# Fixed Universe Computational Geometry

Chris Mario Christoudias [*] Paul Green [†] Eugene Hsu [‡]

## Abstract

We present a survey of algorithms and data structures that solve computational geometry problems under the assumption the input is drawn from a fixed universe. We show that many geometric problems can be solved faster in a fixed universe on the RAM. We overview the use of sorting, x-fast tries and stratified trees and combine these techniques to give efficient solutions to the orthogonal point location, orthogonal range query, and approximate nearest neighbor problems that beat their the algebraic decision tree lower bounds.

## 1 Introduction

Many computational geometry problems are based on the fact that points are real numbers. Arithmetic operations such as addition, multiplication, square roots, etc can be done in constant time. Under this model, many algorithms have been developed for classic problems such as convex hull and Voronoi diagrams that are optimal (more specifically, $\Omega(n \log n)$ by reduction to sorting under decision tree models).

In many practical applications, however, points can be represented in a fixed universe (in other words, a grid). By assuming a more powerful model of computation (that is, a word RAM), one can often devise data structures and algorithms that perform better than methods intended for algebraic decision tree models. An example outside of computational geometry is sorting, which has a well-know $\Omega(n \log n)$ bound under the decision tree model [CLRS01]. If all elements are assumed to be integer, however, one can use a method such as counting sort to achieve linear time (although it becomes linear in the size of the universe, which is often quite large). An immediate consequence is that many sweepline algorithms in computational geometry can also be solved in "linear" time as well.

Recently, researchers in computational geometry have been discovering the abundance of data structures available for performing efficient operations on an integer grid. As in the above sorting example, these data structures acheive better time bounds than those acheivable under a comparsion based computational model by cleverly exploiting the bit representations of the coordinates. A popular data structure used for fixed universe computational geometry is van Emde Boas' stratfied tree [van77], which can perform predecessor/successor queries in log-logarithmic time and linear space. Other sophisticated data structures include $*$-fast tries which can be cleverly combined and applied to achieve log-logarithmic time bounds for difficult problems in computational geometry.

In this paper we survey some rather nonobvious results that allow for very efficient operations for the problems of orthogonal point location and range queries, and approximate nearest neighbors. We begin in section 2 by describing some methods that have been developed for 1-dimensional fixed universe problems. In particular we discuss the one-dimensional stratified tree and $x$-fast trie. We then describe simple algorithms that can be solved on a grid. Finally, we present a survey of orthogonal point location in Section 4, orthogonal range queries in Section 5, and approximate nearest neighbor in Section 6 and provide conluding remarks in Section 7.

[*] cmch@mit.edu
[†] green@csail.mit.edu
[‡] ehsu@csail.mit.edu

# 2 Background

We begin by defining some common notation used in our paper.

- $U$ is the size of our universe. For simplicity, we assume that it is a power of 2.
- $n$ is the number of objects.
- $k$ is the number of answers in the output (for algorithms with output-sensitive time complexity).

While it is often helpful to think of the universe as the set of integers $[0,U)$ or $[1,U]$, it should be noted that it is often unnecessary to depend on this fact. Stating it another way, many of the results that we describe (with the notable exception of the approximate nearest neighbors described in section 6) do not rely on a uniform grid over the fixed universe. This allows us to generalize results to alternative representations of numbers such as IEEE floating point [Gol91].

We proceed by describing several algorithms and data structures that exploit the fixed universe assumption in one dimension.

## 2.1 Sorting

Sorting has a known lower bound of $\Omega(n \log n)$ under the decision tree computational model. In a fixed universe, the obvious algorithm is counting sort. Assuming a size $U$ universe, this algorithm traverses each point and hashes it in constant time into an array of size $U$. This takes $O(n)$. The array is then traversed in-order and the elements of the non-empty bins reported in sorted order. This gives an $O(n+U)$ running time for counting sort. Unfortunatley, this algorithm has a linear dependence on $U$, and requires $\Theta(U)$ space. Another simple, common integer sorting algorithm is Radix Sort. With this algorithm, a $O(n \log_n U)$ algorithm is acheived. Using dynamic Stratified Trees (see Section 2.3) (e.g. van Emde Boas Priority Queues) sorting can be accomplished in $O(n \log\log U)$ time by n successor queries (each running in $O(\log\log U)$ time). More recently, Andersson *et al.* [AHNR95] proposed an algorithm that allows sorting in $O(n \log\log n)$ time, independent of $U$, and $O(\sqrt{U})$ space. Note, the above sorting algorithms beat the $\Omega(n \log n)$ time bound by assuming a word RAM computational model, i.e. a unit-cost RAM that supports bit opertaions in constant time. In Section 3 we demonstrate how the above sorting techniques can provide simple, efficient solutions to problems in fixed universe computational geometry.

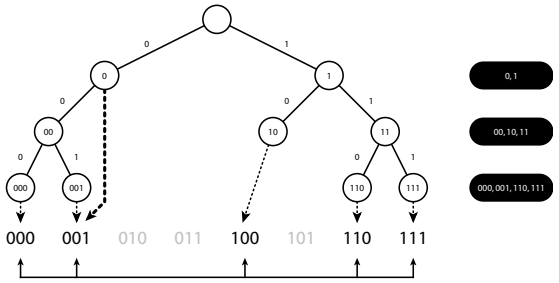



Figure 1: An $x$-fast trie that stores the set $\{0,1,4,6,7\}$ in the universe $[0,7]$. Leaves correspond to elements of the set. Note that the element $100 = 4$ is stored at the node labeled 10 to save space. Each element is connected to adjacent ones in a doubly linked list. The node labeled 0 has no right child, and thus a descendant pointer (thick dashed line) is stored to the maximum element of its left subtree. On the right, the existing prefixes at each level are shown.

## 2.2 $x$-Fast Tries

In a fixed universe, a binary trie is a natural way to store a set of elements for the purpose answering of existence queries in $O(\log U)$ time. Here, we will review a paper by Willard [Wil83] in which he modifies a standard binary trie to allow for $O(\log\log U)$ time existence queries. Furthermore, the resulting data structure will allow successor and predecessor queries in $O(\log\log U)$ time as well.

We begin by building a standard binary trie on all the elements. If a subtrie contains only one element, we save space by making the node a leaf and storing the element there. At this point, each leaf represents an element of the set. Since the height of the trie is $O(\log U)$, it follows that it takes $O(n \log U)$ space. We then augment the trie as follows.

- We add links between all elements and their predecessors and successors.
- If an internal node has no left subtree, we add a special *descendant link* to the smallest element of its right subtree. Analogously, if an internal node has no right subtree, we add a descentant link to the largest element of its left subtree.

An example of this modified trie is shown in figure 1.

Now, assume that we want to perform a successor query on some value $x$ (predecessor queries can be solved analogously). One way to do this is to just walk down the trie according to the binary representation of $x$. At some point, we'll stop because we can't go any farther. We call this node the *stopping node*. This node can also be interpreted as representing the deepest binary prefix of $x$ that is stored in the trie. There are two possible cases.

- Suppose that the stopping node is a leaf. We check whether it corresponds to $x$. If so, then $x$ is in the set, and we can follow the successor link. If not, the leaf node stores some other element $y$. Since $x$ and $y$ have the same binary prefix, and no other elements in the set have that prefix, $y$ is either a predecessor or successor of $x$. Since all elements are connected to their predecessor and successor, we can just walk a constant number of steps "sideways" from $y$ to find the successor of $x$.
- Suppose that the stopping node is not a leaf. There must be a descendant link; otherwise, we would have stopped at a deeper node. Following the descendant link will lead to an element $y$, which is either a predecessor or successor of $x$. We can then walk "sideways" as before to find the successor of $x$.

It is obvious that, after finding the stopping node, both cases are handled in constant time. Thus, the efficiency bottleneck is in finding the stopping node, which, as described, takes $O(\log U)$ time. This is pretty dismal, since we could just store all the elements in a balanced binary tree and get $O(\log n)$ time for all operations.

To make stopping node searches faster, consider storing an $O(U)$ sized array for each level of the trie. We use this array to check whether a certain binary prefix exists at a given level. Specifically, element $p$ of the array stored at level $i$ is null if the prefix $p$ does not exist at level $i$ of the trie. Otherwise, element $p$ stores a pointer to the corresponding node. Clearly, these checks can be performed in constant time.

Using these arrays, we can find the stopping node by performing binary search on the levels of the tree. One can alternatively interpret this as performing binary search on the binary representation of the query. It follows that we can find the stopping node in $O(\log\log U)$ time.

Of course, using these arrays is horribly inefficient with space. Since we're storing an $O(U)$ array at $O(\log U)$ levels, the total space consumption of the arrays is $O(U \log U)$. That's not very good, especially considering that we could just precompute predecessors and successors for all $U$ elements in the universe and store them in arrays. This would give us constant time queries and $O(U)$ space.

To remedy this, we use a data structure described by Fredman *et al.* [FKS82] which allows us to perform constant time existence queries on a set of $n$ elements drawn from a fixed universe using only $O(n)$ space. We use this data structure in place of the arrays at each level of the trie. The query time remains the same, and the total space consumption becomes a much more palatable $O(n \log U)$. This data structure is known as an $x$-fast trie.

It is possible to achieve $O(n)$ space consumption while retaining the $O(\log\log U)$ query times by *pruning*. The resulting data structure is called a $y$-fast trie. In the interest of brevity, we will not describe it here. Details can be found in Willard's paper [Wil83].

