Geometric data structures for computer graphics: an overview

Mark H. Overmars

RUU-CS-87-5
February 1987

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

axis-parallel rectangles (an important step when working with bounding boxes). Another application solves the windowing problem in a set of line segments.

Many geometric data structures are static, i.e., they do not allow for efficient insertions and deletions of objects, or the update algorithms are very complicated. In Section 5 some general methods are presented for turning static data structures into dynamic structures that do allow for insertions and/or deletions. These general methods work for whole classes of data structures and problems.

Finally, in Section 6 we give some concluding remarks and directions for further reading and research.

2 Quad-trees.

Quad-trees were introduced by Finkel and Bentley[12] in 1974 although they were in fact already known much longer in computer graphics. For example, Warnock's hidden surface removal algorithm ([29,30], see also e.g. [23] for a description) is based on quad-trees but he did not name them this way. Moreover there was some difference. In computer graphics quad-trees are normally used as space dividing data structures, i.e., they work in image space. On the other hand, quad-trees as introduced in [12] split the set of objects and work in object space. Many results on and using quad-trees have been obtained. See Samet[25] for a bibliography of over 250 papers on quad-trees and related hierarchical data structures.

In subsection 2.1 we will describe both image space and object space quad-trees and study some basic algorithms that use them. In subsection 2.2 we study some modifications as introduced by Overmars and van Leeuwen[21]. In subsection 2.3, we describe some related results on e.g. k-d trees.

2.1 Simple quad-trees.

Let V be a set of points in the plane and let the points have integer coordinates between 0 and $U-1$. Let us assume that $U = 2^i$ for some i. An *image space quad-tree* T is a 4-ary tree that is defined as follows: When V is empty, T is an empty leaf. When V contains one point T is a leaf containing this one point. Otherwise, T consists of a root w that has four sons w_1, w_2, w_3 and w_4 that each correspond to one quadrant of the $U * U$ grid (see figure 1). Each son w_j is a quad-tree of the points in V lying in the particular quadrant. See figure 2 for an example of a quad-tree.

Theorem 2.1 *An image space quad-tree of n points on a $U*U$ grid needs $\theta(n \log U)$ storage.*

Proof. The depth of the quad-tree is clearly bounded by $\lceil \log U \rceil$. Consider each internal node together with its empty sons (if any). Each such node is on at least one search path to a leaf containing a point in the set. For each point there

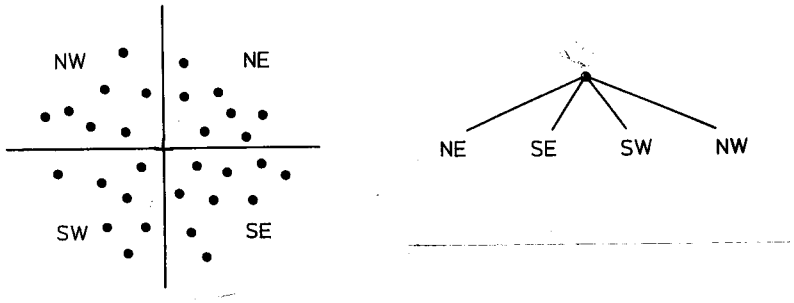

Figure 1: The four quadrants.

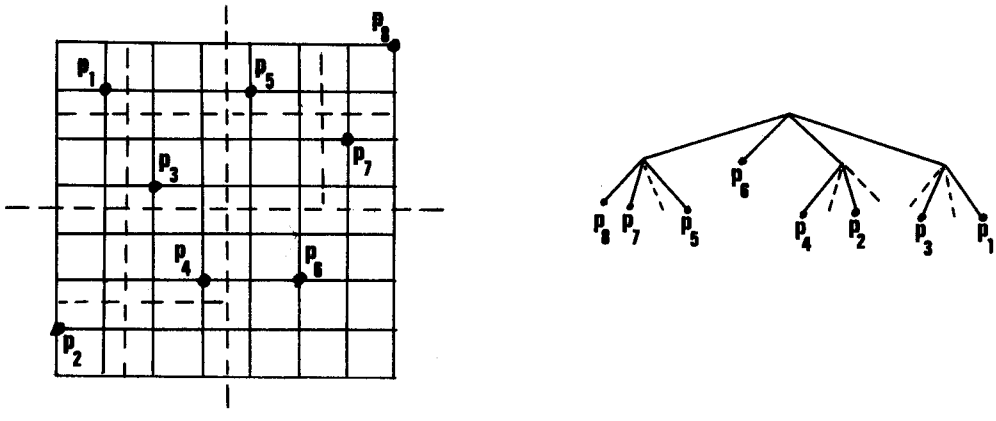

Figure 2: An image space quad-tree.

