Optimal Static Range Reporting in One Dimension

Stephen Alstrup*
The IT University of Copenhagen
stephen@it-c.dk

Gerth Stølting Brodal†
BRICS‡
Dept. of Computer Science
University of Aarhus
gerth@brics.dk

Theis Rauhe*
The IT University of Copenhagen
theis@it-c.dk

ABSTRACT

We consider static one dimensional range searching problems. These problems are to build static data structures for an integer set $S \subseteq U$, where $U = \{0, 1, \ldots, 2^w - 1\}$, which support various queries for integer intervals of U. For the query of reporting all integers in S contained within a query interval, we present an optimal data structure with linear space cost and with query time linear in the number of integers reported. This result holds in the unit cost RAM model with word size w and a standard instruction set. We also present a linear space data structure for approximate range counting. A range counting query for an interval returns the number of integers in S contained within the interval. For any constant $\varepsilon > 0$, our range counting data structure returns in constant time an approximate answer which is within a factor of at most $1 + \varepsilon$ of the correct answer.

1. INTRODUCTION

Let S be a subset of the universe $U = \{0, 1, \ldots, 2^w - 1\}$ for some parameter w. We consider static data structures for storing the set S such that various types of range search queries can be answered for S. Our bounds are valid in the standard unit cost RAM with word size w and a standard instruction set. We present an optimal data structure for the fundamental problem of reporting all elements from S contained within a given query interval. We also provide a data structure that supports an approximate range counting query and show how this can be applied for multi-dimensional orthogonal range searching. In particular, we provide new results for the following query operations.

*Partially supported by a grant from The Danish Natural Science Research Council, grant no. 51-00-0390.

†Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT)

‡Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation.

STOC'01, July 6-8, 2001, Hersonissos, Crete, Greece.

$\mathsf{FindAny}(a, b),\ a, b \in U$: Report any element in $S \cap [a, b]$ or $\perp$ if there is no such element.

$\mathsf{Report}(a, b),\ a, b \in U$: Report all elements in $S \cap [a, b]$.

$\mathsf{Count}_\varepsilon(a, b),\ a, b \in U, \varepsilon \geq 0$: Return an integer k such that $|S \cap [a, b]| \leq k \leq (1 + \varepsilon)|S \cap [a, b]|$.

We let n denote the size of S and let $u = 2^w$ denote the size of universe U. Our main result is a static data structure with space cost $\mathrm{O}(n)$ that supports the query FindAny in constant time. As a corollary, the data structure allows Report in time $\mathrm{O}(k)$, where k is the number of elements to be reported.

Furthermore, we give linear space structures for the approximate range counting problem. We present a data structure that uses space $\mathrm{O}(n)$ and supports $\mathsf{Count}_\varepsilon$ in constant time, for any constant $\varepsilon > 0$.

The preprocessing time for the mentioned data structures is expected time $\mathrm{O}(n\sqrt{\log u})$.

1.1 Related work

Efficient static data structures for range searching have been studied intensively over the past 30 years, for surveys and books see e.g. [1, 18, 20]. In one dimension there has been much focus on the following two fundamental problems: the *membership problem* and the *predecessor problem*. These problems address the following queries respectively:

$\mathsf{Member}(a),\ a \in U$: Return yes if and only if $a \in S$.

$\mathsf{Pred}(a),\ a \in U$: Return the predecessor of a, i.e., $\max(S \cap [0, a])$ or $\perp$ if there are no such element.

The Member query is easily solved by FindAny, Report or $\mathsf{Count}_\varepsilon$ by restricting the query to unit size. On the other hand, it is straightforward to compute these three queries by at most two predecessor queries given an additional sorted (relative to U) list of the points S, where each point is associated its list rank.

An information theoretic lower bound implies that any data structure supporting any of the above queries, including Member, requires at least $\log \binom{u}{n}$ bits, i.e., has linear space cost in terms of $w = \log u$ bit words for $n \le u^{1-\Omega(1)}$. In [12], Fredman, Komlós and Szemeredi give an optimal solution for the static membership problem, which supports Member in constant time and with space cost $\mathrm{O}(n)$. In contrast, the predecessor problem does not permit a data structure with constant query time for a space cost bounded by $n^{\mathrm{O}(1)}$. This was first proved by Ajtai [3], and later Beame and Fich [8] improved Ajtai's lower bound and in addition gave a matching upper bound of

$$\mathrm{O}(\min(\log\log u/\log\log\log u, \sqrt{\log n/\log\log n}))$$

on the query time for space cost $\mathrm{O}(n^{1+\delta})$ for any constant $\delta > 0$. Beam and Fich's lower bound holds for *exact* counting queries, i.e., $\mathsf{Count}_\varepsilon$ where $\varepsilon = 0$. Our result shows that it is possible to circumvent this lower bound by allowing a slack in the precision of the result of the queries.

For data structures with *linear* space cost, Willard [24] provides a data structure with time $\mathrm{O}(\log\log u)$ for predecessor queries. Andersson and Thorup [7] show how to obtain a dynamic predecessor query with bounds $\mathrm{O}(\min(\log\log n \cdot \log\log u/\log\log\log u, \sqrt{\log n/\log\log n}))$. For linear space cost, these bounds were previously also the best known for the queries $\mathsf{FindAny}$, Report and $\mathsf{Count}_\varepsilon$. However, for superlinear space cost, Miltersen *et al.* [19] provide a data structure which achieves constant time for $\mathsf{FindAny}$ with space cost $\mathrm{O}(n \log u)$. Miltersen *et al.* also show that testing for emptiness of a rectangle in two dimensions is as hard as exact counting in one dimension. Hence, there is no hope of achieving constant query time for any of the above query variants including approximate range counting for two dimensions using space at most $n^{\mathrm{O}(1)}$.

