Dynamic Generation of Discrete Random Variates[1]

Yossi Matias[2]
Bell Laboratories
600 Mountain Avenue
Murray Hill, N.J. 07974

Jeffrey Scott Vitter[3]
Department of Computer Science
Duke University
Durham, N.C. 27708–0129

Wen-Chun Ni[4]
Department of Computer Science
Brown University
Providence, R.I. 02912–1910

May 30, 1997

[1]This paper is a combination of two independent works [17] and [24] and collaborative work. A summarized version appears in [19].

[2]Support was provided in part by National Science Foundation grants CCR–8906949 and CCR–9111348. Part of the work was done while the author was at University of Maryland, Institute for Advanced Computer Studies, and at Tel Aviv University. Email: matias@research.bell-labs.com.

[3]Support was provided in part by a National Science Foundation Presidential Young Investigator Award with matching funds from IBM, by NSF research grants CCR–9007851 and CCR–9522047, and by Army Research Office grants DAAL03–91–G–0035 and DAAH04–96–1–0013. Part of the work was done while the author was at Brown University. Part of the revisions were done while the author was visiting Lucent Technologies, Bell Laboratories, Murray Hill, NJ. Email: jsv@cs.duke.edu.

[4]Support was provided in part by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014–91–J–4052, ARPA order 8225. Email: wcn@cs.brown.edu.

Abstract

We present and analyze efficient new algorithms for generating a random variate distributed according to a dynamically changing set of weights. The base version of each algorithm generates the discrete random variate in $O(\log^* N)$ expected time and updates a weight in $O(2^{\log^* N})$ expected time in the worst case. We then show how to reduce the update time to $O(\log^* N)$ amortized expected time. The algorithms are simple, practical, and easy to implement. We show how to apply our techniques to a recent lookup table technique in order to obtain expected constant time in the worst case for generation and update, with no assumptions about the input being made. We give parallel algorithms for parallel generation and update having optimal processor-time product. We also apply our techniques to obtain an efficient dynamic algorithm for maintaining ϵ-heaps of elements; each query is required to return an element whose value is within an ϵ relative factor of the maximal element value. For $\epsilon = 1/\text{polylog}(n)$, each query, insertion, or deletion takes $O(\log\log\log n)$ time.

Keywords: random number generator, random variate, alias, bucket, rejection, dynamic update, approximate priority queue.

1 Introduction

The generation of random variates based on arbitrary finite, discrete distributions has long been a key component of many computer simulations [1],[7], [14], [25]. Given the elements $1, 2, \ldots, N$ and their respective weights $w_1, w_2, \ldots, w_N \geq 0$, we want to design an algorithm to generate a random variate that has value j with probability $w_j / \sum_{1 \leq i \leq N} w_i$. In the static case, when the N weights are fixed, we can utilize the clever optimal algorithm by Walker, commonly called the *alias method*; the time to generate a random variate is constant and the preprocessing cost is $O(N)$ [14], [25].

In this paper we consider the problem in the important and more challenging dynamic case, in which the weights of the elements can be updated dynamically. The relevant measures of efficiency are the generation time and the update time. We can rerun Walker's algorithm each time a weight is updated, but the update cost $O(N)$ is too high. Up until recently, the best known algorithm for the dynamic problem was the binary tree-based scheme developed by Wong and Easton [26], whose generation and update times are both $O(\log N)$. Each generation requires one call to a random number generator that provides a uniform random integer in the range $[0, \sum_{1 \leq i \leq N} w_i)$.

Recently, Rajasekaran and Ross [21] and Greenberg and Vitter [12] developed different algorithms for the dynamic case that do generation and update in constant expected time for various restricted classes of updates. After the submission of the conference version of this paper [19], the authors learned of an interesting recent algorithm due to Hagerup, Mehlhorn, and Munro [13] that does generation and update in constant expected time and linear space when the weights are nonnegative integers and the maximum weight is bounded by a polynomial in N.

In this paper, we introduce practical and efficient randomized algorithms for the general dynamic case that do generation in $O(\log^* N)$ expected time and update in $O(2^{\log^* N})$ time.[1] Our algorithms, presented in Sections 2 and 3, are especially easy to implement, fast in practice, and recommended for general use. In Section 4 we show how to use a more sophisticated approach to achieve $O(\log^* N)$ amortized expected update time; that is, if the total number of updates is $t \geq 0$, the expected total time to do all the updates is $O(t \log^* N)$. The ultimate theoretical improvement is constant expected time per dynamic operation, which we show how to achieve in Section 5. We use the lookup table technique of [13] to adapt our algorithms and obtain expected constant time for generation and update, without the primary restrictions required in [13]. An application of our method to the problem of constant-time prediction for prefetching appears in [16]. Dictionary issues for efficient space utilization are considered in Section 6. In all our algorithms, no assumption is being made about the distributions of the input values and operations. The expectations are over the randomness in the algorithm itself. The implementations of the base algorithms are simple and practical with small constant factors implicit in the big-oh terms.

We also consider the parallel version of the problem, where a batch of operations are given and we would like to process them in parallel. In Section 7 we give parallel algorithms with optimal processors-time product. In particular, a batch of m generations or updates can be processed

[1] We use the standard terminology that $\log^* n$ is the smallest integer k such that k applications of the binary logarithm function applied to n, namely, $\lg(\lg(\ldots\lg(n)))$, is at most 1. For $N \leq 65536$, we have $\log^* N \leq 4$; for $N \leq 2^{65536}$, we have $\log^* N \leq 5$.

on an m-processor CRCW PRAM with optimal speedup, with respect to the algorithms mentioned above. The parallel update algorithm requires the (non-standard) Fetch&Add PRAM [11]; it can be processed with a slow-down of $t = \Theta(\log m / \log\log m)$, with high probability, on a (standard) CRCW PRAM with m/t processors, yielding optimal speedup.

We conclude in Section 8 with an efficient dynamic method for an "approximate" version of the well-known priority queue data type. Priority queues support the operations of insert, delete, and findmax; a findmax query returns an element having the maximum value of all the stored elements. The operations can be implemented in $O(\log n)$ time on the standard heap (see, e.g., [15]), in $O(\log n / \log\log n)$ time when some of the time bounds are allowed to be amortized [8], in $O(\sqrt{\log n}\,)$ amortized expected time when randomization is allowed [8], and in $O(\log\log u)$ time when the element values are integers in the universe $[1, u]$ [23]. The *ϵ-heap* we construct supports the more relaxed query ϵ-findmax, in which the element returned must have a value within an ϵ relative factor of the maximum element value. For $\epsilon = 1/\mathrm{polylog}(n)$, each query, insertion, or deletion takes $O(\log\log\log n)$ time; for $\epsilon = n^{-\mathrm{polylog}(n)}$, each query, insertion, or deletion takes $O(\log\log n)$ time.

2 First Algorithm

In this section, we describe the basic idea of our first algorithm; the analysis and more subtle aspects of it will be discussed in later sections. For completeness, we have included in Section B of the Appendix background information on three important techniques used by our algorithm, namely, the rejection method, table doubling, and dynamic hashing.

Let the initial total weight of the N elements be $W = \sum_{1 \le i \le N} w_i$. We assume the RAM model of computation in which the standard arithmetic operations on integers of value $O(W)$, including the discrete (truncated binary) logarithmic function and generating random integers, take constant time.

Let us regard each of the N elements as a "zeroth-level" element. The idea of our algorithm is to partition the zeroth-level elements by weight into ranges $R_j^{(1)}$, for $j \le \lg W$, such that $R_j^{(1)}$ is associated with the range $[2^{j-1}, 2^j)$. Note that j may be negative, since we do not restrict the elements' weights to be integers. There may be more than one element falling into a range $R_j^{(1)}$, and their total weight, written as $\mathit{weight}(R_j^{(1)})$ is in the range $[2^{j'-1}, 2^{j'})$, for some $j' > j$; we can treat the range $R_j^{(1)}$ as a new "first-level" element with weight $\mathit{weight}(R_j^{(1)})$ and put it into the second-level range $R_{j'}^{(2)}$, defined as $[2^{j'-1}, 2^{j'})$. For those ranges containing only one element, we put them into a *level table* $\mathcal{T}_1$ rather than into a second-level range.

Given a list of ranges $R_{j_1}^{(2)}, R_{j_2}^{(2)}, \ldots, R_{j_n}^{(2)}$ each containing at least two elements, we repeat the same partition process using $R_{j_1}^{(2)}, R_{j_2}^{(2)}, \ldots, R_{j_n}^{(2)}$ as second-level elements. More generally, by applying the same process to each range $R_j^{(\ell)}$ containing at least two elements, for $j \le \lg W$ and $\ell \ge 1$, we can build level-$(\ell+1)$ range $R_k^{(\ell+1)}$, defined as $[2^{k-1}, 2^k)$ for some $k \le \lg W$. The process repeats until there is no range containing at least two elements.

The process is best viewed as a level-by-level, bottom-up construction of a forest of trees. The

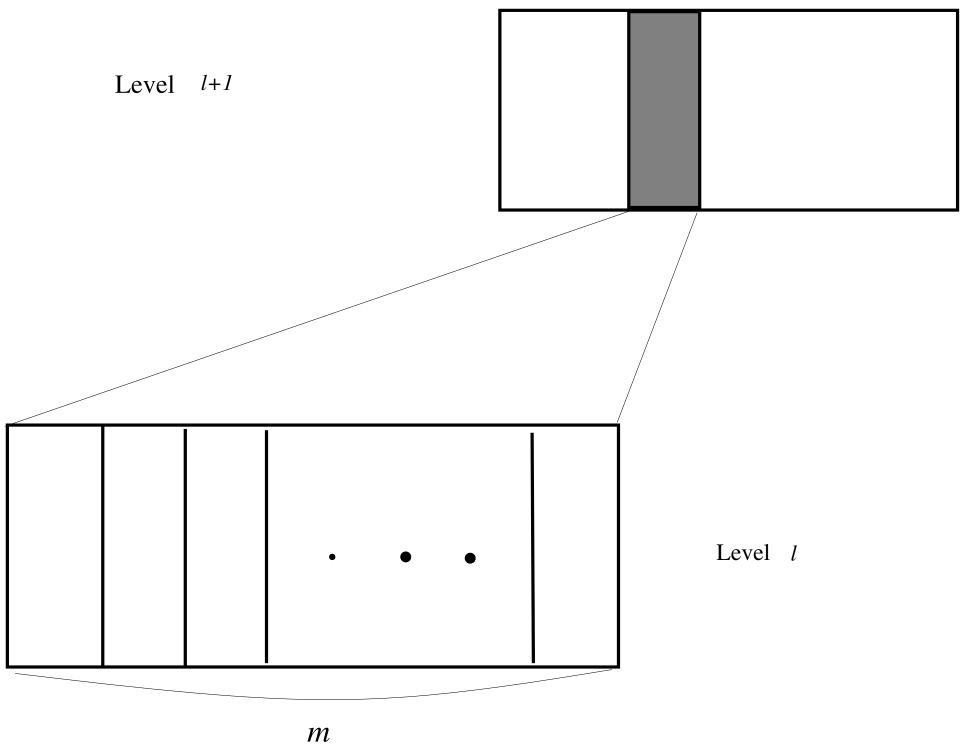


Figure 1: A range with degree m on level ℓ and its parent range on level $\ell+1$.

elements $1, 2, \ldots, N$, are implicit *leaves* in the trees being built and can be regarded as comprising the implicit level 0. Any range (node) on some level $\ell \geq 1$ is called *internal* in the collection of trees. More importantly, there is no distinction between the elements and the nonempty ranges from this viewpoint: they are all treated as nodes in a tree or elements in a set. For $\ell \geq 1$, if $R_i^{(\ell)}$ has at least two children and its total weight is in the range $R_j^{(\ell+1)}$, then $R_i^{(\ell)}$ is a *child* of range $R_j^{(\ell+1)}$; conversely, $R_j^{(\ell+1)}$ is the *parent* of $R_i^{(\ell)}$. A range with only one child is said to be a *root* range and has no parent. We define the *degree* of range $R_j^{(\ell)}$ to be the number of children it has; the degree of a root range is 1. The relation between a range and its parent range is illustrated by Figure 1.

