Cache-Oblivious Planar Orthogonal Range Searching and Counting

Lars Arge[*]
BRICS[†]
Dept. of Computer Science
University of Aarhus
IT Parken, Aabogade 34
8200 Aarhus N, Denmark
large@daimi.au.dk

Gerth Stølting Brodal[‡]
BRICS[†]
Dept. of Computer Science
University of Aarhus
IT Parken, Aabogade 34
8200 Aarhus N, Denmark
gerth@daimi.au.dk

Rolf Fagerberg[§]
Dept. of Mathematics and Computer Science
University of Southern Denmark
5230 Odense M, Denmark
rolf@imada.sdu.dk

Morten Laustsen
BRICS[†]
Dept. of Computer Science
University of Aarhus
IT Parken, Aabogade 34
8200 Aarhus N, Denmark
mol@daimi.au.dk

ABSTRACT

We present the first cache-oblivious data structure for planar orthogonal range counting, and improve on previous results for cache-oblivious planar orthogonal range searching.

Our range counting structure uses $O(N \log_2 N)$ space and answers queries using $O(\log_B N)$ memory transfers, where B is the block size of any memory level in a multilevel memory hierarchy. Using bit manipulation techniques, the space can be further reduced to $O(N)$. The structure can also be modified to support more general semigroup range sum queries in $O(\log_B N)$ memory transfers, using $O(N \log_2 N)$ space for three-sided queries and $O(N \log_2^2 N / \log_2 \log_2 N)$ space for four-sided queries.

Based on the $O(N \log N)$ space range counting structure, we develop a data structure that uses $O(N \log_2 N)$ space and answers three-sided range queries in $O(\log_B N + T/B)$ memory transfers, where T is the number of reported points. Based on this structure, we present a general four-sided range searching structure that uses $O(N \log_2^2 N / \log_2 \log_2 N)$ space and answers queries in $O(\log_B N + T/B)$ memory transfers.

[*]Supported in part by the US National Science Foundation through RI grant EIA–9972879, CAREER grant CCR–9984099, ITR grant EIA–0112849, and U.S.-Germany Cooperative Research Program grant INT–0129182, 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. Part of this work was done while the author was at Duke University.

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

[‡]Partially supported by the Carlsberg Foundation through grant ANS-0257/20 and the Danish Natural Science Research Council through grant 21-04-0389.

[§]Supported in part by the Danish Natural Science Research Council (SNF).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SCG'05, June 6–8, 2005, Pisa, Italy.
Copyright 2005 ACM 1-58113-991-8/05/0006 ...$5.00.

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

Keywords

Cache-Oblivious, orthogonal range searching, range counting, semi-group range queries

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 transfered 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.

For 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 in the access times of main memory and disks, relatively little work has been done in models of multilevel memory. One reason for this is the many parameters in such models. The *cache-oblivious* model was introduced as a way of achieving algorithms that are efficient in arbitrary memory hierarchies without the use of complicated multilevel memory models.

In this paper we develop new and improved cache-oblivious data structures for planar orthogonal range counting and searching. Planar orthogonal range searching is the problem of finding, among a set of N points in the plane, all T points lying in a given query axis-parallel rectangle. The corresponding counting problem returns the number of such points.

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 (or cache) 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 the internal and the external memory. Computation can only occur 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. [21], 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 cache, the relevant block is automatically transfered into the cache. If the cache is full, an *optimal paging strategy* replaces the *ideal* block in cache 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 two-level 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 [21]. As a consequence, an algorithm that is optimal in the two-level model is optimal on *all* levels of an arbitrary multilevel hierarchy.

1.2 Previous results

Range searching has been studied extensively in the RAM model. In the planar case, for example, some of the best known structures (for any fixed $\varepsilon > 0$) answer orthogonal range queries in $O(\log_2 N + T \log^{\varepsilon}(2N/T))$ time using linear space and in $O(\log_2 N + T)$ time using $O(N \log^{\varepsilon} N)$ space, respectively [17, 18]. Refer to a recent survey for further results [2].

In the I/O-model, the B-tree [7, 20] supports one-dimensional range queries in $O(\log_B N + T/B)$ memory transfers using linear space. In two dimensions, one has to use $\Theta(N \log_B N/ \log_B \log_B N)$ space to obtain an $O(\log_B N + T/B)$ query bound [6, 19]. The external range-tree structure obtains these bounds [6]. If only linear space is used then $\Theta(\sqrt{N/B} + T/B)$ transfers, as obtained by the kd-B tree [24, 27], is needed to answer a query. For the problem of range counting, the CRB-tree [22], which is an external version of the compressed range tree [18], answers queries in $O(\log_B N)$ memory transfers and uses $O(N)$ space if bit manipulations of pointers and counters are allowed. Refer to recent surveys for further I/O-model and hierarchical memory model results [4, 28].

Frigo et al. [21] developed cache-oblivious algorithms for sorting, Fast Fourier Transform, and matrix multiplication. Subsequently, a number of other results have been obtained in the cache-oblivious model [1, 5, 8, 9, 10, 11, 12, 13, 14, 15, 16, 26], among them several cache-oblivious B-tree structures with $O(\log_B N)$ search and update bounds [10, 11, 12, 16, 26]. Several of these structures can also support one-dimensional range searching in $O(\log_B N + T/B)$ memory transfers [11, 12, 16] (but at an increased amortized update cost of $O(\log_B N + \frac{1}{B}\log_2^2 N) = O(\log_B^2 N)$ memory transfers). In [13], an algorithm for batched planar orthogonal range searching were developed, which answers a set of $O(N)$ queries using $O(\frac{N}{B}\log_{M/B}\frac{N}{B} + T/B)$ memory transfers, where T is the combined size of the answers.

Agarwal et al. [1] were the first to develop cache-oblivious structures for non-batched planar orthogonal range searching. They developed a cache-oblivious version of a kd-tree that answers planar range queries in $O(\sqrt{N/B}+T/B)$ memory transfers using linear space. It supports updates in $O(\frac{\log_2 N}{B}\log_{M/B} N) = O(\log_B^2 N)$ transfers. 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 $O(\log_B N + T/B)$ memory transfers but using $O(N \log_2^2 N)$ space. The central part of this structure is an $O(N \log_2 N)$ space structure for answering planar three-sided range queries in $O(\log_B N + T/B)$ memory transfers, that is, for finding all T points in a query range $[x_l, x_r] \times [y_b, \infty)$. The analysis of the range tree structure (or rather, the structure for three-sided queries) requires that $B = 2^{2^c}$ for some nonnegative integer constant c.