1.2 Approximate data structures

Several papers discuss the approach of obtaining a speed-up of a data structure by allowing slack of precision in the answers. In [17], Matias *et al.* study an approximate variant of the dynamic predecessor problem, in which an answer to a predecessor query is allowed to be within a multiplicative or additive error relative to the correct universe position of the answer. They give several applications of this data structure. In particular, its use for prototypical algorithms, including Prim's minimum spanning tree algorithm and Dijkstra's shortest path algorithm. The papers [4] and [6] provide approximate data structures for other closely related problems, e.g., for nearest neighbor searching, dynamic indexed lists, and dynamic subset rank.

An important application of our approximate data structure is the static d-dimensional orthogonal range searching problem. The problem is given a set of points in U^d, to compute a query for the points lying in a d-dimensional box $R = [a_1, b_1] \times \cdots \times [a_d, b_d]$. Known data structures providing sublinear search time have space cost growing exponential with the dimension d. This is known as the "curse of dimensionality" [9]. Hence, for d of moderate size, a query is often most efficiently computed by a linear scan of the input. A straightforward optimization of this approach using space $\mathrm{O}(dn)$ is to keep the points sorted by each of the d coordinates. Then, for a given query, we can restrict the scan to the dimension i, where fewest points in S have the ith coordinate within the interval $[a_i, b_i]$. This approach leads to a time cost of $\mathrm{O}(dt(n) + \mathrm{opt})$ where opt is the number of points to be scanned and $t(n)$ is the time to compute a range counting query for a given dimension. Using the previous best data structures for the exact range counting problem, this approach has a time cost of $\mathrm{O}(d \min(\log\log u, \sqrt{\log n/\log\log n}) + \mathrm{opt})$. Using our data structure supporting $\mathsf{Count}_\varepsilon$ and $\mathsf{FindAny}$, we improve the time for this approach to optimal time $\mathrm{O}(d + (1+\varepsilon)\mathrm{opt}) = \mathrm{O}(d + \mathrm{opt})$ within the same space cost. A linear scan behaves well in computational models, which consider a memory hierarchy, see [2]. Hence, even for large values of opt, it is likely that the computation needed to determine the dimension for the scan majorizes the overall time cost.

1.3 Organization

The paper is organized as follows: In Section 2 we define our model of computation and the problems we consider, and state definitions and known results needed in our data structures. In Section 3 we describe our data structure for the range reporting problem, and in Section 4 we describe how to preprocess and build it. Finally, in Section 5 we describe how to extend the range reporting data structure to support approximate range counting queries.

2. PRELIMINARIES

A query $\mathsf{Report}(a, b)$ can be implemented by first querying $\mathsf{FindAny}(a, b)$. If an $x \in S \cap [a, b]$ is returned, we report the result of recursively applying $\mathsf{Report}(a, x-1)$, then x, and the result of recursively applying $\mathsf{Report}(x+1, b)$. Otherwise the empty set is returned. Code for the reduction is given in Figure 2. If k elements are returned, a straightforward induction shows that there are $2k+1$ recursive calls to Report, i.e. at most $2k+1$ calls to $\mathsf{FindAny}$, and we have therefore the following lemma.

LEMMA 1. *If* $\mathsf{FindAny}$ *is supported in time at most* t*, then* Report *can be supported in time* $\mathrm{O}(t \cdot k)$*, where* k *is the number of elements reported.*

The model of computation, we assume throughout this paper, is a unit cost RAM with word size w bits, where the set of instructions includes the standard boolean operations on words, the arbitrary shifting of words, and the multiplication of two words. We assume that the model has access to a sequence of truly random bits.

In our constructions we use the following definitions and results. Given two words x and y, we let $x \oplus y$ denote the binary exclusive-or of x and y. If x is a w bit word and i a nonnegative integer, we let $x \downarrow i$ and $x \uparrow i$ denote the rightmost w bits of the result of shifting x i bits to the right and i bits to the left respectively, i.e. $x \downarrow i = x \ \mathrm{div}\ 2^i$ and $x \uparrow i = (x \cdot 2^i) \bmod 2^w$. For a word x, we let $\mathrm{msb}(x)$ denote the most significant bit position in x that contains a one, i.e. $\mathrm{msb}(x) = \max\{i \mid 2^i \le x\}$ for $x \ne 0$. We define $\mathrm{msb}(0) = 0$. Fredman and Willard in [13] describe how to compute msb in constant time.

THEOREM 1 (FREDMAN AND WILLARD [13]). *Given a* w *bit word* x*, the index* $\mathrm{msb}(x)$ *can be computed in constant time, provided a constant number of words is known which only depend on the word size* w*.*

Essential to our range reporting data structure is the efficient and compact implementation of *sparse arrays*. We define a sparse array to be a static array where only a limited number of entries are initialized to contain specific values. All other entries may contain arbitrary information, and crucial for achieving the compact representation: It is not possible to distinguish initialized and not initialized entries. For the implementation of sparse arrays we will adopt the following definition and result about *perfect hash functions*, where $[n]$ denotes $\{0, 1, \ldots, n-1\}$.