Figure 2 gives a view of the trees built. Each level table $\mathcal{T}_\ell$, for $\ell \geq 1$, contains the nonempty root ranges created during the ℓth iteration of the tree-building process. Each nonempty root range $R_j^{(\ell)}$ is stored in a dynamic hash table, indexed by j and ℓ, so that the total space to store all the root ranges is linear. When we insert the level-ℓ roots into $\mathcal{T}_\ell$, we also compute the total weight of these roots, denoted $weight(\mathcal{T}_\ell)$. In general, we have a forest of trees whose roots may be on different levels. We denote by L the maximum level number of a root. The data structure consists of levels $1, 2, \ldots, L$.

The remaining question is how to store the children for a given internal range $R_j^{(\ell)}$. Within each range $R_j^{(\ell)}$, for $\ell \geq 1$, we keep a dynamic table of buckets. Insertions and deletions of ranges are handled by a worst-case adaptation of the well-known amortized table-doubling technique (see, e.g., Appendix B.2). Each bucket contains one range $R_i^{(\ell-1)}$ such that $weight(R_i^{(\ell-1)}) \in [2^{j-1}, 2^j)$.

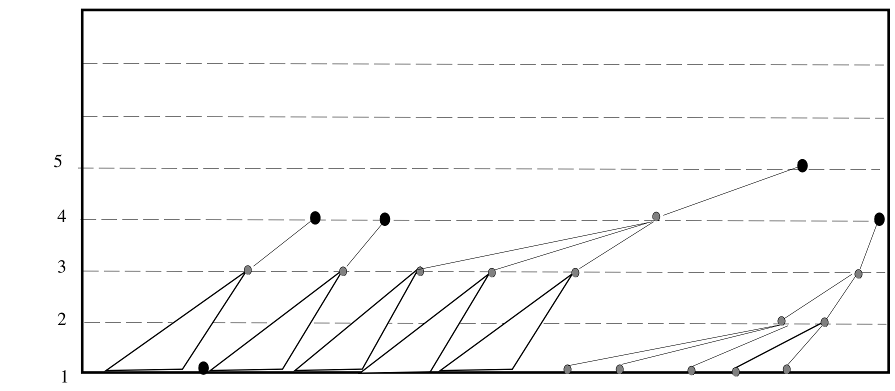


Figure 2: A forest of trees built, with $L = 5$ levels. The horizontal dashed lines mark the levels. Root nodes are denoted by solid circles.

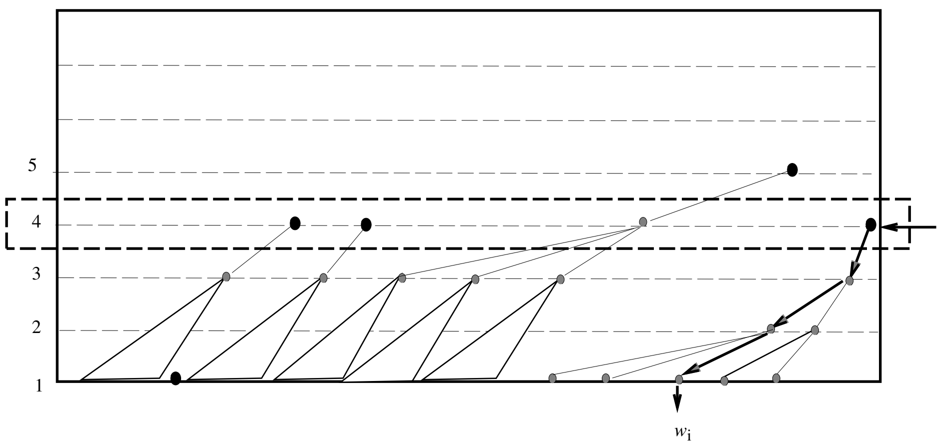


Figure 3: Generating a random variate from level $\ell = 4$.

The total weight $weight(R_j^{(\ell)})$ for $\ell \geq 1$ is defined to be $\sum_i weight(R_i^{(\ell-1)})$, where the summation is taken over the children $R_i^{(\ell-1)}$ of $R_j^{(\ell)}$.

The generation of a random variate according to the current distribution of weights w_1, w_2, …, w_N is done as follows:

Step 1. We choose some $\mathcal{T}_\ell$, where $1 \leq \ell \leq L$, based on the weights of the level tables.

Step 2. We choose one root range $R_j^{(\ell)}$ on level ℓ according to the weight distribution of the ranges.

Step 3. Within $R_j^{(\ell)}$, we use the rejection method (see, e.g., Appendix B.1) to choose one of its children according to their weight distribution. We repeat the process until we reach level 1, where the chosen child is one of the N elements. We output the chosen element.

Steps 2 and 3 are explained pictorially in Figure 3.

In Step 1 we choose one of the levels T_ℓ by generating a uniform random variate $U \in [0,1)$ and setting ℓ to the minimum positive integer such that $U < \sum_{1 \leq k \leq \ell} weight(T_k)$. The value of ℓ is found by a sequential search using values $\ell = 1, 2, \ldots$. In Step 2 we choose a nonempty root range $R_j^{(\ell)}$ on level ℓ by processing the nonempty root ranges $R_{j_1}^{(\ell)}$, $R_{j_2}^{(\ell)}$, ... $R_{j_s}^{(\ell)}$ in sequence, where $j_1 > j_2 > \cdots > j_s$ until we find the minimum value $1 \le j \le s$ such that $U \le \sum_{1 \le k \le j} weight(R_k^{(\ell)})$. The first (largest) index j_1 can be computed easily, for example, as $\lfloor \lg roots(\mathcal{T}_\ell) \rfloor + 1$, where $roots(\mathcal{T}_\ell) = \sum 2^{j_i}$, with the summation taken over all nonempty root ranges j_i on level ℓ. The successive indices j_2, j_3, ...can be obtained by iteratively subtracting 2^{j_1} and taking the discrete log function again. Alternatively, it suffices to step down iteratively from j_1 until we find the values j_i for which $R_{j_i}^{(\ell)}$ is nonempty. Step 3 consists of descending level by level from $R_j^{(\ell)}$ using the rejection method at each step until an element at the bottom level is reached, which we output.

Theorem 1 *The expected cost of the above algorithm for generating a random variate distributed according to the current weights is $O(\log^* N)$, where N is the number of elements.*

Proof: The cost of Step 1 is $O(L)$, since there are L levels to choose from. We show in Theorem 3 that $L \le \log^* N + 1$ in the worst case (although it is typically even smaller). In Step 2, generating one root range $R_j^{(\ell)}$ on level ℓ may cost time linear in the number of nonempty root ranges on level ℓ. Fortunately, the expected time is constant, since the range weights decrease exponentially. Let $R_{j_1}^{(\ell)}$, $R_{j_2}^{(\ell)}$, ..., $R_{j_n}^{(\ell)}$ be the set of nonempty root ranges on level ℓ, where $j_1 > j_2 > \cdots > j_s$. The expected cost of Step 2

$$\sum_{1 \le k \le n} k \cdot weight(R_{j_k}^{(\ell)}) / weight(\mathcal{T}_\ell).$$

This expression can be simplified using the facts that $2^{j_k - 1} \le weight(R_{j_k}^{(\ell)}) < 2^{j_k}$ and $weight(\mathcal{T}_\ell) \ge \sum_{1 \le k \le n} 2^{j_k - 1} \ge 2^{j_1 - 1}$ to yield the following upper bound on the expected Step 2 cost:

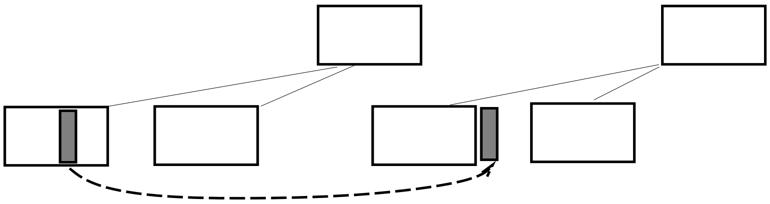
$$\sum_{1 \le k \le n} k \cdot 2^{j_k - j_1 + 1} \le \sum_{1 \le k \le n} k 2^{-k} < 2.$$

In Step 3 we walk down the levels from $R_j^{(\ell)}$ in constant expected time per level, using the rejection method, using a total of $O(\log^* N)$ expected time. □

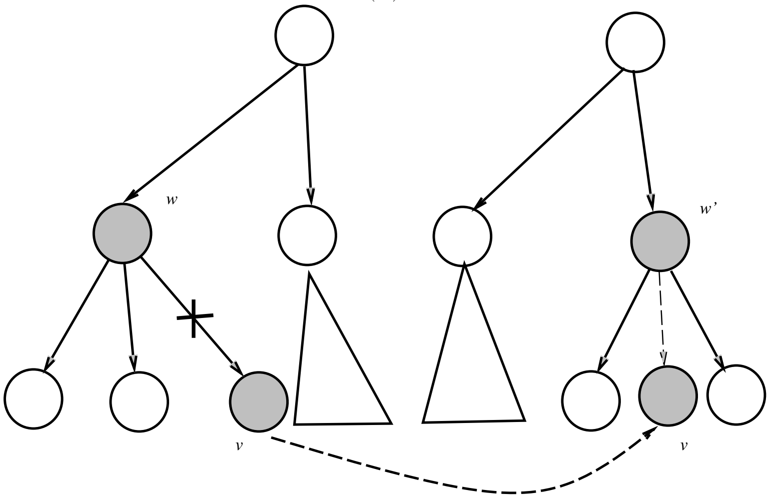
The dynamic scheme of Wong and Easton [26] uses $O(\log N)$ time per generation, but it requires only one call to a random number generator that outputs a uniform number in the range $[0, \sum_{1 \le i \le N} w_i)$. Our algorithm uses an average of at most about $2L$ calls to a uniform random number generator, primarily due to Step 3. It may be possible to use a faster uniform random number generator or to "share" random numbers: The random numbers needed in Step 3 do not usually require the precision of those needed for Wong and Easton's algorithm, especially when $\sum_{1 \le i \le N} w_i$ is large; the maximum precision needed is proportional to the degree of the current node in the tree, which is at most N but is typically very small.