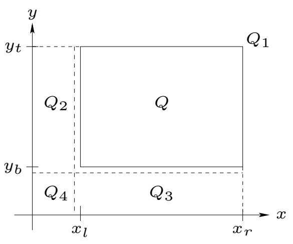
1.3 Our results

In this paper, we develop the first cache-oblivious structures for planar orthogonal *range counting*. Our structures answer queries in $O(\log_B N)$ memory transfers. We first describe a version using $O(N \log_2 N)$ space, and then reduce the space to $O(N)$ assuming that bit manipulations of pointers and counters are allowed. This matches the performance of the CRB-tree [22] in the I/O-model. The structure can be generalized to support semigroup range sum queries in $O(\log_B N)$ memory transfers using $O(N \log_2 N)$ and $O(N \log_2^2 N/ \log_2 \log_2 N)$ space for three-sided and four-sided queries, respectively (assuming that elements in the semigroup can be represented in $O(1)$ space).

For planar orthogonal *range searching*, we develop an improved cache-oblivious three-sided structure without any assumption on B. The structure uses $O(N \log_2 N)$ space and supports queries in $O(\log_B N + T/B)$ memory transfers. Based on this structure, we then give an improved $O(\log_B N + T/B)$ query general (four-sided) structure without any assumption on B and with an improved $O(N \log_2^2 N/ \log_2 \log_2 N)$ space bound.

2. RANGE COUNTING

In this section we describe a cache-oblivious data structures supporting four-sided range counting queries in $O(\log_B N)$ memory transfers. A four-sided range counting query $Q = [x_l, x_r] \times [y_b, y_t]$ can be answered using four two-sided queries; refer to Figure 1. Without loss of generality we therefore only consider two-sided range counting queries $Q = (-\infty, x_r] \times (-\infty, y_t]$ in the rest of this section.



$$
\begin{aligned}
Q_1 &= (-\infty, x_r] \times (-\infty, y_t] \\
Q_2 &= (-\infty, x_l) \times (-\infty, y_t] \\
Q_3 &= (-\infty, x_r] \times (-\infty, y_b) \\
Q_4 &= (-\infty, x_l) \times (-\infty, y_b)
\end{aligned}
$$

Figure 1: Reducing four-sided range counting queries $Q = [x_l, x_r] \times [y_b, y_t]$ to four two-sided queries; $Q = Q_1 - Q_2 - Q_3 + Q_4$.

Our structures are based on a construction of Lueker [23] for range reporting in $O(\log_2 N + T)$ time. The basic structure resembling Lueker's construction [23] is described in Section 2.1. The main contribution of Section 2.2 is then a recursive layout of this structure in memory that enables cache-oblivious range counting queries in $O(\log_B N)$ memory transfers using $O(N \log_2 N)$ space. The layout is a generalization of the layout of the cache oblivious search tree of Prokop [25] to handle non-constant sized secondary structures. In Section 2.3 we then reduce the space to $O(N)$ using standard bit manipulation techniques, assuming that each word contains $\Omega(\log_2 N)$ bits.

2.1 Basic approach

Given a set S of N points in the plane, our basic structure, based on the structure of Lueker [23], is defined as follows: The N points are stored at the leaves of a binary search tree $\mathcal{T}$ in x-coordinate sorted order from left-to-right. A list L_v is associated with each internal node v of $\mathcal{T}$ containing the points stored at the leaves of the subtree rooted at v sorted with respect to y-coordinate. With each point p_i in L_v we store two pointers left(p_i) and right(p_i) to the topmost point p_l and p_r in $L_{\text{left}(v)}$ and $L_{\text{right}(v)}$, respectively, with y-coordinate at most $y(p_i)$; here left(v) and right(v) are the left and right child of v, respectively, and $y(p_i)$ is the y-coordinate of the point p_i. With p_i in L_v we also store the number of points in $L_{\text{left}(v)}$ with y-coordinate at most $y(p_i)$ as leftsum(p_i). Finally, for the root r we store a binary search tree on L_r ordered with respect to y-coordinate. The store uses $O(N \log N)$ space since each point is stored in a list on each level of $\mathcal{T}$. Figure 7 illustrates a tree $\mathcal{T}$ for sixteen points and the links between the associated L_v lists.

To answer a counting query $Q = (-\infty, x_r] \times (-\infty, y_t]$ we perform a top-down traversal of $\mathcal{T}$ for the rightmost leaf storing a point with x-coordinate at most x_r, while in each encountered node v locating the topmost point in L_v with y-coordinate at most y_t: We first search the binary search tree on L_r to locate the topmost point p_i with y-coordinate at most y_t in L_r. Based on the x-coordinate split value in r, we then proceed to left(v) or right(v) following the left(p_i) or right(p_i) pointers. By continuing this process down the tree, we will, as argued below, locate the relevant topmost point with y-coordinate at most y_t in the list L_v in each node v on the search path. Whenever the search continues to the right child, we also add leftsum(p_i) to the output count; if the point stored at the final leaf is contained in Q the count is incremented by one.

It is easy to see that in order to prove that the query procedure correctly counts the number of points from S in Q, we simply have to argue that if p_i is the topmost point in L_v with y-coordinate at most y_t, then left(p_i) and right(p_i) are the topmost points in $L_{\text{left}(v)}$ and $L_{\text{right}(v)}$ with y-coordinate at most y_t. This is implied by the fact that $L_{\text{left}(v)} \subseteq L_v$ and $L_{\text{right}(v)} \subseteq L_v$ for all nodes v.

It it equally easy to see that the query procedure uses $O(\log_2 N)$ time: $O(\log_2 N)$ time to search L_r and $O(1)$ time at each level of $\mathcal{T}$.

2.2 Memory layout

Our method for achieving an efficient cache-oblivious version of the above data structure is based on two ideas. The first idea is a recursively defined memory layout of the L_v lists, developed from the van Emde Boas layout [25] of binary trees. In each step of the van Emde Boas recursion, we further divide the L_v lists for the nodes in the subtree under consideration. The first part for all nodes in the subtree are stored together, then the next part for all nodes, and so forth. The second idea is to ensure locality of reference during a search by adding redundant information to the lists: each list L_v is replaced by a list $\overline{L}_v \supseteq L_v$ consisting of L_v plus some *dummy points*. We will ensure that $\overline{L}_r = L_r$ and $\overline{L}_v \subseteq \overline{L}_{\text{parent}(v)}$ for all nodes v. As in the basic structure, a point $p_i \in \overline{L}_v$ has pointers left(p_i) and right(p_i) to the topmost points p_l and p_r in $\overline{L}_{\text{left}(v)}$ and $\overline{L}_{\text{right}(v)}$ with y-coordinate at most $y(p_i)$. However, dummy points are *not* counted in leftsum(p_i), i.e. leftsum(p_i) denotes the number of points in $L_{\text{left}(v)}$ with y-coordinate at most $y(p_i)$. Since $\overline{L}_v \subseteq \overline{L}_{\text{parent}(v)}$ and $\overline{L}_v \supseteq L_v$ dummy points do not affect the time complexity or the correctness of the query procedure described above. The dummy points are introduced during the recursive layout of the structure in memory, as defined formally below.

