# Simple and Semi-Dynamic Structures for Cache-Oblivious Planar Orthogonal Range Searching

Lars Arge*
Department of Computer Science
University of Aarhus
IT-Parken, Aabogade 34
DK-8200 Aarhus N
Denmark
large@daimi.au.dk

Norbert Zeh†
Faculty of Computer Science
Dalhousie University
6050 University Ave
Halifax, NS B3H 1W5
Canada
nzeh@cs.dal.ca

## ABSTRACT

In this paper, we develop improved cache-oblivious data structures for two- and three-sided planar orthogonal range searching. Our main result is an optimal static structure for two-sided range searching that uses linear space and supports queries in $\mathcal{O}(\log_B N + T/B)$ memory transfers, where $B$ is the block size of any level in a multi-level memory hierarchy and $T$ is the number of reported points. Our structure is the first linear-space cache-oblivious structure for a planar range searching problem with the optimal $\mathcal{O}(\log_B N + T/B)$ query bound. The structure is very simple, and we believe it to be of practical interest.

We also show that our two-sided range search structure can be constructed cache-obliviously in $\mathcal{O}(N \log_B N)$ memory transfers. Using the logarithmic method and fractional cascading, this leads to a semi-dynamic linear-space structure that supports two-sided range queries in $\mathcal{O}(\log_2 N + T/B)$ memory transfers and insertions in $\mathcal{O}(\log_2 N \cdot \log_B N)$ memory transfers amortized. This structure is the first (semi-)dynamic structure for any planar range searching problem with a query bound that is logarithmic in the number of elements in the structure and linear in the output size.

Finally, using a simple standard construction, we also obtain a static $\mathcal{O}(N \log_2 N)$-space structure for three-sided range searching that supports queries in the optimal bound of $\mathcal{O}(\log_B N + T/B)$ memory transfers. These bounds match the bounds of the best previously known structure for this problem; but our structure is much simpler, simple enough, we believe, to be of practical interest.

## Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Geometrical problems and computations*

## General Terms

Algorithms, Design, Theory

## Keywords

Memory hierarchies, cache-obliviousness, data structures, range searching

## 1. INTRODUCTION

The memory systems of modern computers are becoming increasingly complex; they consist of a hierarchy of several levels of cache, main memory, and disk. The access times of different levels of memory often vary by orders of magnitude, and, to amortize the large access times of memory levels far away from the processor, data is normally transferred between levels in large blocks. Thus, it is important to design algorithms that are sensitive to the architecture of the memory system and have a high degree of locality in their memory-access patterns.

When working in the traditional RAM model of computation, one assumes a flat memory system with uniform access time; therefore, algorithms for the RAM model often exhibit low memory-access locality and are thus inefficient in a hierarchical memory system. Although a lot of work has recently been done on algorithms for a two-level memory model, introduced to model the large difference between the access times of main memory and disks, relatively little work has been done in models of multi-level memory hierarchies. One reason for this is the many parameters used in such models to describe the different levels of memory in the hierarchy. The *cache-oblivious* model was introduced as a way of obtaining algorithms that are efficient in arbitrary memory hierarchies without the use of complicated multi-level memory models.

In this paper, we develop improved cache-oblivious data structures for two- and three-sided planar orthogonal range

*Supported in part by the US Army Research Office through grant W911NF-04-1-0278 and by an Ole Rømer Scholarship from the Danish National Science Research Council.
†Supported by the Natural Sciences and Engineering Research Council of Canada and the Canadian Foundation for Innovation.


*SCG'06,* June 5–7, 2006, Sedona, Arizona, USA.

searching. General (*four-sided*) planar orthogonal range searching is the problem of storing a set $S$ of $N$ points in the plane so that all $T$ points lying in an axis-parallel query rectangle $Q = [x_l, x_r] \times [y_b, y_t]$ can be reported efficiently. In *three-sided* range searching, every query satisfies $y_t = \infty$; in *two-sided* range searching, every query satisfies $y_t = \infty$ and either $x_l = -\infty$ or $x_r = \infty$.

## 1.1 Model of Computation

In the two-level *I/O-model* (or *external-memory model*), introduced by Aggarwal and Vitter [3], the memory hierarchy consists of an internal memory of size $M$ and an arbitrarily large external memory partitioned into blocks of size $B$. An *I/O*, or *memory transfer*, transfers one block between internal and external memory. Computation can occur only on data present in internal memory. The complexity of an algorithm in this model (an external-memory algorithm) is measured in terms of the number of memory transfers it performs, as well as the amount of external memory it uses.

In the *cache-oblivious model*, introduced by Frigo et al. [24], algorithms are described in the RAM model, but are analyzed in the two-level I/O-model. It is assumed that, when an algorithm accesses an element that is not stored in internal memory, the relevant block is automatically transferred into internal memory. If the internal memory is full, an *optimal paging strategy* replaces the *ideal* block in internal memory based on the future accesses of the algorithm. Often, it is also assumed that $M > B^2$ (the *tall-cache* assumption). So, informally, cache-oblivious algorithms run in the I/O-model, but cannot make use of $M$ and $B$. Because an analysis of a cache-oblivious algorithm in the two-level model must hold for any block and main memory size, it holds for *any* level of an arbitrary memory hierarchy [24]. As a consequence, a cache-oblivious algorithm that is optimal in the two-level model is optimal on *all* levels of an arbitrary multi-level hierarchy.

## 1.2 Previous Results

Range searching has been studied extensively in the RAM model; refer, for example, to a recent survey [2] for a discussion of results. In the I/O-model, the B-tree [9, 23] supports one-dimensional range queries in the optimal bound of $\mathcal{O}(\log_B N + T/B)$ memory transfers and uses linear space. In two dimensions, $\Theta(N \frac{\log_B N}{\log \log_B N})$ space is necessary to obtain an $\mathcal{O}(\log_B N + T/B)$ query bound [8, 21]. The external range-tree structure [8] achieves these bounds. If only linear space is used, then $\Theta(\sqrt{N/B} + T/B)$ memory transfers, as achieved by the kd-B tree [26, 29], are needed to answer a query. Three-sided (and thus two-sided) queries, on the other hand, can be answered in $\mathcal{O}(\log_B N + T/B)$ memory transfers using the external priority search tree [8], which uses linear space. This structure can also be updated in $\mathcal{O}(\log_B N)$ memory transfers. Refer to recent surveys [4, 30] for further results in the I/O-model and in hierarchical memory models.

In the cache-oblivious model, Frigo et al. [24] developed efficient algorithms for sorting, Fast Fourier Transform, and matrix multiplication. Subsequently, a number of other results have been obtained in this model [1, 5, 6, 7, 10, 11, 12, 13, 14, 17, 18, 19, 20, 28], among them several cache-oblivious B-tree structures with $\mathcal{O}(\log_B N)$ search and update bounds [12, 13, 14, 20, 28]. Several of these structures also support one-dimensional range queries in $O(\log_B N + T/B)$ memory transfers [13, 14, 20] (but at an increased update cost of $\mathcal{O}(\log_B N + \frac{1}{B}\log_2^2 N) = \mathcal{O}(\log_B^2 N)$ amortized memory transfers). A key ingredient in all the cache-oblivious B-tree structures is the so-called van-Emde-Boas layout for storing a balanced constant-degree tree of size $\mathcal{O}(N)$ in memory so that any root-to-leaf path can be traversed cache-obliviously in $\mathcal{O}(\log_B N)$ memory transfers [27].

