Dynamic Range Selection in Linear Space*

Meng He[1], J. Ian Munro[2], and Patrick K. Nicholson[2]

[1] Faculty of Computer Science, Dalhousie University, Canada
[2] David R. Cheriton School of Computer Science, University of Waterloo, Canada,
mhe@cs.dal.ca, {imunro, p3nichol}@uwaterloo.ca

Abstract. Given a set S of n points in the plane, we consider the problem of answering range selection queries on S: that is, given an arbitrary x-range Q and an integer $k > 0$, return the k-th smallest y-coordinate from the set of points that have x-coordinates in Q. We present a linear space data structure that maintains a dynamic set of n points in the plane with real coordinates, and supports range selection queries in $O((\lg n/\lg\lg n)^2)$ time, as well as insertions and deletions in $O((\lg n/\lg\lg n)^2)$ amortized time. The space usage of this data structure is an $\Theta(\lg n/\lg\lg n)$ factor improvement over the previous best result, while maintaining asymptotically matching query and update times. We also present a succinct data structure that supports range selection queries on a dynamic array of n values drawn from a bounded universe.

1 Introduction

The problem of finding the *median* value in a data set is a staple problem in computer science, and is given a thorough treatment in modern textbooks [6]. In this paper we study a dynamic data structure variant of this problem in which we are given a set S of n points in the plane. The *dynamic range median problem* is to construct a data structure to represent S such that we can support *range median queries*: that is, given an arbitrary range $Q = [x_1, x_2]$, return the median y-coordinate from the set of points that have x-coordinates in Q. Furthermore, the data structure must support insertions of points into, as well as deletions from, the set S. We may also generalize our data structure to support *range selection queries*: that is, given an arbitrary x-range $Q = [x_1, x_2]$ and an integer $k > 0$, return the k-th smallest y-coordinate from the set of points that have x-coordinates in Q.

In addition to being a challenging theoretical problem, the range median and selection problems have several practical applications in the areas of image processing [9], Internet advertising, network traffic analysis, and measuring real-estate prices in a region [10].

In previous work, the data structures designed for the above problems that support queries and updates in polylogarithmic time require superlinear space [5]. In this paper, we focus on designing linear space dynamic range selection data

* This work was supported by NSERC and the Canada Research Chairs Program.

structures, without sacrificing query or update time. We also consider the problem of designing *succinct data structures* that support range selection queries on a dynamic array of values, drawn from a bounded universe: here "succinct" means that the space occupied by our data structure is close to the information-theoretic lower bound of representing the array of values [13].

1.1 Previous Work

Static Case: The static range median and selection problems have been studied heavily in recent years [3, 15, 10, 18, 19, 7, 8, 4, 5, 14]. In these problems we consider the n points to be in an array: that is, the points have x-coordinates $\{1, ..., n\}$. We now summarize the upper and lower bounds for the static problem. In the remainder of this paper we assume the word-RAM model of computation with word size $w = \Omega(\lg n)$ bits.

For exact range medians in constant time, there have been several iterations of near-quadratic space data structures [15, 18, 19]. For linear space data structures, Gfeller and Sanders [8] showed that range median queries could be supported in $O(\lg n)$ time[3], and Gagie et al. [7] showed that selection queries could be supported in $O(\lg \sigma)$ time using a wavelet tree, where σ is the number of distinct y-coordinates in the set of points. Optimal upper bounds of $O(\lg n/\lg\lg n)$ time for range median queries have since been achieved by Brodal et al. [4, 5], and lower bounds by Jørgensen and Larsen [14]; the latter proved a cell-probe lower bound of $\Omega(\lg n/\lg\lg n)$ time for any static range selection data structure using $O(n \lg^{O(1)} n)$ bits of space. In the case of range selection when k is fixed for all queries, Jørgensen and Larsen proved a cell-probe lower bound of $\Omega(\lg k/\lg\lg n)$ time for any data structure using $O(n \lg^{O(1)} n)$ space [14]. Furthermore, they presented an adaptive data structure for range selection, where k is given at query time, that matches their lower bound, except when $k = 2^{o(\lg^2 \lg n)}$ [14]. Finally, Bose et al. [3] studied the problem of finding *approximate range medians*. A c-approximate median of range $[i..j]$ is a value of rank between $\frac{1}{c} \times \lceil \frac{j-i+1}{2} \rceil$ and $(2 - \frac{1}{c}) \times \lceil \frac{j-i+1}{2} \rceil$, for $c > 1$.