DEFINITION 1. *A function $h : [m] \to [\ell]$ is* perfect *for a set $S \subseteq [m]$ if h is 1-1 on S. A family $\mathcal{H}$ is an (m, n, ℓ)-family of perfect hash functions, if for all subsets $S \subseteq [m]$ of size n there is a function $h \in \mathcal{H} : [m] \to [\ell]$ that is perfect for S.*

The question of representing efficiently families of perfect hash functions has been throughly studied. Schmidt and Siegel [21] described an $(m, n, \mathrm{O}(n))$-family of perfect hash functions where each hash function can be represented by $\Theta(n + \log\log m)$ bits. Jacobs and van Emde Boas [16] gave a simpler solution requiring $\mathrm{O}(n \log\log n + \log\log m)$ bits in the standard unit cost RAM model augmented with multiplicative arithmetic. Jacobs and van Emde Boas result suffices for our purposes. The construction in [16] makes repeated use of the data structure in [12] where some primes are assumed to be known. By replacing the applications of the data structures from [12] with applications of the data structure from [10], the randomized construction time in Theorem 2 follows immediately.

THEOREM 2 (JACOBS AND VAN EMDE BOAS [16]). *There is an $(m, n, \mathrm{O}(n))$-family of perfect hash functions $\mathcal{H}$ such that any hash function $h \in \mathcal{H}$ can be represented in $\Theta((n \log\log n)/w)$ words and evaluated in constant time for $m \le 2^w$. The perfect hash function can be constructed in expected time $\mathrm{O}(n)$.*

A sparse array A can be implemented using a perfect hash function as follows. Assume A has size m and contains n initialized entries each storing b bits of information. Using a perfect hash function h for the n initialized indices of A, we can store the n initialized entries of A in an array B of size n, such that $A[i] = B[h(i)]$ for each initialized entry $A[i]$. If $A[i]$ is not initialized, $B[h(i)]$ is an arbitrary of the n initialized entries (depending on the choice of h). From Theorem 2 we immediately have the following corollary.

COROLLARY 1. *A sparse array of size m with n initialized entries each containing b bits of information can with expected preprocessing time $\mathrm{O}(n)$ be stored using space $\mathrm{O}(n \cdot b/w)$ words, and lookups are supported in constant time, if $\log\log n \le b \le w$ and $m \le 2^w$.*

For the approximate range counting data structure in Section 5 we need the following result achieved by Fredman and Willard for storing small sets (in [14] denoted Q-heaps; these are actually dynamic data structures, but we only need their static properties). For a set S and an element x we define $\mathrm{rank}_S(x) = |\{y \in S \mid y \le x\}|$.

THEOREM 3 (FREDMAN AND WILLARD [14]). *Let S be a set of w bit words and an integer n, where $|S| \le (\log n)^{1/4}$ and $\log n \le w$. Using time $\mathrm{O}(|S|)$ and space $\mathrm{O}(|S|)$ words, a data structure can be constructed that supports $\mathrm{rank}_S(x)$ queries in constant time, given the availability of a table requiring space and preprocessing time $\mathrm{O}(n)$.*

The result of Theorem 3 can be extended to sets of size $(\log n)^c$ for any constant $c > 0$, by constructing a $(\log n)^{1/4}$-ary search tree of height $4c$ with the elements of S stored at the leaves together with their rank in S, and where internal nodes are represented by the data structures of Theorem 3. Top-down searches then take time proportional to the height of the tree.

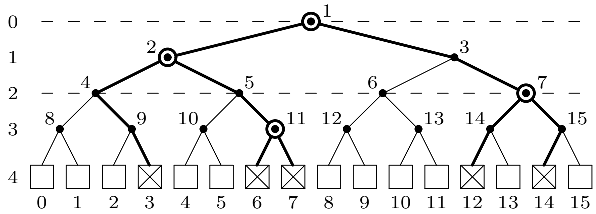
COROLLARY 2. *Let $c > 0$ be fixed constant and S a set of w bit words and an integer n, where $|S| \le (\log n)^c$ and $\log n \le w$. Using time $\mathrm{O}(|S|)$ and space $\mathrm{O}(|S|)$ words, a data structure can be constructed that supports predecessor queries in time $\mathrm{O}(c)$, given the availability of a table requiring space and preprocessing time $\mathrm{O}(n)$.*

3. DATA STRUCTURE

In this section we describe a data structure supporting the query $\mathsf{FindAny}(a, b)$ in constant time. The basic component of the data structure is (the implicitly representation of) a perfect binary tree T with 2^w leaves, i.e. a binary tree where all leaves have depth w (the root has depth zero). The leaves are numbered from left-to-right $\{0, 1, \ldots, 2^w - 1\}$, and the internal nodes of T are numbered $1, \ldots, n-1$. The root is the first node and the children of node v are nodes $2v$ and $2v+1$, i.e. like the numbering of nodes in an implicit binary heap [11, 25]. Figure 1 shows the numbering of the nodes for the case $w = 4$. The tree T has the following properties (see [15]):

FACT 1. *The depth of an internal node v is $\mathrm{msb}(v)$, and the d^{th} ancestor of v is $v \downarrow d$, for $0 \le d \le \mathrm{depth}(v)$. The parent of leaf a is the internal node $2^{w-1} + (a \downarrow 1)$, for $0 \le a < 2^w$. For $0 \le a < b < 2^w$, the nearest common ancestor of the leaves a and b is the $1{+}\mathrm{msb}(a \oplus b)^{\text{th}}$ ancestor of the leaves a and b.*

