A Linear Space Data Structure for Orthogonal Range Reporting and Emptiness Queries

Yakov Nekrich*

Abstract

In this paper we present a linear space dynamic data structure for orthogonal range reporting and emptiness queries. This data structure answers range reporting queries in time $O(\log n \log\log n + k \log^{\varepsilon} n)$ for any $\varepsilon > 0$ and k the size of the answer. Our data structure also supports emptiness and one-reporting queries in time $O(\log n \log\log n)$.

1 Introduction

The orthogonal range reporting problem is to store a set P of points on the plane so that for an arbitrary rectangle $Q = [a, b] \times [c, d]$ all points in $S \cap Q$ can be reported efficiently. Special cases of orthogonal range reporting are *emptiness queries* that answer the question "is $S \cap Q = \emptyset$?" and one-reporting queries that report an arbitrary point from $S \cap Q$ if $S \cap Q \neq \emptyset$.

There are data structures that answer range reporting queries in $O(\log n + k)$ time and $O(n \log^{\varepsilon} n)$ space ([5] in the static case; [9] in the dynamic case). However, the best known linear space data structures for this problem require either superpolylogarithmic query time or a penalty for each point in the answer. The static data structure from [5] supports queries in $O(\log n + k \log \frac{2n}{k+1})$ time. In the dynamic case, the data structure of [8] supports range reporting queries in $O(\sqrt{n \log n} + k)$ and updates in $O(\log n)$ time; the data structure of [5] supports queries in $O((k+1) \log^2 \frac{2n}{k+1})$ time and updates in $O(\log^2 n)$ time.

In this paper we present a dynamic data structure that answers orthogonal range reporting queries in $O(\log n \log\log n + k \log^{\varepsilon} n)$ time where k is the size of the answer. Our data structure supports emptiness queries in $O(\log n \log\log n)$ time; updates are supported in $O(\log^3 n \log\log n)$ time. Thus our dynamic data structure almost matches the upper bound of Chazelle [5] for the range reporting queries in the static case, and achieves almost-optimal time for the emptiness and one-reporting queries.

Our data structure is based on the standard range tree technique [3] that reduces the two-dimensional range reporting to one-dimensional range reporting. The space-efficient solution is obtained by a compact representation of the data structures for one-dimensional queries. Suppose that coordinates of n_v points must be stored in a node v of the range tree; we replace the coordinates by labels from $[1, O(n_v)]$. The data structure that answers one-dimensional queries for the labels stored in v can be implemented with $O(n_v)$ bits using the approach of [4].

*Dept. of Computer Science, University of Bonn, yasha@cs.uni-bonn.de

2 Preliminaries

Let P be the set of points stored in the data structure; let P_x and P_y be the sets of x-coordinates and y-coordinates of the points in P.

The space-efficient data structure presented in this paper uses two important techniques: dynamic range reduction to extended rank space ([9]) and the approach of the compact data structure [4].

The dynamic range reduction to extended rank space is based on maintaining a bijective order-preserving mapping $f : S \to \hat{S}$ for a dynamic set S; f assigns to each element $e \in S$ a label $f(S)$ from a polynomially bounded universe. In this paper we strengthen this condition and require that $f(e)$ for all $e \in S$ belong to the universe of linear size, i.e. $\forall e \in S : f(e) \in [1, O(|S|)]$ and $e_1 < e_2 \Rightarrow f(e_1) < f(e_2)$. If a new element e is inserted into S we assign it a label $f(S)$ and change the values of $f(e_1), f(e_2), \ldots, f(e_s)$ for some $e_1, e_2, \ldots e_s \in S$ so that the order-preserving property of f is maintained. In this case we say that elements $e_1, \ldots, e_s$ are f-*moved.* If an element $e \in S$ is deleted, some elements of S can also be f-moved.