In [17], an algorithm for batched planar orthogonal range searching was developed. This algorithm answers a set of $\mathcal{O}(N)$ queries in $\mathcal{O}(\frac{N}{B}\log_{M/B}\frac{N}{B} + T/B)$ memory transfers, where $T$ is the combined size of the answers. For online planar orthogonal range searching, Agarwal et al. [1] were the first to develop efficient cache-oblivious structures. They developed a cache-oblivious version of a kd-tree that uses linear space and answers range queries in the plane in $\mathcal{O}(\sqrt{N/B} + T/B)$ memory transfers. Point insertions and deletions take $\mathcal{O}\big(\frac{\log_2 N}{B}\log_{M/B} N\big) = \mathcal{O}(\log_B^2 N)$ memory transfers amortized. The structure can be extended to $d$ dimensions. They also developed a cache-oblivious version of a two-dimensional range tree that answers planar range queries in $\mathcal{O}(\log_B N + T/B)$ memory transfers, but using $\mathcal{O}(N \log_2^2 N)$ space. The central part of this structure is an $\mathcal{O}(N \log_2 N)$-space structure for answering three-sided range queries in $\mathcal{O}(\log_B N + T/B)$ memory transfers. The analysis of the range tree structure (or rather, the structure for three-sided queries) requires that $B = 2^{2^c}$, for some non-negative integer constant $c$. Subsequently, Arge et al. [6] developed cache-oblivious structures for planar orthogonal range counting—that is, for counting the points in the query range rather than reporting them—which answer queries in $\mathcal{O}(\log_B N)$ memory transfers. They developed both an $\mathcal{O}(N \log_2 N)$-space and a linear-space structure, but the latter assumes that bit manipulations of pointers and counters are allowed. Using the first structure, they developed a cache-oblivious structure for three-sided range searching with bounds matching the structure of Agarwal et al. [1], but without any assumptions about $B$. Using this structure, they obtained an improved structure for general (four-sided) queries that answers queries in $\mathcal{O}(\log_B N + T/B)$ memory transfers and uses $\mathcal{O}(N \frac{\log_2^2 N}{\log_2 \log_2 N})$ space, without making any assumptions about $B$. To the best of our knowledge, there is currently no (semi-)dynamic structure that can answer queries in $\mathcal{O}(\log N + T/B)$ memory transfers and can be updated in $\mathcal{O}(\log^c n)$ memory transfers, for any constant $c$.

## 1.3 Our Results

In this paper, we develop improved cache-oblivious data structures for two- and three-sided range searching in the plane. Our main result, described in Section 2, is an optimal static structure for two-sided range searching that uses linear space and supports queries in $\mathcal{O}(\log_B N + T/B)$ memory transfers. The structure is the first linear-space cache-oblivious structure for a planar range searching problem with the optimal $\mathcal{O}(\log_B N + T/B)$ query bound. The structure is very simple, consisting of a linear layout (sequence) of the point set $S$ and a search tree over this sequence. Every query is answered by scanning a subsequence of the layout whose length is $\mathcal{O}(T)$. The search tree is used to locate the start position of this subsequence. We believe that the structure will be of practical interest.

We also show that our two-sided range searching structure can be constructed cache-obliviously in $\mathcal{O}(N \log_B N)$

memory transfers. As described in Section 4, using the logarithmic method and fractional cascading, this leads to a semi-dynamic linear-space structure that supports two-sided range queries in $\mathcal{O}(\log_2 N + T/B)$ memory transfers and insertions in $\mathcal{O}(\log_2 N \cdot \log_B N)$ memory transfers amortized. This structure is the first (semi-)dynamic cache-oblivious structure for any planar range searching problem with a linear output cost and logarithmic overhead per query. As already mentioned above, the only previously known dynamic structure that reports the $T$ points contained in a given query in $\mathcal{O}(T/B)$ memory transfers had a query overhead of $\mathcal{O}(\sqrt{N/B})$. The only previous cache-oblivious range search structure that achieves a logarithmic query bound, while performing a linear number of memory transfers in the output size, is static; it seems impossible to make it dynamic using the logarithmic method, while maintaining a logarithmic query bound.

Finally, using a simple standard construction described in Section 3, we also obtain a static $\mathcal{O}(N \log_2 N)$-space structure for three-sided range searching that supports queries in $\mathcal{O}(\log_B N + T/B)$ I/Os. These bounds match the bounds of the best previously known structure [6], but our structure is much simpler. It consists of a binary tree, where each internal node stores two two-sided range search structures for the points in its descendant leaves. Every three-sided query can then be answered by answering two two-sided queries. In particular, our solution does not require the complicated counting structure of [6]. We believe that our structure is of practical interest.

# 2. STATIC TWO-SIDED PLANAR RANGE SEARCHING

In this section, we describe a simple linear-space cache-oblivious structure for storing a set, $S$, of $N$ points in the plane such that two-sided range queries can be answered in optimal $\mathcal{O}(\log_B N + T/B)$ memory transfers. For simplicity, we assume that no two points in $S$ have the same $x$-coordinate.

Our structure consists of two parts: a sequence $\mathcal{L}$ of length $\mathcal{O}(N)$, called the *layout* of $S$, that stores the points in $S$, possibly with duplication; and a balanced binary search tree $Y$ on a subset of the $y$-coordinates of the points in $S$, laid out in memory using the van-Emde-Boas layout [27]. Each leaf of $Y$ stores a pointer into $\mathcal{L}$. A query $(-\infty, x_q] \times [y_q, +\infty)$ is performed by searching for $y_q$ in $Y$, following a pointer from a leaf in $Y$ into $\mathcal{L}$, and scanning forward in $\mathcal{L}$ until a point with $x$-coordinate greater than $x_q$ is found. The search in $Y$ incurs $\mathcal{O}(\log_B N)$ memory transfers, and we show in Section 2.1 that $\mathcal{O}(T)$ points are scanned in $\mathcal{L}$ to answer the query. So the total query cost is $\mathcal{O}(\log_B N + T/B)$ memory transfers. In Section 2.2, we discuss how $\mathcal{L}$ can be constructed in $\mathcal{O}(N \log_B N)$ memory transfers. This proves the main result of this section:

**Theorem 1** *There exists a linear-space cache-oblivious data structure that answers two-sided range queries on a set of $N$ points in the plane in $\mathcal{O}(\log_B N + T/B)$ memory transfers. The structure can be constructed cache-obliviously in $\mathcal{O}(N \log_B N)$ memory transfers.*

## 2.1 The Layout

### 2.1.1 $s_i$-Sparseness

The construction of the layout $\mathcal{L}$ is based on the following idea: Consider storing the points in $S$ in a sequence $S_0$, sorted by their $x$-coordinates. Every two-sided range query $(-\infty, x_q] \times [y_q, +\infty)$ such that $y_q$ is less than the $y$-coordinates of all points in $S$ can be answered by scanning through $S_0$, and reporting the read points, until a point with $x$-coordinate greater than $x_q$ is encountered. Since every point that is read is also reported, the scan takes $\mathcal{O}(1 + T/B)$ memory transfers. The same strategy works for greater values of $y_q$ if we report a point that is read only if its $y$-coordinate is greater than $y_q$. As long as we scan only $\mathcal{O}(T)$ points, the query bound remains $\mathcal{O}(1 + T/B)$. However, we need to deal with queries that would scan too many points.