The cache-oblivious data structure consists of three substructures: X, Y, and $\mathcal{L}$. The structures X and Y are simply van Emde Boas layouts [25] of the base tree $\mathcal{T}$ (without the L_v lists) and of L_r; this cache-oblivious layout supports searches in $O(\log_B N)$ memory transfers [25]. The structure $\mathcal{L}$ is the recursive layout of the $\overline{L}_v$ lists described in the rest of this section.

In the following we let α denote a positive integer. For the $O(N \log_2 N)$ space structure we use $\alpha = 1$, whereas for the linear space data structure in Section 2.3 we use $\alpha = \lfloor \log_2 N \rfloor$. The recursive layout is defined using triples $< C, I, p >$, where I is a y-interval, C is a subtree of $\mathcal{T}$ consisting of the topmost h levels of the subtree rooted at a node v, for some height h, and p is a dummy point to be included in all L_u lists for nodes $u \in C$; we require that $|L_u \cap I| \leq \alpha 2^h$ where h is the height of C, and that the y-coordinate of p equals the lower endpoint of I. For the outermost recursion we have $C = \mathcal{T}$ and $I = (y(p), \infty)$, where p is the lowest input point.

The recursive layout of $< C, I, p >$ resembles the van Emde Boas layout: C is partitioned into a top tree C_0 of height $\lfloor h/2 \rfloor$ and $s = 2^{\lfloor h/2 \rfloor}$ bottom trees $C_1, \ldots, C_s$ of

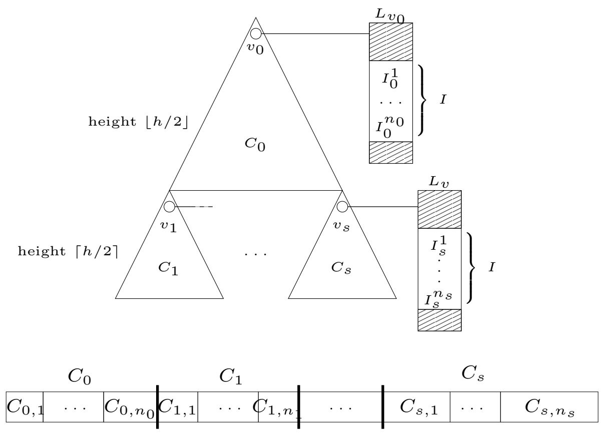


Figure 2: The recursive layout of a subtree C with respect to an interval I.

height $\lceil h/2 \rceil$. The layouts of $C_0, \ldots, C_s$ are stored consecutively in memory, and the layout of each C_i consists of one or more recursive layouts. Let v_i be the root of C_i and h_i the height of C_i. For C_i we split the interval I into $n_i = \lceil |L_{v_i} \cap I|/(\alpha 2^{h_i}) \rceil$ disjoint intervals $I_i^1, \ldots, I_i^{n_i}$, such that $|I_i^1 \cap L_{v_i}| \le \alpha 2^{h_i}$ and $|I_i^j \cap L_{v_i}| = \alpha 2^{h_i}$ for $1 < j \le n_i$. The recursive layouts of C_i are $C_{i,1} = < C_i, I_i^1, p >$ and then $C_{i,j} = < C_i, I_i^j, p_i^j >$, for $1 < j \le n_i$, where p_i^j is the point with minimum y-coordinate in $I_i^j \cap L_{v_i}$. Note that the partitioning of I depends on the points in $L_{v_i} \cap I$ and is different for each C_i. For the base-case of the recursion where C is a single node, $\{p\} \cup (L_v \cap I)$ is simply stored in consecutive memory locations. The recursive layout is illustrated in Figure 2.

The dummy points introduced for the example in Figure 7 are shown in Figure 8. Note that the introduction of p_9 in $L_{\text{left}(r)}$ guarantees that the pointer left(p_9) in L_r points to p_9 in $L_{\text{left}(r)}$, guaranteeing that following the left point of p_9 at the root will end up at a (dummy) point in the same recursive layout of the top tree.

The space required for X and Y is $O(N)$ [25]. The space required for $\mathcal{L}$, not counting dummy points, is $O(N \log_2 N)$, since each input point p appears as a regular point in the $\overline{L}_v$ lists of each of the $O(\log_2 N)$ ancestors v of the leaf storing p. What remains is to argue that the number of dummy points is at most $O(N \log_2 N)$.

Lemma 1. *The total number of dummy points introduced in the recursive layout is $O(N + (N \log_2 N)/\alpha)$.*

Proof. Initially (in the outermost recursive level) we introduce the lowest input point as a dummy point for each node of $\mathcal{T}$, i.e. at most N dummy points. For a recursive layout of $< C, I, p >$ where the subtree C is partitioned into $C_0, \ldots, C_s$ and there are n_i recursive layouts of C_i, we introduce at most $n_i - 1 \le |L_{v_i} \cap I|/(\alpha 2^{h_i})$ dummy points into $\overline{L}_u$ for each node $u \in C_i$. We charge these dummy points to the points in $L_{v_i} \cap I$. Since $|C_i| \le 2^{h_i}$, each point in $L_{v_i} \cap I$ is charged $O((|C_i| \cdot |L_{v_i} \cap I|/(\alpha 2^{h_i}))/|L_{v_i} \cap I|) = O(1/\alpha)$ dummy points. To count how many times a point q can be charged, observe that $q \in L_u$ only for ancestors u of the leaf of $\mathcal{T}$ storing q. Since the recursive layout follows the van Emde Boas layout with top trees and bottom trees, there are $O(\sum_{i=0}^{\log \log n} 2^i) = O(\log n)$ subtrees C rooted at the ancestors of the leaf storing q (of which up to $O(\log_2 \log_2 n)$ subtrees can share a root). Therefore during the recursive layout there are at most $O(\log_2 N)$ different recursive layouts $< C, I, p >$ having an ancestor of q as the root of C and $y(q) \in I$. It follows that each point q is at most charged $O((\log_2 N)/\alpha)$ dummy points. $\square$

For the construction in Section 2.3 we need the following lemma. By a similar argument as in Lemma 1 the number of base cases of the recursive layout, i.e. total number of pieces the L_v lists are cut into, becomes $O(N + (N \log_2 N)/\alpha)$, since each point in an L_v list can be charged a number of base cases created instead of the number of dummy points introduced.