We can efficiently maintain an order preserving mapping $f : S \to [1, O(|S|)]$ using the sparse table technique of [7], [12] (see e.g., [2] for applications of this technique to cache-oblivious B-trees). The following Lemma is a reformulation of the result in [12]

Lemma 1 *We can maintain an order preserving mapping $f : S \to [1, 2|S|[$, so that in the case of an update $O(\log^2 |S|)$ elements of S must be f-moved.*

We maintain two order-preserving mappings $f_x : P_x \to \hat{P}_x$ and $f_y : P_y \to \hat{P}_y$ so that all elements of $\hat{P}_x$ and $\hat{P}_y$ belong to the universe of linear size, $\hat{P}_x \subset [1, O(|P_x|)]$ and $\hat{P}_y \subset [1, O(|P_y|)]$.

Let $\hat{P} = \{(f_x(x), f_y(y)) | (x, y) \in P\}$. A two-dimensional range reporting query $[a, b] \times [c, d]$ on elements of P can be reduced to a query $[\hat{a}, \hat{b}] \times [\hat{c}, \hat{d}]$ on elements of $\hat{P}$ due to the order-preserving property of f_x and f_y: We set $\hat{a} = f_x(succ(a, P_x))$, $\hat{b} = f_x(pred(b, P_x))$, $\hat{c} = f_y(succ(c, P_y))$, and $\hat{d} = f_y(pred(d, P_y))$; here and further $pred(a, S) = \max\{x \in S | x \leq a\}$ and $succ(a, S) = \min\{x \in S | x \geq a\}$. Then $(x, y) \in (Q \cap P) \Leftrightarrow (\hat{x}, \hat{y}) \in (\hat{Q} \cap \hat{P})$. The range reduction technique described above can also be applied to one-dimensional range reporting queries. This approach can be viewed as an extension of the well-known reduction to rank space technique.

Another component of our method is the compact data structure of Blandford and Blelloch[4] that allows us to store a set $S \subset [1, N]$ using $O(|S| \log \frac{N}{|S|})$ bits so that main search operations are supported efficiently. The following reduction is shown in [4]: Given a tree-based data structure D with n elements from the universe $[1, N]$ that uses $O(n)$ words of $\log N$ bits and supports predecessor searches in time $O(t(n))$ and updates in time $O(u(n))$, we can construct a data structure D' that uses $O(n \log \frac{N}{n})$ bits and supports predecessor searches in time $O(t(n))$ and updates in time $O(u(n))$. Using a similar approach we can prove:

Lemma 2 *There exists a data structure D for the elements from the universe $[1, O(n)]$ where n is the number of elements in D; D uses $O(n)$ bits and supports updates, predecessor, and successor queries in time $O(\log \log n)$.*

This Lemma will be proven in the full version of this paper.

3 Description of the Data Structure

Our data structure is based on the standard range trees introduced by Bentley [3] and used in a number of other data structures. We build a tree T_x over P_x; T_x is implemented as a WBB tree [1] with branching parameter 4 and leaf parameter 1, i.e. each leaf contains one element, and each internal node is of degree ρ, $1 \leq \rho \leq 16$. A WBB tree over n elements is of height $O(\log n)$, and a node of height l has between $4^l/2$ and $2 \cdot 4^l$ leaf descendants (see [1] for details). All elements of T_x are stored in the leaves of T_x. We associate a range with each node v of T_x: the range associated to a leaf contains this leaf, and all ranges associated to leaves of T_x are disjoint; the range associated to an internal node v is a union of ranges associated to its children. We say that a point p belongs to a node v if its x-coordinate belongs to the range of v. Let Y_v be the set of y-coordinates of points that belong to a node v. For each node v, except of the root, we store a data structure D_v that contains y-coordinates of all points that belong to v. The data structure D_v allows us to find for each element p in v: the predecessor $pred(p, Y_v)$ and the successor $succ(p, Y_v)$ of p in Y_v; for each child u of v, $pred(p, Y_u)$ and $succ(p, Y_u)$. The y-coordinates of points that belong to the root r of T_x are stored in a binary tree T_r, so that $pred(a, P_y)$ and $succ(a, P_y)$ can be found for any a in $O(\log n)$ time.