2.1 Updating the Weights—Basic Approach

The weights of any element may be modified in an on-line manner, as follows: When the weight w_i in range $R_j^{(1)}$ is changed to $w_i + \Delta$, we must move the corresponding element i to another range $R_k^{(1)}$



(a)



(b)

Figure 4: Two views of the update operation. (a) Moving one bucket from one range to another. (b) The tree view: changing the parent of v from w to w'.

if $w_i + \Delta \notin [2^{j-1}, 2^j)$, in which case we say that element i "changes its parent." A change of an individual weight may thus cause the total weights of two level-1 ranges $R_j^{(1)}$ and $R_k^{(1)}$ to change, which may cause further parent changes higher in the trees.

Coordinating the updates from the bottom up is achieved by associating to each level a queue, as we do in the preprocessing stage. Once the weight of a range has been changed on level ℓ, we reflect the required update to level $\ell+1$ by putting the value changed and the range into the queue. We can view the effect by looking at Figure 4. In Figure 4b, the node v changes its parent node from w to w' because of weight increase. (We can use the table doubling technique of Section B.2 to organize the buckets in each range.) The paths upward from w and w' should be updated accordingly.

The number of ranges affected on level ℓ is no more than 2^ℓ, since each update along an upward path in the data structure spawns at most one new upward update path. In Theorem 3, we show that there are at most $\log^* N + 1$ levels, and hence the total number of ranges affected is bounded by $2^{\log^* N+1}$. By using dynamic hashing, we are able to insert new ranges and delete old ranges in constant expected time. This means that each update takes $O(2^{\log^* N})$ expected time:

Theorem 2 *Updating the weight of any element can be performed in $O(2^{\log^* N})$ expected time in the worst case.*

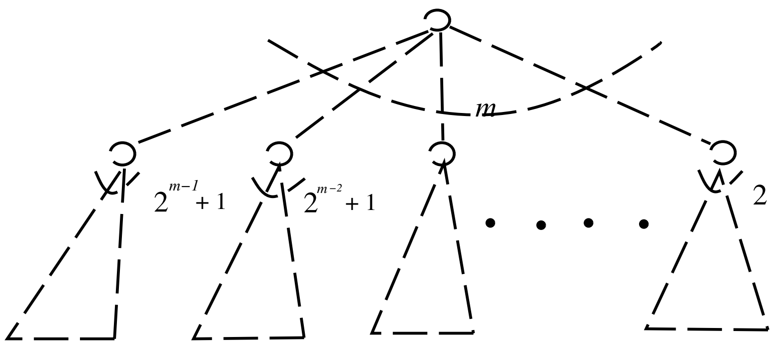


Figure 5: A typical tree built.

2.2 Properties of the Data Structure

In this section we derive some important invariants that are crucial to the analysis of the size and height of the data structure.

Lemma 1 *If the degree of range $R_j^{(\ell)}$ is $m \geq 2$, then weight$(R_j^{(\ell)})$ is in the range $[2^{j'-1}, 2^{j'})$, where $\lg m - 1 < j' - j < \lg m + 1$.*

Proof: Since every bucket in $R_j^{(\ell)}$ represents an element with weight in the range $[2^{j-1}, 2^j)$, we have $weight(R_j^{(\ell)}) \in [m2^{j-1}, m2^j)$. If $weight(R_j^{(\ell)})$ falls into $R_{j'}^{(\ell+1)}$, then $2^{j'-1} \leq weight(R_j^{(\ell)}) < m2^j$ and $m2^{j-1} \leq weight(R_j^{(\ell)}) < 2^{j'}$. The result follows by taking logarithms. □

Lemma 2 *For $\ell \geq 2$, if the degree of range $R_j^{(\ell)}$ is $m \geq 2$, then one of its children has degree at least $2^{m-1}+1$; moreover, the number of $R_j^{(\ell)}$'s grandchildren is at least $2^m + m - 1$.*

Proof: Figure 5 demonstrates the relations between a degree-m node and its children and grandchildren. Let the children of $R_j^{(\ell)}$ be $R_{j_1}^{(\ell-1)}$, $R_{j_2}^{(\ell-1)}$, ..., $R_{j_m}^{(\ell-1)}$, for $j > j_1 > j_2 > \cdots > j_m$. By Lemma 1 and the fact that $j_i \leq j-i$, it follows that $R_{j_i}^{(\ell-1)}$ has at least $2^{j-j_i-1}+1 \geq 2^{i-1}+1$ children. The total number of grandchildren of $R_j^{(\ell)}$ is thus at least $\sum_{1\leq i\leq m}(2^{i-1}+1) = 2^m + m - 1$. □

Lemma 3 *For $\ell \geq k \geq 3$, if the degree of range $R_j^{(\ell)}$ is $m \geq 2$, then the difference in range numbers between the smallest-numbered range on level $\ell - k$ and the smallest-numbered range on level $\ell - k + 1$ among the descendants of $R_j^{(\ell)}$ is at least*

$$2^{2^{\cdot^{\cdot^{2^m}}}}\Big\}k + 1, \tag{1}$$

which is $2^{2^m}+1$ and $2^{2^{2^m}}+1$ for $k = 3$ and $k = 4$, respectively. In addition, the number of descendants of $R_j^{(\ell)}$ on level $\ell - k$ is at least (1).

Proof: By induction on k. We shall demonstrate that the inductive hypothesis is true for either $k = 2$ or $k = 3$. Let us assume that the inductive hypothesis does not hold for the smaller value $k = 2$. Range $R_j^{(\ell)}$'s m children at level $\ell - 1$ occupy contiguous ranges $R_{j_1}^{(\ell-1)}$, $R_{j_2}^{(\ell-1)}$, ..., $R_{j_m}^{(\ell-1)}$, where $j_i = j - i$; otherwise, $j_m \leq j - m - 1$ and the number of $R_{j_m}^{(\ell-1)}$'s children on level $\ell - 2$ is at least $2^m + 1$, by Lemma 1, in which case the inductive hypothesis holds for $k = 2$.

Now suppose that the inductive hypothesis does not hold for the value $k = 3$. There are exactly $2^m + m - 1$ grandchildren of range $R_j^{(\ell)}$ on level $\ell - 2$ occupying contiguous ranges $R_{j-2}^{(\ell-2)}$, $R_{j-3}^{(\ell-2)}$, ..., $R_{j-m-2^m}^{(\ell-2)}$; otherwise by Lemma 1, the number of children of the smallest-numbered range on level $\ell - 2$ is at least $2^{2^m} + 1$, in which case the base case holds for $k = 3$.

The number of ranges on level $\ell - 3$ can be minimized if the ranges on level $\ell - 2$ are ordered by the numbers of their parents on level $\ell - 1$, so we assume that such an ordering occurs. The $2^{i-1} + 1$ children of $R_{j-i}^{(\ell-1)}$ on level $\ell - 2$ occupy contiguous ranges $R_{j-2^{i-1}-i}^{(\ell-2)}, \ldots, R_{j-2^i-i}^{(\ell-2)}$. By Lemma 1, the number of $R_{j-i}^{(\ell-1)}$'s grandchildren on level $\ell - 3$ is at least

$$\left(2^{2^{i-1}-1} + 1\right) + \cdots + \left(2^{2^i-1} + 1\right) = 2^{2^i} - 2^{2^{i-1}-1} + 2^{i-1} + 1.$$

Hence, the number of $R_j^{(\ell)}$'s great grandchildren on level $\ell - 3$ is at least

$$\sum_{1 \le i \le m} \left(2^{2^i} - 2^{2^{i-1}-1} + 2^{i-1} + 1\right) = 2^{2^m} + \frac{1}{2} \sum_{1 \le i < m} 2^{2^i} + 2^m + m - 2. \qquad (2)$$

The number of the smallest-numbered range on level $\ell - 3$ among the great grandchildren of $R_j^{(\ell)}$ is thus at most

$$(j-2) - \left(2^{2^m} + \frac{1}{2} \sum_{1 \le i < m} 2^{2^i} + 2^m + m - 2\right) = j - m - 2^m - 2^{2^m} - \frac{1}{2} \sum_{1 \le i < m} 2^{2^i}.$$

The resulting difference between the smallest range number on level $\ell - 3$ and the smallest range number $j - m - 2^m$ on level $\ell - 2$ among the descendants of $R_j^{(\ell)}$ is at least

$$2^{2^m} + \frac{1}{2} \sum_{1 \le i < m} 2^{2^i} \ge 2^{2^m} + 2.$$

The inductive hypothesis therefore holds for the base case $k = 3$.

For the inductive step, for $k \ge 2$, suppose that the difference in range numbers between the smallest-numbered range on level $\ell - k$ and the smallest-numbered range on level $\ell - k + 1$ among the descendants of $R_j^{(\ell)}$ is at least

$$\left. 2^{2^{\cdot^{\cdot^{2^m}}}} \right\}k + 1.$$

By Lemma 1, the smallest-numbered range on level $\ell - k$ has at least

$$\left. 2^{2^{\cdot^{\cdot^{2^m}}}} \right\}k+1 + 1$$

children, and the inductive hypothesis holds for $k + 1$. □

Each range in the topmost level must be a root and can have degree 1, but all its descendants must have degree ≥ 2. Let us choose ℓ to be one less than the topmost level number; the degree of each non-root range in level ℓ is therefore ≥ 2. Since there are only N elements in the data structure, Lemma 3 implies the following bound on the height of the data structure:

Theorem 3 *The maximum number of levels L in the trees is $\le \log^* N + 1$, where N is the number of elements.*

The space requirement of the algorithm depends on the number of ranges actually put into the table.