## 2.3 Stratified Trees

A Stratified Tree over a universe $[U]$ is a tree where each node represents some range in $[U]$. The root node represents the entire range U. Each internal node $r$ has $\sqrt{U'}$ children (where $U'$ is size of the range $r$ represents). At each successive level of the tree, a range of size $U'$ is split into $\sqrt{U'}$ ranges of size $\sqrt{U'}$. Therefore the height of

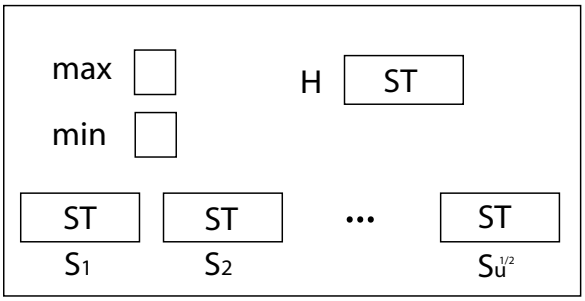



Figure 2: The recursive structure of a Stratified Tree over the universe of size $U$. Each box labeled ST is itself a Stratified Tree over a universe of size $\sqrt{U}$. The max and min element are stored non-recursively. H is a "summary" of the non-empty $S_i$.

a Stratified Tree is $\log\log U$. We now describe the construction and fields of a Stratified Tree data structure.

Given $[U]$ and $P \subseteq [U]$ we build a recursive search structure called a Stratified Tree [van77] (denoted ST($[U]$,$P$), or simply ST) that supports successor queries in $O(\log\log U)$ time. The original stratified tree described by van Emde Boas[van77] uses $O(U)$ space, however by using perfect hashing techniques[DKM$^+$88], the space requirement can be reduced to $O(n)$. For simplicity, the data structure we describe is static, but can easily be made into a dynamic[Wil83] structure (commonly called a Priority Queue) that achieves the same time bounds.

Let $M$ = max$\{P\}$, $m$ = min$\{P\}$, and $P' = P \setminus \{M,m\}$. We non-recursively store max and min at ST, e.g. ST.max = $M$, ST.min = $m$. Next divide $[U]$ into $\sqrt{U}$ ranges, $S_i$, $i \in [\sqrt{U}]$, each of size $\sqrt{U}$. Let $I = \{i | S_i \cap P' \neq \emptyset\}$, be the set of indicies of the non-empty ranges. Insert each $i \in I$ into a hashtable (using perfect hashing [DKM$^+$88]) that maps $i \in I$ to $S_i$. At each non-empty $S_i$, recursively store ST($\sqrt{U}$,$P' \cap S_i$), e.g. build a stratified tree in a universe of size $\sqrt{U}$ for the points that lie in the range $S_i$. Lastly compute $H$ = ST($\sqrt{U}$,$I$). $H$ is a stratified tree that stores the indicies $I$ of the non-empty ranges $S_i$.

### 2.3.1 Successor Queries

We outline the successor procedure for a stratified tree. Let $q$ be a query, and $P$ be values that lie in the universe $[U]$. If $q$ is less that the minimum element stored in our universe $[U]$.min, then the successor of $q$ is clearly $[U]$.min. Suppose $q$ falls within some range $S_i$. If $q$ is smaller than the largest element in $S_i$, then we know that Successor(q) is upper bounded by $S_i$.max, and inparticular, must lie in the range $S_i$. If $q$ happens to be larger than all values stored in range $S_i$, then Successor(q) must be the minimum value in the next non-empty range $S_j$ ($i < j$). Fortunetely $H$, which stores the indicies of non-empty ranges, is also a stratified tree, and thus supports successor queries.

**Successor**($P$,$q$)
  **if** q $< U$.min **then**
    return $U$.min
  **else**
    Let q be in range $S_i$
    **if** q $< S_i$.max **then**
      return Successor($S_i$,q)
    **else**
      k = Successor($H$,$i$)
      return $H[k]$.min
    **end if**
  **end if**

Figure 3: The stratified tree successor procedure.

**Lemma 2.1.** *Successor takes time $O(\log\log U)$ time.*

*Proof.*

$$\begin{aligned} T(U) &= T(\sqrt{U}) + O(1) \\ T(2^m) &= T(2^{m/2}) + O(1) \\ T'(m) &= T'(m/2) + O(1) \\ T'(m) &= O(m^{\log_2 1} \log m) \text{ (by Master's Theorem)} \\ \Longrightarrow T(U) &= O(\log\log U) \end{aligned}$$

□

# 3 Applications of Integer Sorting

We can use the improved sorting bounds in a fixed universe (described in Section 2.1) to solve other computational geometry problems faster than traditional algebraic decision tree based lower bounds. In particular, there are many algorithm that are asymptoticaly limited by a sorting step. For example, *Graham's scan* algorithm [dvOS00] for computing the convex hull, sorts the points and then does $O(1)$ linear scans of the sorted points. In a pure comparison model, this algorithm has $\Theta(n \log n)$ running time, due purely to the sorting step. Using integer sort [AHNR95] and the fixed universe RAM model, we can sort $n$ points in $O(n \log\log n)$ time. The remainder of *Graham's scan* algorithm is unchanged and thus we arrive at a $O(n \log\log n)$ algorithm for Convex Hull. We can leverage our fast convex hull to break the comparison based lower bounds of several other geometric problems. Given the convex hull of a set of points, the rotating calipers[Tou83] technique can be used to solve many problems (e.g. diameter, width, etc.) in $O(n)$ time thus we have $O(n \log\log n)$ algorithms for many common problems. We refer the reader to the survey [Ove88a] which describes more results that are of this nature.

# 4 Orthogonal Point Location

When visiting a new place a common question is "Where am I?". This question is the topic of point location in computational geometry. Provided a map of the universe, point location algorithms seek efficient solutions to this question. In general, this "map" is defined by a set of possibly overlapping regions in $\Re^d$. Given a query point $q \in \Re^d$ point location algorithms return the set of regions that contain $q$.

*Planar* point location algorithms solve point location in 2-dimensions. The typical scenario is outlined in Figure 4. In the figure, a set of $n$ non-crossing line segments define a subdivision of the plane. There are many well known, optimal solutions to planar point location that perform a query in $O(\log n)$ time and use $O(n)$ space. One such algorithm is the persistent data structure algorithm of Sarnak and Tarjan [ST86]. A more commonly used algorithm is that of Seidel [Sei91] that involves randomized incremental construction. An optimal solution to point location in higher dimensions remains somewhat of an open problem [dvOS00]. Nonetheless, efficient solutions solve queries in $O(n^{1-1/\lfloor d/2 \rfloor} \log^{O(1)} n)$ and with linear space [Mat92].

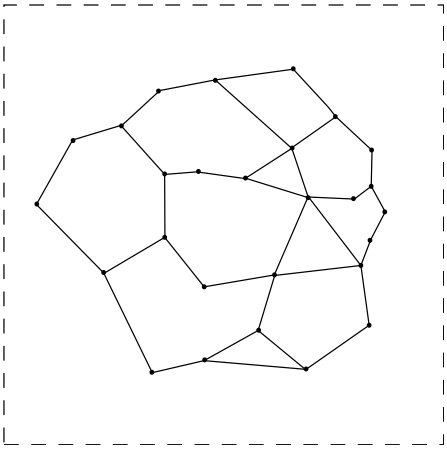


Figure 4: Planar point location on subdivision defined by $n$ line segments.

Thus far we have only considered point location in a real-valued space. In a fixed universe better time bounds can be achieved. For example, a simple but effective solution is to pre-compute the set of regions contained by each point of the integer grid. This algorithm is commonly used in the graphics community for performing planar point location in OpenGL [WNDS99]. It can answer point location queries in constant time, however, it utilizes $O(U^2 \log n)$ space. In the following sections we discuss more space efficient algorithms for performing point location that utilize stratified trees to acheive log logarithmic query times and linear space. Although, these data structures are defined in the context of orthogonal point location (i.e. point location in rectangular subdivisions) de Berg *et al.* discuss extensions of their algorithm to point location on c-oriented polygons and fat triangles. They also discuss how they can perform ray-tracing. The interested reader may consult the paper for a description of the extensions and applications of the orthogonal point location algorithm presented below.

## 4.1 Fixed Universe Orthogonal Point Location

In [dvS92], de Berg *et al.* define a data structure that answers orthogonal point location queries in a fixed universe, in log-logarithmic time and linear space. Here, *orthogonal* is used to specify that the proposed algorithm works on spaces subdivided into rectangles. It is further assumed that these rectangles are disjoint. This algorithm outperforms the poly-log time bounds reported by Edelsbrunner *et al.* for point location in rectangular subdivisions of $d$ dimensions in real-valued spaces [EHH86]. In their paper, de Berg *et al.* apply the above data structure to solve point location queries in one, two and three-dimensions. We present a concise explanation of their method for orthogonal point location in a fixed universe below.

### 4.1.1 The Interval Trie

The interval trie is the underlying data structure used for orthogonal point location. It is used to efficiently store and query a set of one dimensional intervals. Unlike an interval tree [CLRS01], the interval trie is a static data structure that stores redundant copies of each interval. As will be shown in a later section, this property allows the interval trie to be easily modified to support fast point location queries. In a fixed universe, an interval is defined by two endpoints in $[1,U]$ that are either *closed* or *open* depending on whether they are part of the interval. For point location we assume that all intervals are closed on the left and open on the right. An interval trie is a complete binary trie of size $O(U)$ that stores the intervals of $[1,U]$.

An interval trie is illustrated by Figure 5. The leaves of the tree are labeled $1,2,...U$. Each node $\tau$ is associated a range $\rho(\tau)$ that is an interval of $[1,U]$. A leaf node with label $j$ has range $\rho(j) = [j-1/2, j+1/2)$. The range of an internal node is defined by considering the leaves of the subtree rooted at that node. It is computed by taking the union of the range of each leaf in the subtree. Consider an interval $I$ stored by the tree. This interval overlaps a node $\tau$ in one of four ways as illustrated in Figure 6. If $\rho(\tau)$ and $I$ overlap completely then one interval contains the other.

