise a pair (s_i, s_j) cannot change from bad to good (resp. good to bad) since if only $s_j.\text{rank}$ (resp. $s_i.\text{rank}$) increases by 1, it cannot be more than that of s_i (resp. s_j).

To compute B', we use the following inequality.

$$\prod_{i=1}^{n}(x_i + 1/n) \leq e \prod_{i=1}^{n} x_i \qquad x_i \geq 1 \forall i$$

We prove it below.

$$\begin{aligned}\prod_{i=1}^{n}(x_i + 1/n) &\leq \prod_{i=1}^{n}(x_i + x_i/n) = \prod_{i=1}^{n}(x_i(1+1/n)) \\ &= (1+1/n)^n \prod_{i=1}^{n}(x_i) \leq e \prod_{i=1}^{n} x_i\end{aligned}$$

where we used the inequality $(1+1/n)^n \leq e$, $n \in \mathbb{N}$. Now we compute B':

$$\begin{aligned}B' = \prod_{i=1}^{n} B_i' &= \left(\prod_{s_i.\text{dep}<j}(B_i + \frac{1}{2^{s_i.\text{dep}}})\right)\left(\prod_{\substack{s_i.\text{dep}\geq j \\ i\neq r}} B_i\right) B_r' \\ &= \left(\prod_{p=0}^{j-1}\left(\prod_{i=1}^{2^p}(B_i + \frac{1}{2^p})\right)\right)\left(\prod_{\substack{s_i.\text{dep}\geq j \\ i\neq r}} B_i\right) B_r' \\ &\leq \left(\prod_{p=0}^{j-1} eB_i\right)\left(\prod_{\substack{s_i.\text{dep}\geq j \\ i\neq r}} B_i\right) B_r' \\ &= \frac{e^j}{B_r}\prod_{i=1}^{n} B_i = \frac{e^j}{B_r} B\end{aligned}$$

The last line results from multiplying and dividing by B_r and recalling that $B_r' = 1$. Note that $s_r.\text{dep} = j$ and $j < k$. Since $s_r.\text{guest}$ is at depth k before the last access in ON's tree, its rank is at least 2^k. But in the OPT's tree, at most $2^{j+1} - 1$ elements among those 2^k elements are at depth j or less as OPT's tree contains no more than $2^{j+1} - 1$ many elements in the top j levels. So, there are at least $(2^k) - (2^{j+1} - 1)$ many elements whose rank is more than that of $s_r.\text{guest}$ and the depth is more than j in OPT's tree. Hence,

$$B_r \geq 1 + \frac{(2^k) - (2^{j+1} - 1)}{2^j} \geq \frac{2^k}{2^j} = 2^{k-j}$$

$$\Delta\Phi_1 = \log\frac{B'}{B} = \log\frac{e^j B}{B_r B} = \log\frac{e^j}{B_r}$$
$$\leq \log\frac{e^j}{2^{k-j}} = j\log_2 e - (k-j) = j(1+\log_2 e) - k$$

Now we consider $j \geq k$. In this case also, for any server for which $s_i.\mathrm{dep} = j' < j$, it holds that $B'_i \leq B_i + 1$ and for all servers for which $s_i.\mathrm{dep} \geq j$ (excluding s_r), $B'_i = B_i$. Again $B'_r = 1$ but $B_r \geq 1$ since $j \geq k$. By similar calculations, we get $B'\frac{2^j}{B_r}B$ and then, $\Delta\Phi_1 \leq \log 2^j = j \leq 2j - k$. To complete the proof we need to compute the potential change due to OPT's rearrangements between accesses. Consider the potential after OPT adjusted its tree, Φ''. Then the potential change due to OPT's adjustment is

$$\Delta\Phi_2 = \Phi'' - \Phi' = \log B'' - \log B' = \log\frac{B''}{B'}$$

The only operation OPT performs is swap i.e., changing positions between parent and a child. OPT may need to change positions of items during rearrangement between accesses. These can always be done using multiple number of swaps, upwards or downwards or both. Below we compute potential difference due to such a swap. Let OPT access an item z at s_c from depth k', raising it to depth $k' - 1$ by swapping it with its parent z_p at s_p.

For all servers with $s_i.\mathrm{dep} = k' - 1$, except s_p, $B''_i \leq B'_i + 1/2^{k-1}$ holds, as z_p goes to level k' from $k' - 1$ and may become bad to all the servers at level $k' - 1$. For s_p, $B''_p \leq 2B'_c + \frac{2^{k'}}{2^{k'-1}} = 2B'_c + 2 \leq 4B'_c$, as all the items in layer k' may become bad w.r.t. z. Also $B''_c \leq B'_p$ Notice that changes only occur at depth $k' - 1$, nothing will change above or below that.

The computation of B'' is shown in Table I.

Now we compute the potential change due to OPT's single swap:

$$\Delta\Phi_2 = \log\frac{B''}{B'} \leq \log 4e < 4$$

The potential change is less than 4 per swap where OPT must pay one for that swap. If the number of swaps is m for the rearrangement of OPT between any two accesses, the potential change is bounded by $4m$. Putting it all together, we get

$$\begin{aligned}\mathrm{amortized}^{(t)}(\textsc{On}) &\leq \mathrm{cost}^{(t)}(\textsc{On}) + \Delta\Phi_1 + \Delta\Phi_2\\ &\leq k + (j(1+\log_2 e) - k) + 4m\\ &\leq 4(j+m) = 4\mathrm{cost}^{(t)}(\textsc{Opt})\end{aligned}$$

Finally,

$$\begin{aligned}\mathrm{cost}(\textsc{On}) &= \sum^{t}\mathrm{cost}^{(t)}(\textsc{On})\\ &= \sum^{t}\mathrm{amortized}^{(t)}(\textsc{On}) - (\Phi^{(t)} - \Phi^{(0)})\\ &\leq \sum^{t}\mathrm{amortized}^{(t)}(\textsc{On}) \leq \sum^{t} 4\mathrm{cost}^{(t)}(\textsc{Opt})\\ &= 4\mathrm{cost}(\textsc{Opt})\end{aligned}$$

□

Based on Theorem 5 we can now show our working set lower bound:

Theorem 1: Consider a request sequence σ. Any algorithm ALG serving σ using a self-adjusting complete tree, has cost at least $\mathrm{cost}(\textsc{Alg}(\sigma)) \geq WS(\sigma)/4$, where $WS(\sigma)$ is the working set of σ.

Proof: The sum of the access costs of items from an MRU tree is exactly $WS(\sigma)$. For the sake of contradiction assume that there is an algorithm ALG with cost $\mathrm{cost}(\textsc{Alg}(\sigma))$ less than $WS(\sigma)/4$. If follows that Theorem 5 is not true. A contradiction. □

V. Deterministic Algorithm

A. Efficiently Maintaining an MRU Tree

It follows from the previous section that if we can maintain an MRU tree at the cost of *accessing* an MRU tree, we will have a dynamically optimal algorithm. So we now turn our attention to the problem of efficiently maintaining an MRU tree. To achieve optimality, we need that the tree adjustment cost will be proportional to the access cost. In particular, we aim to design a tree which on one hand achieves a good approximation of the MRU property to capture temporal locality, by providing fast *access* (resp. *routing*) to items; and on the other hand is also adjustable at low cost over time.

Let us now assume that a certain item $\sigma^{(t)} = u$ is accessed at some time t. In order to re-establish the (strict) MRU property, u needs to be promoted to the root. This however raises the question of where to move the item currently located at the root, let us call it v. A natural idea to make space for u at the root while preserving locality, is to *push down* items from the root, including item v. However, note that simply pushing items down along the path between u and v (as done in lists) will result in a poor performance in the tree. To see this, let us denote the sequence of items along the path from u to v by $P = (u, w_1, w_2, \ldots, w_\ell, v)$, where $\ell = u.\mathrm{dep}$, *before* the adjustment. Now assume that the access sequence σ is such that it repeatedly cycles through the sequence P, in this order. The resulting cost per request is in the order of $\Theta(\ell)$, i.e., could reach $\Theta(\log n)$ for $\ell = \Theta(\log n)$. However, an algorithm which assigns (and then fixes) the items in P to the top $\log\ell$ levels of the tree, will converge to a cost of only $\Theta(\log\ell) \in O(\log\log n)$ per request: an exponential improvement.

