Dynamic Generation of Discrete Random Variates

Jeffrey Scott Vitter and Wen-Chun Ni

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-92-36
August 1992

Dynamic Generation of Discrete Random Variates

Jeffrey Scott Vitter[*] and *Wen-Chun Ni*[†]

Department of Computer Science
Brown University
Providence, R.I. 02912–1910

August 1992

Abstract

We present and analyze an efficient new algorithm for generating a random variate distributed according to a dynamically changing set of weights. The algorithm can generate the random variate and update a weight each in $O(\log^* N)$ expected time. (For all feasible values of N, we have $\log^* N \leq 5$.) The $O(\log^* N)$ expected update time is amortized; in the worst-case the expected update time is $O(2^{\log^* N})$. The algorithm is simple, practical, and easy to implement.

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

[*]Support was provided in part by a National Science Foundation Presidential Young Investigator Award with matching funds from IBM, by NSF research grant CCR–9007851, and by Army Research Office grant DAAL03–91–G–0035.

[†]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.

Contents

1 Introduction

The generation of random variates based on arbitrary finite, discrete distributions has long been a key component of many computer simulations [6], [10]. Given the elements 1, 2, ..., 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 $O(N)$ [6], [10].

In this paper we consider the problem in the important and more challenging dynamic case, in which the weights of the elements can vary 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 [11], 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 number in the range $[0, \sum_{1 \leq i \leq N} w_i)$.

Recently, Rajeskaran and Ross [8] and Greenberg and Vitter [5] developed different algorithms for the dynamic case that do generation and update in constant expected time for various restricted classes of updates. Independently to our work, Matias [7] developed an algorithm that does generation in $O(\log^* N)$ expected time and with $O(\log^* N)$ expected calls to a uniform random number generator, although general update requires $O(2^{\log^* N})$ expected time.[1]

In this paper, we introduce a very practical and more efficient randomized algorithm for the general dynamic case that does generation and update each in $O(\log^* N)$ expected time. Generation requires an average of $O(\log^* N)$ calls to a uniform $[0, 1)$ random number generator. The $O(\log^* N)$ expected update time is amortized; 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 worst-case expected update time for a single update is $O(2^{\log^* N})$, which is still reasonably small. The expectations are over the randomness in the algorithm; no assumptions are made about the weight updates. The implementation is especially simple and practical with very small constant factors implicit in the big-oh terms.

2 The General Ideas

In this section, we describe the basic idea of our algorithm; the analysis and more subtle aspects of it will be discussed in later sections. For completeness, we have

[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 $\log^* N \leq 5$.

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 $W = \sum_{1 \le i \le N} w_i$. For simplicity, we assume that each weight w_i can be stored in a single computer word; modifications otherwise are straightforward. The idea of our algorithm is to partition the 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 $weight(R_j^{(1)})$ 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 $weight(R_j^{(1)})$ 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 elements 1, 2, ..., 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 $\ell \ge 1$ 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 $\ell \ge 1$, 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.

Each level table $\mathcal{T}_\ell$, for $\ell \ge 1$, 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, as described in Section B.3 in the Appendix, indexed by j and ℓ. When we insert the level-ℓ roots into $\mathcal{T}_\ell$, we also compute the total weight of these roots, denoted $weight(\mathcal{T}_\ell)$. After the preprocessing, 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, ..., L. Figure 2 gives a view of the trees built.

The remaining question is how to store the children for a given internal range $R_j^{(\ell)}$. Within each range $R_j^{(\ell)}$, for $\ell \ge 1$, we keep a dynamic table of buckets. Insertions and deletions of ranges are handled by the table-doubling technique described

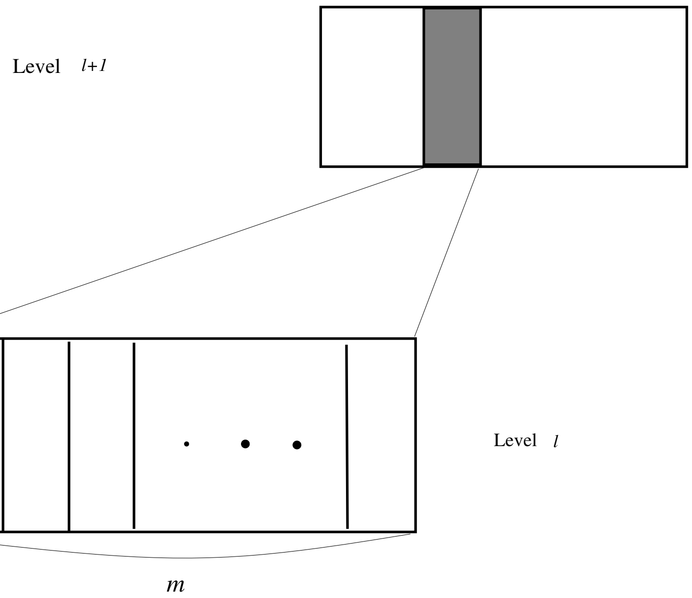


Figure 1: A range with degree m on level ℓ and its parent range on level $\ell + 1$.