Assume that we are willing to scan through $\alpha T$ points, for some $\alpha > 1$, when answering a query with output size $T$. We call a query $S_0$-*dense* if we scan at most $\alpha T$ points in $S_0$ to answer it; otherwise we call it $S_0$-*sparse*. Consider the minimum $y$-coordinate $y_1$ such that there is an $S_0$-sparse query $(-\infty, x_q] \times [y_q, +\infty)$ with $y$-coordinate $y_q = y_1$. Let $S_1$ be the $x$-sorted sequence of all points in $S$ with $y$-coordinates at least $y_1$. Observe that there are no $S_0$-sparse queries with $y$-coordinates less than $y_1$; that is, we can answer any query with $y$-coordinate $y_q < y_1$ efficiently by scanning $S_0$. Queries with $y_q \geq y_1$ can be answered by scanning $S_1$ because all points that may be in the query range have $y$-coordinates at least $y_1$ and are thus stored in $S_1$. Since some of these queries may be $S_1$-sparse, we iterate the process, finding the lowest $y$-coordinate $y_2$ such that there is an $S_1$-sparse query with $y_q = y_2$ and constructing a sequence $S_2$ that contains all points with $y$-coordinates at least $y_2$. In the same fashion, we construct sequences $S_3, S_4, \ldots, S_k$ and $y$-coordinates $y_3, y_4, \ldots, y_k$ until $|S_k| = \mathcal{O}(1)$. Note that $y_1 < y_2 < \cdots < y_k$. Figure 1(a) illustrates this construction.

The main problem with the above idea is that, if we define the layout $\mathcal{L}$ to be the concatenation of sequences $S_0, S_1, \ldots, S_k$, it may use more than linear space, since the total size of sequences $S_0, S_1, \ldots, S_k$ is $\Theta(N^2)$ in the worst case; refer to Figure 1(b).

### 2.1.2 Reducing the Space Requirements

In order to reduce the space requirements to linear, we modify the above construction to ensure that every sequence $S_i$ is identical to the next sequence $S_{i+1}$ in as large a suffix as possible, while guaranteeing that any query $(-\infty, x_q] \times [y_q, +\infty)$ with $y_q = y_{i+1}$ is $S_{i+1}$-dense. Then, instead of storing all of $S_i$ in $\mathcal{L}$, we only store the prefix $L_i$ of $S_i$ that is different from $S_{i+1}$. Thus, the layout $\mathcal{L}$ consists of the concatenation of sequences $L_0, L_1, \ldots, L_k$, where $L_k = S_k$.

More formally, to define the layout $\mathcal{L}$, we apply the following iterative process, starting with $S_0 = S$ and $y_0 = -\infty$: For the current sequence $S_i$, we find the lowest $y$-coordinate $y_{i+1} > y_i$ such that there is an $S_i$-sparse query with $y$-coordinate $y_{i+1}$. Let $x_{i+1}$ be the maximal $x$-coordinate such that the query $(-\infty, x_{i+1}] \times [y_{i+1}, +\infty)$ is $S_i$-sparse. Then we define $L_i$ to be the prefix of $S_i$ containing all points with $x$-coordinates at most $x_{i+1}$, and $R_i$ to be the suffix of $S_i$ containing all points with $x$-coordinates greater than $x_{i+1}$. Thus, $S_i = L_i \circ R_i$. Let $L_i'$ be the subsequence of $L_i$ containing all points with $y$-coordinates at least $y_{i+1}$. Then we

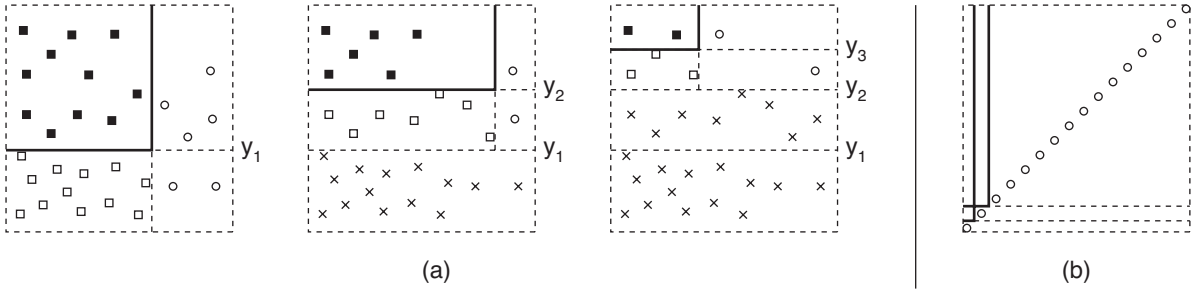


**Figure 1: (a) The construction of sets $S_0, S_1, \ldots S_k$ using the iterative process described in Section 2.1.1 with $\alpha = 2$. The three figures show sets $S_0$, $S_1$, and $S_2$ and queries $(-\infty, x_1] \times [y_1, +\infty)$, $(-\infty, x_2] \times [y_2, +\infty)$, and $(-\infty, x_3] \times [y_3, +\infty)$ that are sparse with respect to these sets and have minimal $y$-coordinates among all such queries. In each figure, the crosses represent points in $S$ that do not belong to the current set $S_i$, squares represent points that are scanned by the query, and solid squares represent points that are actually reported. (b) Defining $\mathcal{L}$ to be the concatenation of sequences $S_0, S_1, \ldots, S_k$ results in the use of $\Theta(N^2)$ space: The first shown query is $S_0$-sparse and leads to the creation of a sequence $S_1$ of size $N-1$; the second shown query is $S_1$-sparse and leads to the creation of a sequence $S_2$ of size $N-2$. This continues, creating $N$ sequences of total size $\Theta(N^2)$.**

define $S_{i+1} = L'_i \circ R_i$. This process continues until we obtain a sequence $S_k$ of constant size; for this sequence, we define $L_k = S_k$ and $R_k = \emptyset$. The layout $\mathcal{L}$ is the concatenation of sequences $L_0, L_1, \ldots, L_k$ (refer to Figure 2).

**Observation 1** *For $0 \le i \le k$, $S_i = L_i \cup L_{i+1} \cup \cdots \cup L_k$.*

**Observation 2** *Every sequence $L_i$, $0 \le i < k$, contains at least $\frac{\alpha-1}{\alpha}|L_i|$ points that are not in $S_{i+1}$.*

*Proof.* Sequence $S_{i+1}$ is the concatenation of $L'_i$ and $R_i$. $R_i$ does not contain any points from $L_i$. $L'_i$ contains at most $|L_i|/\alpha$ points from $L_i$ because the query $(-\infty, x_{i+1}] \times [y_{i+1}, +\infty)$ is $S_i$-sparse. □

Using Observation 2, we can prove that layout $\mathcal{L}$ uses linear space.