Another basic idea is to try and keep the MRU property at every step. Let us call this strategy MAX-PUSH. Consider a request to item u which is at depth $u.\mathrm{dep} = k$. Initially u is moved to the root. Then the MAX-PUSH strategy chooses for each depth $i < u.\mathrm{dep}$, the *least* recently accessed (and with maximum rank) item from level i: formally, $w_i = \arg\max_{v \in V : v.\mathrm{dep} = i} v.\mathrm{rank}$. We then push w_i to the host of w_{i+1}. It is not hard to see that this strategy will actually maintain a perfect MRU tree. However, items with the maximum rank in different levels, i.e., $w_i.\mathrm{host}$ and $w_{i+1}.\mathrm{host}$, may not be in a parent-child relation. So to push w_i to $w_{i+1}.\mathrm{host}$, we may need to travel all the way from $w_i.\mathrm{host}$ to the root and then from the root to $w_{i+1}.\mathrm{host}$, resulting in a cost proportional to i per level i. This accumulates a rearrangement

$$
\begin{aligned}
B'' = \prod_{i=1}^{n} B''_i &\le \left(\prod_{s_i.\mathrm{dep}<k'-1} (B''_i)\right)\left(\prod_{\substack{s_i.\mathrm{dep}\ge k' \\ i\neq q}} B''_i\right)(B''_c)(B''_p)\left(\prod_{\substack{s_i.\mathrm{dep}=k'-1 \\ i\neq p}} (B''_i)\right) \\
&\le \left(\prod_{s_i.\mathrm{dep}<k'-1} (B'_i)\right)\left(\prod_{\substack{s_i.\mathrm{dep}\ge k' \\ i\neq q}} B'_i\right)(B'_p)(4B'_c)\left(\prod_{\substack{s_i.\mathrm{dep}=k'-1 \\ i\neq p}} (B'_i + \tfrac{1}{2^{k'-1}})\right) \\
&= \left(\prod_{s_i.\mathrm{dep}<k'-1} (B'_i)\right)\left(\prod_{\substack{s_i.\mathrm{dep}\ge k' \\ i\neq q}} B'_i\right)(B'_p)(4B'_c)\left(e\prod_{\substack{s_i.\mathrm{dep}=k'-1 \\ i\neq p}} (B'_i)\right) \\
&= 4e\left(\prod_{s_i.\mathrm{dep}<k'-1} (B'_i)\right)\left(\prod_{s_i.\mathrm{dep}\ge k'} B'_i\right)(B'_p)(B'_c)\left(\prod_{\substack{s_i.\mathrm{dep}=k'-1 \\ i\neq p}} (B'_i)\right) \\
&= 4eB'
\end{aligned}
$$

TABLE I
COMPUTATION OF B''

Algorithm 1: MOVE-HALF (Upon request to u in tree)

1: **access** $u = s.\mathrm{guest}$ along the tree branches (cost: $u.\mathrm{dep}$)
2: let v be the item with the highest rank at depth $\lfloor u.\mathrm{dep}/2 \rfloor$
3: **swap** u along tree branches to node v (cost: $\frac{3}{2}u.\mathrm{dep}$)
4: **swap** v along tree branches to server s (cost: $\frac{3}{2}u.\mathrm{dep}$)

cost of $\sum_{i=1}^{k} i > k^2/2$ to push all the items with maximum rank at each layer, up to layer k. This is not proportional to the original access cost k of the requested item and therefore, leads to a non-constant competitive ratio as high as $\Omega(\log n)$.

Later, in Section VI, we will present a randomized algorithm that maintains a tree that approximates an MRU tree at a low cost. But first, we will present a simple deterministic algorithm that does not directly maintain an MRU tree, but has cost that is proportional to the MRU cost and is hence dynamically optimal.

B. The MOVE-HALF Algorithm

In this section we propose a simple deterministic algorithm, MOVE-HALF, that is proven to be dynamically optimal. Interestingly MOVE-HALF does not maintain the MRU property but its cost is shown to be competitive to the *access cost* on an MRU tree, and therefore, to the working set lower bound.

MOVE-HALF is described in Algorithm 1. Initially, MOVE-HALF and OPT start from the same tree (which is assumed w.l.o.g. to be an MRU tree). Then, upon a request to an item u, MOVE-HALF first accesses u and then interchanges its position with node v that is the highest ranked item positioned at half of the depth of u in the tree. After the interchange the tree remains the same, only u and v changed locations. See Figure 1 (b) for an example of MOVE-HALF operation where v_6 at depth 3 is requested and is then interchanged with v_4 at depth 1 (assuming it is the highest rank node in level 1).

The *access cost* of MOVE-HALF is proportional to the access cost of an MRU tree.

Theorem 6: Algorithm MOVE-HALF is 4 access-competitive to an MRU algorithm.

Before going to the proof of Theorem 6, we discuss several properties of MOVE-HALF. First, we show that whenever any item v moves down in MOVE-HALF's tree, its depth is at most twice plus one when compared to its depth in an MRU tree.

Lemma 1: Whenever some item v moves down to depth h in MOVE-HALF's tree, it is at least at depth $\lfloor h/2 \rfloor$ in an MRU tree.

Proof: Upon a request of some item u, say, from depth h, let v replace u at depth h in MOVE-HALF's tree, from depth $\lfloor h/2 \rfloor$. At the time of this request, v must be the highest ranked item at depth $\lfloor h/2 \rfloor$ and accordingly, is replaced by u. As the depth of the root is zero, the total number of items in depth $\lfloor h/2 \rfloor$ is exactly $2^{\lfloor h/2 \rfloor}$. So $v.\mathrm{rank} \ge 2^{\lfloor h/2 \rfloor}$ at the time u is requested. Accordingly the position of v in an MRU tree is at least at depth $\lfloor h/2 \rfloor$ (see Equation 1). Therefore, the depth of v in MOVE-HALF's tree is at most twice plus one when compared to its depth in an MRU tree. □ Next, let $t = 0$ or a time where an item v was moved down in MOVE-HALF's tree. Let $t' > t$ be the first time that v was requested in σ after time t. Then we can claim the following:

Claim 1: If the depth of v in MOVE-HALF's tree is h at time t', then its depth in an MRU tree at time t' is at least $\lfloor h/2 \rfloor$.

Proof: For the case $t = 0$, since initially v is at the same depth in both trees, the claim follows trivially. If $t > 0$, then let t'' be the most recent time before t' that v was moved down. Then at time t'', item v was moved from some depth $\lfloor h/2 \rfloor$ to h. At time t'', according to Lemma 1, the depth of v in an MRU tree was at least $\lfloor h/2 \rfloor$. Clearly v's depth remains unchanged in MOVE-HALF's tree at time t', since time t'' was the most recent move down of v. Also since we consider the first request of v after time t, it means that the rank of v could only increase between t and t'. So its depth in an MRU tree

could not decrease from $\lfloor h/2 \rfloor$. □

We can now prove Theorem 6.