Lemma 2. *The total number of base cases in the recursive layout is $O(N + (N \log_2 N)/\alpha)$.*

Finally, we are ready to bound the number of memory transfers used to answer a range counting query (using the query proceedure described in Section 2.1): First observe that the dummy points ensure that a search in a layout $< C, I, p >$ cannot leave the layout at internal nodes of C using the left(p_i) and right(p_i) pointers, because all nodes in C are guaranteed to have a dummy point with y-coordinate $\min(I) = y(p)$. Next observe that the size of $< C, I, p >$ is $O(|C|(1 + |L_v \cap I|))$, where v is the root of C, since p and each point from $L_v \cap I$ can at most be added once as dummy points to all L_u lists for $u \in C$. If C has height at most h this is $O(\alpha 2^{2h})$. Thus if C has height at most $\frac{1}{2} \log_2 B$ and $\alpha = 1$, the layout of $< C, I, p >$ fits into $O(1)$ blocks, and a path through it can be traversed in $O(1)$ memory transfers. Since a query path from the root to a leaf of $\mathcal{T}$ can be covered by layouts corresponding to subtrees of height between $\frac{1}{4} \log_2 B$ and $\frac{1}{2} \log_2 B$, each fitting into $O(1)$ blocks, it follows that a query performs $O(\frac{\log_2 N}{(\log_2 B)/4}) = O(\log_B N)$ memory transfers.

Theorem 1. *There exists a cache-oblivious data structure for storing N points in the plane using $O(N \log_2 N)$ space, such that a four-sided range counting query can be answered in $O(\log_B N)$ memory transfers.*

2.3 Linear space

In this section we describe how the range counting data structure in Section 2.2 using $O(N \log_2 N)$ space can be compressed to use $O(N)$ space (memory words) using bit manipulation techniques; we assume that the memory consists of $W \ge \log_2 N$ bit words, and that it is possible to perform shifts, additions, and boolean operations in $O(1)$ time.

Let $\alpha = \lfloor \log_2 N \rfloor$; by Lemma 1 the total number of dummy points introduced (and base cases of the recursive layout) is $O(N)$. By Lemma 2 it follows that the $O(N \log_2 N)$ points are laid out in the $\overline{L}_v$ lists in $O(N)$ *chunks*, i.e. the base case in the recursive layout, each of size $O(\log_2 N)$.

Lemma 3. *Each chunk of a list $\overline{L}_v$ contains left and right pointers to points in at most $O(1)$ different chunks.*

Proof. Consider a chunk c of $\overline{L}_v$, and let u be a child of v. In the recursive layout let I be the y-interval spanning c for the recursion where v is in the top tree C_0 and u is the root of a bottom tree C_i. The dummy points ensure that all pointers in c to $\overline{L}_u$ point to points with y-coordinate in I. Since u is the root of the bottom tree C_i, the recursive layout ensures that $L_u \cap I$ is partitioned into a sequence of chunks of size exactly α and one last chunk of size at most α plus possibly one dummy points. Since the $O(\alpha)$ pointers in the chunk c in $\overline{L}_v$ point to consecutive points in $\overline{L}_u$, it follows that at most $O(1)$ chunks of $\overline{L}_u$ can possibly be hit by a pointer from c. $\square$

In the following we describe how each of the $O(N)$ chunks can be stored in $O(1)$ words, implying $O(N)$ space in total. A pointer to a point in a chunk is represented by a pair $\langle \text{chunk}, \text{offset} \rangle$, i.e. a pointer to the chunk containing the point and the offset of the point within the chunk relative to the lowest point in the chunk.

For a chunk we store $\text{left}(p_0)$, $\text{right}(p_0)$, and $\text{leftsum}(p_0)$ for the lowest point p_0 in the chunk. For each of the following points $p_1, p_2, \ldots$ we store three bits: $\Delta\text{leftsum}(i)$, $\Delta\text{left}(i)$, and $\Delta\text{right}(i)$, where $\Delta\text{leftsum}(i) = \text{leftsum}(p_i) - \text{leftsum}(p_{i-1})$, $\Delta\text{left}(i) = 0$ if and only if $\text{left}(p_i) = \text{left}(p_{i-1})$, and $\Delta\text{right}(i) = 0$ if and only if $\text{right}(p_i) = \text{right}(p_{i-1})$. The bit values Δleftsum, Δleft, and Δright are stored in $O(1)$ words. Finally we store explicit pointers $\text{left}(p_i)$ if $\text{left}(p_i)$ and $\text{left}(p_{i-1})$ point to distinct chunks, and similarly for $\text{right}(p_i)$. From Lemma 3 there are only $O(1)$ of such pointers, implying total $O(1)$ space for a chunk.

Given a pointer $\langle c, i \rangle$ to an implicit point p_i in a chunk c, we compute $\text{leftsum}(p_i)$ as $\text{leftsum}(p_0) + \sum_{j=1}^{i} \Delta\text{leftsum}(j)$. To compute $\text{left}(p_i)$ we find the highest explicit left pointer $\text{left}(p_k) = \langle q, o \rangle$ stored in the chunk with $k \leq i$, and return the pointer $\langle q, o + \sum_{j=1}^{i} \Delta\text{left}(j) - \sum_{j=1}^{k} \Delta\text{left}(j) \rangle$. Right pointers are computed similarly.

The sums $\sum_{j=1}^{i} \Delta\text{leftsum}(j)$ and $\sum_{j=1}^{i} \Delta\text{left}(j)$ can be computed by the operation $\text{bitcount}(w, i)$. Given a word w and an integer i $\text{bitcount}(w, i)$ returns the number of bits equal to one among the i least significant bits of w, for $1 \leq i \leq W$.

If $\text{bitcount}(w, i)$ is not supported in constant time, we together with w store an additional word w', allowing $\text{bitcount}(w, i)$ to be computed in constant time using additions, shifting and boolean operations. Without loss of generality we assume $W = 2^a$ for some integer a. Let $b = 2^{\lceil \log_2 a \rceil}$. The word w' consists of $\lfloor W/b \rfloor$ subblocks $w'_0, \ldots, w'_{\lfloor W/b \rfloor - 1}$ of b bits each, where block w'_j stores $\text{bitcount}(w, j \cdot b)$.