Lemma 4 *The total number of nonempty ranges is $O(N)$, where N is the number of elements, and the total storage space used by the data structure is $O(N)$.*

Proof: Each tree constructed by the algorithm is height-balanced. With the exception of root ranges, every range in the trees has degree at least 2. This means that the total number of nodes in each height-balanced tree is of the same order as the number of the leaves of the tree, which is N. The dynamic hash tables used to store the ranges for each level occupy $O(N)$ space collectively. $\square$

The universal hashing schemes of Section B.3 can be bypassed in favor of simple table lookup at the cost of a super-linear bound on storage space.

3 Second Algorithm

As in the algorithm of Section 2, we regard the N elements as "zeroth-level" elements, and we use a similar partitioning. The zeroth-level elements are partitioned by weight into ranges R_j, such that R_j is associated with the range $[2^{j-1}, 2^j)$; we again denote by $weight(R_j)$ the total weight of elements in the range R_j. We can treat the range R_j as a new "first-level" element with weight $weight(R_j) \in [2^{j'-1}, 2^{j'})$, for some $j' \geq j$. However, before putting the ranges R_j into the next-level ranges we partition them into *intervals*. We consider the $\log N$ sized integer intervals $I_t = [t\lceil \log N \rceil, \ldots, (t+1)\lceil \log N \rceil - 1]$, for each integer t, and we assign each range R_j to the interval I_{t_j} that contains j. We continue constructing the data structure within each interval *separately*. For each range R_j in I_{t_j}, we have

$$2^{t_j \lceil \log N \rceil} \leq weight(R_j) \leq N \cdot 2^{(t_j+1)\lceil \log N \rceil - 1}.$$

We normalize the weights of all ranges in interval I_t by dividing their value by $2^{t\lceil \log N \rceil}$; that is, the weight of each range R_j is normalized to be $weight'(R_j) = weight(R_j) \cdot 2^{-t_j \lceil \log N \rceil}$. We now have

$$1 \leq weight'(R_j) \leq N \cdot 2^{\lceil \log N \rceil - 1} < N^2.$$

Each element therefore belongs to some range R_j which belongs to an interval I_{t_j} and has a normalized weight $weight'(R_j)$. The total weight of ranges in an interval I_t is denoted as the interval's weight $weight(I_t)$. The weight of each non-empty interval is kept as part of the data structure.

For each interval I_t that contains at least two ranges, we can treat its ranges R_j, where $j \in [t\lceil \log N \rceil, \ldots, (t+1)\lceil \log N \rceil - 1]$, as new "first-level" elements with weights $weight'(R_j)$ and construct the data structure recursively. In the recursive data structure, the ranges of elements in the ℓth level are the elements of the $(\ell+1)$st level.

Lemma 5 *The recursive data structure is of size $O(N)$ and of depth at most $\log^* N$.*

Proof: Each element in the $(\ell+1)$st recursive level contains (as a range) at least two elements from the ℓth recursive level. Therefore, the number of elements in the $(\ell+1)$st recursive level is at most half the number of elements in the ℓth level, which implies a total of at most $2N$ elements at all levels of the data structure. The size of the data structure is clearly linear in the number of elements it contains at all levels.

The number of ranges within an interval in the first level is at most $\log N$. By induction, the number of elements in an interval in the ℓth level is at most $\log^{(\ell)} N$; therefore, each interval in the $(\log^* N)$th level contains at most one element. $\square$

The data structure is stored in a similar manner as the data structure of the first algorithm, using dynamic hashing.

We will use only a portion of the data structure for the generation procedure. More specifically, in the recursive data structure at most four intervals are used; these intervals are called the "significant" intervals. Let I_r be the rightmost non-empty interval, and I_{r-1} and I_{r-2} be the two smaller (possibly empty) intervals next to it. The four significant intervals include I_r, I_{r-1}, and I_{r-2}; the exact definition will be given later. The significant intervals are so named because of the following property:

Lemma 6 *The total weight of the non-significant intervals is at most a $1/N$ fraction of the total weight of the significant intervals.*

Proof: Consider a range R_j that belongs to a non-significant interval I_{t_j}. We have $t_j < r-2$ and thus $j < r\lceil \log N\rceil - 2\lceil \log N\rceil$. The total weight of the non-significant intervals is therefore at most

$$N \cdot 2^{(r-2)\lceil \log N\rceil} \leq \frac{1}{N} \cdot 2^{r\lceil \log N\rceil}$$

while the total weight of the significant intervals is at least $2^{r\lceil \log N\rceil}$. $\square$

The generation of a random variate according to the current distribution of weights w_1, w_2, $\ldots$, w_N is done as follows:

Step 1. We choose between the significant intervals and the non-significant intervals with the appropriate probabilities.

Step 2. If the non-significant intervals are chosen, then we generate a random variate from the elements in the non-significant intervals by applying any linear time algorithm (e.g., [26]), and halt.

Step 3. If the significant intervals are chosen, we choose one of them, say, I_t, with the appropriate probability, and proceed to the next step.

Step 4. Within I_t, we choose a range R_j according to the weight distribution of the ranges by applying the generation procedure recursively.

Step 5. Within R_j, we use the rejection method (see, e.g., Appendix B.1) to choose one of the elements according to their weight distribution.

As mentioned above, the generation algorithm uses only the recursive data structures that are constructed on the significant intervals. The dynamic nature of the problem may cause a non-empty interval to become empty at some future point, thereby causing a non-significant interval to become significant. The complete construction guarantees that when this happens, the appropriate data structure is available.

Step 1 takes constant expected time, given the total weight of the significant intervals. By Lemma 6, the contribution of Step 2 to the expected generation time is $O(1)$. We will show in Lemma 7 that the significant intervals can be found in Step 3 in constant expected time. The rejection method in Step 5 takes constant expected time. Let $G(N)$ be the expected generation time. Steps 1–3 and 5 take constant expected time and Step 4 takes $G(\log N)$ expected time. Thus, $G(N)$ can be expressed by the relation $G(N) = G(\log N) + O(1)$, implying $G(N) = O(\log^* N)$.

There is one issue still to be resolved, namely, to justify the assumption that the significant intervals can be found in Step 3 in constant expected time. It turns out that keeping the sum of weights is sufficient to find the rightmost non-empty interval I_r and thereby the significant interval. This summation trick is also used in Step 2 of Section 2 and in Section 8 and is based on the following observation:

Lemma 7 *We have*

$$r\lceil \log N \rceil \le \log \sum_{1 \le i \le N} w_i < (r+2)\lceil \log N \rceil .$$

Proof: Let j_1 be the maximum j so that R_j is a non-empty range. It is easy to verify that $2^{j_1} \le \sum_{1 \le i \le N} w_i \le N \cdot 2^{j_1}$. By taking logarithms we have $j_1 \le \lceil \log \sum_{1 \le i \le N} w_i \rceil \le j_1 + \lceil \log N \rceil$. Since $r\lceil \log N \rceil \le j_1 \le (r+1)\lceil \log N \rceil - 1$ the result follows. $\square$

Let $t_1 = \lfloor \log \sum_{1 \le i \le N} w_i / \lceil \log N \rceil \rfloor$. By Lemma 7 either $t_1 = r$ or $t_1 = r+1$. We define the *significant intervals* as $SI = \{I_t : t = t_1+1, t_1, t_1-1, t_1-2\}$. The intervals I_r, I_{r-1} and I_{r-2} are in SI, as required. (Note that the forth significant interval may be either I_{r-3} or I_{r+1}.) The summation trick lets us find SI in constant expected time, by only keeping track of the sum $\sum_{1 \le i \le N} w_i$, and by computing t_1.

On-line update of the weight of any element is similar to what is done in the first algorithm. An updated weight w_i of an element in range R_j requires the updates of $weight(R_j)$, $weight'(R_j)$, $weight(I_{t_j})$, and the total weight $\sum_{1 \le i \le N} w_i$. If an update moves an element from one range to a different range, then it implies two updates in the next recursive level, implying a total of $O(2^{\log^* N})$ updates. This gives us the following theorem:

Theorem 4 *The expected cost for generating a random variate according to the current weights is $O(\log^* N)$, where N is the number of elements. Updating the weight of any element can be done in $O(2^{\log^* N})$ expected time in the worst case.*

4 Modification to Achieve $O(\log^* N)$ Update Time

In this section we show how to modify our basic algorithms in order to achieve the desired $O(\log^* N)$ expected update time when amortized over the sequence of updates. That is, if there are t updates, for any $t \geq 0$, the expected time to complete all t updates is $O(t \log^* N)$. In contrast, the expected update time for the basic algorithms derived in Sections 2 and 3 is $\Theta(2^{\log^* N})$ in the worst case. The approach can be generalized to reduce the amortized expected update time from $O(\log^* N)$ to $O(1)$ at the expense of increasing the expected generation time from $O(\log^* N)$ to $O(a^{\log^* N})$, for some constant $a > 2$.

For simplicity, we restrict our attention to the first algorithm, from Section 2; the techniques can be adapted equally well to the second algorithm, of Section 3.

The key to achieving this better amortized bound is by considering the following parameters:

1. We introduce "tolerance" into the ranges to allow "lazy updating." We choose a tolerance factor $0 \leq b < 1$. For convenience, we choose b so that $\frac{2+b}{1-b}$ is power of 2. (Previously we used $b = 0$.) We relax the range of weights that can be stored in the range $R_j^{(\ell)}$ associated with the interval $[2^{j-1}, 2^j)$ by tolerating weights in the interval $[(1-b)2^{j-1}, (2+b)2^{j-1})$. We associate range $R_j^{(\ell)}$ with the tolerated interval $[(1-b)2^{j-1}, (2+b)2^{j-1})$. Note that the resulting set of tolerated ranges overlap. However, when an element with weight w is inserted into a level-ℓ range, it is inserted into the unique range $R_j^{(\ell)}$ where $2^{j-1} \leq w < 2^j$. The element must change its weight by at least the tolerance $b2^{j-1}$ of range $R_j^{(\ell)}$ before it is moved to another range.

2. We modify the criteria defining roots and require that each non-root node have degree at least $d = \frac{1}{2}(\frac{2+b}{1-b})^2 2^c$, where c is a nonnegative integer to be specified later. (Previously we used $d = 2$.) The number d is the minimally allowable number of buckets in a non-root range; from the graph-theoretic viewpoint, it is the minimal degree of the non-root nodes in the trees we build.

4.1 Properties of the Modified Data Structure

In this more general setting, we must modify Lemmas 1–3 and Theorem 3 in order to take into account the tolerance b and degree bound d. In this section we derive new versions, which we call Lemmas 1′–3′ and Theorem 3′. Using a larger value of d slightly decreases the worst-case bound on the number L of levels from that of Theorem 3. For example, if we take $b = 0.4$ and $c \geq 1$, Theorem 3′ shows that the maximum height L of the trees is $\leq \log^* N - 1$.