Proof:[Proof of Theorem 6] We analyze the access costs for an arbitrary item u during the entire run of the algorithm. Let t_1 be the time of the first request to u during the execution of σ. Let d_1 be the first time that u was moved down by MOVE-HALF. Then define t_i, $i > 1$ to be the first time after time d_{i-1} that u is requested. And let d_i be the first time after t_i that u is moved down by MOVE-HALF. Assume that the depth of u at time t_i is L. Then according to Claim 1 its depth at an MRU tree is $\lfloor L/2 \rfloor$. Let $t_i^1, t_i^2, \ldots, t_i^j$ denote all the requests for u between t_i and d_i. A total of j requests to u without any move down of u by MOVE-HALF. We can bound the access cost of MOVE-HALF on these requests as follows. If $j = 1$ it is L, if $j > 1$ then:

$$\begin{aligned} access(u, t_i^1, t_i^j)(\text{MOVE-HALF}) &\leq L + \lfloor L/2 \rfloor + ... + \lfloor L/2^{j-1} \rfloor \\ &\leq 2L \end{aligned}$$

On the other hand the access cost of an MRU algorithm for the same set of requests is bounded as follows. If $j = 1$ it is $\lfloor L/2 \rfloor$, if $j > 1$ then,

$$\begin{aligned} access(u, t_i^1, t_i^j)(\text{MRU}) &\geq \lfloor L/2 \rfloor + 1 + 1 + ... + 1 \\ &\geq \lfloor L/2 \rfloor + j - 1 \geq L/2 \end{aligned}$$

Therefore, for each i we have

$$\frac{access(u, t_i^1, t_i^j)(\text{MOVE-HALF})}{access(u, t_i^1, t_i^j)(\text{MRU})} \leq 4$$

This leads to the results that the total access cost for u in MOVE-HALF is 4-competitive to the total access for u in an MRU tree. Since this is true for each item in the sequence, MOVE-HALF is 4-access competitive compared to an MRU tree. □

Theorem 2: MOVE-HALF algorithm is dynamically optimal.

Proof:[Proof of Theorem 2] Using Theorem 5 and Theorem 6, MOVE-HALF is 16-access competitive. It is easy to see from Algorithm 1 that total cost of MOVE-HALF's tree is 4 times the access cost. Considering these, MOVE-HALF is 64-competitive. □

In the coming section we show techniques to maintain MRU trees cheaply. This is another way to maintain dynamic optimality.

VI. RANDOMIZED MRU TREES

The question of how, and if at all possible, to maintain an MRU tree deterministically (where for each request $\sigma^{(t)}$, $\sigma^{(t)}.depth = \lfloor \log \sigma^{(t)}.\text{rank} \rfloor$) at low cost is still an open problem. But, in this section we show that the answer is affirmative with two relaxations: namely by using randomization and approximation. We believe that the properties of the algorithm we describe next may also find applications in other settings, and in particular data structures like skip lists [22].

At the heart of our approach lies an algorithm to maintain a constant approximation of the MRU tree at any time. First we define MRU(β) trees for any constant β.

Definition 8 (MRU(β) Tree): A tree T is called an *MRU*(β) tree if it holds for any item u and any time that, $u.\text{dep} = \lfloor \log u.\text{rank} \rfloor + \beta$.

Note that, any MRU(0) tree is also an MRU tree. In particular, we prove in the following that a constant additive approximation is sufficient to obtain dynamic optimality.

Theorem 7: Any online MRU(β) algorithm is $4(1 + \lceil \frac{\beta}{2} \rceil)$ access-competitive.

Proof: According to Theorem 5, MRU(0) trees are 4 access-competitive. Here we only need to prove it for $\beta > 0$. Let us consider an algorithm ON(β) that maintains an MRU(β) tree for some β. For each request v that ON(β) needs to serve, if $v.\text{dep} = k$ is an MRU tree, then ON(β) needs to pay, in the worst case, $k + \beta$ (while ON(0) will pay k). According to ON(β), the item of rank 1 is always at depth 0 and the item of rank 2 is always at level 1. For every level $k \geq 2$, we have, $k + \beta \leq (1 + \lceil \frac{\beta}{2} \rceil)k$. For the special case of $k = 1$, the item with rank 3 can also be at most at depth 2, so the formula holds. Overall, using Theorem 5 we have:

$$\text{cost}(\text{ON}(\beta)) \leq (1 + \lceil \frac{\beta}{2} \rceil)\text{cost}(\text{ON}(0)) \leq 4(1 + \lceil \frac{\beta}{2} \rceil)\text{cost}(\text{OPT})$$

Hence ON is $4(1 + \lceil \frac{\beta}{2} \rceil)$ access-competitive. □

To efficiently achieve an MRU(β) tree, we propose the RANDOM-PUSH strategy (see Algorithm 2). This is a simple randomized strategy which selects a random path starting at the root, and then steps down the tree to depth $k = u.\text{dep}$ (the accessed item depth), by choosing uniformly at random between the two children of each server at each step. This can be seen as a simple k-step random walk in a directed version of the tree, starting from the root of the tree. Clearly, the adjustment cost of RANDOM-PUSH is also proportional to k and its actions are independent of any oblivious online adversary. The main technical challenge of this section is proving the following theorem.

Theorem 8: RANDOM-PUSH maintains an MRU(4) (Definition 8) tree in expectation, i.e., the expected depth of the item with rank r is less than $\log r + 3 < \lfloor \log r \rfloor + 4$ for any sequence σ and any time t.

To analyze RANDOM-PUSH and eventually prove Theorem 8, we will define several random variables for an arbitrary σ and time t (so we ignore them in the notation). W.l.o.g., let v be the item with rank i at time t and let $D(i)$ denote the depth of v at time t. First we note that by induction, it can be shown that the support of $D(i)$ is the set of integers $\{0, 1, \ldots, i-1\}$.

To understand and upper bound $D(i)$, we will use a Markov chain $\mathcal{M}_i$ over the integers $0, 1, 2 \ldots, i-1$, which denote the possible depths in the RANDOM-PUSH tree, see Figure 2. For each depth in the chain $0 \leq j < i-1$, the probability to move to depth $j+1$ is 2^{-j}, and the probability to stay at j, is $1-2^{-j}$, for $j = i-1$; it is an absorbing state. This chain captures the idea that the probability of an item at level j to be pushed down the tree by a random walk (to level larger than j) is 2^{-j}. The chain *does not* describe exactly our RANDOM-PUSH algorithm and $D(i)$, but we will use it to prove an upper bound on $D(i)$. First, we consider a random walk described exactly by the Markov chain $\mathcal{M}_i$ with an initial state 0. Let $\ell(i, w)$

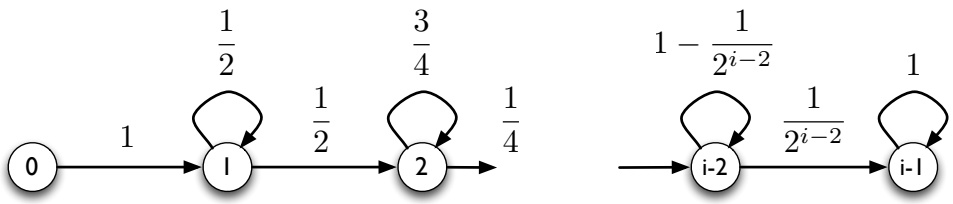

Fig. 2. The Markov chain $\mathcal{M}_i$ that is used to prove Theorem 8 and Lemma 2: possible depths for item of rank i in the complete tree.

denote the random variable of the last state of a random walk of length w on $\mathcal{M}_i$. Then we can show:

Lemma 2: The expected state of $\ell(i,w)$ is such that $\mathbb{E}[\ell(i,w)] < \lceil \log w \rceil + 1$, and $\mathbb{E}[\ell(i,w)]$ is concave in w.

To prove Lemma 2 we use the following corollary.