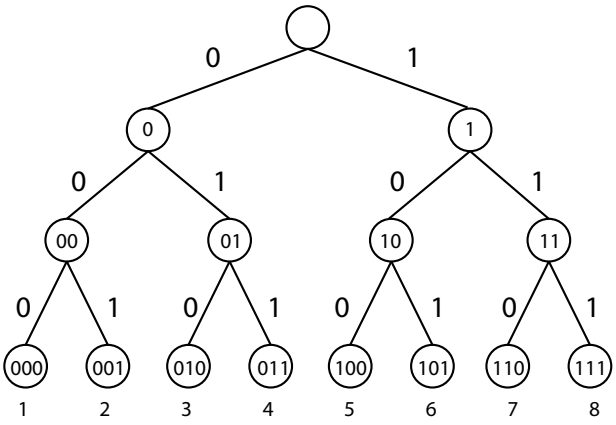



Figure 5: Interval trie built on universe of size $U = 8$.

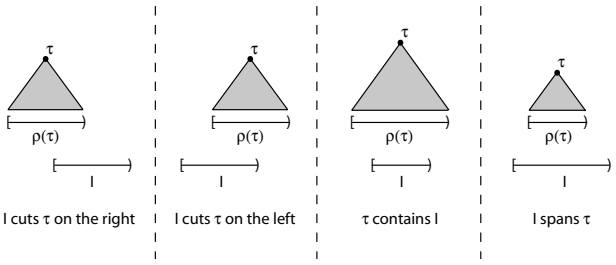



Figure 6: An interval $I$ can overlap a node $\tau$ of the interval trie in one of four ways.

Otherwise, we say that $I$ cuts $\tau$.

We insert an interval into the tree by giving it to each node that it cuts in the interval tree as illustrated by Figure 7. If the interval cuts a node from the right we store its lower bound in that node. If it cuts the node from the left we store its upper bound. Note that each interval is associated a level $l_I$ in the trie, at which it begins to cut nodes. $I$ is contained by exactly one node in each level above $l_I$ and is cut by exactly two nodes at and below this level as demonstrated in Figure 7. This is an important property that we exploit in a later section to achieve efficient space bounds. With the exception of the above properties, the interval trie can be viewed as a binary trie of that stores the endpoints of each interval (see Section 2.2).

Consider storing a set of intervals of $[1,U]$ into the trie.

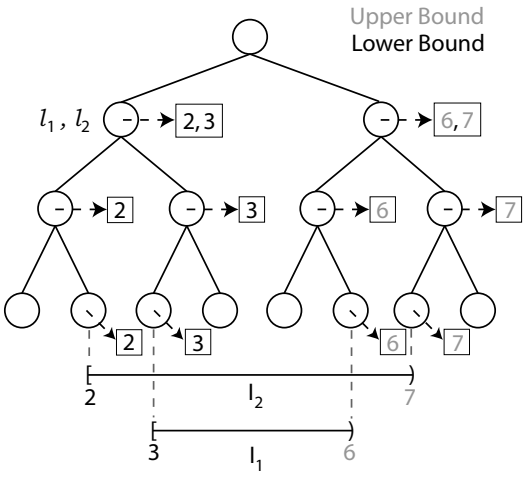



Figure 7: The endpoints of each interval are stored by the nodes it cuts in the interval tree. Each interval cuts two nodes starting at some level $l_I$ in the interval tree.

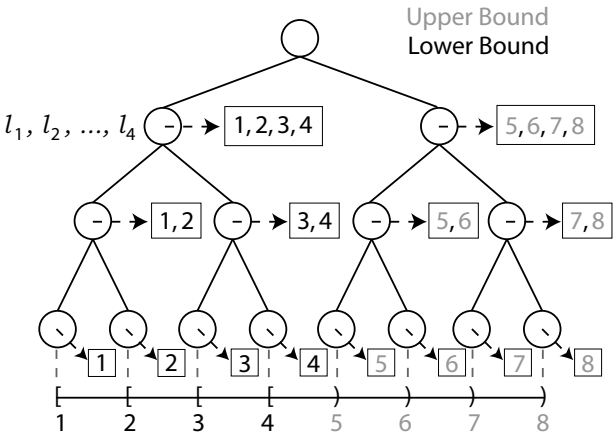



Figure 8: A set of intervals, in which each interval cuts $O(\log U)$ levels of the tree. Such a worst case scenario results in an $O(n \log U)$ query time.

In the worst case all the intervals are stored at each level of the tree (see Figure 8). We can use this trie to perform one-dimensional point location: given a query point $q$, we traverse down the trie comparing it to the intervals stored by its corresponding node at each level. Along this path we report the intervals that contain $q$. $q$ is compared to at most $n$ intervals at each node along its path from the root to a leaf of the trie, which results in a query time of $O(n \log U + k)$. Here, $k$ is the number of intervals that contain $q$. In the case of orthogonal point location, each interval is disjoint and thus $q$ is contained by at most one interval. The query time is therefore reduced to $O(n \log U)$.

As presented, the interval trie results in an in-efficient solution to one-dimensional point location. In the following section we demonstrate how to alter the interval trie to acheive log-logarithmic query time and linear space.

### 4.1.2 Orthogonal Point Location in One Dimension

In one dimension, orthogonal point location subdivides the universe into $n$ disjoint intervals of $[1, U]$. Consider storing these intervals into an interval trie (see Figure 9). Note, because the intervals are disjoint, each node of the interval trie is cut by at most two intervals. As mentioned

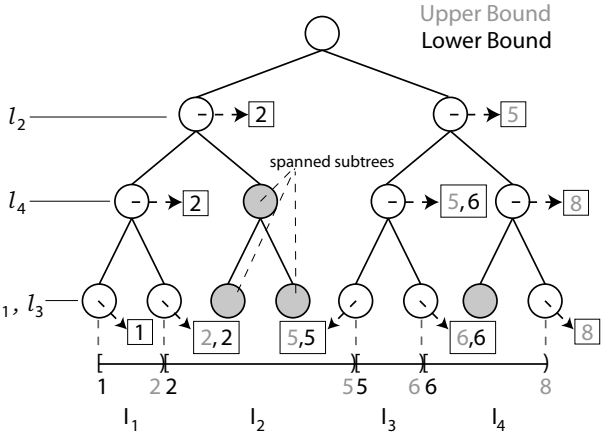



Figure 9: Orthogonal point location in one dimension - the interval trie stores disjoint intervals of $[1, U]$ that subdivide the universe. Each node of the interval tree is either cut or spanned by some interval.

above, direct use of this data structure results in inefficient query and space bounds for one-dimensional point location. Below we demonstrate how to achieve more efficient bounds by altering this data structure.

To improve query time, the interval trie is represented as an x-fast trie (see Section 2.2). Using this data structure, a query is satisfied by performing binary search on the levels of the interval trie as described below. Since the interval trie has $\log U$ levels, this algorithm results in $O(\log\log U)$ query time.

Consider the query point $q$ and level $l$ of the interval trie. Using the perfect hashing scheme of the x-fast trie we can locate the subtree that contains $q$ at level $l$ in constant time. As depicted in Figure 9, the root, $\tau$, of this subtree is either cut or spanned by an interval. If it is spanned, then $\tau$ is empty, i.e. it does not store the endpoints of any intervals. Clearly, the interval that spans $\tau$ cuts nodes higher in the tree. To find this interval, we therfore continue the search at a higher level. If it is cut, it will contain the endpoints of the (at most) two intervals that cut it. In constant time we check if $q$ belongs to either of these intervals. If so we stop and report the interval containing $q$. If $q$ is in neither interval, then $q$ is inside an interval that is contained by $\tau$ (see Figure 9). We therefore continue the search deeper in the tree.

The space consumption of the interval trie is improved by superimposing a level-search tree onto its levels as illustrated by Figure 10. A *level-search tree* is a $b$-ary tree that stores the levels of the interval trie. As depicted in the figure, each node of the level-search tree is assigned $b-1$ levels. The key insight is that maintaining a full interval trie of size $O(u)$ is waistful if $n \ll U$, as is often the case. A level-search tree is used as a sparse representation of the trie, in which only full nodes (i.e. nodes storing intervals) are maintained by the level-search tree.

We refer to the combined data structure as a deBerg tree. Intervals are stored into the deBerg tree as illustrated in Figure 11. An interval $I$ is given to the two nodes it cuts at level $l_I$ of the interval trie, associated with node $v$ in the level search tree. Traversing a downward path from $v$ to the root of the level-search tree, $I$ is also given to an associated level of each node of the level-search tree along this path. Note, at a given level of the interval trie, an interval is stored in at most two nodes. Assuming the level search tree has height $h$, an interval is stored in at most $h$ levels. For $n$ intervals this results in $O(nh)$ space for the deBerg tree. Note the level-search tree has $O(\log U)$ leaves and therefore its height is found as $h = \Theta(\log_b \log U) = \Theta(\log\log U / \log b)$.

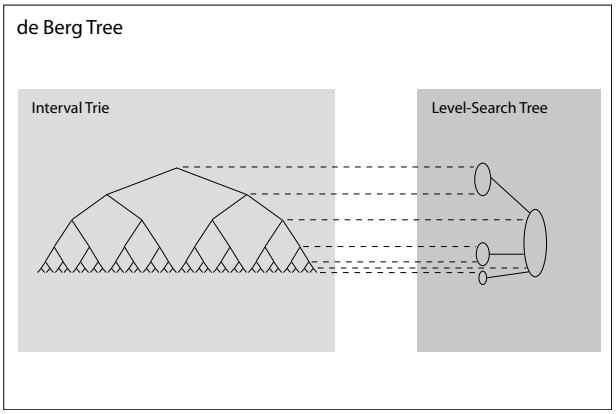



Figure 10: A deBerg tree. The deBerg tree is formed by overlaying a level-search tree over the leves of the interval trie. The dotted lines that extend from the interval trie, specify how its levels are allocated to each node of the level search tree. Note, each level-search tree node is assigned at most $b-1$ levels from the interval trie (in the above deBerg tree $b = 3$).