**Lemma 1** *Layout $\mathcal{L}$ uses at most $\frac{\alpha}{\alpha-1}N = \mathcal{O}(N)$ space.*

*Proof.* For every subsequence $L_i$ in $\mathcal{L}$, we charge the space required to store $L_i$ to the points in $L_i$ that are not in $S_{i+1}$. By Observation 2, there are at least $\frac{\alpha-1}{\alpha}|L_i|$ such points in $L_i$, that is, we charge each such point for $\alpha/(\alpha-1)$ points in $L_i$. Since every point $p$ is charged only in the last sequence $L_i$ that contains $p$, every point is charged only once. Thus, every point in $S$ is charged at most $\alpha/(\alpha-1)$; the space bound follows. □

### 2.1.3 Range Queries

To answer two-sided range queries using layout $\mathcal{L}$, we create a balanced binary search tree $Y$ over the $y$-coordinates $y_0, y_1, \ldots, y_k$ and lay it out using the van-Emde-Boas layout [27]; we let the leaf of $Y$ containing $y_i$ store a pointer to the beginning of $L_i$ in $\mathcal{L}$.

To answer a range query $(-\infty, x_q] \times [y_q, +\infty)$, we search $Y$ to find the leaf containing the maximal $y$-coordinate $y_i$ that satisfies $y_i \le y_q$. Then we follow the pointer from this leaf into $\mathcal{L}$ and scan forward until we encounter a point $p'$ with $x$-coordinate $x_{p'} > x_q$ or we reach the end of $\mathcal{L}$; we report every point $p$ we read that satisfies $y_p \ge y_q$ and has an $x$-coordinate $x_p$ greater than the $x$-coordinate of the last reported point.

As we prove below, this simple procedure answers a query in $\mathcal{O}(\log_B N + T/B)$ memory transfers. By stopping the scan as soon as we see the first point $p'$ with $x_{p'} > x_q$, we ensure that we report only points in the $x$-range $(-\infty, x_q]$; as we will see, this condition also ensures that we inspect only $\mathcal{O}(T)$ points. The condition for reporting points ensures that we report only points in the $y$-range $[y_q, +\infty)$ and that every point is reported only once.

**Lemma 2** *The layout $\mathcal{L}$ and search tree $Y$ can be used to answer any two-sided range query in $\mathcal{O}(\log_B N + T/B)$ memory transfers, where $T$ is the number of reported points.*

*Proof.* We first prove that the above procedure correctly answers any two-sided range query. It is obvious that the procedure never reports a point that is not in the query range, nor does it output any point twice because the $x$-coordinates of reported points are strictly increasing. Thus, we only have to prove that the procedure does not miss a point in the query range.

Assume for the sake of contradiction that there exists a point $p$ in the query range that is not reported, and assume that we start the scan of $\mathcal{L}$ at the first element of sequence $L_i$. We may fail to report $p$ for three reasons: (1) $p \notin S_i$, that is, $p$ is not in $L_i \cup L_{i+1} \cup \cdots \cup L_k$. (2) $p$ is stored after the position where we terminate the scan. (3) We report a point $p'$ with $x_{p'} > x_p$, but there is no occurrence of $p$ in $\mathcal{L}$ between the first element of $L_i$ and $p'$.

First assume that $p \notin S_i$ (Case 1). By Observation 1, we have $L_0 \cup L_1 \cup \cdots \cup L_k = S$ and $L_i \cup L_{i+1} \cup \cdots \cup L_k = S_i$. Thus, there must exist a sequence $L_j$, $j < i$, such that $p \in L_j$. Choose $j$ maximally so. Since $p \notin S_{j+1}$, we have $y_p < y_{j+1} \le y_i$. On the other hand, we start the scan of $\mathcal{L}$ at the first element of $L_i$ because $y_i \le y_q < y_{i+1}$. Since $p$ is in the query range, we have $y_p \ge y_q$, that is, $y_p \ge y_q \ge y_i \ge y_{j+1} > y_p$. This is the desired contradiction.

In both, Cases 2 and 3, we encounter a point $p'$ with $x_{p'} > x_p$ before we encounter $p$ for the first time. For Case 3, this is true by definition. In Case 2, we terminate the scan

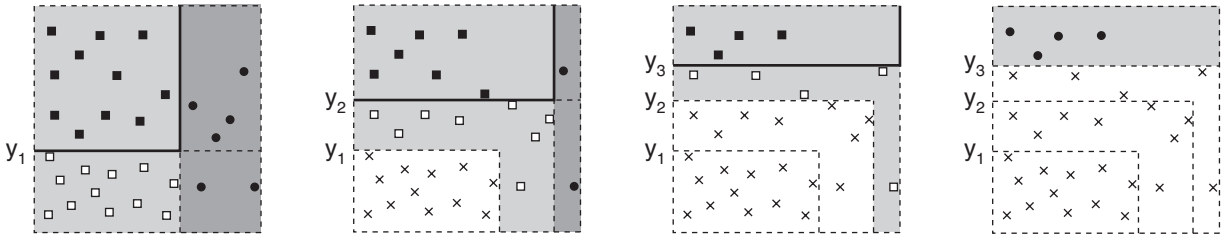


**Figure 2: The construction of sequences $S_0, S_1, \ldots, S_k$, $L_0, L_1, \ldots, L_k$ and $R_0, R_1, \ldots, R_k$ using the iterative process described in Section 2.1.2 with $\alpha = 2$. (In this example, $k = 3$.) The points in the light grey regions constitute sequences $L_0, L_1, L_2, L_3$. The points in the dark grey regions constitute sequences $R_0, R_1, R_2, R_3$; note that $R_2 = R_3 = \emptyset$. For $i = 0, 1, 2, 3$, $S_i = L_i \circ R_i$. In each figure, the query $(-\infty, x_{i+1}] \times [y_{i+1}, +\infty)$ defining $L_i$ and $R_i$ is shown in bold. The layout $\mathcal{L}$ is the concatenation of sequences $L_0, L_1, L_2, L_3$.**

because we encounter a point $p'$ with $x_{p'} > x_q$. However, since $p$ is in the query range, we have $x_p \leq x_q$, that is, $x_p < x_{p'}$. Next we show that we have to encounter a copy of point $p$ before encountering a point $p'$ with $x_{p'} > x_p$, thereby proving that Cases 2 and 3 cannot occur.

So assume that we encounter some point $p'$ with $x_{p'} > x_p$ before encountering $p$ for the first time. Since the points in each sequence $L_i$ are sorted by $x$-coordinates, $p'$ must belong to some sequence $L_h$, $h \geq i$, and the first occurrence of $p$ must be in some sequence $L_j$ with $j > h$. By Observation 1, $L_{h+1} \cup L_{h+2} \cup \cdots \cup L_k = S_{h+1}$. Hence, $p \in S_{h+1} = L'_h \circ R_h$. Since $p' \in L_h$ and $x_p < x_{p'}$, $p \notin R_h$ because $S_h = L_h \circ R_h$ and $S_h$ is sorted by $x$-coordinates. Hence, $p \in L'_h$. However, since $L'_h \subseteq L_h$, this implies that $p \in L_h$, a contradiction.