To compute $\text{bitcount}(w, i)$ the precomputed sum of the bits of the first $\lfloor i/b \rfloor$ blocks of w can be extracted from w' by appropriate shifting and masking, whereas the sum of the $i \bmod b$ least significants bits of the $\lfloor i/b \rfloor + 1$st block of w can be looked up using a precomputed table stored in $O(1)$ words (note that in the cache oblivious model a table of size $\omega(1)$ words could cause a memory transfer at each level of $\mathcal{T}$). More precisely, we compute $\text{bitcount}(w, i)$ as

$$\begin{gathered}(w' >> (i \wedge \neg(b-1))) \wedge (b-1) \\ +\text{Count}[(w >> (i \wedge (\neg(b-1)))) \wedge ((1 << (i \wedge (b-1))) - 1)]\end{gathered}$$

where $\text{Count}[x]$ denotes the number of bits equal to one in x for $0 \leq x < 2^b$. To compute $\text{Count}[x]$ we partition x into at most four groups of at most $a/2$ bits and count the number of one bits in each group separately. For x containing at most $a/2$ bits we can store $\text{Count}[x]$ as an array of $2^{a/2}$ entries each consisting of at most $\lceil \log_2 \frac{a}{2} \leq 2a \rceil \leq 2a$ bits. By using $2^{\lceil \log_2 \lceil \log_2 \frac{a}{2} \rceil \rceil} \leq 2a$ bits for each entry we in total use $2a \cdot 2^{a/2} = O(W)$ bits which can be stored in $O(1)$ precomputed words and be looked up in $O(1)$ time by appropriate shifting and bit masking.

We have argued that a compressed chunk uses $O(1)$ space and supports pointer traversals in $O(1)$ time. To bound the number of memory transfers used by a query we observe that a recursive layout $< C, I, p >$ where the root of C is v satisfies $|L_v \cap I| \leq \alpha |C|$. The total number of chunks to be layed out recursively for $< C, I, p >$ is bounded by $O(\alpha |C|^2 / \alpha) = O(|C|^2)$. As in Section 2.2 this implies that searches use $O(\log_B N)$ memory transfers.

THEOREM 2. *There exists a cache-oblivious data structure for storing N points in the plane using $O(N)$ space, such that a four-sided range counting query can be answered in $O(\log_B N)$ memory transfers.*

3. THREE-SIDED RANGE QUERIES

In this section we develop an $O(N \log_2 N)$ space structure for answering tree-sided range queries $Q = [x_l, x_r] \times [y_b, \infty)$ on a set S of N points in the plane using $O(\log_B N + T/B)$ memory transfers. We first describe a linear space structure which requires the output size T to be in an interval $[\bar{T}, 2\bar{T})$ for a fixed value $\bar{T}$. This structure utilizes ideas from the three-sided structure of Agarwal et al. [1], which in turn is inspired by the external priority search tree of Arge et al. [6]. Then we describe how to use this structure in an $O(N \log_2 N)$ space structure for unknown output size T.

3.1 Known output size structure

In the following we let $\bar{T}$ be a fixed value and we assume that the output size $T \in [\bar{T}, 2\bar{T})$. Our linear space structure consists of $2N/\bar{T}$ structures of size $O(\bar{T})$. To define the structure, we first consider dividing the plane into $N/\bar{T}$ vertical *slabs* $X_1, X_2, \ldots, X_{N/\bar{T}}$ containing $\bar{T}$ points from S each. Using these slabs we then define $2N/\bar{T} - 1$ *buckets*. A bucket is a rectangular region of the plane that completely spans one or more consecutive slabs and is unbounded in the positive y-direction, like a three-sided query. Each bucket contains $\bar{T}$ points and is constructed as follows: We start with $N/\bar{T}$ *active* buckets $b_1, b_2, \ldots, b_{N/\bar{T}}$ corresponding to the $N/\bar{T}$ slabs. The x-range of the slabs define a natural linear ordering on these buckets. We then imagine sweeping a horizontal sweep line from $y = -\infty$ to $y = \infty$. Every time the total number of points above the sweep line in two adjacent active buckets b_i and b_j in the linear order falls to $\bar{T}$, we mark b_i and b_j as *inactive*. Then we construct a new active bucket spanning the slabs spanned by b_i and b_j with a bottom y-boundary equal to the current position of the sweep line. This bucket replaces b_i and b_j in the linear ordering of active buckets intersected by the sweepline. The total number of buckets constructed in this way is $2N/\bar{T} - 1$, since we start with $N/\bar{T}$ buckets and the number of active buckets decreases by one every time a new bucket is constructed. Note that the procedure defines an *active* y-interval for each bucket in a natural way. Buckets overlap but the set of buckets with active y-intervals containing a given y-coordinate (the buckets active when the sweep line was at that value) are non-overlapping and span all the slabs. This means that the active y-intervals of buckets spanning a given slab are non-overlapping, and that the x-ranges and active y-intervals of all the buckets define a

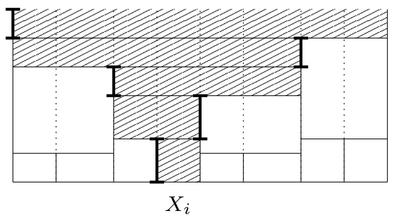


Figure 3: Slabs (dotted lines) and subdivision defined by the x-ranges and active y-intervals (solid lines) of the $2N/\bar{T} - 1$ buckets. The buckets spanning the slab X_i and their active y-intervals are highlighted.

planer (rectangular) subdivision of size $2N/\bar{T} - 1$. Refer to Figure 3.

After defining the $2N/\bar{T} - 1$ buckets, we are now ready to present the three-sided query data structure. It simply consists of a list $\mathcal{B}_i$ for each of the $2N/\bar{T} - 1$ buckets b_i storing the $\bar{T}$ points in b_i, as well as a cache-oblivious point location structure $\mathcal{T}$ on the subdivision defined by the buckets [10]. Since the subdivision is of size $O(N/\bar{T})$, the point location structure uses $O(N/\bar{T})$ space and answers a query in $O(\log_B(N/\bar{T})) = O(\log_B N)$ memory accesses [10]. The layout of the structure simply consists of $O(N)$ memory locations containing $\mathcal{T}$ followed by $\mathcal{B}_1, \ldots, \mathcal{B}_{2N/\bar{T}-1}$.

To answer a three-sided query $Q = [x_l, x_r] \times [y_b, \infty)$ where $|Q \cap S| \in [\bar{T}, 2\bar{T})$, we consider the buckets whose active y-intervals contain y_b. These buckets are non-overlapping and together they contain all points in Q since they span all slabs and have bottom y-boundary below y_b. A constant number, say K, of the buckets are intersected by Q: The two buckets containing x_l and x_r, as well as at most 3 buckets with x-range completely between x_l and x_r (since by construction every two adjacent active buckets contain at least $\bar{T}$ points above y_b). Refer to Figure 4. Thus to answer the query Q we can simply scan the list $\mathcal{B}_i$ of each of these buckets b_i and report the relevant points using $K \cdot O(\bar{T}/B) = O(T/B)$ memory transfers. To find the K buckets we first query $\mathcal{T}$ to find the bucket b_l whose active range contains (x_l, y_b). If b_l spans slabs $X_l, X_{l+1}, \ldots, X_m$ we then query $\mathcal{T}$ to find the bucket containing points in X_{m+1} with y-coordinate y_b. We continue this procedure for each of the K intersected active buckets. Since each query on $\mathcal{T}$ uses $O(\log_B N)$ memory transfers, the query Q is answered in $O(\log_B N + T/B)$ memory transfers in total.