Suppose a query $Q = [a, b] \times [c, d]$ must be answered. The search procedure starts in the root r of T_x. For each visited node v, if the range of v is a subset of $[a, b]$, we report all points in v with y-coordinates in $[c, d]$. Otherwise, if D_v contains points with y-coordinates in $[c, d]$ we visit all children of v whose range intersects with $[a, b]$. For each node v of T_x visited by our search procedure, we find $c_v = succ(c, Y_v)$ and $d_v = pred(d, Y_v)$. We can find c_r and d_r using T_r in $O(\log n)$ time. If c_v, d_v are known and u is a child of v, we can find c_u and d_u using D_v: $c_u = succ(c_v, Y_u)$ and $d_u = pred(d_v, Y_u)$ for $Y_u \subset Y_v$. Since the number of nodes visited by the search procedure is $O(\log n)$ the search time is $O(\log n \cdot t(n))$, if data structures D_v answer predecessor queries in time $t(n)$ and if we ignore the time to report the points in the answer.

4 Space-efficient Implementation

In this section we describe a compact representation of data structures D_v. Our solution consists of two key components: dynamic range reduction to extended rank space and the compact representation of a data structure in a bounded universe in the spirit of [4].

Firstly, we apply the reduction to extended rank space to the elements of P, i.e. we store the elements of range-reduced set $\hat{P}$ in the data structure. W.l.o.g., we assume that all points have different y-coordinates. Then each point in P can be identified by its y-coordinate in $\hat{P}$ using a table of size $O(n)$. This allows us to store only the y-coordinates of points in the nodes of T_x.

Second, we apply reduction to extended rank space in each node of the tree T_x but in the root node r: If v is a child of r, there is a mapping f_v that assigns to each element $e \in P_y$ stored in v a v-label $f_v(e) \in [1, O(|Y_v|)]$. If u is a child of some non-root node v, there is a mapping f_u that assigns to each element stored in node u a u-label $f_u(e) \in [1, O(|Y_u|)]$. Thus f_u maps the v-labels of elements in u (that belong to the universe $[1, O(|Y_v|)]$) to u-labels (that belong to the universe $[1, O(|Y_u|)]$). Observe that each v-label together with a node v in which it is stored uniquely identifies a point $p \in P$ since all f_u are bijective. Further in this paper we denote by Y_v the set of v-labels stored in a node v; we denote by $Y_{v,u}$ the set of v-labels of elements that belong to a child u of v.

We say that element $e \in Y_u$ *corresponds to an element* $e' \in Y_v$ (e is the *corresponding element of* e'), if there is a path $v, n_1, n_2, \ldots, n_s, u$ between nodes v and u in T_x such that $e_{n_1} = f_{n_1}(e'), e_{n_2} = f_{n_2}(e_{n_1}), \ldots, e = f_u(e_{n_s})$ or $e_{n_s} = f_u^{-1}(e), \ldots, e_{n_1} = f_{n_2}^{-1}(e_{n_2}), e' = f_{n_1}^{-1}(e_{n_1})$. In other words, $e \in Y_u$ corresponds to $e' \in Y_v$ if e and e' are labels of the same point in P.

In the next section we will show that it is possible to store both f_v and D_v with $O(m)$ bits, where m is the number of elements in Y_v.

Suppose the query $[a, b] \times [c, d]$ is to be answered; applying reduction to extended rank space we reduce it to a query $[\hat{a}, \hat{b}] \times [\hat{c}, \hat{d}]$ in $\hat{P}$. The search procedure, described in the previous section, starts at the root r of T_x. Using T_r the predecessor of d in Y_r, $d_r = pred(d, Y_r)$, and the successor of c in Y_r $c_r = succ(c, Y_r)$ can be found in $O(\log n)$ time. If the search procedure visits a node u that is a child of some node v, we find $succ(c_v, Y_{v,u})$, $pred(d_v, Y_{v,u})$ and set $c_u = f_u(succ(c_v, Y_{v,u}))$, $d_u = f_u(pred(d_v, Y_{v,u}))$. The predecessor and successor queries can be answered in $O(\log\log n)$ time using Lemma 2.