Using the deBerg tree, point location is performed in a similar fashion to binary search on the interval trie explained above (see Figure 12):

- At a node $v$ of the level search tree we check its $b-1$ assocated levels for subtrees with a full root, that contains our query point.
- Suppose a set of such subtrees exist. We check $q$ against the (at most) two intervals stored at each full node. If $q$ is found to be within an interval, we stop and report the interval. Otherwise, $q$'s interval is contained by the deepest subtree - we continue the search in the child directly under the deepest associated level in which a subtree with a full root was found. (Cases 1 and 2 of Figure 12).

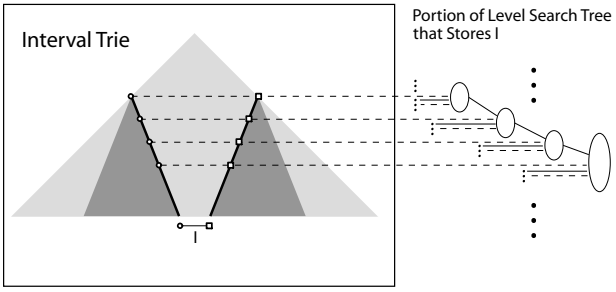



Figure 11: An interval is given to the levels of each node $v$ in the level-search tree, along the path from the child node assigned level $l_I$ to the root. Note, the interval $I$ cuts all the nodes of the paths along the black solid lines in the interval trie. In this fashion, an interval is assigned to at most $2h$ levels of the interval trie, implicitly represented by the deBerg data structure.

.

- Suppose no such subtree exists in $v$. This indicates that the interval containing $q$ spans all the levels associated with $v$ and we continue our search in the highest child of $v$. (Case 3 in Figure 12)

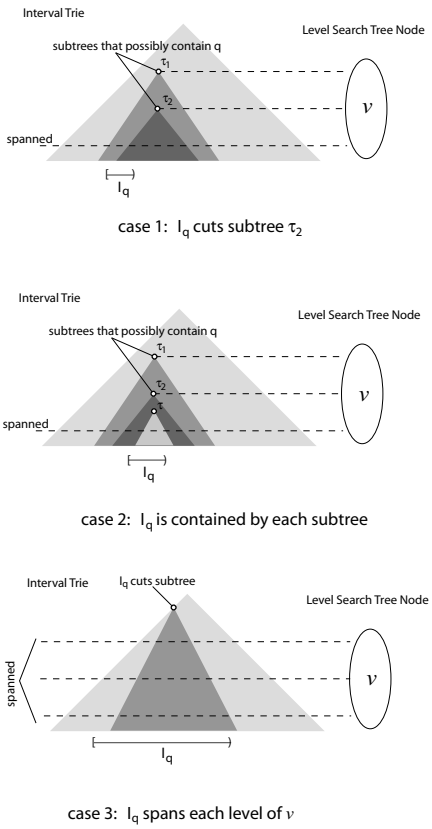



Figure 12: At a level-search node, $v$, a query point $q$ is found in one of three locations. Let $I_q$ denote the interval that contains $q$. Case 1: If $I_q$ cuts a subtree stored by a level of $v$,we stop our search at that node and report $I_q$. Case 2: Each subtree with a full root, possibly cut by $I_q$, actually contain $I_q$. In this scenario, $I_q$ cuts a subtree directly below the deepest full root subtree of $v$. As shown in the figure, this is because the next level below this deepest subtree is spanned by $I_q$. This indicates that $I_q$ cuts a node ($\tau$) of the interval trie that is stored by the subtree rooted at the child of $v$ in between these levels in the interval trie. Case 3: If $I_q$ spans all the levels of $v$ it cuts a node higher in the tree and we therefore check $v$'s highest child for this node.

The proposed algorithm checks a query point against the $b-1$ levels of a level-search tree node in constant time. This is done at most $h$ times to find its associated interval (i.e. a path traversal in the level-search tree is performed). The resulting query time is $O(hb)$.

The deBerg tree of Figure 10 exhibits an inherent tradeoff between space and time depending on whether we hold $h$ or $b$ constant. Holding $h$ constant gives linear space and $O((logU)^{1/h})$ query time, whereas assuming $b$ as constant gives $O(\log\log U)$ query time and $O(n\log\log U)$ space. In one dimension, this tradeoff can be compensated for by applying the prunning technique of van Emde Boas [van77] to achieve $O(\log\log U)$ query time while maintaining linear space. In the following section we demonstrate how this stratified tree may be extended to handle orthogonal point location queries in two and three dimensions.

### 4.1.3 Orthogonal Point Location in Two and Three Dimensions

To perform orthogonal point location in two and three dimensions we add secondary structure to a one-dimensional deBerg tree. This is analagous to the range trees of Section 5.1.1. We first consider the two-dimensional case below.

Consider a rectangular subdivision constructed from $n$ disjoint rectangles. Such a subdivision is illustrated in Figure 13. We store this rectangular subdivision using a two-dimensional deBerg tree, defined as follows:

- Project each rectangle onto the x-axis and store these intervals into a one-dimensional deBerg tree (middle of Figure 13).
- Next, consider a node $\tau$ of the interval trie implicitly stored by the one-dimensional deBerg tree. This node is assigned a range $\rho(\tau) = [x_{\min}, x_{\max})$ defined by the subtree rooted at $\tau$. In two dimensions, this range is defined by two vertical lines situated at $x = x_{\max}$ and $x = x_{\min}$ as illustrated in Figure 13. These vertical lines intersect a set of rectangles in the subdivision. The rectangles whose x-intervals cut these lines are stored by $\tau$.
- Consider the intervals formed by the intersection of each vertical line with the rectangles that cut them. To locate the rectangle that contains a query point $q$, for each vertical line we perform a one-dimensional point location on its associated intervals. The intervals of each line, however, are not a subdivision in general, since rectangles that span $\tau$ generate "gaps" betweem the intervals along each line (see bottom of Figure 13). To remedy this we fill these gaps, which at most doubles the number of intervals stored by $\tau$. We then build one-dimensional deBerg trees on the subdivision of each vertical line and store each tree at $\tau$.

Consider the space bounds of the two dimensional deBerg tree. As in its one-dimensional version, the x-interval of each rectangle is assigned to $O(2h)$ nodes of the deBerg tree. Therefore, inserting each rectangle into

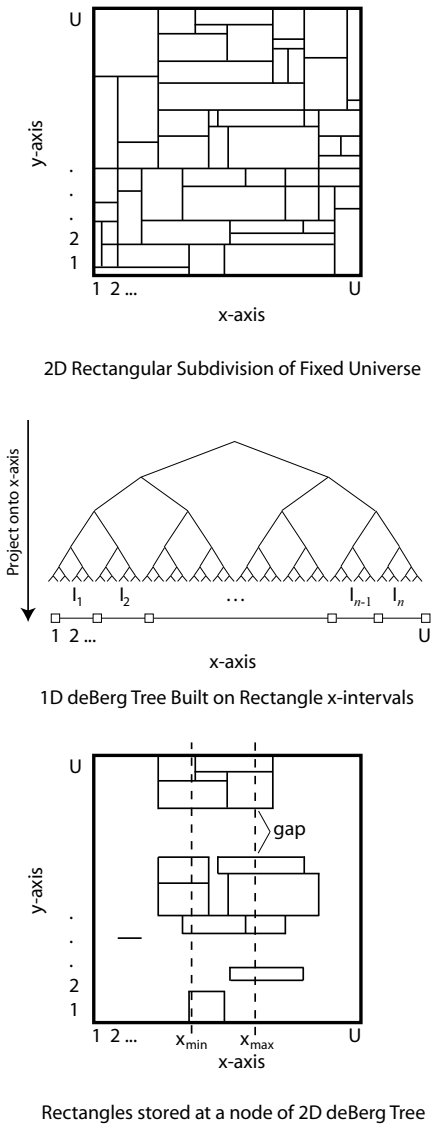



Figure 13: Orthogonal point location in two dimensions. (top) A rectangular subdivision of the fixed universe of size $[U]^2$. (middle) Each rectangle is projected onto the x-axis and its x-interval is stored by the primary structure of the two-dimensional deBerg tree. (bottom) Each node of the two-dimensional deBerg tree stores is cut by a set of rectangles, stored by the secondary structure of the tree.

the tree gives $O(nh)$ storage space. This space bound assumes that we represent the one-dimensional stratified trees of each node using the pruning technique of [dvS92] to achieve linear space and a $O(\log\log U)$ query time. Note, in order to form the secondary structure at each node of the deBerg tree we needed to perform a gap filling step. Since this can only at most double the number of intervals stored by the secondary structure, the space bound is conserved.

