The Online House Numbering Problem: Min-Max Online List Labeling*

William E. Devanny[1], Jeremy T. Fineman[2], Michael T. Goodrich[3], and Tsvi Kopelowitz[4]

1 **University of California, Irvine, CA, USA**
wdevanny@uci.edu
2 **Georgetown University, Washington, D.C., USA**
jfineman@cs.georgetown.edu
3 **University of California, Irvine, CA, USA**
goodrich@uci.edu
4 **University of Waterloo, Ontario, Canada**
kopelot@gmail.com

Abstract

We introduce and study the *online house numbering* problem, where houses are added arbitrarily along a road and must be assigned labels to maintain their ordering along the road. The online house numbering problem is related to classic online list labeling problems, except that the optimization goal here is to minimize the maximum number of times that any house is relabeled. We provide several algorithms that achieve interesting tradeoffs between upper bounds on the number of maximum relabels per element and the number of bits used by labels.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases house numbering, list labeling, file maintenance

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.33

1 Introduction

In this paper we study a new version of the fundamental *online monotonic list labeling problem* [14, 6, 23] (OMLL), where the goal is to maintain labels for a dynamic ordered list of at most n elements that, due to monotonicity requirements of appropriate applications, must have (integer) labels that strictly increase in the direction of the ordering. When a new element is inserted into the list, either between two existing elements or at an endpoint of the list, we must assign a label to the new element that is consistent with the order of the list. To avoid labels becoming too long, algorithms for list-labeling problems relabel elements from time to time thereby maintaining the ordering using relatively few bits for the labels. There are several common variants of OMLL that differ in the number of bits allowed for each label. For example, the special case of $\log n + O(1)$ bits[1] is known as the file-maintenance problem [27, 26, 6, 7], where labels are viewed as corresponding to addresses in a size $O(n)$ array.

* This research was supported in part by NSF grants CCF-1617727, 1228639, CCF-1526631, CCF-1514383 and CCF-16375, and by the Canada Research Chair for Algorithm Design. This article is also supported in part by DARPA under agreement no. AFRL FA8750-15-2-0092. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

[1] Unless another base is indicated, all logarithms in this paper are base 2.

25th Annual European Symposium on Algorithms (ESA 2017).
Editors: Kirk Pruhs and Christian Sohler; Article No. 33; pp. 33:1–33:15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Solutions for OMLL are used as foundational building blocks in several areas of computer science, ranging from cache-oblivious data structures [4, 5, 10] to distributed computing [17], and they play a central role in deamortization [14, 9, 16]. As an illustration of the data structure's central role in the field, it is used as a black box in order-maintenance data structures [14, 23], which themselves are used as black boxes throughout computer science (for examples see [2, 15, 13, 20, 1, 18, 24]).

The focus of previous work solving the OMLL problem in the RAM model has been on minimizing the worst-case or amortized number of relabels per update. For example, when using $O(\log n)$ bits per label the worst-case number of relabels per update is known to be $O(\log n)$ [23], which is tight [11]. A particular element, however, can be relabeled as many as $\Omega(n)$ times during a sequence of n insertions using existing algorithms. This paper considers the goal of minimizing the maximum number of times an element in the list is relabeled, while using only a small number of bits per label. We refer to this version of OMLL as the *online house numbering problem*, since it captures the challenges that take place when maintaining a strictly increasing numbering for a collection of houses representing their order along a road. When a new house is built between any two existing houses (or at either end of the row of houses), this new house needs to be assigned a house number. If no such integer house number is available, however, then other houses need to be renumbered (or relabeled) to make room for a number for the new house. Formally stated, the online house numbering problem is to maintain a labelling of an initially empty ordered list subject to n operations of the form, $\mathsf{insert}(x, a)$: insert x immediately after a in the ordered list. Remarkably, existing solutions for list labeling problems do not seem to lead to efficient solutions for the online house numbering problem.

The online house numbering problem raises some interesting combinatorial questions while also addressing label-update complexity, which is motivated from use of solid-state memories, like flash memory, that have an upper bound on the number of erasures that can occur for any memory cell [8, 25, 28]. For example, consider a database with an ordered set of large records, where each record maintains a label respecting the order. Due to the use of these modern types of memory, the number of times that the label is changed must be minimized, since each relabeling entails rewriting that area in memory. A typical assumption in models for solid-state memories is that the algorithm or data structures also have access to a sublinear amount of additional *scratch space* for computational purposes (see Ben-Aroya and Toledo [3]), which is exempt from the erasure limits. In the context of our online house numbering, this would mean that each element in the data structure has a fixed record containing, e.g., the label and any other auxiliary information that is updated whenever a label changes (for our solution, we also store a counter as part of the record). Any additional components of the data structure must be restricted to the $o(n)$ scratch space.

There are two competing objectives that we consider in designing solutions for the online house numbering problem. The first objective is to minimize, over all elements in the list, the maximum number of times that the label of the element changes throughout the n insertions. Notice that with large labels, a trivial solution in which no relabels are needed is obtainable by assigning x the average of a and b, where b is the element succeeding x. This trivial solution requires $\Omega(n)$ bits per label, and so if each word of memory contains $\Theta(\log n)$ bits (which is a standard assumption), each label requires $\Omega(n/\log n)$ words. A large number of words directly impacts the efficiency of establishing the order of two elements, since comparing their labels entails scanning that many words. Thus, the second objective is to minimize the number of bits used in labels.

Since we are interested in minimizing two competing objectives, we express the complexities of our data structures using a pair of functions. A data structure supporting n insertions with $g(n)$ maximum relabels and using $h(n)$ bits per label is said to have complexity of $\langle g(n), h(n)\rangle$. Notice that $h(n) \geq \lceil \log n \rceil$ since n elements must be labeled. If one is interested in $h(n) = O(\log n)$ (constant number of words per label), then the OMLL lower bounds of [11] imply that $g(n) = \Omega(\log n)$. Thus, if there existed a solution for online house numbering with complexity $\langle O(\log n), O(\log n)\rangle$, it would be asymptotically optimal.