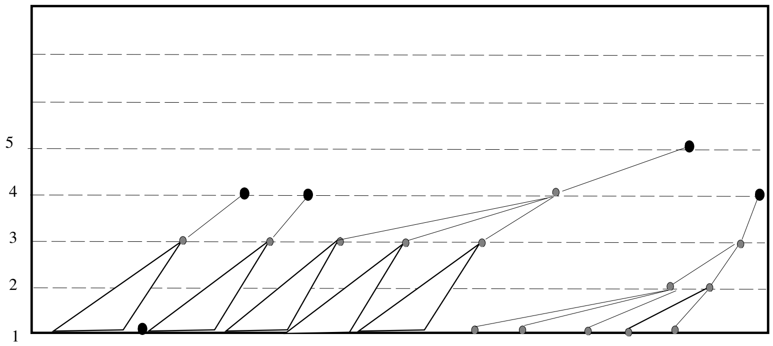


Figure 2: A forest of trees built, with $L = 5$ levels. The break lines mark the levels. Root nodes are denoted by solid circles.

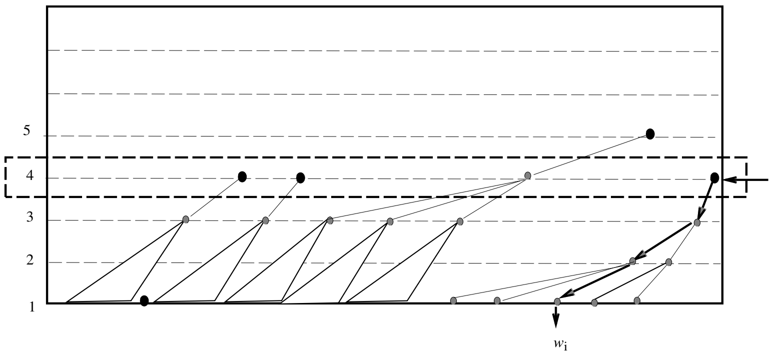


Figure 3: Generating a random variate from level $\ell = 4$.

in Section B.2 in the Appendix. Each bucket contains one range $R_i^{(\ell-1)}$ such that $weight(R_i^{(\ell-1)}) \in [2^{j-1}, 2^j)$. 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 $w_1, w_2, \ldots, w_N$ 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 Section B.1 in the Appendix) 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. We show in Sections 3 and 5 that the expected time to generate a random variate is $O(\log^* N)$.

On-line update of the weight of any element is permitted in the dynamic case. 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 $R_k^{(1)}$ 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. The total expected update time is $O(2^{\log^* N})$ in the worst case. Our main result, described later, yields an $O(\log^* N)$ amortized expected update time. The details and analysis will be given in Sections 4–6.

3 The Generation Method

Generating the random variate based on the current values of the weights consists of the three steps outlined in Section 2. 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 $U < \sum_{1 \le k \le \ell} weight(T_k)$. 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 is computed as $\lfloor \lg roots(\mathcal{T}_\ell) \rfloor + 1$. The successive indices j_2, j_3, …can be obtained by iteratively subtracting 2^{j_i} 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, as described below:

algorithm *generate_range*($R_j^{(\ell)}$);
begin;
$\ell' := \ell$;
while $\ell' > 0$ **do**
    begin
    $j :=$*bucket_rejection*($R_j^{(\ell')}$);
    $\ell' := \ell' - 1$
    end;
return(j)
end;

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). Only one call to a uniform random number generator is needed for Step 1. 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. The optimistic side is that its 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 E of Step 2 is $\sum_{1\le k\le n} k\cdot weight(R_{j_k}^{(\ell)})/weight(\mathcal{T}_\ell)$. Since $2^{j_k-1} \le weight(R_{j_k}^{(\ell)}) < 2^{j_k}$ and $weight(\mathcal{T}_\ell) \ge \sum_{1\le k\le n} 2^{j_k-1} \ge 2^{j_1-1}$, we have

$$E < \sum_{1\le k\le n} k\cdot 2^{j_k-j_1+1} \le \sum_{1\le k\le n} k2^{-k} < 2.$$

Step 2 requires only one call to a uniform random number generator. In Step 3 we walk down the levels from $R_j^{(\ell)}$ in constant expected time per level, by Corollary 1 in Section B.1, using a total of $O(\log^* N)$ expected time and $O(\log^* N)$ expected calls to a uniform random number generator.