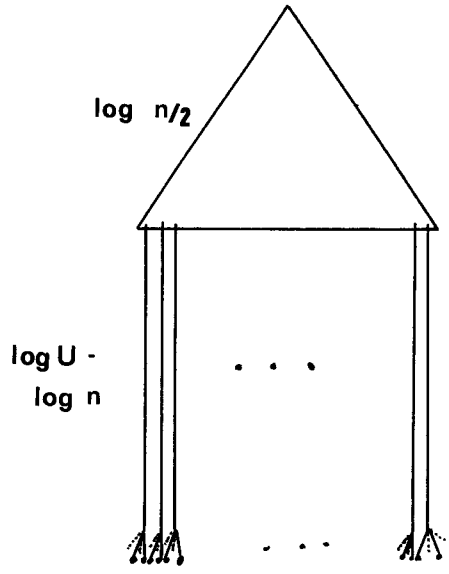

Figure 3: A bad example.

are $O(\log U)$ nodes on the search path. Hence, the amount of storage is bounded by $O(n \log U)$.

Unfortunately, this is also a lowerbound. We can easily construct a quad-tree that looks as follows: The top of the quad-tree consists of a complete 4-ary tree with $n/2$ sons. Each leaf corresponds to an area that contains exactly 2 points, and these points will only be separated on the lowest level of the quad-tree. See figure 3 for the situation we get. It is easy to see that the tree uses storage $\Omega(n + (\log U - \log n)n)$ and when U is large compared to n this is $\Omega(n \log U)$. □

On the other hand, when U is of the same order as n (or smaller than n) the quad-tree only uses $O(n)$ storage.

Quad-trees can be used for many searching problems. As an example let us consider the simple exact match query that asks whether a given point p is in the quad-tree. The searching algorithm works as follows: Let w be the current node in the tree we look at (in the beginning the root). If w is an empty leaf, p is not present. If w is a non-empty leaf containing p we found p. If it does not contain p then p is not present. If w is not a leaf, determine the quadrant p lies in and continue the search in the appropriate son.

Theorem 2.2 *Exact match queries on an image space quad-tree can be performed in time $O(\log U)$.*

Proof. Follows from the above discussion and the fact that the depth of an image space quad-tree is $O(\log U)$. □

Also, for example, range queries, that ask for all points that lie in a given axis-parallel rectangle R (e.g. a window), can be performed on quad-trees. To

this end we again search down the tree but continue the search at all sons of an internal node where there might possibly be answers. Let w be the current node we visit (in the beginning the root). If w is an empty leaf we don't do anything. If it is a leaf that contains a point p we report p if it lies in R. If w is an internal node we look at the way R intersects the four quadrants at w. For each quadrant there are three possibilities: (i) R completely covers the quadrant. In this case we report all points in the quadrant (i.e., in the corresponding subtree).(ii) R does not intersect the quadrant. In this case we do nothing in the quadrant. (iii) R partially intersects the quadrant. In this case we continue the search in the corresponding son of w. As R might partially intersect all quadrants we sometimes continue the search at all four sons. On the other hand, especially when R is small, we often have to search in only one son.

It can easily be shown that the search time is always bounded by $O(k + 2^d)$ where d is the depth of the quad-tree and k the number of reported answers. (This follows from the fact that case (iii) only occurs when the quadrant is intersected by one of the boundaries of R and any horizontal or vertical line intersects at most $O(2^d)$ quadrants in the tree.) Unfortunately, when the quad-tree is not well balanced, d might be $\log U$ and in that case the query time is $\Omega(k + U)$.

Image space quad-trees are dynamic. Insertions and deletions of points can easily be performed in time $O(\log U)$. Details are left to the reader.