1.1 Our Results

In this paper we describe two data structures that are close to the target bound of $\langle O(\log n), O(\log n)\rangle$, but each solution introduces an extra logarithmic factor in one of the functions. In a third solution, we investigate the dependence on the leading constant of $h(n)$ and provide a solution with complexity $\langle O(n^\epsilon), \log n + O(1/\epsilon)\rangle$. Our solutions, which can be adapted to work with $o(n)$ scratch space (deferred to the full version of the paper), establish the following results.

▶ **Theorem 1.** *There exists a house numbering data structure with complexity* $\langle O(\log^2 n), O(\log n)\rangle$*.*

▶ **Theorem 2.** *There exists a house numbering data structure with complexity* $\langle O(\log n), O(\log^2 n)\rangle$*.*

▶ **Theorem 3.** *For any positive constant* ϵ*, there exists a house numbering data structure with complexity* $\langle O(n^\epsilon), \log n + O(1/\epsilon)\rangle$*.*

Proofs of Theorems 1 and 3 appear in Section 3. Theorem 2 is deferred to the full version of the paper. Our solution complexities exhibit an interesting feature: the online house numbering problem seems to exhibit a different tradeoff from OMLL, depending more strongly on the label lengths.

Overview of Challenges and Techniques. The main idea of our approach is that once a particular element has been relabeled many times, structural restrictions assure that this element will not be relabeled much in the future. To achieve this goal, we employ a tree-like structure similar to an (a, b)-tree (or B-tree) that stores at most $O(\gamma)$ elements in nodes of the tree, where $\gamma \geq 2$ is a parameter controlling the tradeoff between the two objectives. (For the purpose of this overview it is helpful to assume $\gamma = 2$.) Roughly speaking, the inorder traversal of this tree corresponds to the order of elements in the list. The elements in a node each have a local label, which is local to that node. The global label assigned to an element corresponds to the concatenation of the local labels on the path from the root to the element (with 0s padded at the end if the element is in an inner node). We require the local labels of elements to respect the order of elements in each node, thereby guaranteing that the global labels respect the total order of the list. To simplify things when extending to nonconstant γ, we employ (classic) file-maintenance data structures within each node for maintaining the local labels. Notice that changing the local label of an element also changes the global labels, which must be stored explicitly, of all elements in that element's subtree.

Our main strategy is to employ node splits to "promote" elements that have been relabeled too many times to higher levels in the tree. Promoting an element e that is currently in node u entails: splitting the elements of u around e into two new nodes, moving e into u's parent, and making the two new nodes children of u's parent. The intuition behind the promotions is that elements in higher nodes are less affected by insertions, and hence these elements need not be relabeled as often. Element promotions happen due to three possible reasons:

(1) if the element has been relabeled too many times since its last promotion,
(2) if the node has too many elements, which would cause the performance of the file-maintenance black box to degrade, or
(3) if the node becomes sufficiently imbalanced, according to a non-obvious weight-balance rule.

The key component of the analysis is ensuring that the height H of the data structure is well bounded, i.e., that elements are never promoted too far. The height H not only places an upper bound on the length of the labels, but in conjunction with the first trigger for element promotion also directly implies a bound on the number of times each element can be relabeled. We emphasize that the analysis bounding H leverages a potential argument in an atypical and non-obvious way, which we view as a surprising component of our data structure.

1.2 Related Prior Work

There is no prior work for the online house numbering problem, but it is closely related to the classic *file maintenance* and *online list labeling* problems for which several authors have shown how to achieve optimal polylogarithmic update times, in either worst-case or amortized senses (e.g., see [6, 14, 22, 23, 11, 7]).

Regarding algorithms in computational models that capture the challenges of solid-state memory, Ben-Aroya and Toledo [3] provide competitive analyses for several such algorithms, but they do not study OMLL problems as a specific topic of interest. See also the work of Irani *et al.* [21]. Subsequent work on efficiently implementing specific data structures and algorithms in such models includes methods for database algorithms [12] and hash tables [19].

2 Preliminaries

In our house numbering data structures, we make use of instances of file maintenance data structures. The following lemma highlights the features that our algorithms leverage. Here, a file-maintenance data structure corresponds to an array, where placing an element in the ith slot in the array corresponds to assigning a label of i to the element.

▶ **Lemma 4.** *For any capacity η, there exists a file maintenance data structure with the following properties:*

- *The data structure assigns to each element a slot in the range $[1, 4\eta]$. Slots are such that a is before b if and only if in the total order a's slot is before b's slot.*
- *If the data structure has at most η elements then it can be split into two data structures with each element being moved at most once.*
- *Starting from an empty data structure, or a data structure that is the output of a split, as long as the number of elements in the structure does not exceed η, the amortized number of elements that are moved to a new slot per insertion is $O(\log^2 \eta)$.*

Proof. A data structure by Itai et al. satisfies these conditions [22]. More detail is given in the full version of the paper. ◀

Using the notation of the statement of Lemma 4, a file maintenance data structure f is characterized by a **capacity** (i.e., η), a **slot range** (i.e., $[1, 4\eta]$), and an amortized **moving cost** $cost(\eta)$ (i.e., $O(\log^2 \eta)$), which are all static. The capacity specifies how many elements can be inserted while still maintaining the $cost(\eta)$ bound. In addition, we define the **usage** of f, denoted $usage(f)$, to be the number of elements currently inside f.

3 A Generic House Numbering Data Structure

We describe our data structure in terms of several variables, namely κ_1, κ_2, κ_3, π_1, and π_2. These will allow us to balance various overheads in the data structure and shall be fixed in the analysis. Additionally, our house numbering data structure is parameterized by a value $\gamma \geq 2$, which controls a tradeoff between the label lengths and the number of relabels performed. Setting γ to a constant attains the $\langle O(\log^2 n), O(\log n)\rangle$ data structure. When considering a data structure for flash memory, the data structure itself, in addition to the actual list, should reside in scratch space. We ignore this issue in the current section but address it in the full version of the paper.