Corollary 1:

$$\sum_{i=0}^{w} \binom{w}{i} (\frac{w-1}{w})^{w-i} \cdot (\frac{1}{w})^{i} \cdot i = 1$$

Proof:

$$\begin{aligned}
\sum_{i=0}^{w} \binom{w}{i} (\frac{w-1}{w})^{w-i} (\frac{1}{w})^{i} i &= \sum_{i=1}^{w} \binom{w}{i} (\frac{w-1}{w})^{w-i} (\frac{1}{w})^{i} i \\
&= \sum_{i=1}^{w} \frac{w!}{i!\ (w-i)!} (\frac{w-1}{w})^{w-i} (\frac{1}{w})^{i} i \\
&= \sum_{i=1}^{w} \frac{(w-1)!}{(i-1)!\ (w-i)!} (\frac{w-1}{w})^{w-i} (\frac{1}{w})^{i-1} \\
&= \sum_{\eta=0}^{w-1} \frac{(w-1)!}{\eta!\ (w-1-\eta)!} (\frac{w-1}{w})^{w-1-\eta} (\frac{1}{w})^{\eta} \\
&\quad \text{(substituting } i \text{ by } \eta+1) \\
&= \left(\frac{1}{w} + \frac{w-1}{w}\right)^{w-1} = 1
\end{aligned}$$

□

Proof:[Proof of Lemma 2] First note that $\mathbb{E}[\ell(i,w)]$ is strictly monotonic in w and can be shown to be concave using the decreasing *rate* of increase: for $w_1 < w_2$, $\mathbb{E}[\ell(i,w_1)] - \mathbb{E}[\ell(i,w_1-\Delta)] \geq \mathbb{E}[\ell(i,w_2)] - \mathbb{E}[\ell(i,w_2-\Delta)]$. To bound $\mathbb{E}[\ell(i,w)]$, we consider the state of another random walk, $\ell'(i,w)$, that starts on state $k = \lceil \log w \rceil$ in a modified chain $\mathcal{M}_i'$. The modified chain $\mathcal{M}_i'$ is identical to $\mathcal{M}_i$ up to state k, but for all states $j > k$, the probability to move to state $j+1$ is $\frac{1}{w} > 2^{-k}$ and the probability to stay at j is $\frac{w-1}{w} < 1-2^{-k}$. So clearly the walk on $\mathcal{M}_i'$ makes faster progress than the walk on $\mathcal{M}_i$ from state k onward. The expected progress of the walk on $\mathcal{M}_i'$ which starts from state k, is now easier to bound and can shown to be: $\mathbb{E}[\ell'(i,w)] \leq \sum_{j=0}^{w} j\binom{w}{j}(\frac{w-1}{w})^{w-j}(\frac{1}{w})^{j} = 1$. But since $\ell(i,w)$ starts at state 0, we have $\mathbb{E}[\ell(i,w)] < \mathbb{E}[\ell'(i,w)] \leq k+1 = \lceil \log w \rceil + 1$. □

Next, in Lemma 4, we bound the expected number of times that v could potentially be pushed down by a random push, i.e., the number of requests to items at a lower depth than v. Later we will use this as the length w of the random walk on $\mathcal{M}_i$. But, to do so we first state the following lemma.

Lemma 3: For every σ, t and $i > j$, we have that in RANDOM-PUSH, $\mathbb{E}[D(i)] > \mathbb{E}[D(j)]$.

Proof: Let u be an item with rank $j < i$. Hence, it was requested more recently than v (which has rank i). The inequality follows from the fact that conditioning that v and u first reached the *same* depth (after the last request of u) then by symmetry their expected progress of depth will be the same from that point. More formally, let D_{uv} be a random variable that denotes the depth when u's depth equals the depth of v for the *first* time (since the last request of u where its depth is set to 0); and -1 if this never happens. Then by the law of total probability,

$$\begin{aligned}
\mathbb{E}[D(i)] &= \mathbb{E}_{D_{uv}}[\mathbb{E}[D(i) \mid D_{uv}]] \\
&= \sum_{k=-1}^{i-1} \mathbb{P}(D_{uv} = k) \sum_{\ell=0}^{i-1} \ell \cdot \mathbb{P}(D(i) = \ell \mid D_{uv} = k)
\end{aligned}$$

(and similar for $D(j)$). But since the random walk (i.e., push) is independent of the servers' ranks, we have for $k \geq 0$ that $\mathbb{E}[D(i) \mid D_{uv} = k] \geq \mathbb{E}[D(j) \mid D_{uv} = k]$. But additionally there is the possibility that they will never be at the same depth (after the last request of u) and that v will always have a higher depth, so $\mathbb{E}[D(i) \mid D_{uv} = -1] > \mathbb{E}[D(j) \mid D_{uv} = -1]$. The claim follows. □

Now, let W_i be a random variable that denotes the number of requests for items with higher depth than v, since v's last request until time t. The following lemma bounds the number of such requests.

Lemma 4: The expected number of requests for items with higher depth than v, since v was last requested, is bounded by $\mathbb{E}[W_i] \leq 2i - 1$.

Proof: We can divide W_i into two types of requests: $W_i^{>}$ requests for items with higher rank and depth than v at the time of their request, and $W_i^{<}$ requests for items with lower rank but higher depth than v at the time of their request. Then $W_i = W_i^{>} + W_i^{<}$. Clearly $W_i^{>} \leq i$ since every such request increases the rank of v and this happens i times (note that some of these requests may have lower depth than v). $W_i^{<}$ is harder to analyze. How many requests for items are there that have lower rank than v at the time of the request, but are below v in the tree (i.e., have higher depth than v)? Note that such requests do not increase v's rank, but *may* increase its depth. Let u be an item with rank $j < i$, hence u was more recently requested than v (maybe several times). Let X_j denote the number of requests for u (since v was last requested) in which it had a higher depth than v. Then $W_i^{<} = \sum_{j=1}^{i-1} X_j$. We now claim that $\mathbb{E}[X_j] \leq 1$. Assume by contradiction that $\mathbb{E}[X_j] > 1$. But then this implies that we can construct a sequence σ' for

which the expected depth of u will be larger than the expected depth of v, contradicting Lemma 3. Putting it all together:

$$\mathbb{E}[W_i] = \mathbb{E}[W_i^{>} + W_i^{<}] \leq \mathbb{E}[i] + \mathbb{E}[\sum_{n=1}^{i-1} X_n] \leq i + \sum_{n=1}^{i-1} \mathbb{E}[X_n] \leq 2i - 1$$

□

We recall some of the known results related to Stochastic Domination [48].

Definition 9 (Stochastic Domination): Let $\mathtt{X}$ and $\mathtt{Y}$ be two random variables, not necessarily on the same probability space. The random variable $\mathtt{X}$ is *stochastically smaller than* $\mathtt{Y}$, denoted by $\mathtt{X} \preceq \mathtt{Y}$, if $\mathbb{P}[\mathtt{X} > z] \leq \mathbb{P}[\mathtt{Y} > z]$ for every $z \in \mathbb{R}$. If additionally $\mathbb{P}[\mathtt{X} > z] < \mathbb{P}[\mathtt{Y} > z]$ for some z, then $\mathtt{X}$ is *stochastically strictly less than* $\mathtt{Y}$, denoted by $\mathtt{X} \prec \mathtt{Y}$.

Theorem 9 (Stochastic Order): Let $\mathtt{X}$ and $\mathtt{Y}$ be two random variables, not necessarily on the same probability space.

1) Suppose $\mathtt{X} \prec \mathtt{Y}$. Then $\mathbb{E}[U(\mathtt{X})] < \mathbb{E}[U(\mathtt{Y})]$ for any strictly increasing function U.
2) Suppose $\mathtt{X}_1 \prec \mathtt{Y}_1$ and $\mathtt{X}_2 \prec \mathtt{Y}_2$, for four random variables $\mathtt{X}_1, \mathtt{Y}_1, \mathtt{X}_2$ and $\mathtt{Y}_2$. Then $a\mathtt{X}_1 + b\mathtt{Y}_1 \prec a\mathtt{X}_2 + b\mathtt{Y}_2$ for any two constants $a, b > 0$.
3) Suppose U is a non-decreasing function and $\mathtt{X} \prec \mathtt{Y}$ then $U(\mathtt{X}) \prec U(\mathtt{Y})$.
4) Given that $\mathtt{X}$ and $\mathtt{Y}$ follow the binomial distribution, i.e., $\mathtt{X} \sim Bn(n_1, p_1)$ and $\mathtt{Y} \sim Bn(n_2, p_2)$, then $\mathtt{X} \preceq \mathtt{Y}$ if and only if the following two conditions holds: $(1-p_1)^{n_1} \geq (1-p_2)^{n_2}$ and $n_1 \leq n_2$.