Image space quad-trees can also be used for storing geometric objects other than points, like e.g. lines, polygons, etc. See e.g. Samet, Shaffer and Webber [26] for examples of quad-trees storing line segments.

Object space quad-trees where defined by Finkel and Bentley[12]. Let V be a set of n points in some arbitrary real two-dimensional space. A quad-tree is constructed as follow: Let $p = (p_1, p_2)$ be some point in the set. We store p in the root of the quad-tree. This splits the space in four quadrants of points with $x \geq p_1$ and $y \geq p_2$, $x \geq p_1$ and $y < p_2$, $x < p_1$ and $y < p_2$, and $x < p_1$ and $y \geq p_2$. These four quadrants become the sons of p and, if not empty, are structured as quad-trees of the points in them. See figure 4 for an example of an objects space quad-tree.

Theorem 2.3 *For each set of n points an object space quad-tree of depth $\lceil \log n \rceil$, using $O(n)$ storage, can be constructed in time $O(n \log n)$.*

Proof. As each node contains a point of the set, each quad-tree uses $O(n)$ storage. To obtain a quad-tree of depth $\lceil \log n \rceil$ we choose as splitting point p the median according to the x-coordinate. When more points have the same x-coordinate as the median, we can always choose one of them that splits the set such that each quadrant contains at most half of the points. The median can be found in time $O(n)$. Repeating this method for each quadrant constructs the

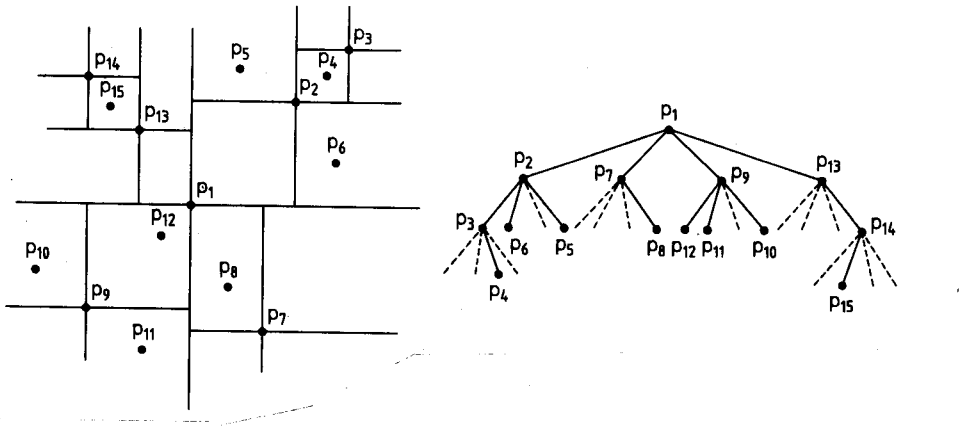

Figure 4: An object space quad-tree.

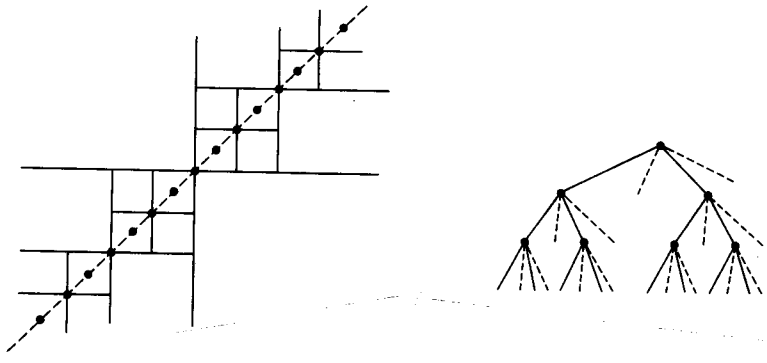

Figure 5: A bad object space quad-tree.

required tree in $O(n \log n)$ time. □

As a quad-tree is a 4-ary tree you might hope to be able to reduce the depth to $\lceil \log_4 n \rceil$. Unfortunately, this is impossible. When, for example, all points lie on a diagonal line, for each splitting point at least two of the quadrants will be empty. Hence, one of the quadrants contains at least $n/2$ points which leads to a depth of at least $\lceil \log n \rceil$ (see figure 5).