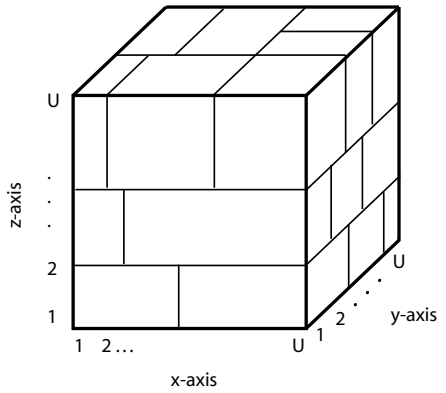
The algorithm for performing two-dimensional point location on the above data structure is analogous to the one-dimensional case:

- At a node $v$ of the level search tree we check its $b-1$ associated levels for a subtree with full root, that contains our query point.
- Suppose a set of such subtrees exist. Each subtree stores a set of rectangles that cut its root. To find if $q$ is within a rectangle, we perform a one dimensional query at each full root node using the 1D deBerg trees that these nodes store. If $q$ is found to be inside a rectangle, we stop and report it. Otherwise, $q$ is inside a rectangle that is contained by the deepest subtree - we continue the search in the child of $v$ that is directly under the deepest level in which a subtree with a full root was found.
- Suppose no such subtree exists in $v$. This indicates that the rectangle containing $q$ spans all the levels associated with $v$ and we continue our search in the highest child of $v$.

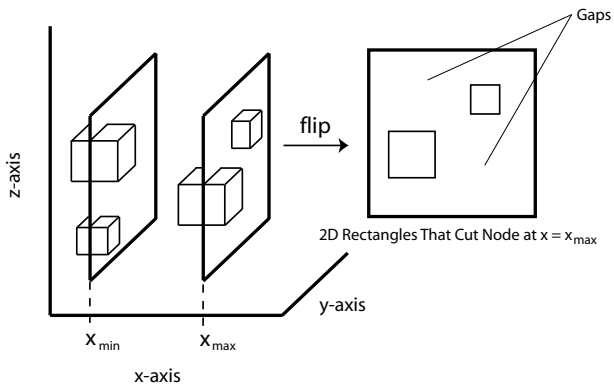
As in one-dimension, the above algorithm checks at most $O(hb)$ subtrees of the deBerg tree to satisfy a query. At each subtree two one-dimensional point locations are performed resulting in an $O(hb(\log\log U))$ query time.

Next, consider three-dimensional orthogonal point location. A rectangular subdivision in three dimensions is illustrated by Figure 14. We store this rectangular subdivision using a three-dimensional deBerg tree, defined as follows:

- Store the x-intervals of each rectangle into a one dimensional deBerg tree.




3D Rectangular Subdivision in a Fixed Universe




Rectangles Stored at a Node of the 3D deBerg Tree

Figure 14: Orthogonal point location in three dimensions. (top) A rectangular subdivision of the fixed universe of size $[U]^3$. (bottom) Each node of the three-dimensional deBerg tree is cut by a set of rectangles stored by the secondary structure of the tree.

- Next, consider a node $\tau$ of the interval trie implicitly stored by the one-dimensional deBerg tree. This node is assigned a range $\rho(\tau) = [x_{\min}, x_{\max})$ defined by the subtree rooted at $\tau$. In three dimensions, this range is defined by the two planes $x = x_{\max}$ and $x = x_{\min}$ that are parallel to the $yz$-plane as illustrated in Figure 14. These planes intersect a set of rectangles in the subdivision. The rectangles whose x-intervals cut these lines are stored by $\tau$.
- Consider the 2D rectangles formed by the intersection of each plane with the rectangles that cut them (see bottom of Figure 14). To locate the rectangle that contains a query point $q$, for each plane we perform a two-dimensional point location on its associated rectangles. The rectangles of each plane, however, are not a subdivision in general, since rectangles that span $\tau$ generate "gaps" between the rectangles along each line (see Figure 14). To remedy this we fill these gaps with "dummy" rectangles. This increases the number of rectangles stored by $\tau$ by a constant factor. We then build two-dimensional deBerg trees on the subdivision on each plane and store each tree at $\tau$.

As in its one and two dimensional analogs, the three-dimensional deBerg tree stores each rectangle in at most $2h$ nodes. Similar to the one-dimensional deBerg tree, a pruning method can be applied to the two-dimensional deBerg tree to achieve $O((\log\log U)^2)$ in linear space. Utilizing this data structure for its secondary structure, the three-dimensional deBerg tree has $O(nh)$ space. Note, the gap filling algorithm increases the number of rectangles stored by the data structure, but only by a constant factor and thus it maintains linear space. To perform a three-dimensional point location query follows directly from the algorithm for two-dimensional stratified trees presented above. The main difference being that this algorithm performs two-dimensional as opposed to one-dimensional point location queries at each node of the deBerg tree resulting in an $O(hb(\log\log U)^2)$ query time.

Comparing the space and time bounds found for the deBerg tree in one, two, and three dimensions one finds that they all satisfy $O(nh)$ space and $O(hb(\log\log U)^{d-1})$ query time, where $d$ is dimension. Table 1 presents a summary of the above space and time bounds for performing orthogonal point location in fixed universes of one, two, and three dimensions. The resulting time bounds achieved under the fixed unverse assumption beat those achieved for real-valued spaces, in which query time is $O(\log^d n)$ as presented by Edelsbrunner. Unfortunately, the deBerg tree breaks down in higher dimensions: in higher dimensions the deBerg tree does not yield linear space. An elegant extension of the stratified tree to higher dimensions is therefore difficult. Nonetheless, the data structure presented by de Berg *et al.* demonstrate the efficiency achievable under a fixed universe assumption.

| Dimension | Query Time | Space |
|---|---|---|
| one | $O(\log\log U)$ | $O(n)$ |
| two | $O((\log\log U)^2)$ | $O(n)$ |
| three | $O((\log U)^{1/h}(\log\log U)^2)$ | $O(n)$ |

Table 1: Summary of space and time bounds acheived by the data structure of de Berg in one, two, and three dimensions on a fixed universe. Note the one and two dimensional trees require a pruning step to achieve log-logarithmic query time with linear space.

# 5 Orthogonal Range Queries

Suppose that we have a database of people, where each person is parameterized by age, weight, and height. One question that we might want to answer is "show me all people who are between 21 and 25 years old, weigh between 130 and 170 pounds, and are less than 6 feet tall." This is known as an orthogonal range query. Besides being an interesting computational geometry problem, it comes up often in database applications.

More formally, suppose we are given a set of $n$ points in $\Re^d$. Each point $\mathbf{p}$ is represented as a vector $(p_1, \ldots, p_d)$. Given an orthogonal range query $\mathbf{q}^- = (q_1^-, \ldots, q_n^-)$ and $\mathbf{q}^+ = (q_1^+, \ldots, q_n^+)$, we wish to find all points $\mathbf{p}$ that satisfy $q_1^- \leq p_1 \leq q_1^+, \ldots, q_n^- \leq p_1 \leq q_n^+$. An example of such a query in two dimensions is shown in figure 15.

In this section, we begin by describing two common solutions to the problem under the algebraic decision tree model. We then proceed by reviewing the data structure

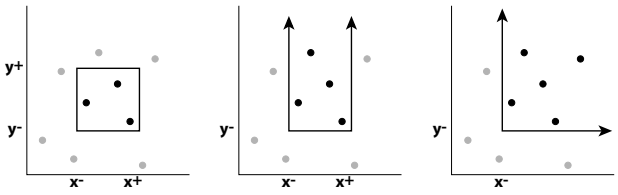



Figure 15: Left, an orthogonal range query on $(x^-, x^+, y^-, y^+)$. Middle, a half-infinite range query on $(x^-, x^+, y^-)$. Right, a dominance query on $(x^-, y^-)$. Black points are part of the query answer, gray points are not.

proposed by Overmars [Ove88b] for orthogonal range queries in fixed universes.

## 5.1 Background

Under the algebraic decision tree model, two common ways to solve for orthogonal range queries are range trees and priority search trees. In this section, we describe these two data structures.

### 5.1.1 Range trees

Range trees [dvOS00] are a well-known data structure, capable of answering orthogonal range queries in $O(\log^d n + k)$ using $O(n \log^{d-1} n)$ storage, where $d$ is the number of dimensions and $k$ is the size of the output. We will only give a brief high-level description; the interested reader should refer to the aforementioned reference.

To begin, we define a one-dimensional range tree as a binary search tree with points stored at leaves. Each node in this tree stores a value. The value of a leaf node is simply the stored point. The value of an internal node is the largest point stored in the left subtree of that node. This binary tree clearly takes $O(n)$ space.

A query can be performed by searching for the two endpoints of the range. Starting from the root, the resulting search paths are the same for a while, and then they fork at some node $b$. The parts of the two paths under $b$ form a "blanket" over a set of subtrees. By traversing all $O(\log n)$ subtrees that hang directly under this blanket, we can report the query result in $O(\log n + k)$ time, where $k$ is the number of points.

To extend range trees to two dimensions (call the first dimension $x$ and the second dimension $y$), we build a one-dimensional range tree on all points, stored by $x$-coordinate. We call this the *primary tree*. For a node $v$ in the primary tree, we define $S(v)$ be the set of points stored in the subtree rooted at $v$. For each node $v$, we associate a *secondary tree* that stores all nodes in $S(v)$, stored by $y$-coordinate.

When we perform a query, we first perform a query on the primary tree with the given $x$-range. As before, this results in a blanket and a set of $O(\log n)$ subtrees that hang directly under this blanket. We then query the secondary trees stored at the roots of these subtrees with the given $y$-range. This results in a query time of $O(\log^2 n + k)$, since we are doing queries on $O(\log n)$ secondary trees. The space consumption turns out to be $O(n \log n)$.

The same technique can be applied inductively to higher dimensions; in general, $d$-dimensional points are inserted into a one-dimensional primary range tree by their first coordinate, and secondary $(d-1)$-dimensional range trees are stored at each node of the primary range tree.

### 5.1.2 Priority search trees

A priority search tree [McC85, dvOS00] is a data structure that answers *half-infinite range queries* in the form "which points lie between some $x^-$ and $x^+$ and above some $y^-$?" as shown in figure 15.