If the range of node u is contained in $[a, b]$, we identify all elements in Y_u between c_u and d_u. A label $e \in Y_u$ is in the interval $[c_u, d_u]$ if and only if the corresponding element $e_r \in Y_r$ is in the interval $[\hat{c}, \hat{d}]$. In this way we can find the labels of all points in $[\hat{a}, \hat{b}] \times [\hat{c}, \hat{d}]$ in $O(\log n + k)$ time. For each $e \in Y_u$, $c_u \le e \le d_u$, we determine its "original" coordinates, i.e. we find the element e_r in Y_r that corresponds to e and a point in $\hat{P}$ with y-coordinate e_r.

To find e_r for some label $e \in Y_u$, we apply inverse range reduction. That is, we compute $e_{v_1} = f_u^{-1}(e)$, $e_{v_2} = f_{v_1}^{-1}(e_{v_1}), \ldots, e_r = f_{v_s}^{-1}(e_{v_s})$ where $v_1, v_2, \ldots, v_s$ are nodes in T_x on the path from u to the root r. It will be shown in Lemma 3 that each $f_v^{-1}(e)$ can be computed in $O(\log\log n)$ time. This would incur a penalty of $O(\log n \log\log n)$ for each point in the answer. The penalty can be sufficiently reduced if we store for the elements in some nodes u the corresponding elements in $Y_{u'}$ where u' is an ancestor of u on some higher level (we say that a node u is on level d if the path from u to the root consists of d edges). For each $e \in Y_u$ and for every node u on level $l = t\lfloor\sqrt{\log n}\rfloor$, $t = 1, 2, \ldots$, we store the corresponding element of $Y_{u'}$ where u' is the ascendant of u on level $(t-1)\lfloor\sqrt{\log n}\rfloor$. Suppose we look for the element e_r corresponding to some $e \in Y_u$ for some node u. Then we can find in $O(\sqrt{\log n}\log\log n)$ time $e_{u'}$ that corresponds to the ancestor u' of u on level $t\lfloor\sqrt{\log n}\rfloor$ for some t. Once $e_{u'}$ is known, we can find e_r in time $O(\sqrt{\log n}\log\log n)$. It will be shown in Lemma 3 that each data structure in a node u that stores for each $e \in Y_u$ the corresponding element $e' \in Y_{u'}$ where u' is situated $\lfloor\sqrt{\log n}\rfloor$ levels above u requires $O(|Y_u|\sqrt{\log n})$ bits. Since the total number of elements in all such data structures is $O(n\sqrt{\log n})$, these additional data structures use $O(n \log n)$ bits. We can reduce the penalty for each point in the answer from $O(\sqrt{\log n}\log\log n)$ to $O(\log^{1/4} n \log\log n)$ using $O(n \log n)$ additional bits: For each $e \in Y_u$ where u is on level $t\lfloor\log^{1/4} n\rfloor$ we store the corresponding element $e_r \in Y_{u'}$ for u' on level $(t-1)\lfloor\log^{1/4} n\rfloor$. For each $e \in Y_u$ where u is on level $t\lfloor\log^{3/4} n\rfloor$ we store the corresponding element $e_r \in Y_{u''}$ for u'' on level $(t-1)\lfloor\log^{3/4} n\rfloor$. For each $e \in Y_u$ for some node u we compute the corresponding $e_{u'}$ for u' on level $t_1\lfloor\log^{1/4} n\rfloor$, then compute $e_{u''}$ that corresponds to $e_{u'}$ and belongs to a node on level $t_2\lfloor\log^{1/2} n\rfloor$, then compute $e_{u'''}$ that corresponds to $e_{u''}$ and belongs to a node on level $t_3\lfloor\log^{3/4} n\rfloor$ for some t_1, t_2, t_3; finally, we compute e_r that corresponds to $e_{u'''}$. The four above steps take time $O(\log^{1/4} n \log\log n)$.