Lemma 4. *Provided that the output size $T \in [\bar{T}, 2\bar{T})$ for a fixed value $\bar{T}$, there exists a cache-oblivious data structure for storing N points in the plane using $O(N)$ space, such that a three-sided range query can be answered in $O(\log_B N + T/B)$ memory transfers.*

3.2 General structure

Our general structure for answering three-sided queries on a set S of n points in the plane consists of a structure $\mathcal{C}$ for three-sided range counting, as well as $\log_2 N$ structures $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_{\log_2 N}$ for answering queries with known approximate output size. The structure $\mathcal{C}$ is implemented using the $O(N \log N)$ space (four-sided) counting structures described in Section 2 and each $\mathcal{T}_i$ is the linear space structures described above with $\bar{T} = 2^i$. Thus overall the structures uses $O(N \log_2 N)$ space.

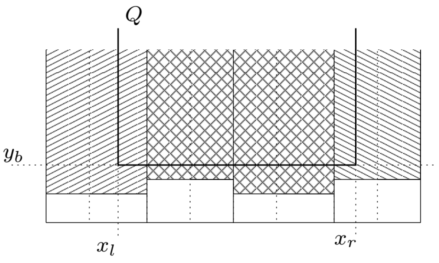


Figure 4: Buckets active at y_b. Only a constant number of the buckets are intersected by the query Q.

To answer a query Q we simply query $\mathcal{C}$ to compute the output size $T = |Q \cap S|$ and then we query the structure $\mathcal{T}_i$ where $2^i < T \leq 2^{i+1}$. Since the query on $\mathcal{C}$ uses $O(\log_B N)$ memory transfers (Theorem 1) and the query on $\mathcal{T}_i$ uses $O(\log_B N + T/B)$ memory transfers (Lemma 4) we obtain the following.

Theorem 3. *There exists a cache-oblivious data structure for storing N points in the plane using $O(N \log_2 N)$ space, such that a three-sided range query can be answered in $O(\log_B N + T/B)$ memory transfers.*

4. FOUR-SIDED RANGE QUERIES

Using our structure for three-sided queries, we can construct a cache-oblivious structure for general (four-sided) range queries on a set S of N points in the plane. The structure is similar to the internal and external range trees [6, 17] and utilizes the notion of "multislabs" that has been used in several external data structures (refer e.g. to [4]).

Our structure consists of a fanout $f = \sqrt{\log_2 N}$ base tree $\mathcal{T}$ on the N points in S sorted by x-coordinates. The base tree has height $O(\log_2 N / \log_2 \log_2 N)$ and can be laid out in memory by storing each node as a small $\log_2 \log_2 N$ height tree and using the standard van Emde Boas layout such that a root-leaf path can be traversed cache-obliviously in $O(\log_B N)$ memory accesses. With each node v of $\mathcal{T}$ we naturally associate a *slab* X_v: The slab X_l associated with a leaf l is defined by the x-interval formed by two consecutive points; the slab X_v of an internal node v is the union of the slabs associated with the children $v_1, v_2, \ldots, v_f$ of v. Thus the slab X_v is divided into f sub-slabs by the slabs associated with the children of v. We define a *multislab* of v to be a continuous range of its sub-slabs, that is, $X_v[i:j] = \bigcup_{l=i}^{j} X_{v_l}$ is the multislab consisting of subslabs X_{v_i} through X_{v_j}, for $1 \leq i \leq j \leq f$. There are $O(f^2) = O(\log_2 N)$ multislabs at v. Refer to Figure 5.

For an internal node v, let S_v be the points of X_v (i.e. the points from S residing in leaves below v). We store the points of S_v in $O(\log_2 N)$ secondary structures associated with v: One structure $\mathcal{L}_v$ on S_v for answering three-sided queries with the opening to the left, one structure $\mathcal{R}_v$ on S_v for answering queries with the opening to the right, and for each multislab $X_v[i:j]$ of v one cache-oblivious search tree $\mathcal{M}_{X_v[i:j]}$ on the points in S_v with x-coordinates in $X_v[i:j]$. The points in each $\mathcal{M}_{X_v[i:j]}$ tree are stored in y-coordinate order. The secondary structures are stored separately from the memory layout of the base tree $\mathcal{T}$. Separately we also

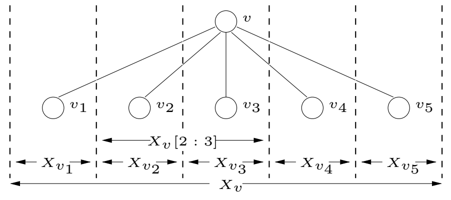


Figure 5: Node v of base tree $\mathcal{T}$. The slab X_v associated with v is divided into subslabs by the slabs associated with the children $v_1, \ldots, v_5$.

store for each node v of $\mathcal{T}$ an array of pointers to v's associated structures; the node v itself stores a pointer to the array. Using this array and index calculation based on i and j, we can access $\mathcal{L}_v$, $\mathcal{R}_v$, and any single $\mathcal{M}_{X_v[i:j]}$ in $O(1)$ memory transfers.

Since each point in S_v is stored in $O(f) = O(\log_2 N)$ linear space search trees $\mathcal{M}_{X_v[i:j]}$, and in two $O(N \log_2 N)$ space structures $\mathcal{L}_v$ and $\mathcal{R}_v$, the secondary structures of v occupy $O(|S_v| \log_2 |S_v|)$ space in total. Since each point is stored in the secondary structures of the $O(\log_2 N / \log_2 \log_2 N)$ nodes on one root-leaf path of $\mathcal{T}$, our structure uses $O(N \log_2^2 N / \log_2 \log_2 N)$ space overall.

To answer a range query $Q = [x_l, x_r] \times [y_b, y_t]$, we search down $\mathcal{T}$ using $O(\log_B N)$ memory transfers to find the first node v where x_l and x_r are contained in different children v_i and v_j of v. To answer Q on S_v, and thus on S, we then first query $\mathcal{R}_{v_i}$ to find the points from X_{v_i} lying in Q and query $\mathcal{L}_{v_j}$ to find the points from X_{v_j} lying in Q. Finally, if $j > i+1$, we perform a (one-dimensional) range query with $[y_b, y_t]$ on $\mathcal{M}_{X_v[i+1:j-1]}$ to find the remaining points from $X_{v_{i+1}}, X_{v_{i+2}}, \ldots, X_{v_{j-1}}$ lying in Q. It is easy to see that this correctly reports all T points in Q; refer to Figure 6. Since each of the three queries uses $O(\log_B N + T/B)$ memory access, we have obtained the following.