The tree. Our house-numbering structure is a perfect rooted $(4\kappa_1\gamma + 1)$-ary tree T. Each internal node u in the tree maintains an instance f_u of a $\kappa_1\gamma$-capacity file-maintenance data structure (à la Lemma 4) and the leaves of the tree are associated with space for a single element. The leaves store the actual elements e in the tree, but leaves may be empty. Each element e also maintains a **relabel counter** $c(e)$.

The internal nodes store (conceptual) copies of elements, which we call **representatives**, that have been promoted to a higher level in the tree. We refer to all the copies of a particular element e as **the representatives** e. Representatives are analogous to duplicate keys in internal nodes of a B$^+$ tree, with each non-empty node containing exactly one representative that has been promoted to the parent node.

Each file-maintenance data structure f_u assigns slots in the range $[1, 4\kappa_1\gamma]$ to the representatives in node u. Equivalently, the file-maintenance structure specifies how to store the representatives in a size-$\kappa_1\gamma$ array, starting from slot 1. We use the 0th slot in the array for a special **dummy representative** d_u^-, which corresponds to the only representative in node u that has also been promoted to the parent. (As such, d_u^- is a representative of the leftmost left element in u's subtree.) Note that since the slot storing the dummy representative is not part of the file-maintenance structure f_u, the dummy representative never moves from slot 0. The i-th slot in f_u corresponds to the i-th child of u in an inorder tree walk. For representative r in f_u let $s(r)$ denote the slot in f_u that is assigned to r.

Without yet worrying about precisely how elements are labelled, we state a property about how they must appear in T. Naturally, the order property constrains the way we label elements.

▶ **1 (Order Property).** *An inorder traversal of T encounters the elements in their house numbering order.*

We use T_u to denote the subtree rooted at node u. Let v be the parent of u in T. Since u is represented as a slot s in the slot range of f_v, we will abuse notation and sometimes denote T_u by T_s. A subtree is empty if it contains no elements.

The labels. The label for an element e, denoted $\ell_T(e)$, is based on the root-to-leaf path down to the leaf node containing e. In particular, labels are base-$(4\kappa_1\gamma + 1)$ numbers with a number of digits equal to the height of the tree. Consider the path $u_1, u_2, \ldots, u_{H_T+1}$ down to the leaf containing e, where u_1 is the root, u_{H_T+1} is the leaf containing e, and H_T is the height of T. For $1 \leq i \leq H_T$ let s_i denote the slot of u_{i+1} in f_{u_i}. Then e's label is the concatenation of digits $s_1, s_2, \ldots, s_{H_T}$. An example of determining the labels of elements is depicted in Figure 1. The label of each element uses $\lceil \log(4\kappa_1\gamma + 1)\rceil$ bits per level of T for a total of $H_T\lceil \log(4\kappa_1\gamma + 1)\rceil$ bits.

Notice that by construction and the Order Property, the labels of elements respect the order of the elements in the house numbering.

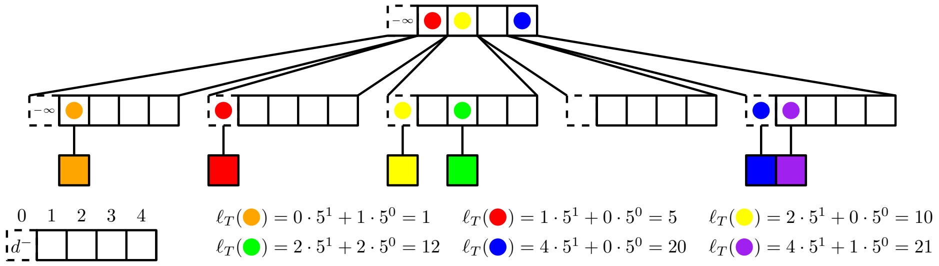
−∞
−∞
0 1 2 3 4
d^-
$\ell_T(●) = 0 \cdot 5^1 + 1 \cdot 5^0 = 1$
$\ell_T(●) = 1 \cdot 5^1 + 0 \cdot 5^0 = 5$
$\ell_T(●) = 2 \cdot 5^1 + 0 \cdot 5^0 = 10$
$\ell_T(●) = 2 \cdot 5^1 + 2 \cdot 5^0 = 12$
$\ell_T(●) = 4 \cdot 5^1 + 0 \cdot 5^0 = 20$
$\ell_T(●) = 4 \cdot 5^1 + 1 \cdot 5^0 = 21$

Figure 1 This tree illustrates how elements are labelled based on their representative's node's file maintenance labels and the node labels of their parents. Empty leaf nodes are omitted and we note that the root node is violating the Capacity Property.

Relabeling and subtrees. To maintain the Order Property, whenever a representative r is moved from slot s to slot s' in f_u, all of the elements and file-maintenance representatives in T_s are moved to the same exact location, but in $T_{s'}$. The following property will guarantee that this movement does not violate the Order Property.

▶ **2 (Representative Property).** *The representatives for an element e induce a path from the parent of the leaf containing e to the highest representative. Each representative of e except for the highest one is the dummy representative of its corresponding node.*

Following the Representative Property, we abuse notation and refer to the highest representative of an element e as the canonical representative of e, and denote this representative by $r(e)$.

3.1 Insertions

We now discuss the implementation of the insert(x, a) operation. Let u be the parent of the leaf node containing a, and assume for now that $usage(f_u) < \kappa_1\gamma$. A new representative r of x is inserted into f_u immediately after the representative representing a in f_u (possibly causing elements in f_u to change slots). Because this insertion is into a file maintenance data structure, this insertion may cause some movement of other elements. Element x is placed into the leaf node corresponding to the slot assigned to r in f_u.