For a node v in T, we let $\mathsf{left}(v)$ and $\mathsf{right}(v)$ denote the left and right children of v, and we let T_v denote the subtree rooted at v and S_v denote the subset of S where $x \in S_v$ if, and only if, $x \in S$, and leaf x is a descendent of v. We let P be the subtree of T consisting of the union of the internal nodes on the paths from the root to the leaves in S, and we let V be the subset of P consisting of the root of T and the nodes where both children are in P. We denote V the set of *branching* nodes. Since each leaf-to-root path in T contains w internal nodes, we have $|P| \le n \cdot w$, and since V contains the root and the set of nodes of degree two in the subtree



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
B	1			0	1		1								
D	0	0	1	1	1		0				0			1	1

Figure 1: The binary tree T for the case w=4, $S = \{3, 6, 7, 12, 14\}$, and $H = 2$. The set S induces the sets $P = \{1, 2, 3, 4, 5, 7, 9, 11, 14, 15\}$ and $V = \{1, 2, 7, 11\}$, and the two sparse arrays B and D.

defined by P, we have $|V| = n - 1$, if both children of the root are in P and otherwise $|V| = n$.

To answer a query $\mathsf{FindAny}(a, b)$, the basic idea is to compute the nearest common ancestor v of the nodes a and b in constant time. If $S \cap [a, b] \neq \emptyset$, then either $\max S_{\mathsf{left}(v)}$ or $\min S_{\mathsf{right}(v)}$ is contained in $[a, b]$, since $[a, b]$ is contained within the interval spanned by v, and a and b are spanned by the left and right child of v respectively. Otherwise whatever computation we do cannot identify an integer in $S \cap [a, b]$. At most nw nodes satisfy $S_v \neq \emptyset$. E.g. to compute $\mathsf{FindAny}(8, 13)$, we have $v = 3$, $\max S_{\mathsf{left}(v)} = \max S_6 = \perp$, and $\min S_{\mathsf{right}(v)} = \min S_7 = 12$. By storing these nodes in a sparse array together with $\min S_v$ and $\max S_v$, we obtain a data structure using space $\mathrm{O}(nw)$ words, which supports $\mathsf{FindAny}$ in constant time. In the following we describe how to reduce the space usage of this approach to $\mathrm{O}(n)$ words.

We consider the tree T as partitioned into a set of layers each consisting of H consecutive levels of T, where $H = 1 \uparrow (\mathrm{msb}(w) \downarrow 1)$, i.e. $H = 2^{\lfloor \frac{1}{2} \log w \rfloor}$, or equivalently H is the power of two, where $\frac{1}{2}\sqrt{w} < H \leq \sqrt{w}$. For a node u, we let $\pi(u)$ denote the nearest ancestor z of u, such that $\mathrm{depth}(z) \bmod H = 0$. If $\mathrm{depth}(u) \bmod H = 0$, then $\pi(u) = u$. Since H is a power of 2, we can compute $x \bmod H$ as $x \wedge (H - 1)$, i.e. for an internal node u, we can compute $\pi(u) = u \downarrow (\mathrm{depth}(u) \wedge (H - 1))$. E.g. in Figure 1, $H = 2$ and $\pi(9) = 9 \downarrow (3 \wedge (2 - 1)) = 9 \downarrow 1 = 4$.

The data structure for the set S consists of three sparse arrays B, D, and V, each being implemented according to Corollary 1. The arrays B and D will be used to find the nearest ancestor of a node in P that is a branching node.

- B : A bit-vector that for each node z in P with $\pi(z) = z$ (or equivalently $\mathrm{depth}(z) \bmod H = 0$), has $B[z] = 1$ if, and only if, there exists a node u in V with $\pi(u) = z$.
- D : A vector that for each node u in P where $\pi(u) = u$ or $B[\pi(u)] = 1$ stores the distance to the nearest ancestor v in V of u, i.e. $D[u] = \mathrm{depth}(v) - \mathrm{depth}(u)$.
- V : A vector that for each branching node v in V stores a record with the fields: left, right, m and M, where $V[v].m = \min S_v$ and $V[v].M = \max S_v$ and left (and right respectively) is a pointer to the record of the nearest descendent u in V of v in the left (and right respectively) subtree of v. If no such u exists, then $V[v].\mathrm{left} = v$ (respectively $V[v].\mathrm{right} = v$).

```
Proc Report(a, b)
    x = FindAny(a, b)
    if x ≠⊥ then
        Report(a, x − 1)
        output(x)
        Report(x + 1, b)

Proc FindAny(a, b)
    if a ≤ b then
        H = 1 ↑ (msb(w) ↓ 1)
        d = msb(a ⊕ b)
        u = ((1 ↑ (w − 1)) + (a ↓ 1)) ↓ d
        z = u ↓ ((w − 1 − d) ∧ (H − 1))
        v = B[z] ? V[u ↓ D[u]] : V[z ↓ D[z]]
        for x ∈ { v.left.m, v.left.M, v.right.m, v.right.M }
            if x ∈ [a, b] then return x
    return ⊥
```

Figure 2: Implementation of the queries Report and $\mathsf{FindAny}$.