Dynamic Case: Gfeller and Sanders [8] presented an $O(n \lg n)$ space data structure for the range median problem that supports queries in $O(\lg^2 n)$ time and insertions and deletions in $O(\lg^2 n)$ amortized time. Later, Brodal et al. [4, 5] presented an $O(n \lg n/\lg\lg n)$ space data structure for the dynamic range selection problem that answers range queries in $O((\lg n/\lg\lg n)^2)$ time and insertion and deletions in $O((\lg n/\lg\lg n)^2)$ amortized time. They also show a reduction from the *marked ancestor problem* [1] to the dynamic range median problem. This reduction shows that $\Omega(\lg n/\lg\lg n)$ query time is required for any data structure with polylogarithmic update time. Thus, there is still a gap of $\Theta(\lg n/\lg\lg n)$ time between the upper and lower bounds for linear and near linear space data structures.

[3] In this paper we use $\lg n$ to denote $\log_2 n$.

In the restricted case where the input is a dynamic array A of n values drawn from a bounded universe, $[1, \sigma]$, it is possible to answer range selection queries using a dynamic wavelet tree, such as the succinct dynamic string data structure of He and Munro [11]. This data structure uses $nH_0(A)+o(n \lg \sigma)+O(w)$ bits[4] of space, the query time is $O(\frac{\lg n \lg \sigma}{\lg \lg n})$, and the update time is $O(\frac{\lg n}{\lg \lg n}(\frac{\lg \sigma}{\lg \lg n}+1))$.

1.2 Our Results

In Section 2, we present a *linear space* data structure for the dynamic range selection problem that answers queries in $O((\lg n/\lg \lg n)^2)$ time, and performs updates in $O((\lg n/\lg \lg n)^2)$ amortized time. This data structure can be used to represent point sets in which the points have *real coordinates*. In other words, we only assume that the coordinates of the points can be compared in constant time. This improves the space usage of the previous best data structure by a factor of $\Theta(\lg n/\lg \lg n)$ [5], while maintaining query and update time.

In Section 3, we present a succinct data structure that supports range selection queries on a dynamic array A of values drawn from a bounded universe, $[1..\sigma]$. The data structure occupies $nH_0(A) + o(n \lg \sigma) + O(w)$ bits, and supports queries in $O(\frac{\lg n}{\lg \lg n}(\frac{\lg \sigma}{\lg \lg n} + 1))$ time, and insertions and deletions in $O(\frac{\lg n}{\lg \lg n}(\frac{\lg \sigma}{\lg \lg n} + 1))$ amortized time. This is a $\Theta(\lg \lg n)$ improvement in query time over the dynamic wavelet tree, and thus closes the gap between the dynamic wavelet tree solution and that of Brodal et al. [5].

2 Linear Space Data Structure

In this section we describe a linear space data structure for the dynamic range selection problem. Our data structure follows the same general approach as the dynamic data structure of Brodal et al. [5]. However, we make several important changes, and use several other auxiliary data structures, in order to improve the space by a factor of $\Theta(\lg n/\lg \lg n)$.

The main data structure is a weight balanced B-tree [2], T, with branching parameter $\Theta(\lg^{\varepsilon_1} n)$, for $0 < \varepsilon_1 < 1/2$, and leaf parameter 1. The tree T stores the points in S at its leaves, sorted in non-decreasing order of y-coordinate[5]. The height of T is $h_1 = \Theta(\lg n/\lg \lg n)$ *levels*, and we assign numbers to the levels starting with level 1 which contains the root node, down to level h_1 which contains the leaves of T. Inside each internal node $v \in T$, we store the smallest and largest y-coordinates in $T(v)$. Using these values we can acquire the path from the root of T to the leaf representing an arbitrary point contained in S in $O(\lg n)$ time; a binary search over the values stored in the children of an arbitrary internal node requires $O(\lg \lg n)$ time per level.