The insertion respects the Order Property, so x receives a valid label. The insertion causes some number of other representatives in f_u to be relabelled and also increases the usage of f_u. Eventually f_u will reach capacity. The capacity of the file maintenance instances needs to be respected and so when f_u reaches capacity we move around representatives in T to create room (thereby guaranteeing again that f_u is below capacity before the next insertion). This is captured by the following property.

▶ **3 (Capacity Property).** *For any internal node u in T, $usage(f_u) < \kappa_1\gamma$.*

In order to maintain the Capacity Property, the data structure employs *promoting* canonical representatives to higher nodes in T. The promotion procedure is detailed in Section 3.2.

Relabel counters. The relabel counter of an element is incremented whenever the label of the element is changed. To prevent any one element from being relabelled too many times, we enforce a bound on the relabel counter. Recall that the $cost(\eta)$ function is defined in Section 2.

Algorithm 1 Insert x into the house numbering data structure immediately after element a.

```
function INSERT($x, a$)
    $u \leftarrow$ parent of $a$'s leaf
    insert a representative of $x$ into $f_u$
    move any relabeled elements to their new leaves
    place $x$ into the corresponding leaf of $u$
    repeatedly fix property violations using promote()
end function
```

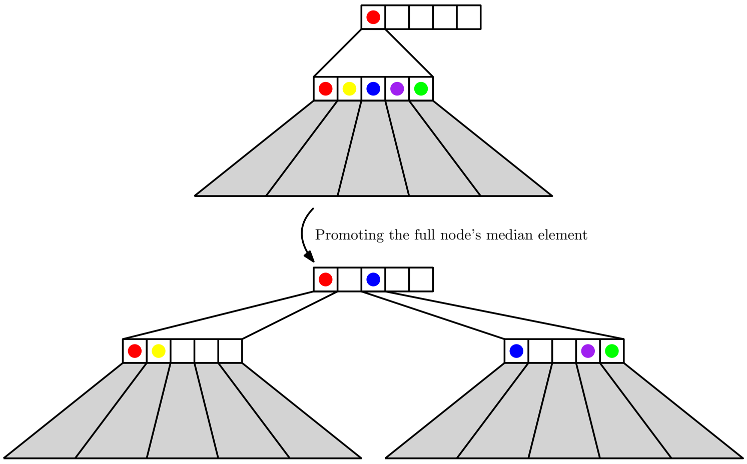

Figure 2 The blue median representative of the full file maintenance data structure is promoted dividing the subtree into two parts.

▶ **4. Counter Property:** *For any element e, $c(e) < \kappa_2\, cost(\kappa_1\gamma)$.*

In order to enforce the Counter Property, whenever the counter of element e reaches its threshold, $r(e)$ is moved one level higher in the tree by a *promotion operation*, which we describe shortly, and sets $c(e) = 0$.

Notice that moving a representative to a new slot higher up in T will tend to relabel more elements compared to moving a representative to a new slot in a lower node in T. This presents a subtle challenge. Consider two representatives in f_u where u is relatively high up in T, such that one representative r has every file maintenance instance in $T_{s(r)}$ half full while the other representative r' has every file maintenance instance in $T_{s(r')}$ just below capacity. This implies that the number of elements contained in the two subtrees differ exponentially in the height of the subtrees. The consequence of this imbalance is that insertions of elements into the lighter subtree $T_{s(r)}$ can cause frequent promotions into f_u, each time causing r' to move to a new slot in f_u. When r' moves to a new slot, all of the elements in $T_{s(r')}$ must be relabeled. This imbalance creates some difficulties in keeping a tab on the complexities of the data structure. To overcome these difficulties, we enforce a requirement on the data structure to have the following property, which helps ensure a promotion does not relabel too many elements that are too high in the tree by restricting the weight of any subtree. For a representative r in f_u where u has height h in T, let $w(r) = \gamma^h$.

▶ **5. Balance Property:** *For any node u, $\sum_{node\ v \in T_u} \sum_{canonical\ representative\ r \in f_v} w(r) < \kappa_3 \gamma^{height(u)}$.*

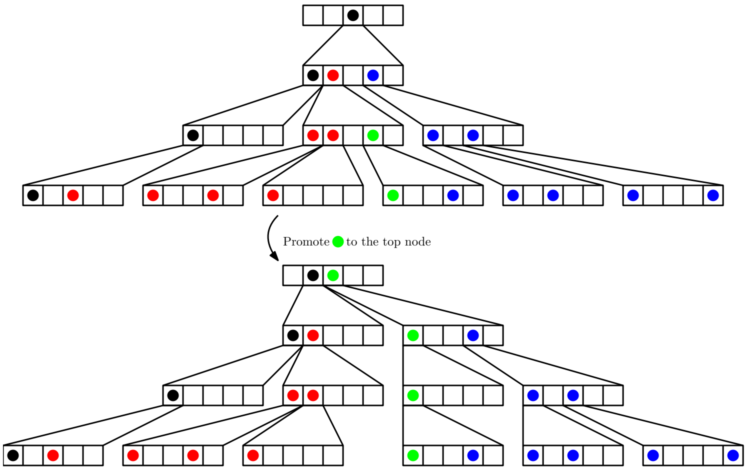

Figure 3 The black representative's subtree passed the threshold of the Balance Property and the weighted median representative, shown in green, is promoted. The subtrees of representatives less and greater than the median are copied and possibly shifted if they need a new root representative. Empty subtrees are omitted.