We can repeat the above trick p times and obtain a data structure that uses $O(2^p n \log n)$ bits and answers range reporting queries in $O(\log n \log\log n + k(2^p)\log^{1/2^p} n \log\log n)$ time. By choosing $p = \log(1/\varepsilon')$ for some $\varepsilon' < \varepsilon$, we obtain the time bounds stated in the introduction.

5 Analysis

Lemma 3 *Given a set $S \subset U = [1, O(m)]$ such that $|S| = m$ and an order-preserving mapping $f : S \to [1, N_m]$ where N_m is a function of m, S can be stored in a data structure A using $O(m \log \frac{N_m}{m})$ bits so that:*
(a) Given $e \in U$ we can determine whether $e \in S$ and if $e \in S$ find a pointer to e in A in $O(1)$ time
(b) Given a pointer to element $e \in S_i$, $f(e)$ can be computed and changed in $O(\log\log n)$ time
(c) Given a pointer to $pred(e, S)$ for $e \notin S$ and $f \in [1, O(N_m)]$ such that $f(pred(e, S)) < f' < f(succ(e, S))$ a new element e with $f(e) = f'$ can be inserted into A in $O(\log m)$ time. Given a pointer to $e \in S$, e can be deleted from A in $O(\log m)$ time.

Proof. The data structure A can be constructed using a slight modification of the approach of [4] and some additional tricks.

We maintain the elements of S as a doubly-linked list of blocks L_e. Each node in the list L_e stores a block B_i that consists of elements $e_{i,1} < e_{i,2} < \ldots < e_{i,b_i}$. Blocks B_i are organized in exactly the same way as in [4]. We store the value of the first element in a block with $O(\log m)$ bits; other values are *difference coded*, i.e. we store the differences $e_{i,j} - e_{i,j-1}$ between consecutive elements in the block. All differences are encoded using a logarithmic code, so that the difference $e_{i,j} - e_{i,j-1}$ can be encoded with $\log(e_{i,j} - e_{i,j-1})$ bits. We choose the number of elements in a block B_i so that $\sum_{j=2}^{b_i} \log(e_{i,j} - e_{i,j-1}) = O(\log m)$. For each block B_i we store the number of elements in B_i with $O(\log\log m)$ bits (since $b_i = O(\log m)$). We can insert and delete elements into a block, split a block into two blocks, and merge two blocks in $O(1)$ time; we can also find the predecessor and the successor of $v \in U$ in a block B_i, and the k-th element in a block B_i in $O(1)$ time. (see

[4] for the proof). The values $f(e)$ for $e \in S$ are also stored in a list of blocks L_f organized in the same way as L_e but with blocks of size $O(\log N_m)$. Observe that elements $e_{i,1}, \ldots, e_{i,b_i}$ are stored in one block in L_e, but $f(e_{i,1}) \ldots, f(e_{i,b_i})$ may be stored in a number of different blocks of L_f. We will describe in the full version of this paper how for any $e \in L_e$ $f(e) \in L_f$ can be found.

All blocks L_e and L_f take $O(m \log \frac{N_m}{m})$ bits and updates of individual blocks in L_e and L_f can be performed in $O(1)$ time given a pointer to a (predecessor) of the updated element (a proof can be found in [4]). A complete description of update operations will be given in the full version of this paper. A pointer to an element $e \in S$ consists of a pointer to the block B_i in which e is stored, and the index of e in B_i.

We split the universe U into $O(m/\log m)$ intervals of size $\log m$, and we store an array $V[]$ with entries that correspond to those intervals: $V[i]$ contains a pointer to the first element in L_e that belongs to interval $[i \log m, (i+1) \log m]$ or $NULL$ if $[i \log m, (i+1) \log m] \cap S = \emptyset$. If some $e \in S \cap [i \log m, (i+1) \log m]$ is stored in a block B_i, all other elements in $S \cap [i \log m, (i+1) \log m]$ are stored in a constant number of blocks that follow B_i in L_e. To determine for some $v \in U$, whether v belongs to S, we check a constant number of blocks after $V[v/\log m]$.