[4] $H_0(A)$ denotes the 0th-order empirical entropy of the multiset of values stored in A. Note that $H_0(A) \leq \lg \sigma$ always holds.

[5] Throughout this paper, whenever we order a list based on y-coordinate, it is assumed that we break ties using the x-coordinate, and vice versa.

Following Brodal et al. [5], we store a *ranking tree* $R(v)$ inside each internal node $v \in T$. The purpose of the ranking tree $R(v)$ is to allow us to efficiently make a branching decision in the main tree T, at node v. Let $T(v)$ denote the subtree rooted at node v. The ranking tree $R(v)$ represents all of the points stored in the leaves of $T(v)$, sorted in non-decreasing order of x-coordinate. The fundamental difference between our ranking trees, and those of Brodal et al. [5], is that ours are more space efficient. Specifically, in order to achieve linear space, we must ensure that the ranking trees stored in each level of T occupy no more than $O(n \lg \lg n)$ bits in total, since there are $O(\lg n / \lg \lg n)$ levels in T. We describe the ranking trees in detail in Section 2.1, but first discuss some auxiliary data structures we require in addition to T.

We construct a red-black tree S_x that stores the points in S at its leaves, sorted in non-decreasing order of x-coordinate. As in [5], we augment the red-black tree S_x to store, in each node v, the count of how many points are stored in $T(v_1)$ and $T(v_2)$, where v_1 and v_2 are the two children of v. Using these counts, S_x can be used to map any query $[x_1, x_2]$ into r_1, the rank of the successor of x_1 in S, and r_2, the rank of the predecessor of x_2 in S. These ranking queries, as well as insertions and deletions into S_x, take $O(\lg n)$ time.

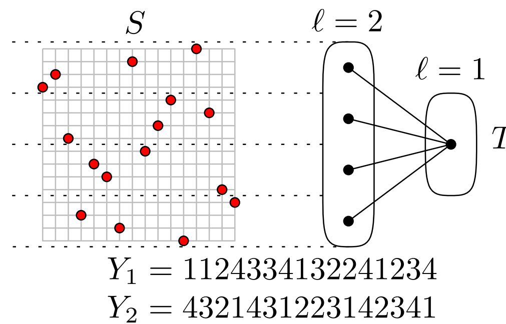

Fig. 1. The top two levels of an example tree T, and the corresponding strings Y_1 and Y_2 for these levels. Each node at level 2 has exactly 4 children.

We also store a string $Y(v)$ for each node v in T. This string consists of all of the of the points in $T(v)$ sorted in non-decreasing order of x-coordinate, where each point is represented by the index of the child of node v's subtree in which they are contained, i.e., an integer bounded by $O(\lg^{\varepsilon_1} n)$. However, for technical reasons, instead of storing each string with each node $v \in T$, we concatenate all the strings $Y(v)$ for each node v at level ℓ in T into a string of length n, denoted by Y_ℓ. Each chunk of string Y_ℓ from left to right represents some node v in level ℓ of T from left to right within the level. See Figure 1 for an illustration. We represent each string Y_ℓ using the succinct dynamic string data structure of He and Munro [12]. Depending on the context, we refer to both the string, and

also the data structure that represents the string, as Y_ℓ. Consider the following operations on the string Y_ℓ:

- $\texttt{access}(Y_\ell, i)$, which returns the i-th integer, $Y_\ell[i]$, in Y_ℓ;
- $\texttt{rank}_\alpha(Y_\ell, i)$, which returns the number of occurrences of integer α in $Y_\ell[1..i]$;
- $\texttt{range_count}(Y_\ell, x_1, x_2, y_1, y_2)$, which returns the total number of entries in $Y_\ell[x_1..x_2]$ whose values are in the range $[y_1..y_2]$;
- $\texttt{insert}_\alpha(Y_\ell, i)$, which inserts integer α between $Y_\ell[i-1]$ and $Y_\ell[i]$;
- $\texttt{delete}(Y_\ell, i)$, which deletes $Y_\ell[i]$ from Y_ℓ.