Algorithm 2 Promote an element x into a node u

```
function PROMOTE($x, u$)
    $v \leftarrow$ node containing the canonical representative for $x$
    remove $x$ from $f_v$
    $a \leftarrow$ predecessor of $x$ in $f_u$
    insert a new canonical representative of $x$ after $a$ in $f_u$
    move the entire subtree of any relabeled elements
    split the subtree below $a$'s canonical representative into:
            - $T_1$ a subtree of elements $< x$ and $>= a$
            - $T_2$ a subtree of elements $>= x$
    place $T_1$ below $a$'s canonical representative
    place $T_2$ below $x$'s canonical representative
end function
```

3.2 Promotions

For element e, the *promotion* of $r = r(e)$ from f_u to f_v, where v is a proper ancestor of u, is performed as follows. Let $\hat{r}$ be the representative in the slot in f_v that contains e in its subtree:

1. Insert r', which is a new representative of e, into f_v immediately after $\hat{r}$ in the order f_v is maintaining (this may cause some elements in f_u to change their slots).
2. Any element in $T_{s(\hat{r})}$ that is after e (inclusive) and its representatives in $T_{s(\hat{r})}$ are moved into identical locations in the subtree of $T_{s(r')}$.
3. The previous step partitions some file maintenance instances into two pieces. Each such instance respaces the representatives it contains according to the split operation of Lemma 4. Notice that if the dummy representative is part of one piece, the data structure adds a new dummy representative in the other piece. This new dummy representative is a representative of e.

Examples of promotions are shown in Figures 2 and 3.

▶ **Lemma 5.** *Promotions preserve the Order and Representative Properties.*

Proof. Before the promotion, the Order Property held and so an inorder traversal encountered the elements with label less than $\ell_T(e)$, then e, and then the elements with label greater than $\ell_T(e)$. After the promotion, the inorder traversal will traverse $T_{s(\hat{r})}$ which contains the elements less than e and then $T_{s(r')}$ which contains e followed by the elements greater than e. The two new subtrees preserved the original inorder traversal of their contained elements. So the inorder traversal before and after the promotion traverses the elements in T in the exact same order and the Order Property holds. The last step of a promotion ensures that the representatives of e still behave properly with respect to the Representative Property. Since we split along the path of elements less than and greater than e, no other path of representatives was altered and the Representative Property still holds. ◀

The only operations we perform on T are insertions of elements at leaves and promotions. Both of these preserve the Order and Representative Properties. Violations of the Capacity, Balance, and Counter Properties are fixed by promoting certain representatives. When a node u with parent v violates the Capacity Property, promote the median representative in f_u (which must be a canonical representative) into f_v. When the subtree of $s(r)$ becomes too heavy and violates the Balance Property, promote the weighted median canonical representative in the subtree of $s(r)$ into the node that contains r. When $c(e)$ passes the threshold of the Counter Property, promote the canonical representative of e into the parent of its current node, and reset $c(e)$ to be zero. Figure 3 shows a promotion due to a violation of the Balance Property and Figure 2 shows a promotion due to a violation of the Capacity Property.

Promoting a representative may introduce new property violations. For example, suppose f_u for some internal node u contains $\kappa_1\gamma$ representatives and its parent v has exactly $\kappa_1\gamma - 1$ representatives. Promoting the median representative from f_u to f_v will cause the f_v to violate the Capacity Property.

The very rough pseudocode in Algorithms 1 and 2 describes the high level steps for insertions and promotions. We describe exactly how violations are processed next.

Property violations. Since several properties may be violated at the same time, we employ the following prioritization for fixing these violations. We process the property violations by alternating between processing all violations of the Capacity and Balance Properties in a highest first fashion and then processing a single violation of the Counter Property. Algorithm 3 shows this procedure in pseudocode.

It is not yet clear that this processing terminates. We address this in Section 4. Whenever the initial insertion or a promotion causes an element to be relabeled, we increment the corresponding relabel counter.

During the processing of violations, a given relabel counter may be increased well past the bound in the Counter Property. But our potential argument only allows us to charge for relabelings that occur when the counter is at or below the threshold. To keep from relabeling the corresponding element each time an above-threshold counter is pushed even higher during a single (recursive) house numbering insertion, we perform the invariant violation processing on a logical copy of the data structure and only relabel elements with the final label. While a relabel counter may be incremented many times, any element is only relabelled at most once per insert operation.

Algorithm 3 Process the violations in the tree

```
function PROCESS_VIOLATIONS
    while there is a violated property do
        while there is a violation of the capacity or balance properties do
            process the highest capacity or balance violation
            (capacity violations have priority)
        end while
        if there is a violation of the counter property then
            process one violation of the counter property
        end if
    end while
end function
```

4 Bounding the Height and Complexities

Let $H(n)$ denote an upper bound on the maximum possible height of a canonical representative in our house-numbering data structure after n elements are inserted. (We shall bound $H(n)$ as a function of γ in Lemma 9.) Then we can directly bound the length of labels at $H(n) \cdot \lceil \log(4\kappa_1\gamma + 1) \rceil$ bits. In particular, if κ_1 and γ are constants, we will prove that $H(n) = O(\log n)$ and hence the labels use $O(\log n)$ bits. Moreover, since we guarantee the Counter Property, each element e can be relabeled at most $\kappa_2\, cost(\kappa_1\gamma)$ times before $r(e)$ is moved up a level. So the maximum number of times that an element is relabeled is $O(\kappa_2 H(n)) = O(\kappa_2 \log n)$, assuming $\kappa_1\gamma$ is a constant and $H(n) = O(\log n)$.

The intuition behind our height analysis is as follows. Each insertion causes $cost(\kappa_1\gamma)$ representatives to be relabeled. Thus we need roughly κ_2 insertions to trigger enough relabels that a single representative could be promoted by the Counter Property. In other words, at most a $1/\kappa_2$ fraction of representatives are promoted due to insertions of elements and the Counter Property. This argument extends up the tree; promotions into height-h nodes can cause at most a $1/\kappa_2$ fraction of representatives to be promoted from height h. If this were the only effect, we would see $(1/\kappa_2)^h$ representatives promoted to height h.

This challenge turns out to be even more complex, since each promotion into a height-h node u also causes the elements in subtrees of any locally relabeled representatives in u to be completely relabeled. The Balance Property helps us to bound the total weight of representatives in these subtrees by $\kappa_3\gamma^h$. By increasing κ_2 enough, we effectively amortize the high number of relabelings due to moving a subtree against the geometrically decreasing number of promotions to that height, i.e., about $\kappa_3\gamma^h/\kappa_2^h$ per insertion. Since there are some "feedback" effects that arise from the interaction of fixing property violations, the analysis must proceed with care.