We now have all we need to prove Theorem 8. The proof follows by showing that $\mathbb{E}[D(i)] \leq \log i + 3$.

Proof:[Proof of Theorem 8] Let $D(i, w)$ be a random variable that denotes the depth of v conditioning that there are w requests of items with higher depth than v, since the last request for v. Note that by the total probability law, we have that

$$\mathbb{E}[D(i)] = \mathbb{E}_{W_i}[\mathbb{E}[D(i, W_i)] = \sum_{w=1}^{|\sigma|} \mathbb{P}(W_i = w)\mathbb{E}[D(i, W_i = w)]$$

Next we claim that $D(i, w)$ is stochastically less [48] than $\ell(i, w)$, denoted by $D(i, w) \preceq \ell(i, w)$.

This is true since the transition probabilities (to increase the depth) in the Markov chain $\mathcal{M}_i$ are at least as high as in the Markov chain that describes $D(i, w)$. The probability that a random walk to depth higher than v's depth visits v (and pushes it down) is exactly 2^{-j} where j is the depth of v. Since $D(i, w) \preceq \ell(i, w)$, it will then follow from Theorem 9 that $\mathbb{E}[D(i, w)] \leq \mathbb{E}[\ell(i, w)]$. Clearly we also have $\mathbb{E}_{W_i}\mathbb{E}[D(i, W_i)] \leq \mathbb{E}_{W_i}\mathbb{E}[\ell(i, W_i)]$. Let $f_i(W_i) = \mathbb{E}[\ell(i, W_i)]$ be a random variable which is a function of the random variable W_i. Recall that $f_i()$ is concave, then by Jensen's inequality [21] and Lemma 4 we get:

$$\begin{aligned}\mathbb{E}[D(i)] &= \mathbb{E}_{W_i}[\mathbb{E}[D(i, W_i)]] \leq \mathbb{E}_{W_i}[\mathbb{E}[\ell(i, W_i)]] \\ &= \mathbb{E}_{W_i}[f_i(W_i)] \leq f_i(\mathbb{E}[W_i]) \leq f_i(2i-1) \\ &= \mathbb{E}[\ell(i, 2i-1)] \leq \lceil \log 2i \rceil + 1 \leq \log i + 3\end{aligned}$$

□

Algorithm 2: RANDOM-PUSH (Upon access to u in tree)

1: **access** $s = u$.host along tree branches (cost: u.dep)
2: let $v = s_1$.guest be the item at the current root
3: **move** u to the root server s_1, setting s_1.guest $= u$ (cost: u.dep)
4: employ RANDOM-PUSH to **shift** down v to depth s.dep (cost: u.dep)
5: let w be the item at the end of the push-down path, where w.dep $= s$.dep
6: **move** w to s, i.e., setting s.guest $= w$ (cost: u.dep $\times$ 2)

It now follows almost directly from Theorems 7 and 8 that RANDOM-PUSH is dynamically optimal.

Theorem 3: The RANDOM-PUSH algorithm is dynamically optimal on expectation.

Proof: Let the t-th requested item have rank r_t, then the access cost is $D(r_t)$. According to the RANDOM-PUSH (Algorithm 2), the total cost is $5D(r_t)$ which is five times the access cost on the MRU(4) tree. Formally, using Theorem 7 and Theorem 8, the expected total cost is:

$$\begin{aligned}&\mathbb{E}\left[\text{cost}(\text{RANDOM-PUSH})\right] = \mathbb{E}\left[\sum_{i=1}^{t} 5D(r_i)\right] \\ &= 5\sum_{i=1}^{t}\mathbb{E}\left[D(r_i)\right] \leq 5\sum_{i=1}^{t}(\log(r_i)+3) \\ &\leq 5\sum_{i=1}^{t}(\lfloor\log(r_i)\rfloor + 4) \leq 5\sum_{i=1}^{t}\text{cost}^{(t)}(\text{MRU}(4)) \\ &\leq 5\cdot\text{cost}(\text{MRU}(4)) = 60\cdot\text{cost}(\text{OPT})\end{aligned}$$

□

VII. WORKING EXAMPLES OF OUR ALGORITHMS

In this section, we provide an example how our algorithms work on actual requests. Recall the definition of the working set and the rank of an item. We consider a complete binary tree of 15 servers hosting 15 items such that initially server i hosts item i. In our initial configuration item 1 is at the root of the tree and for any internal item (node) i, its left child is item $2i$ and right child is item $2i+1$. This initial configuration is obtained by serving a request sequence of items $(1, 2, \cdots, 15)$. Accordingly the working set for item j is $\{1, 2, \cdots, j-1, j\}$ and hence the rank of item j is j. If item i has rank j, we denote it by (i, j). See Figure 3 (a)) for the initial configuration of the tree. First we show an example for MOVE-HALF (Algorithm 1).

Working of MOVE-HALF (Algorithm 1): Consider the initial configuration mentioned above and assume the next three requests are for items $10, 15, 15$. Consider the request 10. Note that the ranks of items 1 to 9 increase by 1 each, as 10 is added to the working set of each of them. As the working set of 10 becomes $\{10\}$, its rank becomes 1. Observe that, for items 11 to

15, the rank remains the same as 10 was already present in the working sets according to our initial configuration. Before the request, item 10 was at depth 3, so according to the algorithm, it needs to swap place with the highest-ranked item at depth $\lfloor \frac{3}{2} \rfloor$ which is item 3. Figure 3 (b) shows the new configuration after the swap, and the path between items 10 and 3. Next item 15 is requested. The ranks of items 1 to 14 increase by 1 each, as 15 is added to the working set of each of them. The working set of 15 becomes $\{15\}$, so its rank becomes 1. Item 15 was at depth 3 before the request and hence it changes position with item 2 which was the highest-ranked item at depth 1 as shown in Figure 4 (a). Finally, item 15 is requested again. Since no new item is added to any of the items' working sets, the rank remains the same for all items. However, item 15 changes position with item 1 by moving up from depth 1 to depth 0 (see Figure 4 (b)).

Working of RANDOM-PUSH(Algorithm 2): We consider the same initial configuration (see Figure 5 (a)). Let us assume that item 10 which is at depth 3, is the next request we need to serve. As before, the rank of 10 becomes 1 and the ranks of all items from 1 to 9 increase by 1. The other ranks remain the same. Algorithm RANDOM-PUSHpushes the item that is at the root (item 1 in this case) along a random path from the root to depth 3. Figure 5 presents on example where the random path ended at item 9 so item 1 moves down to depth 1, item 2 moves down to depth 2, item 4 moves down to depth 3 and item 4 occupies the server where item 9 was hosted. Now item 9 moves to the server which hosted item 10 at the same depth 3 (since the host server of item 10 is vacated) and item 10 is moved to the root of the tree where item 1 was hosted (shown using red curved lines in Figure 5). Note that, the items position changes do not happen along the curved paths in the figure; these changes happen along the tree edges only using swaps.

VIII. RELATED WORK

Given the explosive growth of communication traffic, great efforts have been made over the last years to improve the efficiency and performance of networks, especially in the context of datacenters. While traditionally, datacenter network topologies are static and oblivious to the traffic they serve [1], [31], [38], [39], [41], [49], [50], [57], emerging reconfigurable optical technologies enable more dynamic topologies [27], [32]. Dynamic but demand-oblivious topologies such as RotorNet, Opera and Sirius [13], [43], [44] provide periodic direct connectivity which saves bandwidth and can significantly improve throughput; dynamic and demand-aware topologies such as ProjecToR [29], [40], ReNets [12], SplayNets [46], [47] as well as many others [8], [20], [25], [26], [29], [33], [34], [36], [51], [54], [59] allow to optimize the topology even further, e.g., to optimally serve elephant flows. It has recently been demonstrated by the Cerberus architecture [30] that the best topology is often a combination of these technologies, which depends on the traffic pattern.