Let $W = \lceil \frac{\lceil \lg n \rceil^2}{\lg \lceil \lg n \rceil} \rceil$. The following lemma summarizes the functionality of these data structures for succinct *dynamic strings* over small universe:

Lemma 1 ([12]). *Under the word RAM model with word size $w = \Omega(\lg n)$, a string $Y_\ell[1..n]$ of values from a bounded universe $[1..\sigma]$, where $\sigma = O(\lg^\mu n)$ for any constant $\mu \in (0, 1)$, can be represented using $nH_0(Y_\ell) + O(\frac{n \lg \sigma \lg \lg n}{\sqrt{\lg n}}) + O(w)$ bits to support* access*,* rank*,* range_count*,* insert *and* delete *in $O(\frac{\lg n}{\lg \lg n})$ time. Furthermore, we can perform a batch of m update operations in $O(m)$ time on a substring $Y_\ell[i..i+m-1]$ in which the j-th update operation changes the value of $Y_\ell[i+j-1]$, provided that $m > \frac{5W}{\lg \sigma}$.*

The data structure summarized by the previous lemma is, roughly, a B-tree constructed over the string $Y_\ell[1..n]$, in which each leaf stores a *superblock*, which is a substring of $Y_\ell[1..n]$ of length at most $2W$ bits. We mention this because the ranking tree stored in each node of T will implicitly reference these superblocks instead of storing leaves. Thus, the leaves of the dynamic string at level ℓ are *shared* with the ranking trees stored in nodes at level ℓ.

As for their purpose, these dynamic string data structures Y_ℓ are used to translate the ranks r_1 and r_2 into ranks within a restricted subset of the points when we navigate a path from the root of T to a leaf. The space occupied by these strings is $O((n \lg(\lg^{\varepsilon_1} n) + w) \times \lg n / \lg \lg n)$ bits, which is $O(n)$ words. We present the following lemma:

Lemma 2. *Ignoring the ranking trees stored in each node of T, the data structures described in this section occupy $O(n)$ words.*

In the next section we discuss the technical details of our space-efficient ranking tree. The key idea to avoid using linear space per ranking tree is to *not* actually store the points in the leaves of the ranking tree, sorted in non-decreasing order of x-coordinate. Instead, for each point p in ranking tree $R(v)$, we implicitly reference the the string $Y(v)$, which stores the index of the child of v that contains p.

2.1 Space Efficient Ranking Trees

Each ranking tree $R(v)$ is a weight balanced B-tree with branching parameter $\lg^{\varepsilon_2} n$, where $0 < \varepsilon_2 < 1 - \varepsilon_1$, and leaf parameter $\Theta(W / \lg \lceil \lg n \rceil) =$

$\Theta((\lg n / \lg \lg n)^2)$. Thus, $R(v)$ has height $\Theta(\lg n / \lg \lg n)$, and each leaf of $R(v)$ is implicitly represented by a substring of $Y(v)$, which is stored in one of the dynamic strings, Y_ℓ.

Internal Nodes: Inside each internal node u in $R(v)$, let q_i denote the number of points stored in the subtree rooted at the i-th child of u, for $1 \leq i \leq f_2$, where f_2 is the degree of u. We store a *searchable partial sums structure* [20] for the sequence $Q = \{q_1, ..., q_{f_2}\}$. This data structure will allow us to efficiently navigate from the root of $R(v)$ to the leaf containing the point of x-coordinate rank r. The following lemma summarizes the functionality of this data structure:

Lemma 3 ([20]). *Suppose the word size is $\Omega(\lg n)$. A sequence Q of $O(\lg^\mu n)$ nonnegative integers of $O(\lg n)$ bits each, for any constant $\mu \in (0, 1)$, can be represented in $O(\lg^{1+\mu} n)$ bits and support the following operations in $O(1)$ time:*

- `sum`(Q, i) *which returns* $\sum_{j=1}^{i} Q[j]$,
- `search`(Q, x) *which returns the smallest i such that* `sum`$(Q, i) \geq x$,
- `modify`(Q, i, δ) *which sets $Q[i]$ to $Q[i] + \delta$, where $|\delta| \leq \lg n$.*

This data structure can be constructed in $O(\lg^\mu n)$ time, and it requires a precomputed universal table of size $O(n^{\mu'})$ bits for any fixed $\mu' > 0$.