We define a priority search tree as a binary tree in which each node represents a point. It can be constructed recursively as follows. Given a set of points, we select the point with the highest $y$-coordinate as the root. We then select a vertical line with $x$-coordinate $s$ and partition the remaining points into sets $L$ and $R$, depending on which side of the vertical line they fall. We store the value of $s$ at the root, and recursively construct two priority search trees for the sets $L$ and $R$, which become the children of the root. This construction process is shown in figure 16.

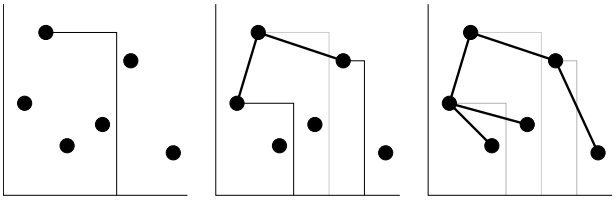


Figure 16: Recursive construction of a priority search tree.

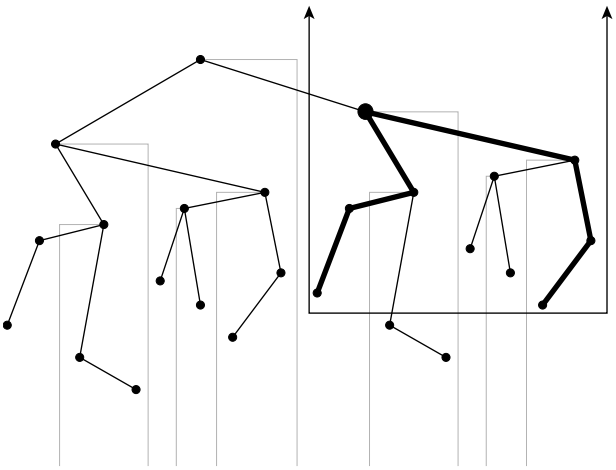


Figure 17: A half-infinite range query forms a blanket (thick line) in the priority search tree. The big node is the fork.

For simplicity, we are assuming that no two points share the same $x$-coordinate. We also note that choosing $s$ to make $|L| \approx |R|$ at each recursive call will result in a balanced priority search tree. In the remainder of this discussion, we assume that $s$ is chosen in such a manner at each recursive step.

To answer a query $(x^-, x^+, y^-)$, we perform simultaneous searches on $x^-$ and $x^+$ and identify the node $b$ where the search paths "fork." As in range trees, the search paths under $b$ form a "blanket" over a set of subtrees, as shown in figure 17. Since we assume that the priority search tree is balances, we assume that the size of the blanket is $O(\log n)$. The points directly on the blanket may or may not lie in the half-infinite range. However, since there are $O(\log n)$ of them, we can just check each one.

We also have to examine each of the $O(\log n)$ subtrees under the blanket. By construction, we know that all nodes in these subtrees necessarily lie between $x^-$ and $x^+$. Thus, we only have to check whether points in these subtrees lie above $y^-$. For each subtree, we can output such nodes in a recursive manner as follows. First, we check the root of the subtree; if it lies below $y^-$, then by construction we know that all nodes under the root also lie below $y^-$, and we terminate the search. If the root lies above $y^-$, we output the root, and recurse on its children. It is easy to see that the time it takes to process each subtree is proportional to how many answers are in the subtree. Thus, the total query time is $O(\log n + k)$, where $k$ is the total number of answers. Since each node in the priority search tree corresponds to a point, the space requirement is clearly $O(n)$.

At this point, we note that a priority search tree can only answer half-infinite range queries for a certain infinite direction (as described, the infinite direction of a query is "up"). The same priority search tree can not be used for "down," "left," or "right" infinite directions, although we could easily modify the construction to build trees for any desired direction.

It remains to be seen how priority search trees can be used for actual orthogonal range queries. To do this, we store all points in a regular balanced binary search tree $T$, indexed by $y$-coordinate. For each internal node $v$ that is a left child of its parent, we store an "up" priority search tree that contains all points in the subtree of $T$ rooted at $v$. Analogously, we store a "down" priority search tree for each right internal node.

Now, given an orthogonal range query $(x^-, x^+, y^-, y^+)$, we search for $y^-$ and $y^+$ in $T$ and find the node $b$ where the search paths fork. All answers to the query must lie in the subtree rooted at $b$. Since the $y$-coordinate of the point stored at $b$ lies between $y^-$ and $y^+$, all points in the left subtree of $b$ have $y$-coordinate less than $y^+$. Thus, we can query the "up" priority search tree stored at the left child of $b$ with $(x^-, x^+, y^-)$ without worrying about reporting any extra points. By parallel reasoning, we can query the "down" priority search tree stored at the right child of $b$ with $(x^-, x^+, y^+)$. Finally, we check the point stored at $b$ and output the point if it lies in the range.

The query time is $O(\log n)$ to find the forking node in $T$, and $O(\log n + k)$ to run two priority search tree queries. As for space, we note that each on each of the $O(\log n)$ levels of $T$, each of the $n$ points is stored in exactly one of the priority search trees (which take linear space). It follows that the total space consumption is $O(n \log n)$.

## 5.2 Dominance queries

We begin our discussion of fixed universe orthogonal range queries by developing a method to answer two-dimensional *dominance queries*. These are queries of the form "given a query $(x^-, y^-)$, report all points whose $x$-coordinate is $\geq x^-$ and $y$-coordinate is $\geq y^-$." An example is shown in figure 15. Again, we follow the description given by Overmars [Ove88b]. With substantial loss of generality, we assume that all points do not share $x$-coordinates. We will remove this assumption later.

All points are stored in an $x$-fast trie $T$ (section 2.2), indexed by $x$-coordinate. The basic idea is to overlay a priority search tree (section 5.1.2) on top of $T$. We will describe how to do this algorithmically. We create an extra field in every node of $T$ to store an *upper point*. Initially, the upper points of all leaves are set to be their corresponding points in $T$, and the upper points of all nonleaf nodes are null. Then, we iterate through the upper points on the leaves in order of decreasing $y$-coordinate. For each upper point, we percolate it up $T$ as far as possible; that is, we move it up the path to the root and stop when we hit a node with an upper point already stored.

The result of this is a priority search tree. To allow for fast searches, we store the following additional information at each leaf $\delta$ of $T$.

- A standard priority search tree $P_\delta$ that contains all $O(\log U)$ upper points on the path from $\delta$ to the root of $T$.
- A linked list $R_\delta$ that contains the $O(\log U)$ subtree roots *with upper points* that are hanging off the right side of the path from $\delta$ to the root. We sort the list by decreasing $y$-coordinate of the upper points.

To solve a dominance query $(x^-, y^-)$, we do the following. We begin by searching for the successor of $x^-$ in $T$ in $O(\log \log U)$ time. Call the successor $\delta$. As with standard priority search trees, the path from $\delta$ to the root forms a blanket (or rather, a half-blanket) which "covers" upper points that are possible answers for the dominance query. All the upper points on the blanket are stored in $P_\delta$, and thus we can just query $P_\delta$ for $(x^-, \infty, y^-)$. Since the size of the $P_\delta$ is $O(\log U)$, this search takes $O(\log \log U + k_P)$ time, where $k_P$ is the number of answers in $P_\delta$.

We also have to look in all the $O(\log U)$ right subtrees stored in $R_\delta$. As described in section 5.1.2, we can process each subtree in time proportional to the number of answers that lie in that subtree. However, we can't check all $O(\log U)$ subtrees if we want to retain our nice $O(\log \log U + k)$ time bound. Thankfully, we have that the subtrees in $R_\delta$ are stored in decreasing order of the $y$-coordinate stored in the upper point at the root. So, we can process the subtrees in that order. If we reach a subtree whose root upper point is below $y^-$, we know that no subtrees following that one in $R_\delta$ will contain any answers. Thus, we can terminate the search. The result is that we can find all answers in right subtrees in $O(k_R)$ time, where $k_R$ is the number of answers in $R_\delta$.

In summary, it takes $O(\log \log U)$ to find the successor, $O(\log \log U + k_P)$ to process $P_\delta$, and $O(\log \log U + k_R)$ to process $R_\delta$. The total query time, then, is $O(\log \log U + k)$, where $k$ is the total number of answers.

As for the amount of space, we know that the $x$-fast trie itself takes $O(n \log U)$. Each of the $n$ leaves stores a $P_\delta$ and $R_\delta$ with $O(\log U)$ space each. Thus, total space consumption remains at $O(n \log U)$.

To fix our original assumption that all points have unique $x$-coordinates, we simply group all points with the same $x$-coordinate together. Within the group, the points are stored in a linked list ordered by decreasing $y$-coordinate. The element with the largest $y$-coordinate is chosen as the "representative" of the group in the dominance query data structure.

If the group representative is a valid answer in the dominance query, we can walk down the list until we reach a point below $y^-$. Since the number of operations that we must do for each group is proportional to the number of answers it stores, we retain $O(\log \log U + k)$ queries.

## 5.3 Half-infinite range queries

In the section 5.2, we described how storing the list $R_\delta$ allows us to efficiently find all answers under the induced blanket of a dominance query. To handle half-infinite range queries (in the "up" direction), it is necessary to store more information at each leaf $\delta$. As before, we store structures $P_\delta$. Instead of storing $R_\delta$, we store an array of $\log U$ lists $R_\delta^1, \ldots, R_\delta^{\log U}$. Like $R_\delta$, these lists store subtree roots with upper points that are hanging off the right side of the path from $\delta$ to the root. The difference is that list $R_\delta^i$ only stores subtree roots that are strictly below level $i$ of the tree (where the root is at level 1). Additionally and analogously, we store lists $L_\delta^1, \ldots, L_\delta^{\log U}$ for left-hanging subtree roots. As in the previous section, all of these lists are sorted in order of decreasing $y$-coordinate.