Given the above data structure $\mathsf{FindAny}(a, b)$ can be implemented by the code in Figure 2. If $a > b$, the query immediately returns $\perp$. Otherwise the value H is computed, and the nearest common internal ancestor u in T of the leaves a and b is computed together with $z = \pi(u)$. Using B, D, and V we then compute the nearest common ancestor branching node v in V of the leaves a and b. In the computation of v an error may be introduced, since the arrays B, D and V are only well defined for a subset of the nodes of T. However, as we show next, this only happens when $S \cap [a, b] = \emptyset$. Finally we check if one of the m and M values of v.left and v.right is in $[a, b]$. If one of the four values belongs to $[a, b]$, we return such a value. Otherwise $\perp$ is returned.

As an example consider the query $\mathsf{FindAny}(8, 13)$ for the set in Figure 1. Here $d = 2$, $u = (8 + 4) \downarrow 3 = 3$, $z = 3 \downarrow ((3 - 2) \wedge 1) = 3 \downarrow 1 = 1$. Since $B[1] = 1$, we have $D[u] = 1$, and $v = V[u \downarrow D[u]] = V[3 \downarrow 1] = V[1]$. The four values tested are the m and M values of $V[2]$ and $V[7]$, i.e. $\{3, 7, 12, 14\}$, and we return 12.

THEOREM 4. *The data structure supports* $\mathsf{FindAny}$ *in constant time and* Report *in time* $\mathrm{O}(k)$*, where* k *is the number of elements reported. The data structure requires space* $\mathrm{O}(|S|)$ *words.*

PROOF. The correctness of $\mathsf{FindAny}(a, b)$ can be seen as follows: If $S \cap [a, b] = \emptyset$, then the algorithm returns $\perp$, since before returning an element there is a check to find if the element is contained in the interval $[a, b]$. Otherwise $S \cap [a, b] \neq \emptyset$.

If $a = b \in S$, then by Fact 1 the computed $u = 2^{w-1} + a \downarrow 1$ is the parent of a and $z = u \downarrow ((w - 1) \wedge (H - 1)) = u \downarrow$

$(\mathrm{depth}(u) \bmod H) = \pi(u)$. We now argue that v is the nearest ancestor node of the leaf a that is a branching node. If $S_{\pi(u)} = \{a\}$, then $T_{\pi(u)} \cap V = \emptyset$ and $B[\pi(u)] = 0$, and v is computed as $V[\pi(u) \downarrow D[\pi(u)]]$, which by definition of D is the nearest ancestor of $\pi(u)$ that is a branching node. Otherwise $|S_{\pi(u)}| \geq 2$, implying $T_{\pi(u)} \cap V \neq \emptyset$ and $B[\pi(u)] = 1$. By definition $D[u]$ is then defined such that $V[u \downarrow D[u]]$ is the nearest ancestor of u that is a branching node. We conclude that the computed v is the nearest ancestor of the leaf a that is a branching node. If the leaf a is contained in the left subtree of v, then $v.\mathrm{left} = v$ and $v.m = a$. It follows that $v.\mathrm{left}.m = a$. Similarly, if the leaf a is contained in the right subtree of v, then $v.\mathrm{right}.M = a$.

For the case where $S \cap [a, b] \neq \emptyset$ and $a < b$, we have by Fact 1 that the computed node u is the nearest common ancestor of the leaves a and b, where $\mathrm{depth}(u) = w - (d + 1)$, and that $z = u \downarrow ((w - 1 - d) \wedge (H - 1)) = u \downarrow (\mathrm{depth}(u) \bmod H) = \pi(u)$. Similarly to the case $a = b$, we have that the computed node v is the nearest ancestor of the node u that is a branching node. If $v = u$, i.e. v is the nearest common ancestor of the leaves a and b, then $S_{\mathsf{left}(v)} \cap [a, b] \neq \emptyset$ or $S_{\mathsf{right}(v)} \cap [a, b] \neq \emptyset$. If $|S_{\mathsf{left}(v)}| \geq 2$ and $S_{\mathsf{left}(v)} \cap [a, b] \neq \emptyset$, then $v.\mathrm{left} \neq v$ and $v.\mathrm{left}.M \in [a, b]$. If $|S_{\mathsf{left}(v)}| = 1$ and $S_{\mathsf{left}(v)} \cap [a, b] \neq \emptyset$, then $v.\mathrm{left} = v$ and $v.\mathrm{left}.m \in [a, b]$. Similarly if $S_{\mathsf{right}(v)} \cap [a, b] \neq \emptyset$, then either $v.\mathrm{right}.m \in [a, b]$ or $v.\mathrm{right}.M \in [a, b]$. Finally we consider the case where $v \neq u$, i.e. either $u \in T_{\mathsf{left}(v)}$ or $u \in T_{\mathsf{right}(v)}$. If $u \in T_{\mathsf{left}(v)}$ and $|S_{\mathsf{left}(v)}| = 1$, then $v.\mathrm{left} = v$ and $S_{\mathsf{left}(v)} = \{v.m\} = \{v.\mathrm{left}.m\} \subseteq [a, b]$. Similarly if $u \in T_{\mathsf{right}(v)}$ and $|S_{\mathsf{right}(v)}| = 1$, then $v.\mathrm{right} = v$ and $S_{\mathsf{right}(v)} = \{v.M\} = \{v.\mathrm{right}.M\} \subseteq [a, b]$. If $u \in T_{\mathsf{left}(v)}$ and $|S_{\mathsf{left}(v)}| \geq 2$, then $T_{v.\mathrm{left}}$ is either a subtree of $T_{\mathsf{left}(u)}$ or $T_{\mathsf{right}(u)}$, implying that $v.\mathrm{left}.M \in [a, b]$ or $v.\mathrm{left}.m \in [a, b]$ respectively. Similarly if $u \in T_{\mathsf{right}(v)}$ and $|S_{\mathsf{right}(v)}| \geq 2$, then either $v.\mathrm{right}.M \in [a, b]$ or $v.\mathrm{right}.m \in [a, b]$.