Having proved that the query procedure correctly answers any two-sided range query, it remains to bound the number of memory transfers it incurs. Since the search for $y_q$ in $Y$ incurs $\mathcal{O}(\log_B N)$ memory transfers, it suffices to show that the query procedure scans only $\mathcal{O}(T)$ consecutive points in $\mathcal{L}$, which incurs $\mathcal{O}(1 + T/B)$ memory transfers.

Since the query starts scanning at the first element of $L_i$, the query scans all points in sequences $L_i, L_{i+1}, \ldots, L_{j-1}$, for some $j \geq i$, and finally terminates the scan after reading a prefix $P_j$ of sequence $L_j$. First observe that the query is $S_i$-dense, that is, if we were to answer the query on $S_i$, we would scan a set $S'$ of $K \leq \alpha T$ points, which is exactly the set of points in $S_i$ that have $x$-coordinates no greater than $x_q$. Next observe that all the points we scan in $\mathcal{L}$ are contained in $S'$ because, by Observation 1, $L_i \cup L_{i+1} \cup \cdots \cup L_k = S_i$ and all scanned points have $x$-coordinates no greater than $x_q$. Finally, by Observation 2, every sequence $L_i$ contains at least $\frac{\alpha-1}{\alpha}|L_i|$ points that do not occur in any subsequent sequence $L_j$. Hence, $K \geq \sum_{h=i}^{j-1} \frac{\alpha-1}{\alpha}|L_h| + |P_j|$, that is, $\sum_{h=1}^{j-1} |L_h| + |P_j| \leq \frac{\alpha}{\alpha-1}K$. Since $K \leq \alpha T$, we scan no more than $\frac{\alpha^2}{\alpha-1}T = \mathcal{O}(T)$ points in $\mathcal{L}$. □

## 2.2 Construction of the Layout

In this section, we describe an efficient algorithm for constructing the layout $\mathcal{L}$ described in Section 2.1. Our algorithm uses a sweep-line approach: Starting at $-\infty$, we sweep a horizontal line upwards across the plane, while maintaining the invariant that, by the time the sweep line reaches a $y$-coordinate $y$ with $y_i \leq y < y_{i+1}$, we have constructed sequences $L_0, L_1, \ldots, L_{i-1}$ and $S_i$. Every time the sweep line reaches a point $p$, we check whether there is a query $(-\infty, x_q] \times [y_q, +\infty)$ with $y_q = y_p$ that is $S_i$-sparse. This is equivalent to detecting whether there exists an $x$-coordinate $x_q$ such that less than $1/\alpha$ of the points in $S_i$ with $x$-coordinates at most $x_q$ are above the sweep line. If there is such a query, then $y_p = y_{i+1}$ and we need to construct $L_i$ and $S_{i+1}$. To do so, we let $q$ be the point with $y_q = y_p$ and with maximal $x$-coordinate $x_q$ such that the query $(-\infty, x_q] \times [y_q, +\infty)$ is $S_i$-sparse. Then we scan through $S_i$ until we encounter the first point with $x$-coordinate greater than $x_q$, while copying all scanned points to $\mathcal{L}$ and removing all points with $y$-coordinates less than $y_q$ from $S_i$. The copied points constitute sequence $L_i$, and the resulting sequence is $S_{i+1}$.

It is easy to see that, ignoring the cost of testing for every point $p \in S$ whether there is an $S_i$-sparse query with bottom boundary $y_p$, the sweep uses $\mathcal{O}(|\mathcal{L}|/B) = \mathcal{O}(N/B)$ memory transfers: When constructing $L_i$ and $S_{i+1}$, we scan and copy $|L_i|$ points, and, by Lemma 1, $L_0 \circ L_1 \circ \cdots \circ L_k = \mathcal{L}$ contains $\mathcal{O}(N)$ points. Below we describe a data structure on $S$ that allows us to check, using $\mathcal{O}(1)$ memory transfers, whether there is an $S_i$-sparse query $(-\infty, x_q] \times [y_q, +\infty)$ with $y_q = y_p$. If there is such a query, we can use the structure to find the point $q$ with $y_q = y_p$ and maximal $x$-coordinate $x_q$ such that the query $(-\infty, x_q] \times [y_q, +\infty)$ is $S_i$-sparse; this takes $\mathcal{O}(\log_B N)$ memory transfers. The structure needs to be updated each time the sweep line reaches a point in $S$ and each time a point is removed from $S_i$. We show that this can be done using $\mathcal{O}(\log_B N)$ memory transfers per update. This leads to an $\mathcal{O}(N/B + N \cdot \log_B N) = \mathcal{O}(N \log_B N)$ construction algorithm.

To describe our data structure, we need a few definitions. Every point in $S$ can be in one of three different states w.r.t. the current set $S_i$ and the current sweep line: *dead*, if it is not in $S_i$; *useful*, if it is in $S_i$ and above the sweep line; or *useless*, if it is in $S_i$ and below the sweep line. We refer to the useful and useless points as *alive*. In other words, a query whose bottom boundary is on the sweep line scans only alive points and outputs all useful points it scans. For any subsequence $S'$ of $S_i$, we define $u(S')$ and $w(S')$ to be the number of useful and useless points in $S'$, respectively. We define the *surplus* of $S'$ as $s(S') = (\alpha - 1)u(S') - w(S')$. A query is $S_0$-dense if the surplus of the prefix of $S_0$ scanned by the query is non-negative.

With these definition, we can now define our data structure. It is a balanced binary tree $\mathcal{T}$ over the points in $S$, sorted by their $x$-coordinates. Each point in $\mathcal{T}$ is marked as *dead*, *useful* or *useless*. At any time during the sweep, the alive (useful and useless) points constitute $S_i$; initially,

all points in $\mathcal{T}$ are marked as useful. Each internal node $v$ in $\mathcal{T}$ represents the subsequence $S_v$ of $S_i$ consisting of the alive points in the subtree rooted at $v$. Node $v$ stores two labels: the surplus $s(v) = s(S_v)$ of $S_v$ and a *tolerance* $t(v)$, which is the minimum surplus of any prefix of $S_v$; initially, $s(v) = (\alpha - 1)|S_v|$ and $t(v) = \alpha - 1$. Below we will show how to update $s(v)$ and $t(v)$ as the sweep line moves upwards and points get marked as useless or dead. The structure of the tree $\mathcal{T}$ itself remains static, and $\mathcal{T}$ is laid out in memory using the van-Emde-Boas layout; we can thus traverse a root-to-leaf path in $\mathcal{T}$ using $\mathcal{O}(\log_B N)$ memory transfers [27].

To see how $\mathcal{T}$ can be used to determine whether there is an $S_i$-sparse query $(-\infty, x_q] \times [y_q, +\infty)$ with $y_q = y_p$ when reaching a point $p$, that is, to determine whether there is an $x$-coordinate $x_q$ such that less than $1/\alpha$ of the alive points with $x$-coordinates at most $x_q$ are useful, we need the following observation.