Our focus in this paper is on such dynamic demand-aware topologies. In particular, we follow the approach by Avin et al. who showed that efficient demand-aware networks can be built from ego-trees [7], [9], [12], [45]. However, so far it is only known how to build such demand-aware networks that provide static optimality guarantees [12]. With this paper, we presented a building block for competitive demand-aware networks, a constant-competitive ego-tree, which can be used to enhance existing systems such as ReNets [12].

The self-adjusting tree networks considered in this paper feature an interesting connection to self-adjusting datastructures. Interestingly, while we have shown in this paper that dynamically optimal algorithms for tree networks exist, the quest for constant competitive online algorithms for binary search trees remains a major open problem [53]. Nevertheless, there are self-adjusting binary search trees that are known to be *access optimal* [14], but their rearrangement cost it too high.

In the following, we review this related work on datastructures in more details.

Dynamic List Update: Linked List (LL). The dynamically optimal linked list datastructure is a seminal [52] result in the area: algorithms such as Move-To-Front (MTF), which moves each accessed item to the front of the list, are known to be 2-competitive, which is optimal [2], [5], [52]. We note that the Move-To-Front algorithm results in the Most Recently Used property where items that were more recently used are closer to the head of the list. The best known competitive ratio for randomized algorithms for LLs is 1.6, which almost matches the randomized lower bound of 1.5 [4], [55].

Binary Search Tree (BST). In contrast to CTs, self-adjustments in BSTs are based on *rotations* (which are assumed to have unit cost). While BSTs have the working set property, we are missing a matching lower bound: the *Dynamic Optimality Conjecture*, the question whether splay trees [53] are dynamically optimal, continues to puzzle researchers even in the randomized case [3]. On the positive side, over the last years, many deep insights into the properties of self-adjusting BSTs have been obtained [19], including improved (but non-constant) competitive ratios [16], regarding weaker properties such as working sets, static, dynamic, lazy, and weighted, fingers, regarding pattern-avoidance [18], and so on. It is also known (under the name *dynamic search-optimality*) that if the online algorithm is allowed to make rotations for free after each request, dynamic optimality can be achieved [14]. Known lower bounds are by Wilber [56], by Demaine et al. [23]'s interleaves bound (a variation), and by Derryberry et al. [24] (based on graphical interpretations). It is not known today whether any of these lower bounds is tight.

Unordered Tree (UT). We are not the first to consider *unordered* trees and it is known that existing lower bounds for (offline) algorithms on BSTs also apply to UTs that use rotations: Wilber's theorem can be generalized [28]. However, it is also known that this correspondance between ordered and unordered trees no longer holds under weaker measures such as *key independent processing costs* and in particular *Iacono's measure* [35]: the expected cost of the sequence which results from a random assignment of keys from the search tree to the items specified in an access request sequence. Iacono's work is also one example of prior work which shows that for specific scenarios, working set and dynamic optimality properties are equivalent. Regarding the current work, we note that the

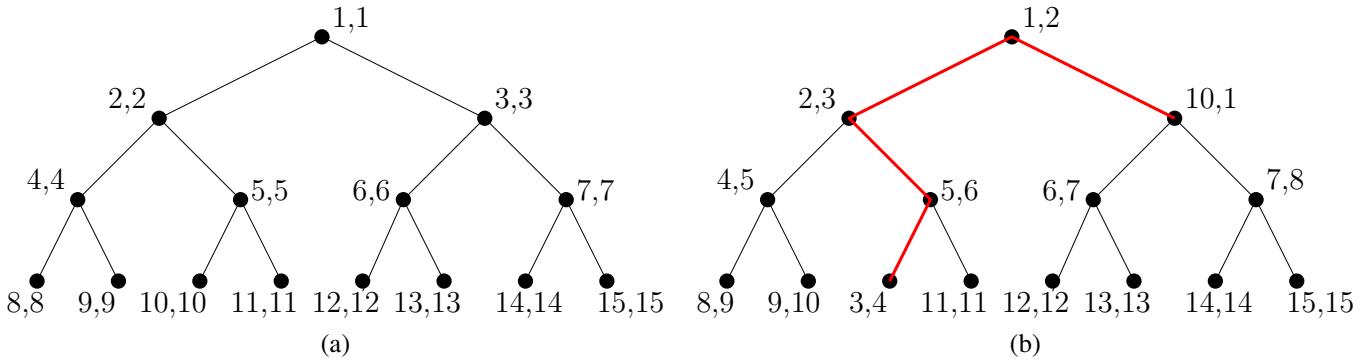

Fig. 3. (a) Initially each server i hosts item i with rank i, denoted by (i, i). (b) Item 10 is requested in the sequence. The red path shows the exchange of the position of item 10 with the highest-ranked item at depth 1, i.e., item 3.

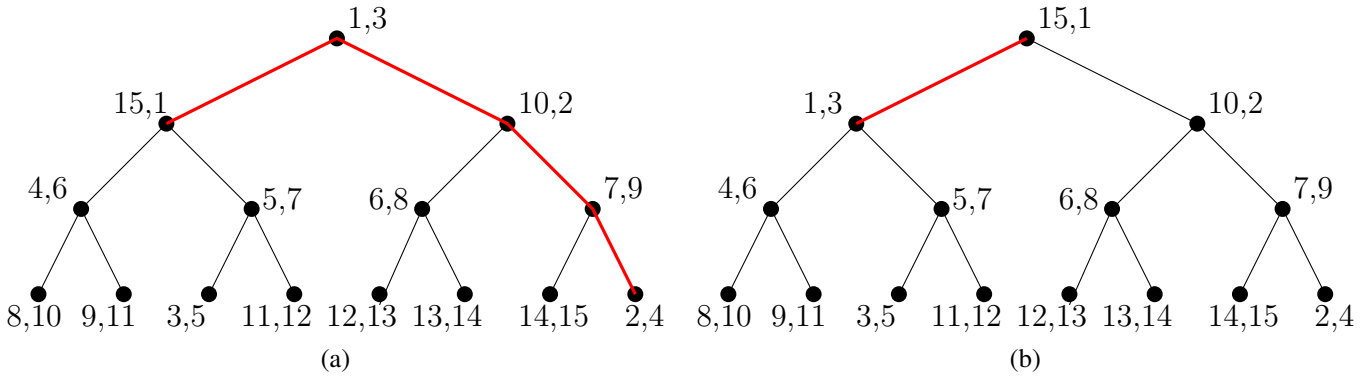

Fig. 4. (a) Item 15 is requested in the online sequence. The red path shows the exchange of position of item 15 with the highest ranked item at depth 1, i.e., item 2. (b) Again 15 comes up in the online sequence. The red path shows the exchange of position of item 15 with the highest-ranked item at depth 0, i.e., item 1.