THEOREM 4. *There exists a cache-oblivious data structure for storing N points in the plane using $O(N \log_2^2 N / \log_2 \log_2 N)$ space, such that a four-sided range query can be answered using $O(\log_B N + T/B)$ memory transfers.*

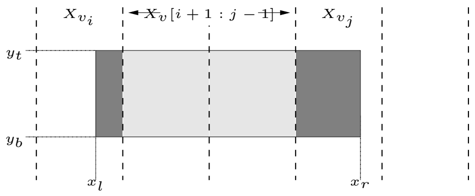


Figure 6: Answering query in v by answering three-sided queries on points in X_{v_i} and X_{v_j}, as well as a range query on points in multislab $X_v[i+1, j-1]$.

5. SEMIGROUP RANGE SUM QUERIES

In this section we describe how the range counting structure in Section 2 can be modified to support *semigroup range sum queries*, where each point has an associated weight from some semigroup and the output of a range query is the sum (with respect to the semigroup operator, e.g. max) of the weights of the points contained within the query range.

Three-sided semigroup range sum queries are supported in $O(\log_B N)$ memory transfers using $O(N \log_2 N)$ space, and four-sided semigroup range sum queries are supported in $O(\log_B N)$ memory transfers using $O(N \log_2^2 N / \log_2 \log_2 N)$ space. The space bounds hold for semigroups where points can be represented in $O(1)$ space.

5.1 Three-sided queries

The three-sided data structure is identical to the $O(N \log_2 N)$ space cache oblivious range counting data structure described in Section 2, except that a point p_i in a list $\overline{L}_v$ now stores two values: leftsum(p_i) and rightsum(p_i). The value leftsum(p_i) is the sum of the weights of the points in $L_{\text{left}(v)}$ with y-coordinate at most $y(p_i)$. Similarly, rightsum(p_i) is the sum of the weights of the points in $L_{\text{right}(v)}$ with y-coordinate at most $y(p_i)$.

To perform a query $Q = [x_l, x_r] \times (-\infty, y_t]$ we follow the path to the leaf of $\mathcal{T}$ storing the point with the smallest x-coordinate larger than or equal to x_l, and the path to the leaf storing the largest x-coordinate less than or equal to x_r. For each of the visited nodes v we locate the point p_i in $\overline{L}_v$ with largest y-coordinate that is less than or equal to y_t. For all nodes on the search path to x_l and x_r, respectively, that are not nodes on the search path to x_r and x_l, respectively, we sum up rightsum(p_i) and leftsum(p_i), respectively. Finally, we add the weight of the two leaves if they are contained within the query rectangle.

Since the data structure is identical to the data structure in Section 2, except for the additional $O(1)$ information at the nodes of the $\overline{L}_v$ lists, the $O(N \log_2 N)$ space bound follows from Section 2. Similarly following each of the two root-to-leaf paths uses $O(\log_B N)$ memory transfers, i.e. a three-sided semigroup range sum query requires in total $O(\log_B N)$ memory transfers.

THEOREM 5. *There exists a cache-oblivious data structure for storing N points in the plane using $O(N \log_2 N)$ space, such that a three-sided semigroup range sum query can be answered in $O(\log_B N)$ memory transfers.*

5.2 Four-sided queries

To support four-sided semigroup range sum queries we apply the approach used in Section 4 for reducing four-sided range queries to three-sided range queries, by using a fanout $\sqrt{\log_2 N}$ base tree $\mathcal{T}$ where each node v spans a slab which is partitioned into $\sqrt{\log_2 N}$ sub-slabs by the children of v. In the following we will adopt the notion used in Section 4.

For each node v of $\mathcal{T}$ we store S_v in two three-sided structures (as described in Section 5.1) $\mathcal{L}_v$ and $\mathcal{R}_v$, respectively, to answer three-sided queries with the opening to the left and right, respectively. For each of the $O(\log_2 N)$ multislabs $X_v[i:j]$ we have a cache-oblivious search tree $\mathcal{M}_{X_v[i:j]}$ storing the points in $S \cap X_v[i:j]$ at the leaves sorted with respect to y-coordinate. With each node u in $\mathcal{M}_{X_v[i:j]}$ we store the two sums leftsum(u) and rightsum(u) of the weights of the points in the left and right subtree of u, respectively.

A query is performed in a similar way as described in Section 4. First we search $\mathcal{T}$ to find the node v where x_l and x_r are contained in different sub-slabs X_{v_i} and X_{v_j} of v. We then query $\mathcal{R}_{v_i}$ and $\mathcal{L}_{v_j}$ with $[x_l, \infty) \times [y_b, y_t]$ and $(-\infty, x_r] \times [y_b, y_t]$, respectively. If $i+1 < j$, we query $\mathcal{M}_{X_v[i+1:j-1]}$ with

$[y_b, y_t]$, where we sum up the rightsum(u) and leftsum(u) values for the nodes u on the search paths for y_b and y_t, respectively, i.e. summing up the weight of the subtrees of $\mathcal{M}_{X_v[i+1:j-1]}$ spanned completely by the range $[y_b, y_t]$.

A node v in the base tree occupies $O(|S_v| \log_2 |S_v|)$ space for the two three-sided structures $\mathcal{L}_v$ and $\mathcal{R}_v$, and $O(|S_v|)$ for each of the $O(\log_2 N)$ multislab structures $\mathcal{M}_{X_v[i:j]}$. Therefore each level of the base-tree $\mathcal{T}$ uses $O(N \log_2 N)$ space, with a total of $O(N \log_2^2 N / \log_2 \log_2 N)$ space. A query uses $O(\log_B N)$ memory transfers since the searches in $\mathcal{T}$ and $\mathcal{M}_{X_v[i+1:j-1]}$ use $O(\log_B M)$ memory transfers and the two three-sided semigroup range sum queries also use $O(\log_B M)$ memory transfers.

THEOREM 6. *There exists a cache-oblivious data structure for storing N points in the plane using $O(N \log_2^2 N / \log_2 \log_2 N)$ space, such that a four-sided semigroup range sum query can be answered using $O(\log_B N)$ memory transfers.*

6. CONCLUSION

We have presented cache-oblivious data structures for various planar range problems all achieving optimal query time bounds. Four-sided range counting queries can be answered in $O(\log_B N)$ memory transfers using $O(N)$ space. General semigroup range sum queries can be answered in $O(\log_B N)$ memory transfers, using $O(N \log_2 N)$ space for three-sided queries and $O(N \log_2^2 N / \log_2 \log_2 N)$ space for four-sided queries. Range reporting queries can be answered in $O(\log_B N + T/B)$ memory transfers, where T is the number of reported points, using space $O(N \log_2 N)$ for three-sided range queries and $O(N \log_2^2 N / \log_2 \log_2 N)$ space for four-sided range queries.