**Observation 3** *When the sweep line is at point $p$, a query $(-\infty, x_q] \times [y_q, +\infty)$ with $y_q = y_p$ is $S_i$-sparse if and only if the subsequence of $S_i$ scanned when answering the query has negative surplus.*

*Proof.* Let $S'$ be the prefix of $S_i$ that is scanned when answering the query $(-\infty, x_q] \times [y_q, +\infty)$ on $S_i$. The query reports exactly the points in $S'$ that are useful. The surplus of $S'$ is negative if and only if $(\alpha - 1)u(S') < w(S')$, that is, $\alpha u(S') < u(S') + w(S') = |S'|$. Thus, the surplus of $S'$ is negative if and only if the query is $S_i$-sparse. □

Since the tolerance of the root of $\mathcal{T}$ is the minimum surplus of any prefix of $S_i$, Observation 3 states that there is an $S_i$-sparse query if and only if the root has negative tolerance, which can be checked using a single memory access.

If we detect that there is an $S_i$-sparse query when reaching $p$, that is, if the root has negative tolerance, we can find the point $q$ with $y_q = y_p$ and maximal $x$-coordinate $x_q$ such that the query $(-\infty, x_q] \times [y_q, +\infty)$ is $S_i$-sparse as follows: We traverse a root-to-leaf path in $\mathcal{T}$ while maintaining the cumulative surplus $s^*$ of the prefix of $S_i$ represented by all subtrees to the left of the traversed path; initially, $s^* = 0$. At a node $v$ with left child $l$ and right child $r$, we test whether $s^* + s(l) + t(r)$ is negative. If it is, we add $s(l)$ to $s^*$ and proceed to $r$; otherwise, we proceed to $l$. Then we report the point $q$ whose $y$-coordinate is $y_p$ and whose $x$-coordinate is the $x$-coordinate of the point in the leaf reached by this procedure.

**Lemma 3** *When the sweep line is at point $p$, we can use $\mathcal{T}$ to determine in $\mathcal{O}(1)$ memory transfers whether there is an $S_i$-sparse query $(-\infty, x_q] \times [y_q, +\infty)$ with $y_q = y_p$. If so, it takes $\mathcal{O}(\log_B N)$ memory transfers to find the point $q$ with $y_q = y_p$ and maximal $x$-coordinate $x_q$ such that the query $(-\infty, x_q] \times [y_q, +\infty)$ is $S_i$-sparse.*

*Proof.* The complexity bounds follow immediately from the above discussion and the fact that we traverse a single root-to-leaf path to find $q$. To prove that the reported point, $q'$, is indeed $q$, we need to prove that $q'$ is the rightmost point such that the prefix of $S_i$ ending at $q'$ has negative surplus (Observation 3). To do so, it suffices to prove that, for every visited node $v$, the longest prefix of $S_i$ with negative surplus ends with an element in $S_v$. We do so by induction: For the root, the claim is trivial. So assume that the claim holds for the current node $v$, and let $l$ and $r$ be its two children. It suffices to prove that we continue the search at $r$ if and only if there is a prefix of $S_i$ with negative surplus that ends with an element in $S_r$. To do so, we observe that $s^* + s(l) + t(r)$ is the minimum surplus of any prefix of $S_i$ ending with a point in $S_r$. Indeed, $t(r) = \min_{p' \in S_r} t_{p'}$ and, for every point $p' \in S_r$, the prefix of $S_i$ ending at $p'$ has surplus $s^* + s(l) + t_{p'}$, where $t_{p'}$ is the surplus of the prefix of $S_r$ that ends at $p'$. This implies that there exists a prefix of $S_i$ with negative surplus that ends with an element in $S_r$ if and only if $s^* + s(l) + t(r)$ is negative. □

What remains to show is how $\mathcal{T}$ can be updated using $\mathcal{O}(\log_B N)$ memory transfers when the status of a point $p$ changes from useful to useless, because the sweep line passes $p$, or when the status of $p$ changes from useless to dead, because $p$ is removed from $S_i$.

**Lemma 4** *Tree $\mathcal{T}$ can be updated in $\mathcal{O}(\log_B N)$ memory transfers after a status change of a single point.*

*Proof.* When a point $p$ in a leaf $l$ of $\mathcal{T}$ changes its status from useful to useless, the surplus of each of $l$'s ancestors decreases by $\alpha$: $\alpha - 1$ for the loss of a useful point, and 1 for the gain of a useless point. When $p$ changes its status from useless to dead, the surplus of each ancestor increases by 1 because it loses a useless descendant. Thus, the relevant surplus values can easily be updated in $\mathcal{O}(\log_B N)$ memory transfers by traversing the path from $l$ to the root.

Similarly, only nodes on the path from the root to $l$ change their tolerance values $t(v)$ when $p$ changes its status. The tolerance of a node $v$ with left child $l$ and right child $r$ is $t(v) = \min(t(l), s(l) + t(r))$. Hence, we can also update the tolerances of the relevant nodes as we traverse the path from $l$ to the root. □

During the construction of the layout $\mathcal{L}$, we perform an update and a query on $\mathcal{T}$ whenever the sweep line passes a point in $S$. When constructing $S_{i+1}$ from $S_i$, we also perform one update on $\mathcal{T}$ per point in $S_i \setminus S_{i+1}$. Thus, in total, we answer $N$ queries and perform $2N$ updates on $\mathcal{T}$ to construct $\mathcal{L}$. This costs $\mathcal{O}(N \log_B N)$ memory transfers in total. Since the rest of the construction of $\mathcal{L}$ uses $\mathcal{O}(N/B)$ memory transfers, and since the search tree $Y$ can easily be constructed in $\mathcal{O}\left(\frac{N}{B} \log_B N\right)$ memory transfers, we obtain

**Lemma 5** *The layout $\mathcal{L}$ and the search tree $Y$ for a set of $N$ points in the plane can be constructed in $\mathcal{O}(N \log_B N)$ memory transfers.*

Lemma 5, along with Lemmas 1 and 2, proves Theorem 1.

# 3. STATIC THREE-SIDED PLANAR RANGE SEARCHING

By combining our linear-space structure for two-sided range queries with a standard construction, we obtain a simple $\mathcal{O}(N \log_2 N)$-space structure for three-sided queries. Our three-sided structure consists of a balanced binary base tree $\mathcal{T}$ with the $N$ points in $S$ stored at the leaves, sorted by their $x$-coordinates. $\mathcal{T}$ is laid out in memory using the van-Emde-Boas layout, so that a root-to-leaf path can be traversed cache-obliviously in $\mathcal{O}(\log_B N)$ memory transfers [27]. For

each internal node $v$ in $\mathcal{T}$, let $S_v$ be the points from $S$ stored in the subtree rooted at $v$. We store the points in $S_v$ in two secondary structures, $\mathcal{L}_v$ and $\mathcal{R}_v$, associated with $v$. $\mathcal{L}_v$ is a structure for answering two-sided queries of the form $(-\infty, x_q] \times [y_q, +\infty)$ (with the $x$-opening to the left); $\mathcal{R}_v$ is a structure for answering two-sided queries of the form $[x_q, +\infty) \times [y_q, +\infty)$ (with the $x$-opening to the right).