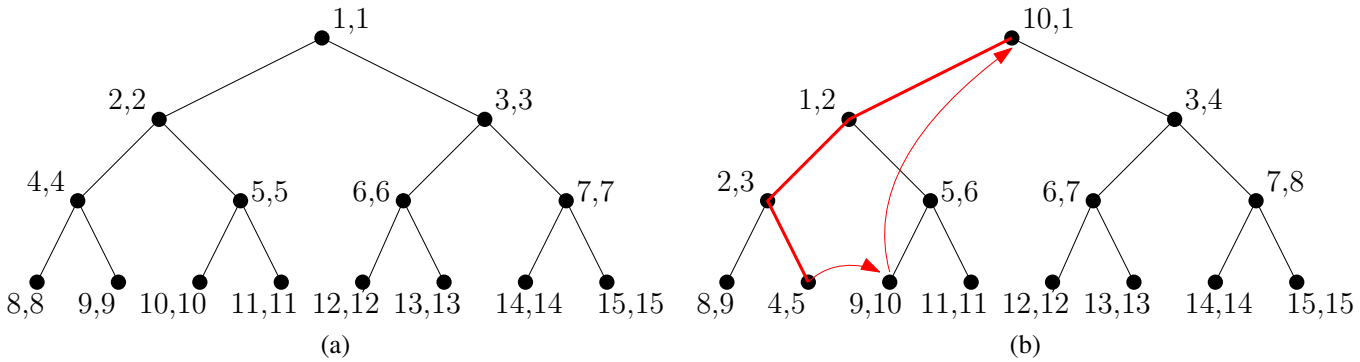

Fig. 5. (a) Initially each server i hosts item i with rank i and it is denoted by (i, i). (b) Item 10 is requested in the online sequence. Red path along the tree edges shows the random push and curve paths indicate movement of item 9 to the position of item 10 and of item 10 to the root.

reconfiguration operations in UTs are more powerful than the swapping operations considered in our paper: a rotation allows to move entire subtrees at unit costs, while the corresponding cost in CTs is linear in the subtree size. We also note that in our model, we cannot move freely between levels, but moves can only occur between parent and child. In contrast to UTs, CTs are bound to be balanced.

Skip List (SL) and B-Trees (BT). Intriguingly, although SLs and BSTs can be transformed to each other [22], Bose et al. [17] were able to prove dynamic optimality for (a restricted kind of) SLs as well as BTs. Similarly to our paper, the authors rely on a connection between dynamic optimality and working set: they show that the working set property is sufficient for their restricted SLs (for BSTs, it is known that the working set is an upper bound, but it is not known yet whether it is also a lower bound). However, the quest for proving dynamic optimality for general skip lists remains an open problem: two restricted types of models were considered in [17], bounded and weakly bounded. In the bounded model, the adversary can never forward more than B times on a given skip list level, without going down in the search; and in the weakly bounded model, the first i highest levels contain no more than $\sum_{j=0}^{i} B^j$ items. Optimality only holds for constant B. The weakly bounded model is related to a complete B-ary tree (similar to our complete binary tree), but there is no obvious or direct connection between our result and the weakly bounded

optimality. Due to the relationship between SLs and BSTs, a dynamically optimal SL would imply a working set lower bound for BST. Moreover, while both in their model and ours, proving the working set property is key, the problems turn out to be fundamentally different. In contrast to SLs, CTs revolve around *unordered* (and balanced) trees (that do not provide a simple search mechanism), rely on a different reconfiguration operation (i.e., swapping or *pushing* an item to its parent comes at unit cost), and, as we show in this paper, actually provide dynamic optimality for their general form. Finally, we note that [17] (somewhat implicitly) also showed that a random walk approach can achieve the working set property; in our paper, we show that the working set property can even be achieved deterministically and without maintaining MRU.

Heaps and Paging. More generally, our work is also reminiscent of *online paging* models for *hierarchies* of caches [58], which aim to keep high-capacity nodes resp. frequently accessed items close to each other, however, without accounting for the reconfiguration cost over time. Similar to the discussion above, self-adjusting CTs differ from paging models in that in our model, items cannot move arbitrarily and freely between levels (but only between parent and child at unit cost).

IX. Conclusion

Our paper opens interesting avenues for future research. On the theoretical front, it remains to determine the (non-asymptotical) optimal competitive ratio, either by closing the gap with a matching lower bound, or by improving our upper bound. On the practical front, it would be interesting to evaluate the performance of our algorithms when used to support (as ego-trees) existing networks such as ReNets. In this context and also more generally, it would further be interesting to study the empirical competitive ratio achieved by our algorithms for existing benchmarks and under realistic workloads. The latter will also require a methodological contribution, as the offline problem is not well-understood and computing the optimal offline solution seems computationally complex.

Acknowledgments. Research supported by the European Research Council (ERC), grant agreement No. 864228 (AdjustNet), Horizon 2020, 2020-2025.

References

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.

[2] Susanne Albers. A competitive analysis of the list update problem with lookahead. *Mathematical Foundations of Computer Science 1994*, pages 199–210, 1994.

[3] Susanne Albers and Marek Karpinski. Randomized splay trees: theoretical and experimental results. *Information Processing Letters*, 81(4):213–221, 2002.

[4] Susanne Albers, Bernhard Von Stengel, and Ralph Werchner. A combined bit and timestamp algorithm for the list update problem. *Information Processing Letters*, 56(3):135–139, 1995.

[5] Susanne Albers and Jeffery Westbrook. Self-organizing data structures. In *Online algorithms*, pages 13–51. Springer, 1998.

[6] Chen Avin, Manya Ghobadi, Chen Griner, and Stefan Schmid. On the complexity of traffic traces and implications. In *Proc. ACM SIGMETRICS*, 2020.

[7] Chen Avin, Kaushik Mondal, and Stefan Schmid. Demand-aware network designs of bounded degree. *Distributed Computing*, 2017.

[8] Chen Avin, Kaushik Mondal, and Stefan Schmid. Demand-aware network designs of bounded degree. In *Proc. International Symposium on Distributed Computing (DISC)*, 2017.

[9] Chen Avin, Kaushik Mondal, and Stefan Schmid. Demand-aware network design with minimal congestion and route lengths. In *Proc. IEEE INFOCOM*, pages 1351–1359, 2019.

[10] Chen Avin and Stefan Schmid. Toward demand-aware networking: A theory for self-adjusting networks. In *ACM SIGCOMM Computer Communication Review (CCR)*, 2018.

[11] Chen Avin and Stefan Schmid. Toward demand-aware networking: A theory for self-adjusting networks. *ACM SIGCOMM Computer Communication Review*, 48(5):31–40, 2019.

[12] Chen Avin and Stefan Schmid. Renets: Statically-optimal demand-aware networks. In *Proc. SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, 2021.

[13] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, et al. Sirius: A flat datacenter network with nanosecond optical switching. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 782–797, 2020.

[14] Avrim Blum, Shuchi Chawla, and Adam Kalai. Static optimality and dynamic search-optimality in lists and trees. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2002.

[15] Shaileshh Bojja Venkatakrishnan, Mohammad Alizadeh, and Pramod Viswanath. Costly circuits, submodular schedules and approximate carathéodory theorems. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, pages 75–88, 2016.

[16] Prosenjit Bose, Karim Douïeb, Vida Dujmović, and Rolf Fagerberg. An o (log log n)-competitive binary search tree with optimal worst-case access times. In *Scandinavian Workshop on Algorithm Theory*, pages 38–49. Springer, 2010.

[17] Prosenjit Bose, Karim Douïeb, and Stefan Langerman. Dynamic optimality for skip lists and b-trees. In *Proc. 19th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1106–1114, 2008.

[18] Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. Pattern-avoiding access in binary search trees. In *Proc. Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 410–423. IEEE, 2015.

[19] Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. The landscape of bounds for binary search trees. *arXiv preprint arXiv:1603.04892*, 2016.

[20] Kai Chen, Ankit Singla, Atul Singh, Kishore Ramachandran, Lei Xu, Yueping Zhang, Xitao Wen, and Yan Chen. Osa: An optical switching architecture for data center networks with unprecedented flexibility. *IEEE/ACM Transactions on Networking (TON)*, 22(2):498–511, 2014.

[21] T.M. Cover and J. Thomas. *Elements of information theory*. Wiley, 2006.

[22] Brian C. Dean and Zachary H. Jones. Exploring the duality between skip lists and binary search trees. In *Proceedings of the 45th Annual Southeast Regional Conference*, ACM-SE 45, pages 395–399, New York, NY, USA, 2007. ACM.

[23] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Patrascu. Dynamic optimality - almost. *SIAM J. Comput.*, 37(1):240–251, 2007.

[24] Jonathan Derryberry, Daniel Dominic Sleator, and Chengwen Chris Wang. *A lower bound framework for binary search trees with rotations*. School of Computer Science, Carnegie Mellon University, 2005.