Object space quad-trees can be used to solve similar problems as image space quad-trees. The time bounds are no longer depending on U but on n only. See e.g. Bentley and Stanat[7] for some time analysis of queries on object space quad trees.

One of the disadvantages of object space quad-trees is that it is hard to perform insertions and deletions and keep the tree balanced when doing so. Especially

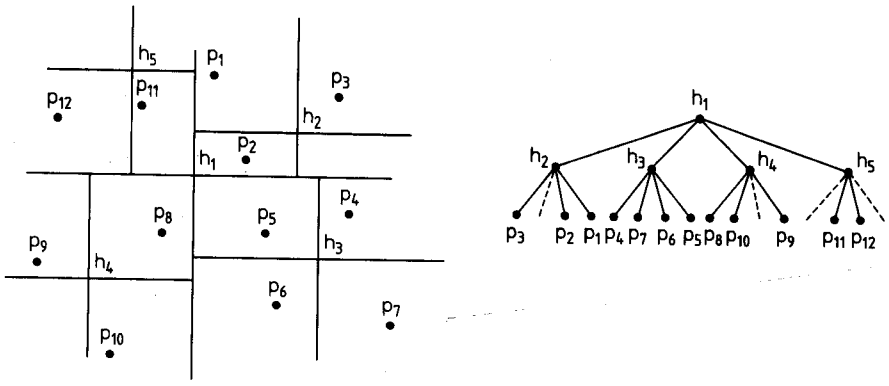

Figure 6: A pseudo quad-tree.

deletions cause problems. For example, when we delete the point p stored at the root, a new root has to be found and this new root splits the set in different subsets as the old root did. Hence, it might require rebuilding large parts of the tree. (See e.g. Kersten and van Emde Boas[13] and Samet[24] for some partial solutions to this problem.)

2.2 Pseudo quad-trees.

Overmars and van Leeuwen[21] introduced a so-called *pseudo quad-tree*, also called a *leaf search quad-tree*, to improve the depth and the update time. A pseudo quad-tree has exactly the same form as a normal quad-tree, except that points from the set are stored at the leaves only and internal nodes store arbitrary points (not in the set) as splitting points. See figure 6 for an example of a pseudo quad-tree.

The depth of pseudo quad-trees can be bounded by $\lceil \log_3 n \rceil$, assuming that no two points have the same x- or y-coordinate. This is based on the following lemma:

Lemma 2.4 *Let V be a set of n points. There exists a point h (not in V) such that each quadrant induced by h contains at most $\lceil n/3 \rceil$ points of V.*

Proof. Choose an x-coordinate h_1 such that $\lceil n/3 \rceil$ of the points in V have x-coordinate $< h_1$ and the other $\lfloor 2n/3 \rfloor$ points have x-coordinate $> h_1$. (This is possible because of the assumption made.) Let V' be the set of points with x-coordinate $> h_1$. Choose h_2 such that half of the points in V' has y-coordinate $< h_2$ and the other half has y-coordinate $> h_2$. Now $h = (h_1, h_2)$ has the required property (see figure 7). □

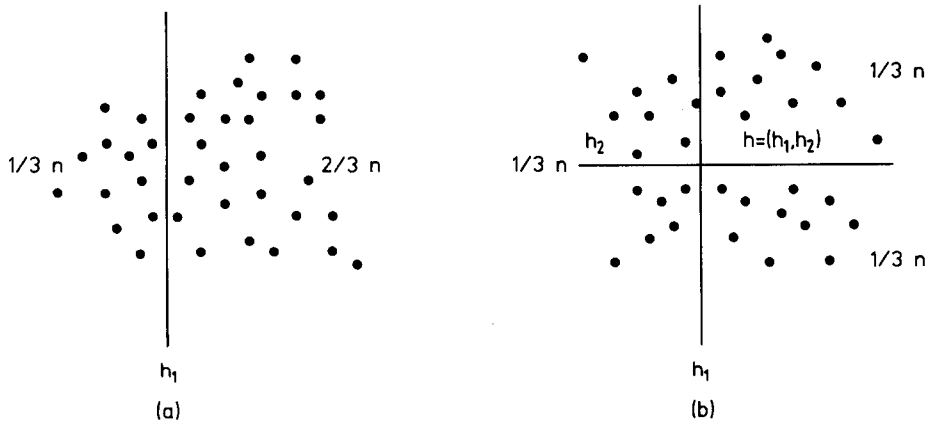

Figure 7: The spliting point.