For conciseness, we refer to the expanded ranges in the modified algorithm simply as ranges; they have tolerance factor $0 < b < 1$ and all ranges except the roots have degree at least $d = \frac{1}{2}(\frac{2+b}{1-b})^2 2^c$, for nonnegative integer c. With these modifications, Lemma 1 takes the following form:

Lemma 1′ *If the degree of range $R_j^{(\ell)}$ is $m \geq d$, then weight$(R_j^{(\ell)})$ is in the range $R_{j'}^{(\ell+1)}$, where* $\lg m - \lg(\frac{2+b}{1-b}) < j' - j < \lg m + \lg(\frac{2+b}{1-b})$.

Proof: Each of the m children of $R_j^{(\ell)}$ has weight in the range $[(1-b)2^{j-1}, (2+b)2^{j-1})$, so $weight(R_j^{(\ell)})$ must be in the range $[m(1-b)2^{j-1}, m(2+b)2^{j-1})$. If $weight(R_j^{(\ell)})$ falls into $[(1-b)2^{j'-1}, (2+b)2^{j'-1})$, then $(1-b)2^{j'-1} \leq weight(R_j^{(\ell)}) < m(2+b)2^{j-1}$ and $m(1-b)2^{j-1} \leq weight(R_j^{(\ell)}) < (2+b)2^{j'-1}$. The inequality follows by taking logarithms. □

We can use Lemma 1′ to get the following modification of Lemma 2:

Lemma 2′ *For $\ell \geq 2$, if the degree of range $R_j^{(\ell)}$ is $m \geq d$, then one of its children has degree at least 2^{m-1+c}; moreover, the number of $R_j^{(\ell)}$'s grandchildren is at least $2^{m+c} - 2^c + m$.*

Proof: Let the children of a range $R_j^{(\ell)}$ be $R_{j_1}^{(\ell-1)}, R_{j_2}^{(\ell-1)}, \ldots, R_{j_m}^{(\ell-1)}$, for $j > j_1 > j_2 > \cdots > j_m$. By Lemma 1′, we have $j_1 \leq j - \lg(\frac{2+b}{1-b}) - c$, $j_i \leq j - i + 1 - \lg(\frac{2+b}{1-b}) - c$, and the number of children of $R_{j_i}^{(\ell-1)}$ is at least $\max\{d, 2^{i-1+c}+1\}$. Thus, the total number of grandchildren of $R_j^{(\ell)}$ is $\geq \sum_{1\leq i\leq m}(2^{i-1+c}+1) = 2^{m+c} - 2^c + m$. □

Lemma 3′ *For $\ell \geq k \geq 3$, if the degree of range $R_j^{(\ell)}$ is $m \geq d$, then the difference in range numbers between the smallest-numbered range on level $\ell - k$ and the smallest-numbered range on level $\ell - k + 1$ among the descendants of $R_j^{(\ell)}$ is at least*

$$\left.2^{2^{\cdot^{\cdot^{\cdot^{2^m}}}}}\right\}k + \lg\left(\frac{2+b}{1-b}\right) + 1.$$

In addition, the number of descendants of $R_j^{(\ell)}$ on level $\ell - k$ is at least

$$\left.2^{2^{\cdot^{\cdot^{\cdot^{2^m}}}}}\right\}k.$$

Proof: The full proof is similar to that of Lemma 3, except that the minimum difference of range numbers between a parent node and its largest-numbered child is $c + \lg(\frac{2+b}{1-b})$ rather than 1. This enlarges the differences between the smallest-numbered ranges on adjacent levels and introduces the term $\lg(\frac{2+b}{1-b})$. The details are suppressed for brevity. □

Lemma 3′ can be strengthened substantially, but it suffices for our purposes. As before, we choose ℓ to be one below the topmost level number; the degree of each non-root range in level ℓ is $\geq d$. Let us suppose that $d \geq 16 = 2^{2^2}$. Since there are only N elements in the data structure, Lemma 3′ implies the following improved bound on the height of the data structure (cf. Theorem 3):

Theorem 3′ *The maximum number of levels L of the trees is $\leq \log^* N - 1$, where N is the number of elements.*

4.2 Amortized Analysis of the Modified Algorithm

When a node w is made a child of range $R_j^{(\ell)}$ represented by node x, node w must later change its weight by at least x's tolerance $b2^{j-1}$ in order for it to "change its parent." This tolerance prevents too many insertions and deletions from occurring. When w changes its parent, x loses weight and w's new parent gains weight; two paths of nodes need to be updated: the one upward from node x and the one upward from w's new parent. All the nodes on the two paths should revise their weights to reflect the changes.

To facilitate the amortized analysis, we use an accounting method [22], where we charge C_ℓ units of cost to a level-ℓ node w that changes its parent. Since we only change the weights of one of the N bottom-level elements on level 0, and in the worst case the element will change its parent, we charge C_0 to each dynamic weight update operation. The credits accumulated at each node must pay for the cost of a parent change for that node, when it occurs, plus the cost of processing the resulting two upward update paths.

Suppose that node w changes its parent from x_1 to y_1 during an update. The update path starting from w is defined to be $w \to x_1 \to x_2 \to \cdots \to x_m$, where x_m is a root, and we call this path the *old ancestor path* of w. The *new ancestor path* of w is $w \to y_1 \to y_2 \to \cdots \to y_n$, where y_n is a root.

Let us consider for reasons of brevity only the case in which w is decremented in weight by Δ and changes its parent from x_1 to y_1, and we restrict ourselves to the analysis of the old ancestor path $w \to x_1 \to x_2 \to \cdots \to x_m$. Node w is on level ℓ, and node x_j is on level $\ell + j$. Let node x_j correspond to the range $R^{(\ell)}_{i_j+1}$, for $1 \le j \le m$.

Suppose that the nodes x_1, x_2, $\ldots$ x_{j-1} do not change their parents or become roots as a result of the parent change of w. The change of weight of node x_j due to the update of w is $\mathit{weight}(w) \le (2+b)2^{i_1}$. Let us define $\underline{\delta}(x_j, x_{j+1}) = \mathit{weight}(x_j) - (1-b)2^{i_j}$ to be the difference between the weight of x_j and the lower boundary of the range $R^{(\ell)}_{i_{j+1}+1}$ represented by x_{j+1} at the time when x_j was last inserted into one of x_{j+1}'s buckets (or, equivalently, when x_j changed its parent to x_{j+1}). We have $\underline{\delta}(x_j, x_{j+1}) \ge b2^{i_{j+1}}$. By Lemma 1$'$, we have $2^{i_{j+1}} \ge 2^{i_1}((\frac{2+b}{1-b})2^c)^j$, which gives us $\underline{\delta}(x_j, x_{j+1}) \ge b((\frac{2+b}{1-b})2^c)^j 2^{i_1}$. Therefore, the ratio f_j between x_j's weight change and the tolerated weight change $\underline{\delta}(x_j, x_{j+1})$ satisfies

$$f_j \le \frac{(2+b)2^{i_1}}{b\Big((\frac{2+b}{1-b})2^c\Big)^j 2^{i_1}} = \left(\frac{2}{b}+1\right)\left(\left(\frac{2+b}{1-b}\right)2^c\right)^{-j}.$$

Since the weight change of x_j is at most f_j of the total weight change needed to cause a parent change, it suffices to deposit $f_j C_{\ell+j}$ credits on node x_j during the processing of w's parent change.

Next let us consider the case in which nodes x_1, x_2, $\ldots$ x_{j-1} do not change their parents, but nodes x_1, x_2, $\ldots$ x_k become roots, for $k \le j-1$, as a result of the parent change of w. Nodes x_1, x_2, $\ldots$ x_k do not need credits deposited on them, since they no longer have parents, and the credits can be deferred instead to x_{k+1}, $\ldots$. By similar reasoning to above, the ratio f_j between x_j's weight change and the tolerated weight change $\underline{\delta}(x_j, x_{j+1})$ satisfies

$$f_j \le \left(\frac{2}{b}+1\right)\left(\left(\frac{2+b}{1-b}\right)2^c\right)^{-j+k},$$

and it suffices to deposit $f_j C_{\ell+j}$ credits on node x_j during the processing of w's parent change.

The number of credits deposited on node x_j is at least $C_{\ell+j}$ times the fraction of the tolerance represented by x_j's weight change. Thus, at the future time when the weight of node x_j is out of the range of node x_{j+1} and x_j changes parent, there will be at least $C_{\ell+j}$ credits on x_j to pay for the required updating.

The other cases to consider, such as consideration of the new update path and the case in which w is incremented in weight, are analogous to the ones discussed above and are left to the reader. This gives us the following lemma:

Lemma 4 *The total number of credits allocated to a level-ℓ node between two times it changes parent is at least C_ℓ.*

By the above reasoning, we get the following recurrence on the number of credits C_ℓ needed to perform a parent change of a node on level ℓ:

$$\begin{aligned} C_\ell &\leq 2(L-\ell+1) + 2 \sum_{1\leq j\leq L-\ell} \frac{\left(\frac{2}{b}+1\right)}{\left((\frac{2+b}{1-b})2^c\right)^j} C_{\ell+j} \\ &\leq 2(L-\ell+1) + \frac{2\left(\frac{2}{b}+1\right)}{(\frac{2+b}{1-b})2^c - 1} C_{\ell+1} \qquad (3) \end{aligned}$$

where $C_L = 1$. The first term on the right-hand side corresponds to the minimum cost needed to process the two update paths of length $\leq L-\ell+1$ caused by the parent change. The jth term in the summation represents the credits needed for the two level-$(\ell+j)$ nodes on the old ancestor path and the new ancestor path. If $2(\frac{2}{b}+1) < (\frac{2+b}{1-b})2^c - 1$, the solution to (3) is $C_\ell = O(L-\ell)$.

Lemma 5 *If $c > \lg((\frac{2}{b}+1)(1-b))$, then $C_\ell = O(L-\ell)$, where $L \leq \log^* N - 1$ is the number of levels in the trees.*

We can choose the constants b and c (and thus d) so that the conditions of Theorem $3'$ and Lemma 5 are satisfied. For example, we can choose $b = 0.4$ and $d = 32$. The number of credits we need to allocate for the update of an element's weight is thus $C_0 = O(L) = O(\log^* N)$. This gives us our main result:

Theorem 4 *The amortized expected cost for each update operation is $O(\log^* N)$, where N is the number of input elements.*

With the modification discussed above, the time to implement Steps 1–3 for generating a random variate increases by a multiplicative factor of $1/b$ (because of the effect on the rejection method in Step 3) and an additive factor of $\log d$ (because of the effect on the the weights of the roots in the level table in Step 2). Since $1/b$ and d can be chosen to be to be reasonably small constants, the resulting increase in generation time is not much. A beneficial effect of the modification, which we mentioned above, is that the worst-case bound on the number of levels L decreases slightly as d gets larger. In practice, we can probably avoid this modification and keep $b = 0$ and $d = 2$, or else use a partially modified algorithm with a larger d, but for theoretical and worst-case purposes, the full modification is needed in order to get the $O(\log^* N)$ time bound for generation and update.