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

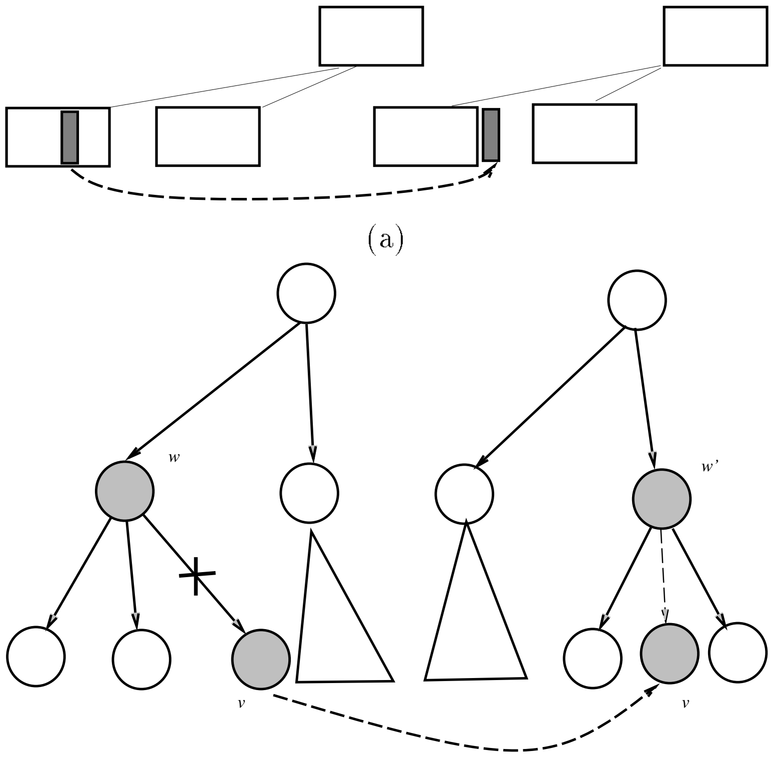
The dynamic scheme of Wong and Easton [11] uses $O(\log N)$ time per generation, but it requires only one call to a random number generator that outputs a uniform number in the range $[0, \sum_{1\le i\le N} w_i)$. Our algorithm uses an average of at most about $2L$ calls to a uniform random number generator, 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.

4 Updating the Weights—Basic Approach

On-line update of weights is permitted in the dynamic case. If we want to change weight w_i to $w_i + \Delta$, the structure of the hierarchy may have to be changed. If the new value of w_i is no longer be in its original range, we must move element i into another range.

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. Since we have at most $\log^* N + 1$ levels, to be shown in Theorem 3, the total number of ranges affected is bounded by $2^{\log^* N+1}$. By using the universal hashing



(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'.

scheme of Section B.3 and its variants, 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.*

The same result that we have described so far, namely, $O(\log^* N)$ expected time and expected calls to a uniform random number generator per generation, but an expected update time of $O(2^{\log^* N})$, was obtained independently by Matias [7] using another interesting technique. Matias inserts elements into ranges (as we do), but "elevates" to the next level all the "significant" ranges, defined as those nonempty ranges whose range numbers are within $4 \lg N$ of the largest range number. Significant ranges may have only one element in them. (In our algorithm, we elevate all ranges with at least two elements in them, although some may be insignificant.) This process is repeated recursively, with N reset to the number of elevated significant ranges; the height of the resulting data structure is $O(\log^* N)$. At each level, elements corresponding to non-significant ranges are generated in a brute-force way; elements corresponding to significant ranges are generated recursively using the constructed tree.

The update time in Matias's algorithm is $O(2^{\log^* N})$ and not $O(\log^* N)$, for a reason similar to the one described above: Changing the weight of a range may cause it in the next level of the data structure to be removed from one parent range and added to a different parent range, resulting in an exponential blowup in work from one level to the next. It is easy to construct sequences of updates where each update requires $\Omega(2^{\log^* N})$ time.