We also store the *matrix structure* of Brodal et al. [5] in each internal each node u of the ranking tree. Let $f_1 = \Theta(\lg^{\varepsilon_1} n)$ denote the out-degree of node $v \in T$, and let $T(v_1), ..., T(v_{f_1})$ denote the subtrees rooted at the children of v from left to right. Similarly, recall that $f_2 = \Theta(\lg^{\varepsilon_2} n)$ denotes the out-degree of $u \in R(v)$, and let $T'(u_1), ..., T'(u_{f_2})$ be the subtrees rooted at each child of u from left to right. These matrix structures are a kind of partial sums data structure defined as follows; we use roughly the same notation as [5]:

Definition 1 (Summarizes [5]). *A* matrix structure *M^u is an $f_1 \times f_2$ matrix, where entry $M^u_{p,q}$ stores the number of points from $\cup_{i=1}^{q} T'(u_i)$ that are contained in $\cup_{i=1}^{p} T(v_i)$. The matrix structure M^u is stored in two ways. The first representation is a standard table, where each entry is stored in $O(\lg n)$ bits. In the second representation, we divide each column into* sections *of $\Theta(\lg^{\varepsilon_1} n)$ bits — leaving $\Theta(\lg \lg n)$ bits of overlap between the sections for technical reasons — and we number the sections $s_1, ..., s_g$, where $g = \Theta(\lg^{1-\varepsilon_1} n)$. In the second representation, for each column q, there is a* packed word *$w^u_{q,i}$, storing section s_i of each entry in column q. Again, for technical reasons, the most significant bit of each section stored in the packed word $w^u_{q,i}$ is padded with a zero bit.*

We defer the description of how the matrix structures are used to guide queries until Section 2.2. For now, we just treat these structures as a black box and summarize their properties with the following lemma:

Lemma 4 ([5]). *The matrix structure M^u for node u in the ranking tree $R(v)$ occupies $O(\lg^{1+\varepsilon_1+\varepsilon_2} n)$ bits, and can be constructed in $o(\lg^{1+\varepsilon_1+\varepsilon_2} n)$ time. Furthermore, consider an update path that goes through node u when we insert a value into or delete a value from $R(v)$. The matrix structures in each node along an update path can be updated in $O(1)$ amortized time per node.*

Shared Leaves: Now that we have described the internal nodes of the ranking tree, we describe the how the leaves are shared between $R(v)$ and the dynamic string over Y_ℓ. To reiterate, we do not actually store the leaves of $R(v)$: they are only conceptual. We present the following lemma that will be crucial to performing queries on the ranking trees:

Lemma 5. *Let u be a leaf in $R(v)$ and S be the substring of $Y(v)$ that u represents, where each value in S is in the range $[1..\sigma]$, and $\sigma = \Theta(\lg^{\varepsilon_1} n)$. Using a universal table of size $O(\sqrt{n} \times \texttt{polylog}(n))$ bits, for any $z \in [1..|S|]$, an array $C_z = \{c_1, ..., c_\sigma\}$ can be computed in $O(\lg n/\lg\lg n)$ time, where $c_i = \texttt{rank}_i(S, z)$, for $1 \le i \le \sigma$.*

We end our discussion of ranking trees by presenting the following lemma regarding their space and construction time:

Lemma 6. *Each ranking tree $R(v)$ occupies $O\left(\frac{m(\lg\lg n)^2}{\lg^{1-\varepsilon_1} n} + w\right)$ bits of space if $|T(v)| = m$, and requires $O(m)$ time to construct, assuming that we have access to the string $Y(v)$.*

Remark 1. Note that the discussion in this section implies that we need not store ranking trees for nodes $v \in T$, where $|T(v)| = O(\lg n/\lg\lg n)^2$. Instead, we can directly query the dynamic string Y_ℓ using Lemma 5 in $O(\lg n/\lg\lg n)$ time to make a branching decision in T. This will be important in Section 3, since it significantly reduces the number of pointers we need.