4.3 Tradeoffs between Update and Generation

We can make the expected amortized update time $O(k)$, $k \geq 1$, by not propagating weight information above the kth level. We instead assign the "approximate weight" degree$(R_j^{(\ell)}) \times (2+b)2^{j-1}$ to range $R_j^{(\ell)}$. We modify Step 3 so that whenever a rejection test fails at level $\ell \geq k$, we restart the entire process with Step 1. The resulting random variate is generated with the correct distribution, but in a backtracking manner, which results in an exponential increase in generation time. Concerning update time, there is no propagation of weight information. The only extra time needed is for changing parents, which happens at most a constant number of times per update, in the amortized sense, because of the use of tolerance and the minimum degree bound.

Theorem 5 *The amortized expected cost for update can be reduced to $O(k)$, $1 \leq k \leq \log^* N$, at the cost of $O(k + a^{\log^* N - k})$ expected generation time, for some constant $a > 2$.*

5 Expected Constant-Time Updates and Generation

A simple lookup table technique for dynamic random variate generation was recently developed by Hagerup, Mehlhorn, and Munro [13]. Their use of the technique provides a constant-time algorithm for generation and update, but only when the weights are integral and bounded by a polynomial in N. In this section we show how to use table lookup with our algorithms to do generation and updates in constant expected time, without any restrictions on the weights. We first give a brief description of the lookup table technique and then show how to incorporate it with our algorithms.

A simple approach for the basic generation problem, already used in [26], is based on maintaining an array of prefix sums W_i of the weights w_i; that is, $W_i = \sum_{1 \leq j \leq i} w_j$, for each $1 \leq i \leq N$. A random variate is generated by first selecting uniformly at random a number $r \in [0, W_N]$, and then choosing $i = i(r)$ such that $W_{i-1} < r \leq W_i$. It is easy to verify that if the weights are non-negative integers then r can be restricted to be an integer (within the same range). When both N and W_N are sufficiently small, the outcomes for all possible values of r can be precomputed and stored in a lookup table. Subsequently, for a given r the appropriate index $i = i(r)$ can be found in constant time. The size of the lookup table, as well as the time it takes to precompute it, are $O(W_N)$. If the weights w_i are integers from the range $[1, m]$, then $W_N \leq mN$.

To handle updates, we need to precompute a lookup table for each possible set of weights. Since each of the N weights can have m possible values, there are at most m^N lookup tables to precompute. Each lookup table is of the type described above, and in addition it stores for each possible update a pointer to the lookup table that corresponds to the updated set of weights. There are mN possible updates, since each update involves selecting one of the N weights and changing its value to one of at most m possible values. Hence, the extra pointers increase the size of each lookup table by only a constant factor, to $O(mN)$. The total space S required for storing all the lookup tables is therefore

$$S = O(Nm^{N+1}), \tag{4}$$

and it takes $O(S)$ time to construct them. The full details of the construction appear in [13].

We apply the table lookup technique as follows. The idea is to use only two levels of recursion of the data structure of Section 3. After two levels of recursion we are left with subproblems consisting of $O(\log\log N)$ ranges R_j, and we have $1 \leq \mathit{weight}(R_j) \leq m$, for $m = (\log N)^{O(1)}$. The weight of each range at this level of recursion is rounded to the next larger integer. Let us refer to subproblems with these parameters as *compact*.

We precompute lookup tables for all possible compact problems. By replacing N by $\log\log N$ and substituting $m = (\log N)^{O(1)}$ in (4), we find that the total space for the lookup tables of the compact subproblems is $S = (\log N)^{O(\log\log N)}$, which is $o(N)$. Thus, storage space remains linear, and precomputation time is $o(N)$.

The generation of a random variate is done as in Section 3, except for the following modification. The data structures for the intervals I_t after two levels of recursion correspond to compact subproblems and are replaced by a pointer to the appropriate lookup table. Whenever we execute Step 4 after two levels of recursion, we generate the range R_j in constant time by lookup in the table for I_t. (Because the range weights are rounded after two levels of recursion, we must use the rejection method to determine whether to actually generate the range; acceptance occurs with probability at least 1/2.) It is easy to verify that the generation takes expected constant time. Inserts and deletes to the I_t data structures can be done by updating pointers in constant time. We have proved the following theorem:

Theorem 6 *The operations of update and generation can be done in expected constant time and linear space, with no restriction made on the input weights.*

6 Dictionary Issues

In the algorithms described so far we have indicated that we use dynamic hashing for time and space efficiency, without elaborating. To be more specific, we use a dictionary data structure that supports the operations of insert, delete, and lookup. We use dictionary algorithms that support each of these operations in constant time, with high probability [6, 4].

Because of the varying sizes of the weights, we may have to reinitialize dictionaries from time to time when we need to insert a weight that is too large and does not belong to the universe handled by the dictionary. We can do the reinitialization, assuming constant-time access to weights, by maintaining a linked list of dictionaries in the order of increasing universe size $U_1, U_2, \ldots, U_t$. Insertions are always made into U_t for the current value of t, and when the universe size of U_t is not large enough for an insertion, we set $t := t + 1$ and append an empty dictionary with a larger universe. This data structure supports constant-time operations, since lookup is not actually required in our application; we have a direct pointer, when needed, to the location of the ith element in the data structure. The only purpose of the dictionary is to limit the total storage required to be linear.

The dictionary algorithms quoted above are based on polynomials over a finite field F_p where p is a prime. This imposes a problem of finding a new prime that is sufficiently large when the universe size increases. To get around it, we will reduce the universe size of each element to $O(K^3)$ first, where K is a tentative upper bound on the length of the future update sequence, and then use a dictionary over F_p, where $p = O(K^3)$ is independent of the universe size.

To reduce the universe, it is sufficient to use a 2-universal hash function [2], which will enable injective universe reduction with high probability. We must find a family of classes of 2-universal hash functions that are easy to compute without *a priori* knowledge about the universe size. Dietzfelbinger *et al.* [5] recently developed such a scheme that allows hash functions to be selected in constant time. Using their scheme, we do not need to have any *a priori* knowledge about the universe size, and we still obtain constant-time algorithms.

7 Parallel Algorithms

In this section we parallelize the previous algorithms in order to perform m updates or m generations in parallel.

The generation procedures do not change the data structure. Therefore, m generations can be done in parallel, if concurrent read is allowed. In the updating procedures, the effect of having several updates in parallel is that several processors may want to update the weight of the same element or range in parallel. Note that even if all parallel updates are assumed to be for distinct elements, at higher levels in the data structure we may have concurrent updates for the same range. In such case, we need to update the range by the sum of these updates. This can be done in constant time on a Fetch&Add PRAM [11].[2] This model is a powerful and non-standard model of CRCW PRAM. However, each step of an m-processor Fetch&Add PRAM can be simulated on standard CRCW models (e.g., on Arbitrary, Priority, Collision, or Tolerant) in $O(\log m / \log\log m)$ time, $O(m)$ space, and $O(m)$ operations, with high probability [9]. As in the sequential case, the memory is managed through a dictionary algorithm: we use a parallel dictionary algorithm which with linear space supports each instruction in $O(\log^* m)$ time and $O(m)$ operations, with high probability [18, 10].

Theorem 7 *The expected cost for generating m random variates according to the current weights is $O(\log^* N)$, using m processors on a* CRCW PRAM*, where N is the number of elements. Updating the weight of m elements can be done in $O(\log^* N)$ amortized expected time and in $O(2^{\log^* N})$ expected time in the worst case, using m processors on a Fetch&Add* PRAM*. It can be done with a slow-down of $t = O(\log m / \log\log m)$ on a (standard)* CRCW PRAM *with m/t processors (optimal speedup).*

8 ϵ-Heap

In this section we show how to apply our techniques to obtain an efficient dynamic algorithm for maintaining approximate priority queues. Given an arbitrary $\epsilon > 0$, we construct an ϵ-heap, so that each query returns an element whose value is within an ϵ relative factor of the current maximal element value. In particular, if the maximal element has value x, the ϵ-heap returns an ϵ-maximum, whose value is in the range $[(1-\epsilon)x, x]$.

Our heap data structure is related to the data structure of Section 3. For consistency we denote the value of an element as its weight. The input elements are partitioned by weight into ranges R_j, such that R_j is associated with the range $[(1+\epsilon)^{j-1}, (1+\epsilon)^j)$. Note that all elements in a range are within an ϵ relative factor from each other. The elements of each range are kept in an arbitrary data

[2] In this model, if two or more processors attempt to write to the same cell in a given step, then their values are added to the value already written in the shared memory location and all prefix sums obtained in the (virtual) serial process are recorded.

structure (e.g., a list, an array). To find an ϵ-maximum, it suffices to find the maximal non-empty range. Then, an arbitrary element from the range can be taken to be an ϵ-maximum.

Each range R_j is now represented by the weight $w(R_j) = j$. The ranges are partitioned into integer intervals of size $\lfloor \log N / \log(1+\epsilon) \rfloor = O((1/\epsilon) \log N)$. We consider the integer interval $I_t = [t\lfloor \log_{1+\epsilon} N \rfloor, \ldots, (t+1)\lfloor \log_{1+\epsilon} N - 1]$, for each integer t, and we assign each range R_j to the interval I_{t_j} that contains j. The weights of each range R_j in interval I_t are now normalized to $\tilde{w}(R_j) = j - t\lfloor \log_{1+\epsilon} N \rfloor \in [1, \ldots, \lfloor \log_{1+\epsilon} N \rfloor]$. For each interval, we keep a separate priority queue of van Emde Boas *et al.* [23]. In addition to the data structure described above, we keep record of $\sum (1+\epsilon)^j$, where the summation is over the non-empty ranges R_j.

To implement an update operation for an element of weight w_i, we first compute its range R_j, by $j = \lceil \log_{1+\epsilon} w_i \rceil$. The data structure for the elements of R_j is then updated. Now the interval I_{t_j} is computed by $t_j = j$ div $\lfloor \log_{1+\epsilon} N \rfloor$, and the update is done in the priority queue of the interval I_{t_j}.