Now, given a half-infinite range query $(x^-, x^+, y^-)$, we begin by searching for $x^-$ and $x^+$ in the $x$-fast trie $T$. Call the resulting leaf nodes $\alpha$ and $\omega$. As with regular priority search trees, the corresponding search paths form a blanket over subtrees we have to examine. We begin by querying $P_\alpha$ and $P_\omega$ to output all nodes on the blanket that are in the range. Then, we search for the node $b$ that is the "fork" of the two search paths. As the depth of $T$ is $O(\log U)$, we can not afford to walk down the trie to find it. However, since $T$ is an $x$-fast trie, we can simply perform a binary search on the levels of $T$ to find $b$ in $O(\log \log U)$ time.

Suppose $b$ is on level $i$ of the tree. In a regular priority search tree, we would want to examine all right subtrees hanging off the left "branch" of the blanket, and all left subtrees hanging off the right "branch." Conveniently, these are exactly the subtrees stored in $R_\alpha^i$ and $L_\omega^i$. We can process these lists just as we did for dominance queries to achieve an $O(\log \log U + k)$ total query time.

The $x$-fast trie itself takes $O(n \log U)$ space, and each of the $n$ leaves stores a priority search tree with $O(\log U)$ elements and $O(\log U)$ lists with $O(\log U)$ elements each. Therefore, total space consumption is $O(n \log^2 U)$.

## 5.4 Putting it all together

Finally, we are equipped to handle fixed universe orthogonal range queries in two dimensions. To do so, we treat the data structure of section 5.3 as a black box that allows us to perform fixed universe half-infinite range queries in $O(\log \log U + k)$ time. Call this data structure a fixed universe priority search tree (FUPST).

Recall from section 5.1.2 how regular priority search trees were used to perform orthogonal range queries in two dimensions: the points are stored in a balanced binary search tree by $y$-coordinate, and each left (right) child stores an "up" ("down") priority search tree. Similarly, we store an $x$-fast trie on the points, *indexed by the $y$-coordinate*, and associate every left (right) child with an "up" ("down") FUPST.

Now, recall how we performed an orthogonal range query in section 5.1.2: we queried the $y^-$ and $y^+$ and found a forking node $b$ in the binary search tree and performed two priority search tree queries on its children. In the $x$-fast trie, we can also find a forking node $b$ using binary search, as described in section 5.3. This search takes $O(\log \log U)$ time. Then, we perform half-infinite range queries on the FUPSTs stored in the two children of $b$. As previously stated, this takes $O(\log \log U + k)$.

Since there are $O(n \log U)$ nodes in the $x$-fast trie, and each node stores a FUPST that takes $O(n \log^2 U)$ space, the total space consumption is $O(n^2 log^3 U)$. It is possible to achieve $O(n \log U)$ by using techniques borrowed from $y$-fast tries (briefly described in section 2.2. However, in the interest of brevity, we omit these details and refer the interested reader to the paper by Overmars [Ove88b].

## 5.5 Higher dimensions

Unfortunately, priority search trees don't scale to higher dimensions very easily. Range trees, as described in section 5.1.1, do. The basic idea is to use the data structure described in section 5.4 as a base case for the recursive construction of a multidimensional range tree. For example, in three-dimensions, we use a one-dimensional range tree as a primary tree and store fixed universe orthogonal range query structures at each internal node. This results in a query time of $O(\log n \log \log U + k)$, compared to $O(\log^3 n + k)$ for a standard multidimensional range tree. We can easily generalize to higher dimensions.

# 6 Approximate Nearest Neighbors

Given a set of Points $P$, a query point $q$, and a metric $d(x,y)$, the *Nearest Neighbor* (NN) problem is to find a point $p_{nn} \in P$ , s.t. $d(q,p_{nn}) \leq d(q,p)\ \forall p \in P$. e.g. The distance from $p_{nn}$ to $q$ is minimal among all points from the set $P$. The ε-*Approximate Nearest Neighbor* (ε-ANN or ANN) problem is to find a $p_{ann} \in P$ s.t. $d(q,p_{ann}) \leq (1+\varepsilon)\ d(q,p_{nn})$. e.g. The distance from $q$ to $p_{ann}$ is atmost a factor of (1+ε) of the distance to true nearest neighbor.

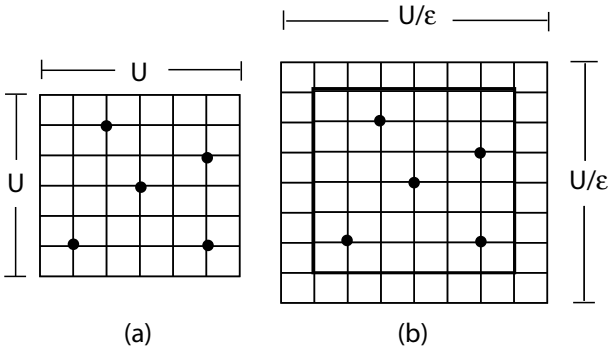



Figure 18: a) The universe $U$ and points $P$. b) The universe after growing by $1/\varepsilon^2$. The dark square denotes the original universe.

## 6.1 Exact Solutions

The standard solution to nearest neighbor queries in 2-dimensions is to first compute the Voronoi diagram of the point set P and then the problem is reduced to planar point location on the faces of the Voronoi diagram. There are a variety of planar point location algorithms [dvOS00, Sei91, ST86] that run in $O(\log N)$ time and use $(N)$ space (here N is the size complexity of the voronoi arrangement). These combined algorithms works well in the 2-dimensional case, however they are not as practical as the dimension increases. For example, it has been shown[dvOS00] that the complexity of the Voronoi diagram of n points in higher dimensions is $\Theta(n^{\lceil d/2 \rceil})$. Also many exact geometric algorithms have an exponential dependence on dimension (for a fixed dimension this exponential dependence disapears in big-$O$ notation) which is known as the "curse of dimensionality". Even if you assume points $P$ lie on a grid, Voronoi based algorithms can not take advantage of any of the fixed universe point location algorithms described above (section 4), because although the points are taken from a fixed universe, the edges of the Voronoi arrangment may not be.

## 6.2 ANN in 2 dimensions

Amir *et al.* [AEIS99] introduced a data structure that answers ε-*Approximate Nearest Neighbor* queries in $O(d + \log\log U)$ time based on a multidimensional extension to the Stratified Tree [van77]. We describe a 2-dimensional variant of this algorithm due to Cary [Car01] that acheieves the same $O(\log\log U)$ query bounds and uses space polynomial in $n$ and ε. We assume the reader is familar with the 1-dimensional stratified tree introduced in section 2.3.

Initially we are given a set of points P with coordinates that lie on the integer grid $[U]^2$. As in the 1-dimensional stratified tree, we will define a recursive data structure that divides our universe into "square root" pieces, each of size "square root" of the size our universe.

This is accomplished as follows. Let $[U']^2$ be the current universe (for some recursive level of the construction), with $U' \leq U$. If $U' \leq 1/\varepsilon^2$ we store the nearest neighbor of each point $p = (i,j) \in [U']^2$ in a table of size $[U']^2 = O(1/\varepsilon^4)$. Thus given a query point $q \in [U']^2$ the nearest neighbor can be found in $O(1)$ time by indexing into the table.

Now assume $U' > 1/\varepsilon^2$. We first grow our universe by a factor $1/\varepsilon^2$, e.g. our $[U']^2$ grid now becomes $[U'/\varepsilon]^2$ (see Figure 18). Next divide $[U'/\varepsilon]^2$ into $U'/\varepsilon^2$ regular squares, $S_{ij}$, each of size $\sqrt{U'} \times \sqrt{U'}$ (see Figure 19). For each square $S_{ij}$, let $p_{ij}$ be the point at the center of $S_{ij}$. Let $B_{ij} = \{\ p \mid p \in P' \wedge d(p,p_{ij}) \leq \sqrt{U'}/\varepsilon\}$ be the set of points in $P'$ within distance $\sqrt{U'}/\varepsilon$ from the center of square $S_{ij}$. If $q$ lies near the boundary of a square, its nearest neighbor may lie in a bordering square (see Figure 19). Intuitively, expanding the neighborhood around $S_{ij}$

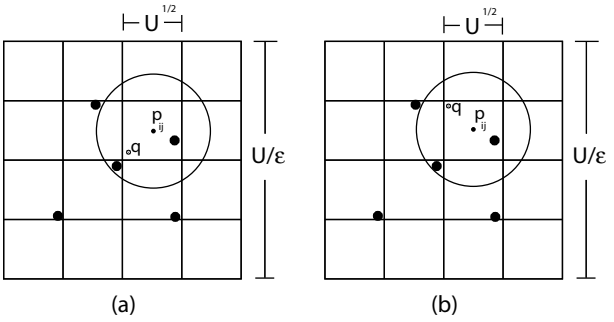



Figure 19: The subdivision of the universe in to $\sqrt{U} \times \sqrt{U}$ squares and the ball of radius $\sqrt{U}/\varepsilon$ around point $p_{ij}$ at the center of square $S_{ij}$. Two possible positions of the query point $q$ are shown. In (a) the exact nearest neighbor lies within ball $B_{ij}$. In figure (b) the exact nearest neighbor lies outside $B_{ij}$. In both cases the nearest neighbor falls in an adjacent square to the square $q$ lies in. In both cases The approximate nearest neighbor returned will be chosen among the points that lie in $B_{ij}$.

allows us to consider points that fall in adjacent squares. As $\varepsilon$ shrinks, we increase the overlapping area and thus improve the approximation. For each non-empty $B_{ij} \neq \emptyset$, store the indicies $(i,j)$ in a hashtable that maps $(i,j)$ to $S_{ij}$. Given a query point $q$ that lies in square $S_{ij}$, we can quickly determine if ball $B_{ij}$ contains any points from $P'$ using this hashtable. Storing the non-empty squares $S_{ij}$ in a hashtable also reduces the amount of space needed because empty squares need not be stored. This reduces the space from $O(U)$ to $O(n^2)$ (this space bound is not tight, and can be improved). In each $S_{ij}$ with non-empty $B_{ij}$, store the recursive ANN structure for the subproblem ANN($[\sqrt{U'}/\varepsilon]^2$ , $B_{ij}$) in the universe of size $[\sqrt{U'}/\varepsilon]^2$ with points $B_{ij}$. Let $I = \{\ (i,j) | S_{ij} \cap P' \neq \emptyset\ \}$ be the set indicies of non-empty squares $S_{ij}$ (Note: set $I$ may be different than the set of indicies stored in hashtable). Lastly, we create the recursive structure $H$ = ANN($[\sqrt{U'}]^2$,$I$) of non-empty indicies, $I$, in the universe of size $[\sqrt{U'}]^2$. $H$ is used during the search procedure. If $q$ lies in an empty square (e.g $B_{ij} = \emptyset$), then we use $H$ to find the nearest non-empty square $S_{hk}$ to square $S_{ij}$. This concludes the construction of ANN($[U']^2$,$P'$).