2.2 Answering Queries

In this section, we explain how to use our space efficient ranking tree in order to guide a range selection query in T. We are given a query $[x_1, x_2]$ as well as a rank k, and our goal is to return the k-th smallest y-coordinate in the query range. We begin our search at the root node v of the tree T. In order to guide the search to the correct child of v, we determine the canonical set of nodes in $R(v)$ that represent the query $[x_1, x_2]$. Before we query $R(v)$, we search for x_1 and x_2 in S_x. Let r_1 and r_2 denote the ranks of the successor of x_1 and predecessor of x_2 in S, respectively. We query $R(v)$ using $[r_1, r_2]$, and use the searchable partial sum data structures stored in each node of $R(v)$, to identify the canonical set of nodes in $R(v)$ that represent the interval $[r_1, r_2]$. At this point we outline how to use the matrix structures in order to decide how to branch in T.

Matrix Structures: We discuss a straightforward, *slow method* of computing the branch of the child of v to follow. The full details of how to use the matrix structures to speed up the query can be found in the original paper [5].

In order to determine the child of v that contains the k-th smallest y-coordinate in the query range, recall that T is sorted by y-coordinate. Let f_1 denote the degree of v, and q_i' denote the number of points that are contained in the range $[x_1, x_2]$ in the subtree rooted at the i-th child of v, for $1 \le i \le d$.

Determining the child that contains the k-th smallest y-coordinate in $[x_1, x_2]$ is equivalent to computing the value τ such that $\sum_{i=1}^{\tau-1} q'_i < k$ and $\sum_{i=1}^{\tau} q'_i \geq k$. In order to compute τ, we examine the set of canonical nodes of $R(v)$ that represent $[x_1, x_2]$, denoted C. The set C contains $O(\lg n / \lg \lg n)$ internal nodes, as well as at most two leaf nodes.

Consider any internal node $u \in C$, and without loss of generality, suppose u was on the search path for r_1, but not the search path for r_2, and that u has degree f_2. If the search path for r_1 goes through child c_q in u, then consider the difference between columns f_2 and q in the first representation of matrix M^u. We denote this difference as M'^u, where $M'^u_i = M^u_{i,f_2} - M^u_{i,q}$, for $1 \leq i \leq f_1$. For each internal node $u \in C$ we add each M'^u to a running total, and denote the overall sum as M'. Next, for each of the — at most — two leaves on the search path, we query the superblocks of Y_ℓ to get the relevant portions of the sums, and add them to M'. At this point, $M'_i = q'_i$, and it is a simple matter to scan each entry in M' to determine the value of τ. Since each matrix structure has f_1 entries in its columns, and by Lemma 5, this overall process takes $O(f_1 \times \lg n / \lg \lg n) = O(\lg^{1+\varepsilon_1} n / \lg \lg n)$ time, since there are $O(\lg n / \lg \lg n)$ levels in $R(v)$. Since there are $O(\lg n / \lg \lg n)$ levels in T, this costs $O((\lg n / \lg \lg n)^2 \times \lg^{\varepsilon_1} n)$ time. This time bound can be reduced by a factor of $f_1 = O(\lg^{\varepsilon_1} n)$, using word-level parallelism and the second representation of the matrix structures [5].