To find the maximal non-empty range (and thereby an ϵ-maximum) we first find the maximal non-empty interval I_r, and then use the priority queue of I_r to find the maximal range in I_r. To find the maximal interval we use a summation trick similar to the one used in Section 3, based on the following lemma:

Lemma 6 *We have*

$$r\lceil \log_{1+\epsilon} N \rceil \leq \log_{1+\epsilon} \sum (1+\epsilon)^j < (r+2)\lceil \log_{1+\epsilon} N \rceil.$$

Proof: Let j_1 be the maximum j so that R_j is a non-empty range. It is easy to verify that $(1+\epsilon)^{j_1} \leq \sum (1+\epsilon)^j < N \cdot (1+\epsilon)^{j_1}$. By taking logarithms we have $j_1 \leq \log_{1+\epsilon} \sum (1+\epsilon)^j < j_1 + \lceil \log_{1+\epsilon} N \rceil$. Since $r\lceil \log_{1+\epsilon} N \rceil \leq j_1 \leq (r+1)\lceil \log_{1+\epsilon} N \rceil - 1$ the result follows. $\square$

Let $t_1 = \lfloor \log_{1+\epsilon} \sum (1+\epsilon)^j / \lceil \log_{1+\epsilon} N \rceil \rfloor$. By Lemma 6 either $t_1 = r$ or $t_1 = r+1$. In this analysis we must relax our computational model to allow truncated logarithms to an arbitrary base, such as $1+\epsilon$, to be done in constant time; this issue is discussed further in Section 9. The only non-constant time operations are therefore the operations on the priority queues on each interval, which take $O(\log \log \lfloor \log_{1+\epsilon} N \rfloor) = O(\log \log (\frac{1}{\epsilon} \log N))$ time. For $\epsilon = n^{-\mathrm{polylog}(n)}$ we get $O(\log \log n)$ time. For $\epsilon = 1/\mathrm{polylog}(n)$ we get $O(\log \log \log n)$ time. Implementation in linear space can be done using dynamic hashing, as for the generation algorithms.

9 Conclusions

We have presented two practical and efficient randomized algorithms for generating a random variate according to a set of weights that can vary dynamically. For simplicity our algorithms are expressed for the case in which the range of the random variate is the set $S = \{1, 2, \ldots, N\}$ for some N, but a simple modification allows S to be any dynamically varying set of cardinality N.

The two algorithms of Sections 2 and 3 use tree-based data structures of height $O(\log^* N)$. In each case, the expected time to generate the random variate is $O(\log^* N)$, and the expected time to update a weight value is $O(2^{\log^* N})$. We have shown in Section 4 how to modify the algorithms by introducing the notion of tolerance and by requiring each non-root node in the trees to have degree at least d, for some large enough d, in order to improve the expected update time from $O(2^{\log^* N})$ to $O(\log^* N)$. The expectations in each algorithm are over the randomness in the algorithms; we make no assumptions about the weight updates.

We have shown in Section 5 how to improve the running time to expected constant time by using the elegant lookup table technique developed by Hagerup, Mehlhorn, and Munro [13]. The table lookup method was used in [13] to get an expected constant-time algorithm for the case when the weights are integers and bounded by a polynomial of N. Our use of table lookup, however, removes all assumptions about the weights. The variance of the running times of our algorithms can also be made to be $o(1)$ so as to get good tail bounds.

Our constant-time algorithm has been applied in [16] to the universal prediction techniques developed in [16]; the resulting prediction algorithm runs in constant expected time. In that application, prediction is done by generating a random variate in which the weights are exponential quantities of the form $w_i = (f_i)^r$, where f_i is an integer frequency and r can be regarded as an fixed integer, both of size $O(N)$. The generation and updates can be done in constant time even when arithmetic operations must be done on finite-precision $O(\log N)$-sized arguments. Element i's weight w_i is approximated from above by $2^{\lceil r \lg f_i \rceil}$, and the first level of the algorithm in Section 3 is applied to these approximated weights, using finite-precision arithmetic on the exponents. The resulting subproblems have polynomially sized weights, and the construction continues as in Section 5. Because of the initial approximation by a power of 2, if element i is selected for generation, a final acceptance-rejection test must be done before actually generating element i; in the test, element i is accepted with probability $(f_i)^r / 2^{\lceil r \lg f_i \rceil} \geq 1/2$. That test can be done in constant expected time using finite precision by generating an exponentially distributed random variate [16].

All our algorithms are implemented in linear space, by using dynamic hashing algorithms. In the course of this application we were led to consider the difficulty of having varying universe, and as a result defined the abstract dictionary problem of supporting the operations of insert, delete, and lookup for the case in which there is no *a priori* known bound on the universe size. The difficulty is how to find quickly the hashing parameters needed for the dynamic hashing. We have assumed that standard operations take constant time on arguments proportional to the maximum weight encountered so far. In our applications for dynamically generating random variates, a simpler version of the dictionary problem arose in Section 6, in which lookup operations are not required by the data structure, and we have an expected constant-time solution, using a new class of hash functions of Dietzfelbinger *et al.* [5]; the main purpose of the dictionary is merely to obtain linear storage space.

The basic algorithms we have developed may be preferable to the modified algorithms for normal use in practice, especially if there are *a priori* upper and lower bounds on the weights, and if the dynamic hashing technique is removed in favor of simple table lookup. However, it may be better to use degree bound $d > 2$ because of its effect on lessening the height of the data structure. Experimentation is needed.

In Section 7 we have considered the problem of generating random variates in parallel and have given parallel algorithms with optimal processor-time product. In particular, a batch of m generations can be processed on an m-processor CRCW PRAM in $O(\log^* m)$ expected time, or in constant expected time using the improved algorithms. A batch of m updates can be processed in $O(\log m \log^* m / \log\log m)$ expected time and $O(m)$ expected number of operations on a CRCW PRAM.

In Section 8 we have presented the ϵ-heap data structure—an approximating alternative to the classic heap—that uses $O(\log\log\log n)$ time per operation for $\epsilon = 1/\text{polylog}(n)$. Recently [20] several improvements were obtained, including the presentation of other ϵ-data structures with operations such as ϵ-findmin and ϵ-successor, and an algorithm that maintains an ϵ-heap in $O(1)$ time per operation, for $\epsilon = 1/\text{polylog}(n)$, and whose use of truncated logarithms is restricted to the reasonable class of binary logarithms.

Acknowledgments

We would like to thank Kurt Mehlhorn for helpful comments and Albert Greenberg and Sanguthevar Rajasekaran for bringing the problem to our attention and for helpful discussions.

References

[1] P. Bratley and B. L. Fox and L. E. Schrage, *A Guide to Simulation.* Springer-Verlag, Second Edition, 1987.

[2] J. L. Carter and M. N. Wegman. Universal Classes of Hash Functions, *Journal of Computer and System Sciences*, 18: 143–154, April 1979.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* McGraw-Hill, New York, NY, 1990.

[4] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial Hash Functions are Reliable. *Proceedings of the 19th Annual International Colloquium on Automata, Languages, and Programming*, Springer LNCS 623, 235–246, 1992.

[5] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A Reliable Randomized Algorithm for the Closest-Pair Problem. Manuscript, Nov. 1992.

[6] M. Dietzfelbinger and F. Meyer auf der Heide. A New Universal Class of Hash Functions and Dynamic Hashing in Real Time, *Proceedings of the 17th Annual International Colloquium on Automata, Languages, and Programming*, Springer LNCS 443: 6–19, July 1990.

[7] B. L. Fox. Simulated Annealing: Folklore, Facts, and Directions, *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing (H. Niederreiter and P.J.-S. Shiue, eds.)*, Lecture Notes in Statistics. Springer-Verlag, 1995.

[8] M. L. Fredman and D. E. Willard. Trans-Dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths. *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, 719–725, 1990.

[9] J. Gil and Y. Matias. Fast and Efficient Simulations among CRCW PRAMs. *J. of Parallel and Distributed Computing*, 23(2):135–148, 1994.

[10] J. Gil, Y. Matias, and U. Vishkin. Towards a Theory of Nearly Constant Time Parallel Algorithms. *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, 698–710, October 1991.

[11] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAulife, L. Rudolph, and M. Snir. The NYU Ultracomputer—Designing an MIMD Shared Memory Parallel Machine. *IEEE Trans. on Comp*, C-32:175–189, 1983.

[12] A. Greenberg and J. S. Vitter. Constant-Time Generation of Dynamic Random Variates, Notes, June 1990.

[13] T. Hagerup, K. Mehlhorn, and I. Munro. Optimal Algorithms for Generating Time-Varying Discrete Random Variates, *Proceedings of the 20th Annual International Colloquium on Automata, Languages, and Programming*, Springer LNCS 700: 253–264, July 1993.

[14] D. E. Knuth. *The Art of Computer Programming*, Volume 2: *Seminumerical Algorithms*. Addison Wesley, Reading, MA, 1981.

[15] D. E. Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*. Addison Wesley, Reading, MA, 1973.

[16] P. Krishnan and J. S. Vitter. Optimal Prediction for Prefetching in the Worst Case, *Proceedings of the Fifth Annual SIAM-ACM Symposium on Discrete Algorithms*, 392–401, January 1994.

[17] Y. Matias. Rolling a Dice with Varying Biases. Manuscript, July 1992.

[18] Y. Matias and U. Vishkin. Converting High Probability into Nearly-Constant Time—with Applications to Parallel Hashing. *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, 307–316, 1991.

[19] Y. Matias, J. Vitter, and W. C. Ni. Dynamic Generation of Discrete Random Variates, *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, Austin, TX, January 1993, 361–370.

[20] Y. Matias, J. Vitter, and N. Young. Approximate Data Structures with Applications. *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, Alexandria, VA, January 1994, 187–194.

[21] S. Rajasekaran and K. W. Ross. Fast Algorithms for Generating Discrete Random Variates with Changing Distributions, *ACM Transactions on Modeling and Computer Simulation*, 3(1):1–19, 1993.

[22] R. E. Tarjan. Amortized Computational Complexity, *SIAM Journal on Algebraic and Discrete Methods*, 6(2): 306–318, 1985.

[23] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and Implementation of an Efficient Priority Queue. *Math. Systems Theory*, 10:99–127, 1977.

[24] J. S. Vitter and W.-C. Ni. Dynamic Generation of Discrete Random Variates, Brown University Technical Report CS–92–36, August 1992.

[25] A. J. Walker. New Fast Method for Generating Discrete Random Numbers with Arbitrary Distributions, *Electronic Letters*, 10(8):127–128, 1974.

[26] C. K. Wong and M. C. Easton. An Efficient Method for Weighted Sampling without Replacement, *SIAM Journal on Computing*, 9(1):111–114, 1980.

Appendix

In the Appendix we give describe the preprocessing algorithm and give some background on the rejection method, table doubling, and universal hashing, which are used by our algorithm.