Since each point from $S$ is stored in two linear-space secondary structures on each of the $\mathcal{O}(\log_2 N)$ levels of $\mathcal{T}$, our structure uses $\mathcal{O}(N \log_2 N)$ space in total. To construct our structure, we sort the points in $S$; build $\mathcal{T}$ and the sets $S_v$, for all nodes $v \in \mathcal{T}$, bottom-up; and finally build the two structures $\mathcal{L}_v$ and $\mathcal{R}_v$ from $S_v$, for every node $v \in \mathcal{T}$. Sorting the points in $S$ requires $\mathcal{O}\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ memory transfers [24]. The construction of $\mathcal{T}$ and of the sets $S_v$ requires $\mathcal{O}(N/B)$ memory transfers per level of $\mathcal{T}$, $\mathcal{O}\left(\frac{N}{B} \log_2 N\right)$ memory transfers in total. Finally, since the total size of all secondary structures at each level is $\mathcal{O}(N)$, the construction of all secondary structures associated with nodes of $\mathcal{T}$ takes $\mathcal{O}(N \log_B N) \cdot \mathcal{O}(\log_2 N)$ memory transfers, by Theorem 1.

To answer a three-sided range query $[x_l, x_r] \times [y_b, \infty)$, we search down $\mathcal{T}$, using $\mathcal{O}(\log_B N)$ memory transfers, to find the first node $v$ such that $x_l$ is contained in the subtree rooted at the left child $l$ of $v$ and $x_r$ is contained in the subtree rooted at the right child $r$ of $v$. Then we query $\mathcal{R}_l$ with the query range $[x_l, +\infty) \times [y_b, +\infty)$ and $\mathcal{L}_r$ with the query range $(-\infty, x_r] \times [y_b, +\infty)$. By Theorem 1, this costs $\mathcal{O}(\log_B N + T/B)$ memory transfers. The correctness of this procedure is obvious. This proves

**Theorem 2** *There exists a cache-oblivious data structure that uses $\mathcal{O}(N \log_2 N)$ space to store $N$ points in the plane and answers three-sided range queries in $\mathcal{O}(\log_B N + T/B)$ memory transfers. The structure can be constructed using $\mathcal{O}(N \log_B N \cdot \log_2 N)$ memory transfers.*

# 4. SEMI-DYNAMIC TWO-SIDED PLANAR RANGE SEARCHING

Using the so-called *logarithmic method* [15, 25], we can support insertions into our two-sided structure. We partition the set $S$ of points into $\log_2 N$ disjoint subsets such that the $i$'th subset $S_i$ is either empty or of size $2^i$. Then we construct a two-sided structure on each set $S_i$, using Theorem 1. We answer a query by querying each of the $\log_2 N$ structures and combining the answers. If $T_i$ is the number of points reported from $S_i$, this takes $\sum_{i=0}^{\log_2 N} \mathcal{O}(\log_B 2^i + T_i/B) = \mathcal{O}(\log_B N \cdot \log_2 N + T/B)$ memory transfers in total. We perform an insertion by finding the first empty subset $S_k$, collecting all points in sets $S_0, S_1, \ldots, S_{k-1}$, and forming $S_k$ from the inserted point and the $\sum_{j=0}^{k-1} 2^j = 2^k - 1$ collected points from these sets. Then we discard all structures on sets $S_0, \ldots, S_{k-1}$ and construct a structure for the new set $S_k$. By Theorem 1, this takes $\mathcal{O}(2^k \log_B 2^k)$ memory transfers. If we divide the construction cost among the $2^k$ points moved to $S_k$, each of them has to pay for $\mathcal{O}(\log_B N)$ transfers. Since a point never moves from a set with higher index to a set with lower index, we charge every point at most $\mathcal{O}(\log_2 N)$ times. The amortized cost of an insertion is thus $\mathcal{O}(\log_B N \cdot \log_2 N)$ memory transfers.

The $\mathcal{O}(\log_B N \cdot \log_2 N + T/B)$ query bound can be improved to $\mathcal{O}(\log_2 N + T/B)$ using fractional cascading [22]. First recall that the two-sided structure for $S_i$ consists of a sequence $\mathcal{L}_i$ of points from $S_i$ and a search tree $Y_i$ on a sorted sequence of $y$-coordinates of a subset of the points in $S_i$; each element in this sequence has a pointer to a point in $\mathcal{L}_i$. The $\log_B N$-term in the query bound is a result of the search in the search trees $Y_i$; after the search, a query on $S_i$ is answered by scanning $\mathcal{O}(T_i)$ consecutive points in $\mathcal{L}_i$. In the following, we describe how to avoid the search in $Y_i$. In fact, we will not need a search tree at all, and we will from now on use $Y_i$ to denote the sequence of $y$-coordinates that would normally be stored in $Y_i$.

Given $\log_2 N + 1$ sorted sequences, $Y_0, Y_1, \ldots, Y_{\log_2 N}$, of elements ($y$-coordinates) such that $|Y_i| \leq 2^i$, our problem is to find in each sequence $Y_i$ the largest element that is no greater than a query element $y_q$. To do so efficiently, we augment each sequence $Y_i$ with some extra elements to obtain a sequence $Y_i'$. More precisely, starting with $Y'_{\log_2 N} = Y_{\log_2 N}$, we repeat the following process for $i = \log_2 N - 1, \log_2 N - 2, \ldots, 0$: We sample every 4'th element from $Y'_{i+1}$ and add all sampled elements to $Y_i$. The result is $Y_i'$. In addition, we augment every element $y \in Y_i \subseteq Y_i'$ with a pointer to the largest element $y' \in Y_i' \setminus Y_i$ such that $y' \leq y$, and we augment every element $y' \in Y_i' \setminus Y_i$ with two pointers, one to the largest element $y \in Y_i$ such that $y \leq y'$ and one to the copy of $y'$ in $Y'_{i+1}$. It is easy to prove by induction that the size $|Y_i'|$ of $Y_i'$ is bounded by $|Y_i| + |Y'_{i+1}|/4 \leq 2^i + 2 \cdot 2^{i+1}/4 = 2 \cdot 2^i$. Given sequences $Y_0, Y_1, \ldots, Y_k$ and $Y'_{k+1}$, we can construct sequences $Y_0', Y_1', \ldots, Y_k'$ iteratively, starting with $Y_k'$: For $i = k, k-1, \ldots, 0$, we simultaneously scan $Y'_{i+1}$ and $Y_i$ to obtain the samples from $Y'_{i+1}$ and merge them with the elements in $Y_i$. This produces $Y_i'$. The pointers for the elements in $Y_i'$ can easily be computed during this merge process. Overall, the construction of $Y_0', Y_1', \ldots, Y_k'$ requires $\sum_{i=0}^{k} \mathcal{O}((|Y_i| + |Y'_{i+1}|)/B) = \sum_{i=0}^{k} \mathcal{O}(2^i/B) = \mathcal{O}(2^k/B)$ memory transfers.