Recursively Searching in T: Let v_τ denote the τ-th child of v. The final detail to discuss is how we translate the ranks $[r_1, r_2]$ into ranks in the tree $R(v_\tau)$. To do this, we query the string $Y(v)$ before recursing to v_τ. We use two cases to describe how to find $Y(v)$ within Y_ℓ. In the first case, if v is the root of T, then $Y_\ell = Y(v)$. Otherwise, suppose the range in $Y_{\ell-1}$ that stores the parent v_p of node v begins at position z, and v is the i-th child of v_p. Let $c_j = \texttt{range_count}(Y(v), z, z + |Y(v)|, 1, j)$ for $1 \leq j \leq f_1$. Then, the range in Y_ℓ that stores $Y(v)$ is $[z + c_{i-1}, z + c_i]$. We then query $Y(v)$, and set $r_1 = \texttt{rank}_\tau(Y(v), r_1)$, $r_2 = \texttt{rank}_\tau(Y(v), r_2)$, $k = k - q'_{\tau-1}$, and recurse to v_τ. We present the following lemma, summarizing the arguments presented thus far:

Lemma 7. *The data structures described in this section allow us to answer range selection queries in $O((\lg n / \lg \lg n)^2)$ time.*

2.3 Handling Updates

In this section, we sketch the algorithm for updating the data structures. We start by describing how insertions are performed. First, we insert p into S_x and look up the rank, r_x, of p's x-coordinate in S_x. Next, we use the values stored in each internal node in T to find p's predecessor by y-coordinate, p'. We update the path from p' to the root of T. If a node v on this path splits, we must rebuild the ranking tree in the parent node v_p at level ℓ, and the dynamic string Y_ℓ.

Next, we update T in a top-down manner; starting from the root of T and following the path to the leaf storing p. Suppose that at some arbitrary node v in this path, the path branches to the j-th child of v, which we denote v_j.

We insert the symbol j into its appropriate position in Y_ℓ. After updating Y_ℓ — its leaves in particular — we insert the symbol j into the ranking tree $R(v)$, at position r_x, where r_x is the rank of the x-coordinate of p among the points in $T(v)$. As in T, each time a node splits in $R(v)$, we must rebuild the data structures in the parent node. We then update the nodes along the update path in $R(v)$ in a top-down manner: each update in $R(v)$ must be processed by all of the auxiliary data structures in each node along the update path. Thus, in each internal node, we must update the searchable partial sums data structures, as well as the matrix structures.

After updating the structures at level ℓ, we use Y_ℓ to map r_x to its appropriate rank by x-coordinate in $T(v_j)$. At this point, we can recurse to v_j. In the case of deletions, we follow the convention of Brodal et al. [5] and use node marking, and rebuild the entire data structure after $\Theta(n)$ updates. We present the following theorem; the technical details will be deferred to the full version of this paper.

Theorem 1. *Given a set S of points in the plane, there is a linear space dynamic data structure representing S that supports range selection queries for any range $[x_1, x_2]$ in $O((\lg n/\lg\lg n)^2)$ time, and supports insertions and deletions in $O((\lg n/\lg\lg n)^2)$ amortized time.*

3 Dynamic Arrays

In this section, we adapt Theorem 1 for problem of maintaining a dynamic array A of values drawn from a bounded universe $[1..\sigma]$. A query consists of a range in the array, $[i..j]$, along with an integer $k > 0$, and the output is the k-th smallest value in the subarray $A[i..j]$. Inserting a value into position i shifts the position of the values in positions $A[i..n]$ to $A[i+1..n+1]$, and deletions are analogous. We present the following theorem; the omitted proof uses standard techniques to reduce the number of word-sized pointers that are required to a constant:

Theorem 2. *Given an array $A[1..n]$ of values drawn from a bounded universe $[1..\sigma]$, there is an $nH_0(A) + o(n\lg\sigma) + O(w)$ bit data structure that can support range selection queries on A in $O(\frac{\lg n}{\lg\lg n}(\frac{\lg\sigma}{\lg\lg n}+1))$ time, and insertions into, and deletions from, A in $O(\frac{\lg n}{\lg\lg n}(\frac{\lg\sigma}{\lg\lg n}+1))$ amortized time. Thus, when $\sigma = O(\texttt{polylog}(n))$ this is $O(\frac{\lg n}{\lg\lg n})$ time for queries, and $O(\frac{\lg n}{\lg\lg n})$ amortized time for insertions and deletions.*