**Search**(P,q)
  **if** $U' \leq 1/\varepsilon^2$ **then**
    return approximate nearest neighbor with table lookup;
  **else**
    Let $S_{ij}$ be the square containing q
    **if** $B_{ij} \neq \emptyset$ **then**
      return Search($B_{ij}$,q);
    **else**
      $S_{hk}$ = Search($H$,$(i,j)$);
      return any $p \in S_{hk}$
    **end if**
  **end if**

Figure 20: The approximate nearest neighbor search procedure

### 6.2.1 ANN queries

The search procedes as follows. If the size of the current universe is less than $1/\varepsilon^2$, we can directly find the nearest neighbor through a table lookup in constant time (remember, we store exact solutions for universes smaller than a preset size). Otherwise we find the square $S_{ij}$ which $q$ lies in (this can be implemented using simple bit tricks, which are allowable in the RAM model), ie $q \in S_{ij}$. If $B_{ij}$ is nonempty (can check this in $O(1)$ time with hashtable) we recurse with $P' = B_{ij}$ and $q$ in universe of size $[\sqrt{U'}/\varepsilon]^2$. If $B_{ij}$ is empty, we search for the closest non-empty square $S_{hk}$ to square $S_{ij}$. To find $S_{hk}$ we recurse with $P' = H$ and $p = (i,j)$ ($(i,j)$ are the indicies of block $S_{ij}$). Finally, we return any point within square $S_{hk}$ as the Approximate Nearest Neighbor..

The reader should notice the similarity to the Successor procedure (Figure 3) described in section 2.3.1. The Nearest Neighbor problem (as well as ANN) can be considered as a generalized Successor problem. The proof that Search runs in time $O(\log\log U)$ is similar to lemma 2.1 and thus we omit it.

We give a sketch of the correctness for Search based on [Car01]. Notice that there are two ways we introduce error into the calculations: If there is a point outside $B_{ij}$ that is closer to $q$ than all points in $B_{ij}$ (see Figure 19.b ) or $q$ lies in an empty square and the point chosen in

$S_{hk}$ is not the closest point to $q$. However, because we are on a $\sqrt{U}$ grid, the error introduced is $O(\sqrt{U})$ (since the nearest point will be atmost one grid square from the point chosen). Finally we observe that the distance from $q$ to its nearest neighbor must be $\Omega(\sqrt{U}/\varepsilon)$ since it must lie outside ball $B_{ij}$. Thus the point chosen is a $(1 + O(\varepsilon))$-Approximate Nearest Neighbor.

As mentioned above the space requirements are polynomial in $n$ and $1/\varepsilon$. Because at each recursive level, the ranges overlap, a point may end up in at most $O(1/\varepsilon^2)$ leaves. At each non-empty leaf we store an array of size $O(1/\varepsilon^4)$. Thus for all of the n points we have $O(n^2/\varepsilon^6)$ space used.

### 6.3 Higher dimensions

As discussed earlier, there are efficient exact solutions to the NN problem in low dimensions (d ¡= 2, d = 3 is arguable). The major benefit of the algorithm described is that It can be extended to higher dimensions. Cary [Car01] extends the ANN structure to higher dimensions and shows a $O(d \log\log U / \log\log\log U)$ query bound. Original results from Amir *et al.* [AEIS99] gave a $O(d + \log\log U + \log 1/\varepsilon)$ result. Both of these approximate algorithms avoid the exponential dependence on dimension. We also mention a technique due to Chan [Cha02] that can anwer ANN queries in $O(\min\{\log\log U, \sqrt{\log n}\})$ without the need for any complex data structures. The algorithm detailed by Chan is based on sorting the points by permutations of the shuffle order [BET99, Cha00] of the points and comparing points that are adjacent in the shuffle ordering. These algorithms trade accuracy for speed in higher dimensions to attempt to break the "curse of dimensionality".

## 7 Conclusion

We have surveyed many computational geometry problems and shown that they can be solved quickly under the fixed universe assumption. By breaking the decision tree "sorting barrier" we were able to give asymptotically faster (independent of the universe size $U$) algorithms for convex hull and other problems based on convex hull. We also presented a class of geometric algorithms and data structures that were dependent on the size of the universe the input is taken from. It is relevent to consider for what size universes are these algorithms preferable to classic algorithms for the same problem. Obviously, if $U = O(n)$, then the fixed universe algorithms will perform better. However, if $U \gg O(n)$, then the fixed universe is possibly a bad choice. We notice, that we are often trading a $O(\log n)$ factor for a $O(\log\log U)$ factor, thus if $n = \omega(\log U)$ then any fixed universe algorithm is asymptotically faster. We also studied an approximate nearest neighbor algorithm that traded some accuracy in the solution to avoid an exponential dependency on the dimension. In each case, we have made some realistic relaxations to our computational model, and arrived at solutions with theoretical advantages.

## References


[ABR00] Stephen Alstrup, Gerth Stolting Brodal, and Theis Rauhe. New data structures for orthogonal range searching. In *IEEE Symposium on Foundations of Computer Science*, pages 198–207, 2000.

[AEIS99] Arnon Amir, Alon Efrat, Piotr Indyk, and Hanan Samet. Efficient regular data structures and algorithms for location and proximity problems. In *IEEE Symposium on Foundations of Computer Science*, pages 160–170, 1999.

[AHNR95] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? In *STOC: ACM Symposium on Theory of Computing (STOC)*, 1995.

[BET99] Marshall W. Bern, David Eppstein, and Shang-Hua Teng. Parallel construction of quadtrees and quality triangulations. *International Journal of Computational Geometry and Applications*, 9(6):517–532, 1999.

[Car01] Matthew Cary. Toward optimal e-approximate nearest neighbor algorithms. *J. Algorithms*, 41(2):417–428, 2001.

[Cha00] Timothy M. Chan. Approximating the diameter, width, smallest enclosing cylinder, and minimum-width annulus. In *Proceedings of the sixteenth annual symposium on Computational geometry*, pages 300–309. ACM Press, 2000.

[Cha02] Timothy M. Chan. Closest-point problems simplified on the ram. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 472–473. Society for Industrial and Applied Mathematics, 2002.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2001.

[DKM+88] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *IEEE Symposium on Foundations of Computer Science*, pages 524–531, 1988.

[dvOS00] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2000.

[dvS92] Mark de Berg, Marc J. van Kreveld, and Jack Snoeyink. Two- and three-dimensional point location in rectangular subdivisions (extended abstract). In *Scandinavian Workshop on Algorithm Theory*, pages 352–363, 1992.

[EHH86] Herbert Edelsbrunner, G. Harring, and David Hilbert. Rectangular point location in $d$ dimensions with applications. *Computer Journal*, 29:76–82, 1986.

[FKS82] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. In *23rd Annual Symposium on Foundations of Computer Science*, pages 165–169, Chicago, Illinois, 3–5 November 1982. IEEE.

[FMNT87] O. Fries, K. Melhorn, S. Naher, and A. Tsakalidis. A $\log\log n$ data structure for three-sided range queries. *Information Processing Letters*, 25(4):269–273, 1987.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[HT02] Yijie Han and Mikkel Thorup. Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space. In *IEEE Symposium on Foundations of Computer Science*, pages 135–144, 2002.

[KO88] Rolf G. Karlsson and Mark H. Overmars. Scanline algorithms on a grid. *BIT*, 28(2):227–241, 1988.

[Mat92] Jiri Matousek. Reporting points in halfspaces. *Computational Geometry Theory and Applications*, 2:169–186, 1992.

[McC85] Edward M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.

[Ove88a] Mark H. Overmars. Computational geometry on a grid: An overview. In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, volume F40, pages 167–184. Springer-Verlag, 1988.

[Ove88b] Mark H. Overmars. Efficient data structures for range searching on a grid. *Journal of Algorithms*, 9(2):254–275, 1988.

[Sei91] Raimund Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry Theory and Application*, 1(1):51–64, 1991.

[ST86] Neil Sarnak and Robert Endre Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29:669–679, 1986.

[Tou83] G. Toussaint. Solving geometric problems with the rotating calipers, 1983.

[van77] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[Wil83] Dan E. Willard. Log-logarithmic worst case range queries are possible in space $\Theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983.

[Wil84] Dan E. Willard. New trie data structures which support very fast search operations. *Journal of Computer and System Sciences*, 28:379–394, 1984.

[WNDS99] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide*. Silicon Graphics, 1999.