Before we turn to bounding the height, we prove a useful lemma.

▶ **Lemma 6.** *If all of the properties hold before an insertion, the processing of the resulting violations will never promote a representative into a node that:*

- *violates the Capacity Property or*
- *contains a representative whose subtree violates the Balance Property.*

Proof. Call a promotion into such a node an *invalid promotion*. We claim that in addition to never performing an invalid promotion, the violation processing maintains the property that violations of the Capacity and Balance Properties each occur at most once in each level of T. We call this the *Once Per Level Property*. When a representative is inserted or promoted into u: the usage of u is increased, the weight of every subtree of every representative on the path to the root is increased, and the relabel counters of any relabeled elements are increased. So an insertion or promotion will only introduce violations of the Capacity Property at u and violations of the Balance Property along the path from u to the root (and some other violations of the Counter Property). For example in Figures 2 and 3, each promotion can only create Counter Property violations lower in the subtree while it may introduce Capacity and Balance Property violations at the root. Hence the initial insertion or promotion from processing a Counter Property violation in a tree with no violations of the Capacity or Balance Properties is valid and will maintain the Once Per Level Property.

When the Once Per Level Property holds, there is some highest violation of each type. There is a highest node violating the Capacity Property and a highest node containing a representative violating the Balance Property. Let u be the higher of these two nodes. If both nodes have equal height, then let u be the highest node violating the Capacity Property. We consider the cases when u violates the Capacity Property or when u does not violate the Capacity Property and violates the Balance Property.

In the first case, f_u violates the Capacity Property by containing $\kappa_1\gamma$ representatives. Because the parent of u is not violating either of the two properties, promoting the median of f_u is valid. That promotion may introduce violations at the parent of u or higher, but they will only be in levels of the tree where there were previously no violations. The violations that were either at u or below will be unaffected by the promotion (except for the one being processed). Therefore the Once Per Level Property still holds after the violation is processed.

In the second case, f_u does not violate the Capacity Property but it does have one representative violating the Balance Property. Because no other representative in f_u violates the Balance Property due to the Once Per Level Property, processing the violation is valid. By promoting the weighted median descendant into u, only f_u can be newly in violation of the Capacity Property and only representatives in ancestors of u can be newly in violation of the Balance Property. Both of these types of new violations are introduced at levels that did not contain a violation of that type before. The splitting of the subtree below into two pieces can only eliminate violations in the levels below u. So after validly processing this violation, the Once Per Level Property holds.

In either case, processing a violation does not make an invalid promotion and maintains the Once Per Level Property. Thus, the invariant processing never promotes a representative into a node violating the Capacity Property or containing a representative whose subtree violates the Balance Property. ◀

Potential argument. To formalize the intuition outlined in the beginning of this section, we analyze our data structure using the following three potential functions, each of which corresponds to one of our properties:

- $\Phi_{fmds} = \pi_1 \sum_{internal\ nodes\ u} \max\left(2\gamma^{height(u)} \cdot usage(f_u) - \kappa_1\gamma^{height(u)+1}, 0\right)$
- $\Phi_{counters} = \sum_e w(r(e))c(e)$
- $\Phi_{tree} = \pi_2 \sum_u \max\left(2\sum_{node\ v\in T(u)} \sum_{canonical\ representative\ r\in f_v} w(r) - \kappa_3\gamma^{height(u)}, 0\right)$

The total potential, $\Phi(T)$, is the sum of these three potential functions, that is $\Phi = \Phi_{fmds} + \Phi_{counters} + \Phi_{tree}$. Each potential function corresponds to one of our three properties and guarantees that when a property is violated we have sufficient potential to "pay" for the promotion.

The next few lemmas show how the potential functions work with the properties. The change in Φ due to a processing a violation can be separated into the two phases of a promotion. First, there is the decrease in potential when the promoted element's relabel counter is reset and the node containing the canonical representative of the element is split. Second, there is the increase in potential due to the insertion of the canonical representative of the element into a higher up node which results in relabeling many other elements, increasing that node's usage, and increasing the weight of every subtree containing the higher up node. Lemma 7 gives an upper bound on the increase in potential due to either an insertion or the second part of a promotion. Lemma 8 gives a lower bound on the decrease in potential due to the first part of a promotion. In conjunction these two Lemmas show that as long as the maximum height $H(n)$ is small, the lower bound on the decrease in potential is greater than the upper bound on the increase in potential. So a promotion results in a net decrease in potential and only insertions of new elements at the leaves increase the potential. Finally Lemma 9 contrasts the amount of potential gained from these insertions with the amount of potential needed to promote one representative to a height of $\log_\gamma n$. Because the former is strictly smaller, the height of the tree must be $H(n) < \log_\gamma n$.

▶ **Lemma 7.** *During a promotion, the insertion of a representative into a file maintenance data structure at height h increases $\Phi(T)$ by at most $(2\pi_1 + \kappa_3\, cost(\kappa_1\gamma) + 2\pi_2\, height(T))\gamma^h$. Moreover, the insertion of an element increases $\Phi(T)$ by at most $2\pi_1 + \kappa_3\, cost(\kappa_1\gamma) + 2\pi_2\, height(T)$.*

Proof. Placing a canonical representative into a node u at height h causes the potential functions to change as follows:

- $\Delta(\Phi_{fmds}) \leq 2\pi_1\gamma^h$, because f_u had its size increased by 1
- $\Delta(\Phi_{counters}) \leq \kappa_3\gamma^h\, cost(\kappa_1\gamma)$, because $cost(\kappa_1\gamma)$ representatives in f_u are relabeled, each causing subtrees with total weight at most $\kappa_3\gamma^h$ to be relabeled.
- $\Delta(\Phi_{tree}) \leq 2\pi_2\, height(T)\gamma^h$, because each representative on the path to the root has the potential of its subtree increased by at most γ^h

The bound on $\Delta(\Phi_{counters})$ is in an amortized sense and is due to Lemma 6. This is because by Lemma 6 we never promote a representative into a node that is violating the Capacity Property, so there are at most $cost(\kappa_1\gamma)$ relabels, or into a node violating the Balance Property, so incrementing the relabel counters of each subtree costs at most $\kappa_3\gamma^h$ potential.