[25] Fred Douglis, Seth Robertson, Eric Van den Berg, Josephine Micallef, Marc Pucci, Alex Aiken, Maarten Hattink, Mingoo Seok, and Keren Bergman. Fleet—fast lanes for expedited execution at 10 terabits: Program overview. *IEEE Internet Computing*, 2021.

[26] Klaus-Tycho Foerster, Monia Ghobadi, and Stefan Schmid. Characterizing the algorithmic complexity of reconfigurable data center architectures. In *Proc. ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2018.

[27] Klaus-Tycho Foerster and Stefan Schmid. Survey of reconfigurable data center networks: Enablers, algorithms, complexity. In *SIGACT News*, 2019.

[28] Michael L Fredman. Generalizing a theorem of wilber on rotations in binary search trees to encompass unordered binary trees. *Algorithmica*, 62(3-4):863–878, 2012.

[29] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. Projector: Agile reconfigurable data center interconnect. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 216–229. ACM, 2016.

[30] Chen Griner, Johannes Zerwas, Andreas Blenk, Manya Ghobadi, Stefan Schmid, and Chen Avin. Cerberus: The power of choices in datacenter topology design (a throughput perspective). In *Proc. ACM SIGMETRICS*, 2022.

[31] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: a high performance, server-centric network architecture for modular data centers. *ACM SIGCOMM Computer Communication Review*, 39(4):63–74, 2009.

[32] Matthew Nance Hall, Klaus-Tycho Foerster, Stefan Schmid, and Ramakrishnan Durairajan. A survey of reconfigurable optical networks. In *Optical Switching and Networking (OSN), Elsevier*, 2021.

[33] Navid Hamedazimi, Zafar Qazi, Himanshu Gupta, Vyas Sekar, Samir R Das, Jon P Longtin, Himanshu Shah, and Ashish Tanwer. Firefly: A reconfigurable wireless data center fabric using free-space optics. In *Proc. ACM SIGCOMM Computer Communication Review (CCR)*, volume 44, pages 319–330, 2014.

[34] Michelle Hampson. Reconfigurable optical networks will move supercomputerdata 100x faster. In *IEEE Spectrum*, 2021.

[35] John Iacono. Key-independent optimality. *Algorithmica*, 42(1):3–10, 2005.

[36] Srikanth Kandula, Jitendra Padhye, and Paramvir Bahl. Flyways to de-congest data center networks. In *Proc. ACM Workshop on Hot Topics in Networks (HotNets)*, 2009.

[37] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In *Proc. 9th ACM Internet Measurement Conference (IMC)*, pages 202–208, 2009.

[38] Simon Kassing, Asaf Valadarsky, Gal Shahaf, Michael Schapira, and Ankit Singla. Beyond fat-trees without antennae, mirrors, and disco-balls. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 281–294. ACM, 2017.

[39] John Kim, Wiliam J Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. In *2008 International Symposium on Computer Architecture*, pages 77–88. IEEE, 2008.

[40] Janardhan Kulkarni, Stefan Schmid, and Pawel Schmidt. Scheduling opportunistic links in two-tiered reconfigurable datacenters. In *33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2021.

[41] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 399–412, 2013.

[42] M. Ghobadi et al. Projector: Agile reconfigurable data center interconnect. In *Proc. ACM SIGCOMM*, pages 216–229, 2016.

[43] William M Mellette, Rajdeep Das, Yibo Guo, Rob McGuinness, Alex C Snoeren, and George Porter. Expanding across time to deliver bandwidth efficiency and low latency. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 1–18, 2020.

[44] William M Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C Snoeren, and George Porter. Rotornet: A scalable, low-complexity, optical datacenter network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 267–280. ACM, 2017.

[45] Maciej Pacut, Wenkai Dai, Alexandre Labbe, Klaus-Tycho Foerster, and Stefan Schmid. Improved scalability of demand-aware datacenter topologies with minimal route lengths and congestion. In *39th International Symposium on Computer Performance, Modeling, Measurements and Evaluation (PERFORMANCE)*, 2021.

[46] Bruna Peres, A de O Otavio, Olga Goussevskaia, Chen Avin, and Stefan Schmid. Distributed self-adjusting tree networks. In *Proc. IEEE INFOCOM*, pages 145–153, 2019.

[47] Stefan Schmid, Chen Avin, Christian Scheideler, Michael Borokhovich, Bernhard Haeupler, and Zvi Lotker. Splaynet: Towards locally self-adjusting networks. *IEEE/ACM Trans. Netw.*, 24(3):1421–1433, June 2016.

[48] Moshe Shaked and J George Shanthikumar. *Stochastic orders*. Springer Science & Business Media, 2007.

[49] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review*, 45(4):183–197, 2015.

[50] Ankit Singla, Chi-Yao Hong, Lucian Popa, and Philip Brighten Godfrey. Jellyfish: Networking data centers, randomly. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, volume 12, pages 17–17, 2012.

[51] Ankit Singla, Atul Singh, Kishore Ramachandran, Lei Xu, and Yueping Zhang. Proteus: a topology malleable data center network. In *Proc. ACM Workshop on Hot Topics in Networks (HotNets)*, 2010.

[52] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, February 1985.

[53] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

[54] Min Yee Teh, Zhenguo Wu, and Keren Bergman. Flexspander: augmenting expander networks in high-performance systems with optical bandwidth steering. *IEEE/OSA Journal of Optical Communications and Networking*, 12(4):B44–B54, 2020.

[55] Boris Teia. A lower bound for randomized list update algorithms. *Information Processing Letters*, 47(1):5–9, 1993.

[56] Robert Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM Journal on Computing*, 18(1):56–67, 1989.

[57] Haitao Wu, Guohan Lu, Dan Li, Chuanxiong Guo, and Yongguang Zhang. Mdcube: a high performance network structure for modular data center interconnection. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 25–36. ACM, 2009.

[58] Gala Yadgar, Michael Factor, Kai Li, and Assaf Schuster. Management of multilevel, multiclient cache hierarchies with application hints. *ACM Transactions on Computer Systems (TOCS)*, 29(2):5, 2011.

[59] Xia Zhou, Zengbin Zhang, Yibo Zhu, Yubo Li, Saipriya Kumar, Amin Vahdat, Ben Y Zhao, and Haitao Zheng. Mirror mirror on the ceiling: Flexible wireless links for data centers. *Proc. ACM SIGCOMM Computer Communication Review (CCR)*, 42(4):443–454, 2012.

Chen Avin received a B.Sc. degree in Communication Systems Engineering from Ben Gurion University, Israel, in 2000. He received the M.S. and Ph.D. degrees in computer science from the University of California, Los Angeles (UCLA) in 2003 and 2006 respectively. He is an associate professor at the Department of Communication Systems Engineering, School of Electrical and Computer Engineering, at the Ben Gurion University of the Negev, Israel. His current research interests are: data-driven graphs and networks algorithms, modeling and analysis with emphasis on demand-aware networks, distributed systems, social networks and randomized algorithms for networking.

Kaushik Mondal is an Assistant Professor at Indian Institute of Technology Ropar. He received his MSc (2008) from Visva Bharati, a Central University in India and PhD (2015) from Indian Institute of Technology Guwahati. Then from August 2015, he worked as postdoc at various institutes namely IIT Guwahati, Ben Gurion University of Negev, Israel and Indian Statistical Institute, Kolkata till June 2019. Then he worked as an Assistant Professor at Indian Institute of Information Technology, Vadodara from July 2019 to January 2020 before joining IIT Ropar. His research interests include network algorithms, distributed algorithms for swarm robots, graph algorithms etc.

Stefan Schmid is a Professor at the Technical University of Berlin, Germany. MSc and PhD at ETH Zurich, Postdoc at TU Munich and University of Paderborn, Senior Research Scientist at T-Labs in Berlin, Associate Professor at Aalborg University, Denmark, and Full Professor at the University of Vienna, Austria. Stefan Schmid received the IEEE Communications Society ITC Early Career Award 2016 and an ERC Consolidator Grant 2019.