The maximum height of Matias's data structure is about the same as the maximum height of our data structure, but the topmost level of Matias's data structure can contain many arbitrarily weighted entries, perhaps 20 depending on the implementation, making the effective height larger. One interesting point is that for typical weight distributions, even for large N, our data structure has height 1 or 2; in fact it is difficult to specify in an informal way distributions for which the height is $\log^* N + 1$. Matias's data structure, on the other hand, can have maximum height for some common distributions. For example, if the weights are increasing powers of 2, our data structure has height 1 and Matias's data structure has maximum height.

The main result of this paper, which we cover in Section 6, is showing how to modify the basic algorithm described so far in order to reduce the update time from $O(2^{\log^* N})$ expected time to $O(\log^* N)$ amortized expected time.

5 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 $R_{j_1}^{(\ell-1)}, R_{j_2}^{(\ell-1)}, \ldots, R_{j_m}^{(\ell-1)}$, 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 $\sum_{1 \leq i \leq m} (2^{i-1} + 1) = 2^m + m - 1$. □

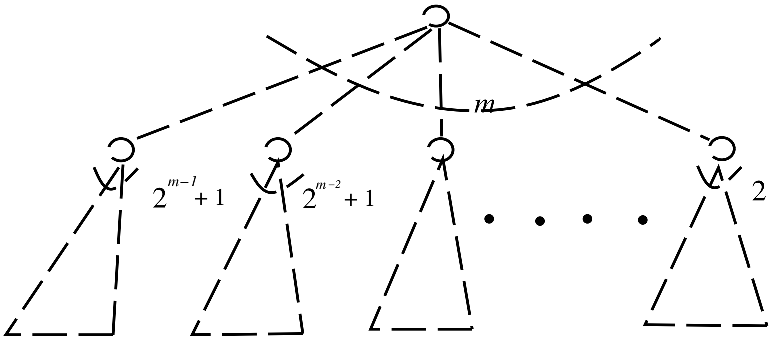


Figure 5: A typical tree built.

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*

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

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

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 $R_{j_1}^{(\ell-1)}, R_{j_2}^{(\ell-1)}, \ldots, R_{j_m}^{(\ell-1)}$, 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 $R_{j-2}^{(\ell-2)}, R_{j-3}^{(\ell-2)}, \ldots, R_{j-m-2^m}^{(\ell-2)}$; 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

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

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. □

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.

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

In this section we present our main result: how to modify our basic algorithm in order to achieve $O(\log^* N)$ 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 unmodified algorithm derived in Theorem 2 is $O(2^{\log^* N})$.

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.

6.1 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*

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

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

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

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.*

6.2 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 [9], 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 $w \rightarrow x_1 \rightarrow x_2 \rightarrow \cdots \rightarrow x_m$, where x_m is a root, and we call this path the *old ancestor path* of w. The *new ancestor path* of w is $w \rightarrow y_1 \rightarrow y_2 \rightarrow \cdots \rightarrow y_n$, 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 $w \rightarrow x_1 \rightarrow x_2 \rightarrow \cdots \rightarrow x_m$. Node w is on level ℓ, and node x_j is on level $\ell + j$. Let node x_j correspond to the range $R_{i_j+1}^{(\ell)}$, for $1 \leq j \leq m$.

Suppose that the nodes x_1, x_2, … 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 $weight(w) \leq (2+b)2^{i_1}$. Let us define $\underline{\delta}(x_j, x_{j+1}) = weight(x_j) - (1-b)2^{i_j}$ to be the difference between the weight of x_j and the lower boundary of the range $R_{i_{j+1}+1}^{(\ell)}$ 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 $\underline{\delta}(x_j, x_{j+1}) \geq b2^{i_{j+1}}$. By Lemma 1′, we have $2^{i_{j+1}} \geq 2^{i_1}((\frac{2+b}{1-b})2^c)^j$, which gives us $\underline{\delta}(x_j, x_{j+1}) \geq b((\frac{2+b}{1-b})2^c)^j 2^{i_1}$. Therefore, the ratio f_j between x_j's weight change and the tolerated weight change $\underline{\delta}(x_j, x_{j+1})$ satisfies

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

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:

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

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 $\le L-\ell+1$ 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.