4 Concluding Remarks

In the same manner as Brodal et al. [5], the data structure we presented can also support orthogonal range counting queries in the same time bound as range selection queries. We note that the cell-probe lower bounds for the static range median and static orthogonal range counting match [17, 14], and — very recently — dynamic weighted orthogonal range counting was shown to have a cell-probe lower bound of $\Omega((\lg n/\lg\lg n)^2)$ query time for any data structure

with polylogarithmic update time [16]. In light of these bounds, it is likely that $O((\lg n/\lg\lg n)^2)$ time for range median queries is optimal for linear space data structures with polylogarithmic update time. However, it may be possible to do better in the case of dynamic range selection, when $k = o(n^\varepsilon)$ for any constant $\varepsilon > 0$, using an adaptive data structure as in the static case [14].

References

1. Alstrup, S., Husfeldt, T., Rauhe, T.: Marked ancestor problems. In: Proc. 39th Annual Symposium on Foundations of Computer Science. pp. 534–543. IEEE (1998)
2. Arge, L., Vitter, J.S.: Optimal external memory interval management. SIAM J. Comput. 32(6), 1488–1508 (2003)
3. Bose, P., Kranakis, E., Morin, P., Tang, Y.: Approximate range mode and range median queries. In: Proc. STACS. LNCS, vol. 3404, pp. 377–388. Springer (2005)
4. Brodal, G., Jørgensen, A.: Data structures for range median queries. In: Proc. ISAAC. LNCS, vol. 5878, pp. 822–831. Springer (2009)
5. Brodal, G., Gfeller, B., Jorgensen, A., Sanders, P.: Towards optimal range medians. Theoretical Computer Science (2010)
6. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms. McGraw-Hill Higher Education, 2nd edn. (2001)
7. Gagie, T., Puglisi, S., Turpin, A.: Range quantile queries: Another virtue of wavelet trees. In: Proc. SPIRE. pp. 1–6. Springer (2009)
8. Gfeller, B., Sanders, P.: Towards optimal range medians. In: Proc. ICALP. LNCS, vol. 5555, pp. 475–486. Springer (2009)
9. Gil, J., Werman, M.: Computing 2-d min, median, and max filters. IEEE Transactions on Pattern Analysis and Machine Intelligence 15(5), 504–507 (1993)
10. Har-Peled, S., Muthukrishnan, S.: Range medians. In: Proc. of the European Symposium on Algorithms. LNCS, vol. 5193, pp. 503–514. Springer (2008)
11. He, M., Munro, J.I.: Succinct representations of dynamic strings. In: String Processing and Information Retrieval. pp. 334–346. LNCS 6393, Springer (2010)
12. He, M., Munro, J.I.: Space Efficient Data Structures for Dynamic Orthogonal Range Counting. In: Proc. WADS. pp. 500–511. LNCS 6844, Springer (2011)
13. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. SFCS. pp. 549–554 (1989)
14. Jørgensen, A., Larsen, K.: Range selection and median: Tight cell probe lower bounds and adaptive data structures. In: Proc. SODA (2011)
15. Krizanc, D., Morin, P., Smid, M.: Range mode and range median queries on lists and trees. Nordic Journal of Computing 12, 1–17 (2005)
16. Larsen, K.: The cell probe complexity of dynamic range counting. Arxiv preprint arXiv:1105.5933 (2011)
17. Pătraşcu, M.: Lower bounds for 2-dimensional range counting. In: Proc. 39th ACM Symposium on Theory of Computing (STOC). pp. 40–46 (2007)
18. Petersen, H.: Improved bounds for range mode and range median queries. In: Proc. SOFSEM. LNCS, vol. 4910, pp. 418–423. Springer (2008)
19. Petersen, H., Grabowski, S.: Range mode and range median queries in constant timeand sub-quadratic space. Inf. Process. Lett. 109, 225–228 (2009)
20. Raman, R., Raman, V., Rao, S.: Succinct dynamic data structures. In: Proc. WADS. pp. 426–437 (2001)