We conclude that if $S \cap [a, b] \neq \emptyset$, then FindAny returns an element in $S \cap [a, b]$.

The fact that FindAny takes constant time follows from Theorem 1 and Corollary 1, since only a constant number of boolean operations and arithmetic operations is performed plus two calls to msb and three sparse array lookups. The correctness of Report and the time $O(k)$ bound follows from Lemma 1.

The space required by the data structure depends on the size required for the three sparse arrays B, D, and V. The number of internal levels of T with depth mod $H = 0$ is $\lceil w/H \rceil$, and therefore the number of initialized entries in B is at most $n\lceil w/H \rceil = O(n\sqrt{w})$. Similarly, the number of initialized entries in D due to $\pi(u) = u$ is at most $n\lceil w/H \rceil$. For the number of initialized entries in D due to $B[\pi(u)] = 1$, we observe that the subtree τ_z of height H rooted at $z = \pi(u)$ by definition contains at least one node from V. If $|\tau_z \cap V| = s$, then τ_z has at most $s + 1$ leaves which are nodes in P, and we have $|\tau_z \cap P| \leq (s + 1)H \leq 2Hs$. Since τ_z contributes to B with at most $2H|\tau_z \cap V|$ entries and $|V| \leq n$, the total number of initialized entries contributed to B due to $B[\pi(u)] = 1$ is bounded by $2Hn$. The number of initialized entries in B is therefore bounded by $2Hn + n\lceil w/H \rceil = O(n\sqrt{w})$. Finally, by definition, V contains at most n initialized entries.

Each entry of B, D, and V requires space: 1, $\lceil \log w \rceil$, and $O(w)$ bits respectively, and B, D, and V have $O(n\sqrt{w})$, $O(n\sqrt{w})$, and at most n initialized entries respectively. The total number of words for storing the three sparse arrays by Corollary 1 is therefore $O((\log w \cdot n\sqrt{w} + w \cdot n)/w) = O(n)$ words. It follows that the total space required for storing the data structure is $O(n)$ words. $\Box$

4. CONSTRUCTION

In this section we describe how to construct the data structure of the previous section in expected time $O(n\sqrt{w})$.

THEOREM 5. *Given an unordered set of n distinct integers each of w bits, the range reporting data structure in Section 3 can be constructed in expected time $O(n\sqrt{w})$.*

PROOF. Initially S can be sorted in space $O(n)$ with the algorithm of Thorup [23] in time $O(n(\log \log n)^2) = O(n\sqrt{w})$ or with the randomized algorithm of Andersson *et al.* [5] in expected time $O(n \log \log n) = O(n\sqrt{w})$. Therefore without loss of generality we can assume $S = \{a_1, \ldots, a_n\}$ where $a_i < a_{i+1}$ for $1 \leq i < n$.

We observe that $v \in V$ if, and only if, v is the root or v, is the nearest common ancestor of a_i and a_{i+1} for some i, where $1 \leq i < n$. Similarly as for the FindAny query, we can by Fact 1 find the nearest common ancestor $v_i \in V$ induced by a_i and a_{i+1} in constant time by the expression

$$v_i = ((1 \uparrow (w - 1)) + (a_i \downarrow 1)) \downarrow \mathrm{msb}(a_i \oplus a_{i+1}) \ .$$

The nodes $v \in V$ form by the pointers $v.\mathrm{left}$ and $v.\mathrm{right}$ a binary tree T_V. The defined sequence $v_1, \ldots, v_{n-1}$ forms an inorder traversal of T_V. Furthermore the nodes satisfy heap order with respect to their depths in T. Recall that $\mathrm{depth}(v_i) = \mathrm{msb}(v_i)$ can be computed in constant time.

The inorder together with the heap order on the depth of the v_i nodes uniquely defines T_V since these are exactly the constraints determining the shape of the *treaps* introduced by Seidel and Aragon [22]. By applying an time $O(n)$ treap construction algorithm [22] to $v_1, \ldots, v_{n-1}$ we get the required left and right pointers for V. The m and M fields for the nodes in V can be constructed in a bottom-up traversal of T_V in time $O(n)$.

The information to be stored in the arrays B and D can by another traversal of T_V be constructed in time linear in the number of nodes to be initialized. Consider an edge (u, v) in T_V, where v is the parent of u in T_V, i.e. v is the nearest ancestor node of u in T that is a branching node or v is the root. Let $u = u_0, u_1, \ldots, u_d = v$ be the nodes on the path from u to v in T such that $\mathrm{depth}(u_i) - \mathrm{depth}(u_{i+1}) = 1$. While processing the edge (u, v) we will compute the information to be stored in the sparse arrays for the nodes $u_0, u_1, \ldots, u_{d-1}$, i.e. the nodes on the path from u to v exclusive v. From the definition of B and D we get the following: For the array B we store $B[\pi(u)] = 1$, if $\mathrm{depth}(\pi(u)) > \mathrm{depth}(v)$, and $B[u_i] = 0$ for all $i = 0, \ldots, d - 1$, where $\mathrm{depth}(u_i) < \mathrm{depth}(\pi(u))$ and $\mathrm{depth}(u_i) \bmod H = 0$. For the array D we store $D[u_i] = \text{depth}(v) - \text{depth}(u_i)$ for all $i = 0, \ldots, d-1$ where $\text{depth}(u_i) \bmod H = 0$ or $\text{depth}(u_i) < H\lceil \text{depth}(v)/H \rceil$ or $\text{depth}(u_i) \geq \text{depth}(\pi(u))$. Finally, we store for the root $B[1] = 1$ and $D[1] = 0$.