□

Now we give a brief sketch of the space and update time analysis of our data structure. A detailed description will be given in the full version of this paper. Using Lemma 3 we can store for each node u the set Y_u and the mapping f_u using $O(|Y_u|)$ bits. The inverse mappings $f^{-1}(u)$ can also be stored with $O(n)$ bits. We can store for all elements in some node u on level l the corresponding elements on level l' using $O(n(l-l'))$ bits. Therefore our data structure uses $O(n \log n)$ bits or $O(n)$ words of size $\log n$.

When a new point (x, y) is inserted into P, we update the information for all nodes T_x whose range contains x. For every such node u in T_x we update the data stored in the data structure D_u and the mapping f_u. According to Lemma 1 we can update f_u by changing the labels of $O(\log^2 n)$ elements in Y_u. This means we must delete and insert $O(\log^2 n)$ elements in D_u, and this takes $O(\log^2 n \log \log n)$ time according to Lemma 2. Since there are $O(\log n)$ nodes u whose range contains x, an insertion takes $O(\log^3 n \log \log n)$ time. Deletions can be processed in the same way as insertions.

We also rebalance the tree T_x: rebalancing in WBB tree is done by splitting nodes, and if a node u contains w elements it is rebuilt only once in a series of $\Omega(w)$ updates affecting the node u (see [1] for details). When u is split, we rebuild all data structures in all children of u in $O(w \log n)$ time. Since every update operation affects $O(\log n)$ nodes in a WBB tree, the amortized cost incurred by rebalancing is $O(\log^2 n)$. Thus our data structure can be updated in $O(\log^3 n \log \log n)$ time.

6 Conclusion

We presented a linear space dynamic data structure that answers emptiness and one-reporting queries in $O(\log n \log \log n)$ time. Existence of the linear space dynamic data structure with optimal $O(\log n)$ query time is an interesting open question.

References

[1] L. Arge, J. S. Vitter, "Optimal External Memory Interval Management", SIAM J. on Computing 32(6), pp. 1488-1508, 2003.

[2] M. A. Bender, E. D. Demaine, M. Farach-Colton, "Cache-Oblivious B-Trees", Proc. 41st FOCS 2000, 399-409.

[3] J. L. Bentley, "Multidimensional Divide-and-Conquer", Commun. ACM 23, pp. 214-229 (1980).

[4] D. K. Blandford, G. E. Blelloch, "Compact Representations of Ordered Sets", Proc. SODA 2004, pp. 11-19.

[5] B. Chazelle, "A Functional Approach to Data Structures and its Use in Multidimensional Searching", SIAM J. on Computing, 17, pp. 427-462, 1988.

[6] P. Elias, "Universal Codeword Sets and Representations of the Integers", IEEE Transactions on Information Theory, 21, pp. 194–203, 1975.

[7] A. Itai, A. G. Konheim, M. Rodeh, "A Sparse Table Implementation of Priority Queues", Proc. 8th ICALP 1981,417-431.

[8] M. Van Kreveld and M.H. Overmars " Divided K-d Trees", Algorithmica 6(6),pp. 840-858, 1991.

[9] Y. Nekrich, "Space Efficient Dynamic Orthogonal Range Reporting" Proc. Symposium on Computational Geometry 2005, pp. 306-313.

[10] J. I. Munro, V. Raman, A. J. Storm, "Representing Dynamic Binary Trees Succinctly", Proc. SODA 2001, pp. 529-536

[11] R. Raman, V. Raman, S. S. Rao, "Succinct Dynamic Data Structures", Proc. WADS 2001, pp. 426-437.

[12] D. E. Willard, "A Density Control Algorithm for Doing Insertions and Deletions in a Sequentially Ordered File in Good Worst-Case Time", Information and Computation 97, pp. 150-204, 1992.