In total these sum up to $(2\pi_1 + \kappa_3\, cost(\kappa_1\gamma) + 2\pi_2\, height(T))\gamma^h$ which upper bounds the increase in all three potential functions. ◀

▶ **Lemma 8.** *If $height(T) \leq H(n)$, there exist settings of π_i's and κ_i's such that promoting a representative to fix a violation does not increase the total potential and $\kappa_2 = O(\gamma H(n))$.*

Proof. Depending on which violation caused the promotion, we must analyze the decrease in potential differently to account for the potential increase from Lemma 7 of $(2\pi_1 + \kappa_3\, cost(\kappa_1\gamma) + 2\pi_2 H(n))\,\gamma^h$ where h is again the height of the node the promoted element is moved into.

- If a Capacity Property violation was processed, then Φ_{fmds} decreased by at least $\pi_1\kappa_1\gamma^h$.
- If a Counter Property violation was processed, then $\Phi_{counters}$ decreased by at least $\gamma^{h-1}\kappa_2\, cost(\kappa_1\gamma)$ due to the potential from $c(e)$.
- If a Balance Property violation was processed, then Φ_{tree} decreased by more than $\pi_2\kappa_3\gamma^h$ because of the subtree containing r.

To ensure the potential available is always at least the potential cost $\pi_1\kappa_1$, $\frac{\kappa_2\,cost(\kappa_1\gamma)}{\gamma}$, and $\pi_2\kappa_3$ must all be greater than $2\pi_1 + \kappa_3\,cost(\kappa_1\gamma) + 2\pi_2 H(n)$. Analyzing the system of inequalities leads to setting $\kappa_1 = 3$, $\kappa_2 = 72\gamma H(n)$, $\kappa_3 = 12H(n)$, $\pi_1 = 24\,cost(3\gamma)H(n)$, and $\pi_2 = 6\,cost(3\gamma)$.

Plugging these values back into the original formula, the potential increases by at most

$$(2(24\,cost(3\gamma)H(n)) + (12H(n))\,cost(3\gamma) + 2(6\,cost(3\gamma))H(n))\gamma^h = 72\,cost(3\gamma)H(n)\gamma^h.$$

On the other hand, the potential decrease is at least

- $(24\,cost(3\gamma)H(n))(3)\gamma^h$ in the case of a Capacity Property violation,
- $\gamma^{h-1}(72\gamma H(n))\,cost(3\gamma)$ in the case of a Counter Property violation, or
- $(6\,cost(3\gamma))(12H(n))\gamma^h$ in the case of a Balance Property violation.

In all three cases, the lower bound of the decrease in potential is equal to $72\,cost(3\gamma)\gamma^h H(n)$ and therefore it is at least the increase in potential due to a promotion. ◀

▶ **Lemma 9.** *For the same setting of π_i's and κ_i's as Lemma 8, and $\gamma \le n$, after n insertions there are no promotions to height above $\log_\gamma n$.*

Proof. Initially the tree is empty and has height zero. By Lemma 8, setting $\log_\gamma n = H(n)$, until $height(T)$ exceeds $H(n)$ promoting a representative does not increase the potential. Thus, while the height bound holds the only mechanism for increasing the potential is by inserting a new element. By Lemma 7, the increase in potential from inserting an element is at most $72\,cost(3\gamma)\log_\gamma n$ and so after n insertions, the potential of the entire data structure is at most $72\,cost(3\gamma)n\log n$.

To complete the proof, we observe that a representative can only be promoted to height h if the total potential in the data structure is at least $72\,cost(3\gamma)\gamma^h\log_\gamma n$. Specifically, the proof of Lemma 8 shows that the potential has to decrease by at least this amount when performing the promotion, so the potential has to exist before the promotion. In order to reach a height of at least $\log_\gamma n + 1$, we would need at least $72\,cost(3\gamma)\gamma^{\log_\gamma n+1}\log_\gamma n > 72\,cost(3\gamma)n\log n$ potential. Thus, a height $\log_\gamma n$ structure cannot have enough potential. ◀

▶ **Theorem 10.** *There exists a house numbering data structure with complexity $\langle O(\gamma\log^2 n)$, $\log_\gamma n \cdot \lceil\log(12\gamma+1)\rceil\rangle$.*

Proof. By Lemma 9, after n insertions there are no promotions into nodes at height higher than $\log_\gamma n$, so a tree of this height suffices. Thus, each label uses $\log_\gamma n$ "digits", where each digit uses $\lceil\log(12\gamma+1)\rceil$ bits, for a total of $\log_\gamma n \cdot \lceil\log(12\gamma+1)\rceil$ bits per label. For the relabel bound, by the Counter Property, each canonical representative is promoted at most $\kappa_2\,cost(3\gamma) = O(\gamma H(n))\,cost(3\gamma) = O(\gamma\log_\gamma n \cdot \log^2\gamma)$ times. Summing on all possible levels and applying Lemma 4, each element is only relabeled $O(\gamma\log^2 n)$ times. ◀

Theorem 1 is a special case of Theorem 10 obtained by setting $\gamma = 2$.

4.1 Achieving $\langle O(n^\epsilon), \log n + O(1/\epsilon)\rangle$

Proof of Theorem 3. Setting $\gamma = n^\epsilon$ in Theorem 10, the number of bits used is $1/\epsilon\lceil\log(12n^\epsilon + 1)\rceil = \log n + O(1/\epsilon)$. The maximum relabel bound becomes $O(n^\epsilon\log^2(n^\epsilon)) = O(n^\epsilon\log^2 n)$. That is when $\gamma = n^\epsilon$, it is an $\langle O(n^\epsilon\log^2 n), \log n + O(1/\epsilon)\rangle$ house numbering data structure. This bound is improved to $\langle O(n^\epsilon), \log n + O(1/\epsilon)\rangle$ by using the same solution with some constant $\epsilon' < \epsilon$. ◀

5 Conclusion

The house numbering problem is an interesting variant of the very well studied file maintenance and online list labelling problems. It poses some unique challenges that previous techniques do not solve. Our two data structures are able to come near optimal for the problem, but an $\langle O(\log n), O(\log n)\rangle$ house numbering data structure remains as an open problem.