Constructing the three sparse arrays, after having identified the $O(n\sqrt{w})$ entries to be initialized, by Corollary 1 takes expected time $O(n\sqrt{w})$. $\square$

5. APPROXIMATE RANGE COUNTING

In this section we provide a data structure for approximate range counting. Let $S \subseteq U$ denote the input set, and let n denote the size of S. The data structure uses space $O(n)$ words and supports $\mathsf{Count}_\varepsilon$ queries in constant time, for any constant $\varepsilon > 0$.

We assume S has been preprocessed such that in constant time we can compute $\mathsf{FindAny}(a, b)$ for all $a, b \in U$. Next we have a sparse array such that we for each element $x \in S$ can compute $\text{rank}_S(x)$ in constant time and can find the successor of x in S in constant time. Both these data structures use space $O(n)$.

Define $count(a, b) = |S \cap [a, b]|$. We need to build a data structure which for any $a, b \in U$ enables the computation of an integer k such that $count(a, b) \leq k \leq (1+\varepsilon)count(a, b)$.

In the following we will use the observation that for $a, b \in S$, $a \leq b$, it is easy to compute the exact value of $count(a, b)$. This value can be expressed as $\text{rank}_S(b) - \text{rank}_S(a) + 1$ and thus the computation amounts to two lookups in the sparse array storing the ranks.

We reduce the task of computing $\mathsf{Count}_\varepsilon(a, b)$ to the case where either a or b are in S. First, it is easy to check if $S \cap [a, b]$ is empty, i.e., $\mathsf{FindAny}(a, b)$ returns $\perp$, in which case we simply return 0 for the query. Hence, we can assume $S \cap [a, b]$ is non-empty and let c be any element in this set and let d be the successor of c in S. Then for any integers k_a and k_b such that $count(a, c) \leq k_a \leq (1+\varepsilon)count(a, c)$ and $count(d, b) \leq k_b \leq (1+\varepsilon)count(d, b)$, it holds that $count(a, b) = count(a, c) + count(d, b) \leq k_a + k_b \leq (1+\varepsilon)count(a, b)$. Hence, we can return $\mathsf{Count}_\varepsilon(a, c) + \mathsf{Count}_\varepsilon(d, b)$ as the answer for $\mathsf{Count}_\varepsilon(a, b)$, where $c \in S \cap [a, b]$ is an integer returned by $\mathsf{FindAny}(a, b)$. Clearly, both calls to $\mathsf{Count}_\varepsilon$ satisfy that one of the endpoints is in S, i.e., the integers c and d. In the following we can thus without loss of generality limit ourselves to the case for a query $\mathsf{Count}_\varepsilon(a, b)$ with $a \in S$. The case $b \in S$ is treated symmetrically.

We start by describing the additional data structures needed, and then how to compute the approximate range counting query using these. Define $p = \lceil \log n \rceil$, and

$$J = \{x \in S \mid (\text{rank}_S(x) - 1) \bmod p = 0\} \cup \max S \ .$$

We construct the following additional data structures (see Figure 3 for an example).

JnodeR : For each element $s \in S$ we store the integer $\mathsf{JnodeR}(s)$ being the successor of s in J.

JumpR : For each element $j \in J$ we store the set $\mathsf{JumpR}(j) = \{y \in S \mid count(j, y) = 2^i \wedge i \in [0, p]\}$.

LN : For each element $j \in J$ we store the set $\mathsf{LN}(j) = \{i \in S \mid j = \mathsf{JnodeR}(i)\}$.

Each of the sets JumpR and LN have size bounded by $p \leq \log|U|$, and hence using the Q-heaps from Corollary 2, we can compute predecessors for these sets in constant time. The Q-heaps have space cost linear in the set sizes. Since the total number of elements in the structures JumpR and LN is $O(|S|)$, the total space cost for these structures is $O(|S|)$. Furthermore, for the elements in S given in sorted order, the total construction of these data structures is also $O(|S|)$.

To determine $\mathsf{Count}_\varepsilon(a, b)$, where $a \in S$, we iterate the following computation until the desired precision of the answer is obtained. Initially let $k = 0$. Let $j = \mathsf{JnodeR}(a)$. If $j \geq b$, return $k + count(a, \mathsf{Pred}_{LN(j)}(b))$. Otherwise, $j < b$, and we increase k by $count(a, j) - 1$. Let $y = \mathsf{Pred}_{\mathsf{JumpR}(j)}(b)$ and $i = \text{rank}_{\mathsf{JumpR}(j)}(y) - 1$. Now $count(j, y) = 2^i \leq count(j, b) < 2^{i+1}$. We increase k by 2^i. Now $k = count(a, y)$ and $count(y, b) < 2^i$. If $y = b$ we return k. If $2^i/k < \varepsilon$, we are also satisfied and return $k + 2^i$. Otherwise we iterate to determine $\mathsf{Count}_\varepsilon(y', b)$, where y' is the successor of y in S.