7 Conclusions

We present practical and efficient randomized algorithms for generating a random variate according to a set of weights that can vary dynamically. In the first algorithm, the expected time to generate the random variate is $O(\log^* N)$, and the expected time to update a weight value is $O(2^{\log^* N})$. Our main result is showing in Section 6 how to modify the algorithm by introducing the notion of tolerance and by requiring each non-root range to contain at least d buckets, for some large enough d, in order to improve the expected update time from $O(2^{\log^* N})$ to $O(\log^* N)$, amortized over the sequence of updates. The expectations in each algorithm are over the randomness in the algorithms; we make no assumptions about the weight updates.

The first algorithm may be preferable to the modified algorithm 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.

The interesting algorithm developed independently by Matias [7] (see Section 4) is roughly comparable to our first algorithm: the maximum height of its data structure is about $\log^* N$, each generation uses on the average $O(\log^* N)$ time and $O(\log^* N)$ calls to a uniform $[0, 1)$ random number generator, and the expected update time is

$O(2^{\log^* N})$. We can show how to modify Matias's algorithm and improve the expected update time in the amortized setting to $O(\log^* N)$, as we did for our algorithm in Section 6, by incorporating our notion of tolerance and by requiring that the significant ranges must have degree $\geq d$, for some large enough d, in order to be promoted to the next higher level. This requires keeping level tables of roots, similar to the ones we use in our algorithm.

References

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

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

[3] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic Perfect Hashing: Upper and Lower Bounds, *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press, 524–531, October 1988.

[4] 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 International Colloquium on Automata, Languages, and Programming*, Springer-Verlag, Lecture Notes in Computer Science, 443: 6–19, July 1990.

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

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

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

[8] S. Rajasekaran and K. W. Ross. Fast Algorithms for Generating Discrete Random Variates with Changing Distributions, Manuscript, February 1992.

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

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

[11] 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, 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$(\mathcal{T}_\ell) := 0$;
roots$(\mathcal{T}_\ell) := 0$;
$Q_{\ell+1} := \emptyset$;
more_than_one := **false**;
while $Q_\ell \neq \emptyset$ **do**
    begin
    $R_i^{(\ell)} :=$ *delete_queue*(Q_ℓ);
    $w_i^* :=$ *weight*$(R_i^{(\ell)})$;
    if there are more than one element in $R_i^{(\ell)}$ **then**
        begin
        Let j be the integer such that $w_i^* \in [2^{j-1}, 2^j)$;
        $R_j^{(\ell+1)} :=$ *find_range*$(j, \ell + 1)$;
        if $R_j^{(\ell+1)} \notin Q_{\ell+1}$ **then** *insert_queue*$(R_j^{(\ell+1)}, Q_{\ell+1})$;
        insert_bucket$(R_i^{(\ell)}, R_j^{(\ell+1)})$;
        delete_range$(R_i^{(\ell)})$;
        more_than_one:=**true**
        end
    else begin
        weight$(\mathcal{T}_\ell) :=$ *weight*$(\mathcal{T}_\ell) + w_i^*$;
        roots$(\mathcal{T}_\ell) :=$ *roots*$(\mathcal{T}_\ell) + 2^i$
        end
    end;
if *more_than_one* **then** *construct_level*$(\ell + 1)$
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 5 *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 Important 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.

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;

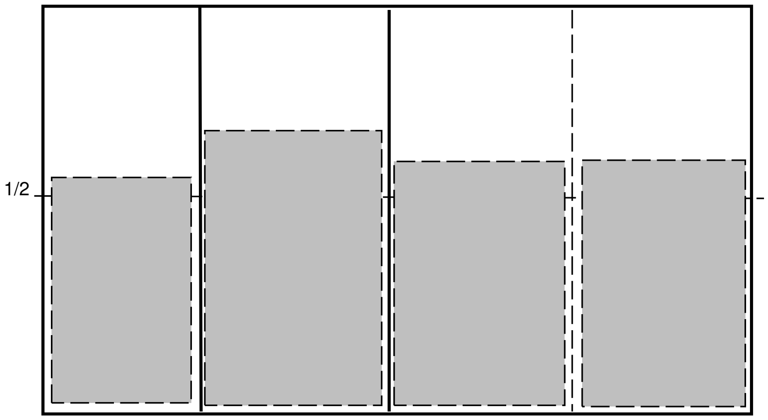


Figure 6: Rejection Method

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.

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 [2]. 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 [1] 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 $\left(\lceil n/m \rceil / (n/m)\right)^2 = O(1)$. 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, 3].