**Lemma 6** *Given sequences $Y_0, Y_1, \ldots, Y_k$ and $Y'_{k+1}$, the augmented lists $Y_0', Y_1', \ldots, Y_k'$ can be constructed cache-obliviously in $O(2^k/B)$ memory transfers.*

Given sequences $Y_0', Y_1', \ldots, Y'_{\log_2 N}$, the largest element $y_i \in Y_i$, for each $0 \leq i \leq \log_2 N$, that is no greater than a query element $q$ can be found efficiently as follows: First we scan $Y_0'$ to find the largest element $y_0' \in Y_0'$ that is no greater than $q$. Then we repeat the following process for $i = 0, 1, \ldots, \log_2 N - 1$: If $y_i' \in Y_i$, then $y_i = y_i'$; otherwise, we find $y_i$ by following the pointer from $y_i'$ to the next smaller element $y_i \in Y_i$. Then we find the largest element $y_i'' \in Y_i' \setminus Y_i$ that is no greater than $y_i'$. If $y_i' \in Y_i' \setminus Y_i$, $y_i'' = y_i'$; otherwise, $y_i'$ stores a pointer to this element. Element $y_i''$ stores a pointer to its occurrence in $Y'_{i+1}$. Since we sampled every 4'th element in $Y'_{i+1}$, $y'_{i+1}$ is among the four elements in $Y'_{i+1}$ that follow $y_i''$ and can thus be identified by inspecting these elements. The total cost of this procedure is $\mathcal{O}(\log_2 N)$ memory accesses because $|Y_0'| = \mathcal{O}(1)$ and, for $i = 0, 1, \ldots, \log_2 N - 1$, we follow a constant number of pointers and scan a constant number of elements in $Y'_{i+1}$.

All that remains is to describe how to maintain the $Y_i'$-sequences during insertions. Recall that we perform an insertion by finding the first empty sequence $S_k$ and then constructing a new two-sided structure, consisting of sequences $\mathcal{L}_k$ and $Y_k$, on the inserted point and the $2^k - 1$ points in $S_0, S_1, \ldots, S_{k-1}$. This requires $\mathcal{O}(2^k \log_B 2^k)$ memory trans-

fers. After the construction of $\mathcal{L}_k$ and $Y_k$, we need to update the sequences $Y'_0, Y'_1, \ldots, Y'_k$. (Sequences $Y'_{k+1}, \ldots, Y'_{\log N}$ remain unchanged). Given $Y'_{k+1}$, we can do so in $\mathcal{O}(2^k/B)$ memory transfers, by Lemma 6. (Note that $Y_0, Y_1, \ldots, Y_{k-1}$ are empty.) Using the same charging argument as at the beginning of this section, we then obtain

**Theorem 3** *There exists a linear-space cache-oblivious data structure that answers two-sided range queries on a set of $N$ points in the plane in $\mathcal{O}(\log_2 N + T/B)$ memory transfers and supports insertions in $\mathcal{O}(\log_B N \cdot \log_2 N)$ memory transfers amortized.*

# 5. CONCLUSIONS AND OPEN PROBLEMS

In this paper, we have presented a simple static linear-space structure for answering two-sided range queries cache-obliviously in the optimal number of memory transfers. We have also demonstrated that, using this structure, we can obtain a structure for three-sided range searching that matches the space and query bounds of the best previous structure for this problem, but is much simpler. Based on our static structure for two-sided range searching, we have also obtained the first semi-dynamic (insertion-only) structure for a planar range searching problem with a logarithmic query bound and polylogarithmic update bound.

The question whether there is a linear-space structure for three-sided range searching with the optimal query bound remains open. In order to make our semi-dynamic structure fully dynamic, it would be necessary to make our static structure support deletions directly. In order to obtain an optimal logarithmic term in the query bound of the semi-dynamic structure, the number of structures in the logarithmic method would have to be reduced to $\mathcal{O}(\log_B N)$. While this is possible in the I/O-model, it is open how to achieve this (or a similar property) cache-obliviously.

After the submission of this paper, Brodal [16] discovered that our static structure for two-sided range searching can be constructed in $\mathcal{O}\big(\frac{N}{B}\log_{M/B}\frac{N}{B}\big)$ memory transfers. This implies an improved amortized insertion bound of $\mathcal{O}\big(\frac{1}{B}\log_{M/B}\frac{N}{B}\cdot\log_2 N\big) = \mathcal{O}(\log_B^2 N)$ for the semi-dynamic structure for two-sided queries and an improved construction bound of $\mathcal{O}\big(\frac{N}{B}\log_{M/B}\frac{N}{B}\log_2 N\big)$ for the static structure for three-sided queries.

# 6. REFERENCES


[1] P. K. Agarwal, L. Arge, A. Danner, and B. Holland-Minkley. Cache-oblivious data structures for orthogonal range searching. In *Proc. ACM Symposium on Computational Geometry*, pages 237–245, 2003.

[2] P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, J. E. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, volume 223 of *Contemporary Mathematics*, pages 1–56. American Mathematical Society, 1999.

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

[4] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.

[5] L. Arge, M. Bender, E. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc. ACM Symposium on Theory of Computation*, pages 268–276, 2002.

[6] L. Arge, G. S. Brodal, R. Fagerberg, and M. Laustsen. Cache-oblivious planar orthogonal range searching and counting. In *Proc. ACM Symposium on Computational Geometry*, pages 160–169, 2005.

[7] L. Arge, M. de Berg, and H. Haverkort. Cache-oblivious R-trees. In *Proc. ACM Symposium on Computational Geometry*, pages 170–179, 2005.

[8] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proc. ACM Symposium on Principles of Database Systems*, pages 346–357, 1999.

[9] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[10] M. Bender, R. Cole, E. Demaine, and M. Farach-Colton. Scanning and traversing: Maintaining data for traversals in memory hierarchy. In *Proc. European Symposium on Algorithms*, pages 152–164, 2002.

[11] M. A. Bender, G. S. Brodal, R. Fagerberg, D. Ge, S. He, H. Hu, J. Iacono, and A. López-Ortiz. The cost of cache-oblivious searching. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 271–282, 2003.

[12] M. A. Bender, R. Cole, and R. Raman. Exponential structures for cache-oblivious algorithms. In *Proc. International Colloquium on Automata, Languages, and Programming*, pages 195–207, 2002.

[13] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 339–409, 2000.

[14] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 29–38, 2002.

[15] J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.

[16] G. S. Brodal. Personal communication, 2006.

[17] G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proc. International Colloquium on Automata, Languages, and Programming*, pages 426–438, 2002.

[18] G. S. Brodal and R. Fagerberg. Funnel heap - a cache oblivious priority queue. In *Proc. International Symposium on Algorithms and Computation, LNCS 2518*, pages 219–228, 2002.

[19] G. S. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proc. ACM Symposium on Theory of Computation*, pages 307–315, 2003.

[20] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 39–48, 2002.

[21] B. Chazelle. Lower bounds for orthogonal range searching: I. the reporting case. *Journal of the ACM*, 37(2):200–212, 1990.

[22] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[23] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[24] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 285–298, 1999.

[25] M. H. Overmars. *The Design of Dynamic Data Structures*. Springer-Verlag, LNCS 156, 1983.

[26] O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter. Bkd-tree: A dynamic scalable kd-tree. In *Proc. International Symposium on Spatial and Temporal Databases, LNCS 2750*, pages 46–65, 2003.

[27] H. Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1999.

[28] N. Rahman, R. Cole, and R. Raman. Optimised predecessor data structures for internal memory. In *Proc. Workshop on Algorithm Engineering, LNCS 2141*, pages 67–78, 2001.

[29] J. Robinson. The K-D-B tree: A search structure for large multidimensional dynamic indexes. In *Proc. SIGMOD International Conference on Management of Data*, pages 10–18, 1981.

[30] J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Computing Surveys*, 33(2):209–271, 2001.