A Preprocessing

In this section we give a detailed description of the preprocessing stage for the first algorithm, described in Section 2., which is used if some of the N weights are initially nonzero. Algorithm *preprocess* partitions the elements into ranges $R_j^{(1)} = [2^{j-1}, 2^j)$, for integer j, and calls algorithm *construct_level* to build the trees from the first level constructed. We use a list of queues to coordinate the events like insertions and deletions. The function *find_range*(i, ℓ) is used to search for the range $R_i^{(\ell)}$ in the level-ℓ hash table organized using universal hashing, as mentioned in Section B.3. If it does not exist, we create one and make $R_i^{(\ell)}$ an empty range. Only level-ℓ ranges containing at least one element are created and put into a level-ℓ hash table. We prove later that the number of ranges created during the execution of the algorithm is $O(N)$. The algorithm *insert_bucket*(*source*, *destination*) is used to insert a range or element called *source* into the range defined by *destination*. It also updates the current total weight in the range *destination*. Since we use an array of buckets in each range to hold the children of the range, generation of one bucket in the range is done by indexing into the array. The insertion and deletion of buckets can be handled by table doubling techniques mentioned in Section B.2. The use of queues Q_ℓ is to avoid the searching of nonempty ranges on the next level.

algorithm *preprocess*;
input weights $w_1, w_2, \ldots, w_N$;
begin
$Q_1 := \emptyset$;
for $i := 1$ **to** N **do**
  begin
  $j := \lfloor \lg w_i \rfloor + 1$; { $w_i \in [2^{j-1}, 2^j)$ }
  $R_j^{(1)} :=$ *find_range*$(j, 1)$;
  insert_bucket$(i, R_j^{(1)})$;
  if $R_j^{(1)} \notin Q_1$ **then** *insert_queue*$(R_j^{(1)}, Q_1)$
  end;
construct_level(1)
end;

We construct a level structure recursively until there are only one-element ranges left. The level weight $weight(\mathcal{T}_\ell)$ is the summation of weights of the root ranges on level ℓ. The method of algorithm *construct_level* is basically the same as that of algorithm *preprocess*. We use the queue Q_ℓ passed from the previous level ℓ to construct the new level $\ell+1$. For any range $R_i^{(\ell)}$ in Q_ℓ containing more than one element, we insert $R_i^{(\ell)}$ into the appropriate range $R_j^{(\ell+1)}$ on level $\ell + 1$ by calling *insert_bucket*$(R_i^{(\ell)}, R_j^{(\ell+1)})$, which also deletes $R_i^{(\ell)}$ from the level table $\mathcal{T}_\ell$. For any range in Q_ℓ that has only one element left, we put it into the level table $\mathcal{T}_\ell$. We also maintain a variable *roots*$(\mathcal{T}_\ell)$ whose bit positions indicate the existence of these root ranges. For example, if the range $R_i^{(\ell)}$ is a root, we just add 2^i to *roots*$(\mathcal{T}_\ell)$. The procedures *insert_queue* and *delete_queue* are trivial to implement such that the cost per call is constant.

```
algorithm construct_level(ℓ)
begin
weight(T_ℓ) := 0;
roots(T_ℓ) := 0;
Q_{ℓ+1} := ∅;
more_than_one := false;
while Q_ℓ ≠ ∅ do
    begin
    R_i^(ℓ) := delete_queue(Q_ℓ);
    w_i^* := weight(R_i^(ℓ));
    if there are more than one element in R_i^(ℓ) then
        begin
        Let j be the integer such that w_i^* ∈ [2^{j-1}, 2^j);
        R_j^(ℓ+1) := find_range(j, ℓ+1);
        if R_j^(ℓ+1) ∉ Q_{ℓ+1} then insert_queue(R_j^(ℓ+1), Q_{ℓ+1});
        insert_bucket(R_i^(ℓ), R_j^(ℓ+1));
        delete_range(R_i^(ℓ));
        more_than_one := true
        end
    else begin
        weight(T_ℓ) := weight(T_ℓ) + w_i^*;
        roots(T_ℓ) := roots(T_ℓ) + 2^i
        end
    end;
if more_than_one then construct_level(ℓ+1)
end;
```

After we construct each level, the total weight of each range is known. Moreover, those ranges containing more than one element will be deleted from the current level; the remaining elements in the table $\mathcal{T}_\ell$ should be the roots of the trees rooted at that level.

Theorem 8 *The preprocessing requires $O(N)$ expected time.*

Proof: We put each range into a queue when it needs to be inserted into the level table $\mathcal{T}_\ell$. When we process ranges on level ℓ, we just pick the elements from the queue and insert them in constant time using dynamic hashing. So the cost is proportional to the number of nonempty ranges on the level, rather than the number of entries on each level. □

In the next section we show that the resulting trees share a common property that they are very "shallow."

B Three Background Techniques

To make the paper self-contained, we review three important techniques whose ideas come into play in our algorithm: the rejection method, table doubling, and dynamic hashing.

B.1 Rejection Method.

The rejectioni method is described in, e.g., [14]. If we want to generate a random variate X with density $f(t)$, we can find another density function $g(t)$ such that $f(t) \leq cg(t)$ for all t, where c is a constant. The function g is selected so that it is relatively easy to compute $g(t)$ and to generate a random variate with density $g(t)$, and the selected constant c is small. The algorithm works as follows:

algorithm *rejection_method*
begin
repeat
  Generate uniform random number $U \in [0, 1)$;
  Generate X according to density $g(t)$
until $U < f(X)/cg(X)$;
return(X)
end;

Proposition 1 *The expected number of iterations to generate X by the rejection method shown above is c.*

We specialize the algorithm to handle the case in which $f(t)$ corresponds to discrete weights $w_1, w_2, \ldots, w_n$, where $1/2 \leq f(i) = w_i \leq 1$ and $cg(i) = 1$, for all $1 \leq i \leq n$. The probability of generating value j equals $w_j / \sum_{1 \leq i \leq n} w_i$.

algorithm *bucket_rejection*(T)
begin
repeat
  Generate uniform random number $U \in [0, 1)$;
  $I = \lfloor Un \rfloor$
until $Un - I < w[I + 1]$;
return$(I + 1)$
end;

Figure 6 gives a graphical view of the rejection method. First we randomly select the table entry and then randomly select a real number between 0 and 1. If the selected number lies in the shaded area, we mark it a "hit"; otherwise, we repeat the process.

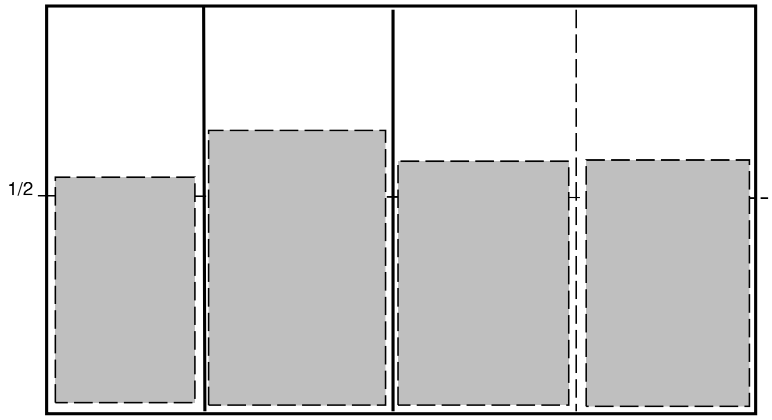


Figure 6: Rejection Method

Corollary 1 *The expected number of iterations in algorithm bucket_rejection is* 2.

B.2 Table Doubling Technique.

A comprehensive treatment of table doubling can be found in [3]. Suppose we want to implement a dynamic table that supports insertion and deletion. In order to use the power of the random-access model, the table is implemented as an array. The size of the table cannot be determined in advance, so dynamic allocation and deallocation of the array is necessary. A trivial algorithm allocates an $(n+1)$-element array when an element is inserted into an n-element array, but this causes worst-case update cost proportional to the size of the array. Since the number of elements in the table is not necessarily the same as the size of the table, let us use α to denote the *load factor* of the table, or its fraction of occupancy. Initially, the table T has size zero. The size of the empty table T becomes 1 when we insert an element into it. Inserting an element into a nonempty table T results in two cases:

1. If $\alpha < 1$, we just insert the new element into one of the free slots.

2. If $\alpha = 1$, the table is full, and we expand the size of the table to twice its original size.

Deleting an element from the table is handled in an analogous way, except that we do not contract the table until $\alpha < 1/4$. The cost for either table expansion or contraction is linear in the size of the table, but the amortized cost for each insertion or deletion is constant.

Proposition 2 *A sequence of m insertion and deletion operations on a dynamic table using the table-doubling method requires $O(m)$ time.*

This algorithm can be modified to run in constant time per operation in the worst case, as follows: In addition to the current table of size n, we also maintain two tables T^+ of size $2n$ and T^- of size $n/2$. If the table T overflows because of insertions, we just reassign T^+ to T, make T the new T^-, and deallocate the old T^-. The new T^+ is initially empty, but is filled up twice as fast as T, so that if T overflows again, T^+ is once again consistent. Deletion is handled in an analogous way.

B.3 Dynamic Hashing

Any single hash function chosen can encounter some bad worst-case inputs that cause linear-time rather than constant-time performance. The remedy devised by Carter and Wegman [2] is to choose a hash function randomly from a good collection H of hash functions and get constant expected performance independent of any particular input sequence.

Let $H = \{h_1, h_2, \ldots, h_m\}$ be a set of hash functions; each h_i is a mapping from $\{0, \ldots, n-1\}$ to $\{0, \ldots, m-1\}$. We say that H is *c-universal* if for every pair of inputs $x \neq y$ in $\{1, 2, \ldots, n-1\}$ the total number of $h \in H$ such that $h(x) = h(y)$ is no more than $c \cdot |H|/m$; that is, only a fraction of c/m of the hash functions in H cause a collision on any pair of inputs.

Proposition 3 *Let H be a c-universal class of hash functions, the expected cost of an insert, delete, or access operation is $O(1 + c\alpha)$, where α is the load factor of the table.*

We can use the c-universal class of hash functions

$$H = \{h_{a,b} \mid h_{a,b}(x) = ((ax + b) \bmod n) \bmod m,\ a, b \in \{0, \ldots, n-1\}\},$$

where $(\lceil n/m \rceil / (n/m))^2 = O(1)$. When the number of elements changes dynamically, the table may have to be expanded or contracted from time to time, but the cost of the rebuilding can be amortized so that the operations still run in amortized constant expected time.

More complicated techniques for implementing the table lookup method in constant expected time are dynamic perfect hashing and its variants [4, 5, 6].