References

1 Amihood Amir, Martin Farach, Ramana M Idury, Johannes A Lapoutre, and Alejandro A Schaffer. Improved dynamic dictionary matching. *Information and Computation*, 119(2):258–282, 1995.

2 Amihood Amir, Gianni Franceschini, Roberto Grossi, Tsvi Kopelowitz, Moshe Lewenstein, and Noa Lewenstein. Managing unbounded-length keys in comparison-driven data structures with applications to online indexing. *SIAM J. Comput.*, 43(4):1396–1416, 2014.

3 Avraham Ben-Aroya and Sivan Toledo. Competitive analysis of flash-memory algorithms. In Yossi Azar and Thomas Erlebach, editors, *European Symp. on Algorithms (ESA)*, volume 4168 of *LNCS*, pages 100–111. Springer, 2006. doi:10.1007/11841036_12.

4 M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

5 M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. *Journal of Algorithms*, 3(2):115–136, 2004.

6 Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In Rolf H. Möhring and Rajeev Raman, editors, *Euro. Symp. on Algorithms (ESA)*, volume 2461 of *LNCS*, pages 152–164. Springer, 2002. doi:10.1007/3-540-45749-6_17.

7 Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, Tsvi Kopelowitz, and Pablo Montes. File maintenance: When in doubt, change the layout! In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, (SODA)*, pages 1503–1522, 2017.

8 R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 91(4):489–502, 2003. doi:10.1109/JPROC.2003.811702.

9 Dany Breslauer and Giuseppe F. Italiano. Near real-time suffix tree construction via the fringe marked ancestor problem. *J. Discrete Algorithms*, 18:32–48, 2013.

10 Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. 13th Annual Symposium on Discrete Algorithms (SODA)*, pages 39–48, 2002. URL: http://www.brics.dk/~gerth/Papers/soda02.ps.gz.

11 Jan Bulánek, Michal Koucký, and Michael E. Saks. Tight lower bounds for the online labeling problem. *SIAM J. Comput.*, 44(6):1765–1797, 2015.

12 Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In *5th Conf. on Innovative Data Systems Research (CIDR)*, pages 21–31, 2011. URL: http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper3.pdf.

13 Richard Cole and Ramesh Hariharan. Dynamic LCA queries on trees. *SIAM J. Comput.*, 34(4):894–923, 2005. doi:10.1137/S0097539700370539.

14 P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *19th ACM Symp. on Theory of Computing (STOC)*, pages 365–372, 1987. doi:10.1145/28395.28434.

15 Paul F. Dietz. Fully persistent arrays (extended array). In *Algorithms and Data Structures, Workshop WADS'89, Ottawa, Canada, August 17-19, 1989, Proceedings*, pages 67–74, 1989.

16 Paul F. Dietz and Rajeev Raman. Persistence, amortization and randomization. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 78–88, 1991.

17 Yuval Emek and Amos Korman. New bounds for the controller problem. *Distributed Computing*, 24(3-4):177–186, 2011.

18 David Eppstein, Zvi Galil, Giuseppe F Italiano, and Amnon Nissenzweig. Sparsification—a technique for speeding up dynamic graph algorithms. *Journal of the ACM (JACM)*, 44(5):669–696, 1997.

19 David Eppstein, Michael T. Goodrich, Michael Mitzenmacher, and Paweł Pszona. Wear minimization for cuckoo hashing: How not to throw a lot of eggs into one basket. In Joachim Gudmundsson and Jyrki Katajainen, editors, *13th Int. Symp. on Experimental Algorithms (SEA)*, volume 8504 of *LNCS*, pages 162–173, Cham, 2014. Springer.

20 David Eppstein, Michael T Goodrich, and Jonathan Z Sun. Skip quadtrees: Dynamic data structures for multidimensional point sets. *International Journal of Computational Geometry & Applications*, 18(01n02):131–160, 2008.

21 Sandy Irani, Moni Naor, and Ronitt Rubinfeld. On the time and space complexity of computation using write-once memory or is pen really much worse than pencil? *Mathematical Systems Theory*, 25(2):141–159, 1992. `doi:10.1007/BF02835833`.

22 Alon Itai, Alan G Konheim, and Michael Rodeh. *A sparse table implementation of priority queues*. Springer, 1981.

23 Tsvi Kopelowitz. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In *IEEE Symp. on Found. of Comp. Sci. (FOCS)*, pages 283–292, 2012. `doi:10.1109/FOCS.2012.79`.

24 Tsvi Kopelowitz, Gregory Kucherov, Yakov Nekrich, and Tatiana A. Starikovskaya. Cross-document pattern matching. *J. Discrete Algorithms*, 24:40–47, 2014.

25 P. Pavan, R. Bez, P. Olivo, and E. Zanoni. Flash memory cells—an overview. *Proceedings of the IEEE*, 85(8):1248–1271, 1997. `doi:10.1109/5.622505`.

26 Dan E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *Proc. International Conference on Management of Data (SIGMOD)*, pages 251–260, 1986.

27 D.E. Willard. Maintaining dense sequential files in a dynamic environment (extended abstract). In *Proc. 14th Annual Symposium on Theory of Computing (STOC)*, pages 114–121, 1982.

28 H.-S.P. Wong, S. Raoux, SangBum Kim, Jiale Liang, John P. Reifenberg, B. Rajendran, Mehdi Asheghi, and Kenneth E. Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010. `doi:10.1109/JPROC.2010.2070050`.