Many problems remain open. Can the space for three-sided and four-sided range queries be reduced to $O(N)$ and $O(N \log_2 N / \log_2 \log_2 N)$ respectively, matching the optimal space bounds for the I/O-model? Can we achieve space $O(N)$ for special cases of the semigroup range sum problem, e.g. for max? Our data structures are for static point sets. Can the data structures be made dynamic, while maintaining optimal query times? Can the techniques be used for higher dimensional queries?

7. 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, Providence, RI, 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 algorithms. In *Proc. ACM Symposium on Theory of Computation*, pages 268–276, 2002.

[6] 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.

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

[8] 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.

[9] 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. 44th Annual Symposium on Foundations of Computer Science*, pages 271–282, 2003.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

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

[16] 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.

[17] B. Chazelle. Filtering search: a new approach to query-answering. *SIAM J. Comput.*, 15(3):703–724, 1986.

[18] B. Chazelle. A functional approach to data structures and its use in multidi mensional searching. *SIAM J. Comput.*, 17(3):427–462, 1988.

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

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

[21] 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.

[22] S. Govindarajan, P. K. Agarwal, and L. Arge. CRB-tree: An efficient indexing scheme for range-aggregate queries. In *Proc. International Conference on Database Theory*, pages 143–157, 2003.

[23] G. S. Lueker. A data structure for orthogonal range queries. In *Proc. 19th IEEE Symposium on Foundations of Computer Science*, pages 28–34, 1978.

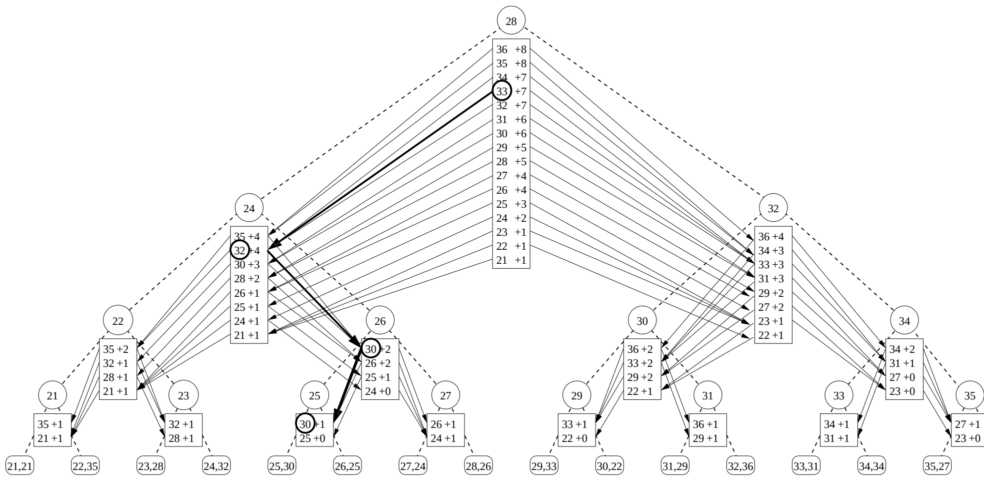
[24] O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter. Bkd-tree: A dynamic scalable kd-tree. In *Proc. International Symposium on Spatial and Temporal Databases*, pages 46–65, 2003.

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

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

[27] 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.

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



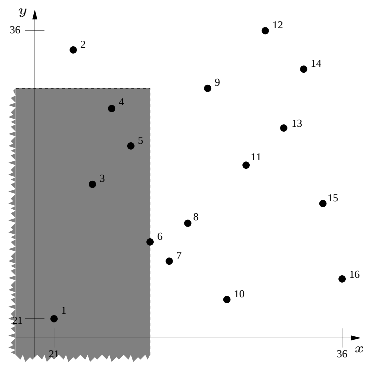


Figure 7: The basic range counting structure $\mathcal{T}$ for 16 points. Leaves show the coordinates of the points and internal nodes show the x-coordinates for the branching. The boxes show the L_v lists, where each line is the y-coordinate of a point p_i and a '+' followed by the $\mathrm{leftsum}(p_i)$ value (note that points are not stored in the data structure, the y-coordinates are only included for illustrative purposes). The search path for the query $Q = (-\infty, 26] \times (-\infty, 33]$ is emphasized.

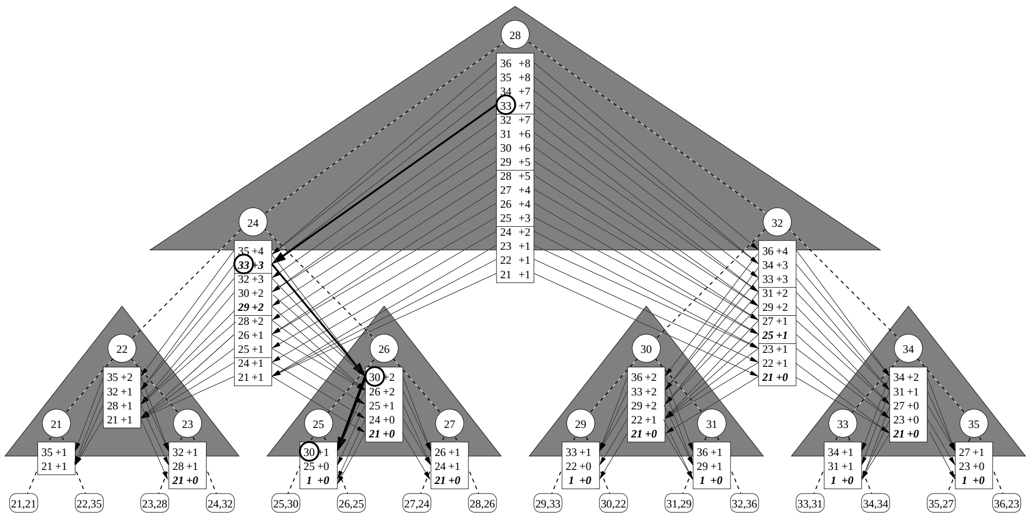


Figure 8: Illustration of the construction of the recursive layout including all dummy points introduced for the example in Figure 7. Dummy points are shown using italic numbers. The shaded triangles depict the toplevel partitioning, and the partitioning of the L_v lists in the top tree correspond to the four recursive layouts of the top tree (there is only one recursive layout for each of the buttom trees).