THEOREM 6. *The data structure uses space $O(|S|)$ words and supports $\mathsf{Count}_\varepsilon$ in time $O(\log(1/\varepsilon))$ for any $\varepsilon > 0$.*

PROOF. From the observations above we conclude that the structure uses space $O(n)$ and expected preprocessing time $O(n)$. Each iteration takes constant time, and next we show that the number of iterations is at most $\ell \leq 1 + \lceil \log(1/\varepsilon) \rceil$. Let $k = 2^I + f$, $f < 2^I$, after the first iteration. In the ℓth iteration we either return $count(a, b)$ or $k + 2^i > count(a, b)$, where $i \leq I - \ell + 1$. In the latter case we have $k < count(a, b) < k + 2^i$. That $k + 2^i \leq (1+\varepsilon)count(a, b)$ follows from $k < count(a, b)$ and $2^i/k < \varepsilon$. Since $2^i < k\varepsilon$, $i \leq I - \ell + 1$ and $k \geq 2^I$, the bound on ℓ follows from $2^{I-\ell+1} < 2^I \varepsilon$. $\square$

6. REFERENCES

[1] P. K. Agarwal. Range searching. In *Handbook of Discrete and Computational Geometry, CRC Press.* 1997.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] M. Ajtai. A lower bound for finding predecessors in Yao's cell probe model. *Combinatorica*, (8):235–247, 1988.

[4] A. Amir, A. Efrat, P. Indyk, and H. Samet. Efficient regular data structures and algorithms for location and proximity problems. In *Proc. 40th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 160–170, 1999.

[5] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57(1):74–93, 1998.

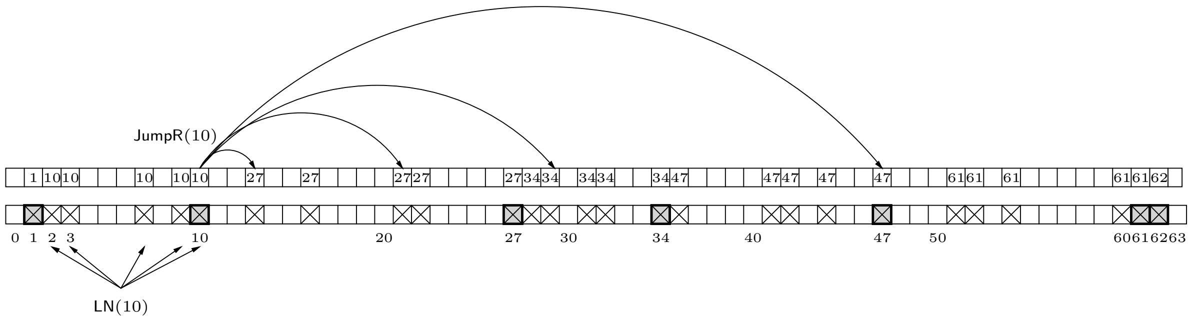


Figure 3: Extension of the data structure to support $\mathsf{Count}_\varepsilon$ queries. $w = 8$, $n = |S| = 27$, **and** $p = \lceil \log n \rceil = 5$**.**

[6] A. Andersson and O. Petersson. Approximate indexed lists. *Journal of Algorithms*, 29(2):256–276, 1998.

[7] A. Andersson and M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proc. 32nd ACM Symposium on Theory of Computing (STOC)*, pages 335–342, 2000.

[8] P. Beame and F. Fich. Optimal bounds for the predecessor problem. In *Proc. 31st ACM Symposium on Theory of Computing (STOC)*, pages 295–304, 1999.

[9] K. L. Clarkson. An algorithm for approximate closest-point queries. In *Proc. 10th Annual Symposium on Computational Geometry*, pages 160–164, Stony Brook, NY, USA, 1994. ACM Press.

[10] M. Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In *13th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1046 of *Lecture Notes in Computer Science*, pages 569–580. Springer Verlag, Berlin, 1996.

[11] R. W. Floyd. Algorithm 245: Treesort3. *Communications of the ACM*, 7(12):701, 1964.

[12] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with O(1) worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[13] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424–436, 1993.

[14] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48:533–551, 1994.

[15] D. Harel and R. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing*, 13(2):338–355, 1984.

[16] C. T. M. Jacobs and P. van Emde Boas. Two results on tables. *Information Processing Letters*, 22(1):43–48, 1986.

[17] Y. Matias, J. S. Vitter, and N. E. Young. Approximate data structures with applications. In *Proc. 5th ACM-SIAM Symp. Discrete Algorithms, SODA*, pages 187–194, 1994.

[18] K. Mehlhorn. *Data Structures and Algorithms: 3. Multidimensional Searching and Computational Geometry*. Springer, 1984.

[19] P. B. Miltersen, N. Nisan, S. Safra, and A. Wigderson. On data structures and asymmetric communication complexity. *Journal of Computer and System Sciences*, 57(1):37–49, 1998.

[20] F. P. Preparata and M. Shamos. *Computational Geometry*. Springer-Verlag, New York, 1985.

[21] J. P. Schmidt and A. Siegel. The spatial complexity of oblivious k-probe hash functions. *SIAM Journal of Computing*, 19(5):775–786, 1990.

[22] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[23] M. Thorup. Faster deterministic sorting and priority queues in linear space. In *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 550–555, 1998.

[24] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.

[25] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.