Continueing choosing splitting points in the above way results in a tree of depth $\lceil \log_3 n \rceil$. Again, choosing all points on a diagonal line, this bound can be proven to be optimal.

To handle the case in which more points have the same x- or y-coordinate efficiently, in [21] so-called extended pseudo quad-trees are define. We won't describe them here.

Again a large number of queries, like e.g. range queries, can be performed on pseudo quad-trees in essentially the same way as for ordinary quad-trees. As the depth of pseudo quad-trees is normally smaller than for ordinary quad-trees the efficiency of the algorithms is improved. For example, for range searching we showed that the complexity is bounded by $O(k + 2^d)$ where d is the depth of the tree. For ordinary quad-trees this depth might be $\lceil \log n \rceil$ resulting in a query time of $O(k + n)$ in the worst case. For pseudo quad-trees we get a worst-case query time of $O(k + 2^{\log_3 n}) = O(k + n^{1/\log 3})$.

Pseudo quad-trees are dynamic. Because all points are stored in the leaves, it is easy to add or remove points. The only problem lies in keeping the tree balanced. In [21] the following theorem is proven:

Theorem 2.5 *For any ϵ, $0 < \epsilon < 2$, there exists an algorithm for inserting and deleting points in a pseudo quad-tree in average time $O(\frac{1}{\epsilon} \log^2 n)$, such that the depth of the tree remains bounded by $\log_{3-\epsilon} n + O(1)$.*

2.3 Related results.

Many structures related to quad-trees have been designed. See [25] for many references to such papers. We will just shortly describe two of them here: multi-dimensional quad-trees and k-d trees.

Quad-trees can easily be generalised to d-dimensional quad-trees for arbitrary d. (For $d = 3$ such structures are often called oct-trees.) Again d-dimensional

quad-trees can be image space or object space. We will only describe the d-dimensional variant of the pseudo quad-tree here.

Let V be a set of n points in a d-dimensional space. A *d-dimensional pseudo quad tree* is defined as follows: It is a tree in which each node has 2^d sons. Points in V are stored in the leaves. Each internal node stores some arbitrary point (not in V). This point splits the d-dimensional space in 2^d regions by drawing hyper-planes parallel to the coordinate-planes through the point. These 2^d regions correspond to the sons of the node.

In [21] some results on d-dimensional quad-trees have been proven. We recall the most important ones.

Lemma 2.6 *Let V be a set of n points in d-dimensional space. There exists a point h (not in V) such that each hyper-quadrant induced by h contains at most $\lceil n/(d+1) \rceil$ points of V.*

Using this lemma we can construct d-dimensional pseudo quad-trees of depth at most $\lceil \log_{d+1} n \rceil$. This result is optimal in the worst case.

Theorem 2.7 *For any ϵ, $0 < \epsilon < d$, there exists an algorithm for inserting and deleting points in a d-dimensional pseudo quad-tree in average time $O(\frac{1}{\epsilon} \log^d n)$, such that the depth of the tree remains bounded by $\log_{d+1-\epsilon} n + O(1)$.*

d-dimensional quad-trees can be used for solving many d-dimensional problems like e.g. d-dimensional range searching. Unfortunately, the query times tend to become worse when the dimension grows.

Bentley[1] (see also [2]) introduced so-called *k-d trees* as an alternative for object space quad trees. A 2-dimensional k-d tree is a binary tree that is constructed as follows: Let V be a set of points in the plane. Choose a point $p \in V$. $p = (p_1, p_2)$ will be stored in the root of the tree. Now split the set V in two subsets V_1 of points with x-coordinate $\leq p_1$ and V_2 of points with x-coordinate $> p_1$. V_1 will be stored in the left subtree of p and V_2 in the right subtree. Now in each subset again take a point. This will become the root of the subtree. Again split the set in two halves but this time with respect to the y-coordinate. On the next level of the tree we again split with respect to x-coordinate, etc. See figure 8 for an example of a k-d tree.

The main advantage of k-d trees over quad-trees is that we always can split the set in an optimal way. This means that we can always construct k-d trees of depth $\lceil \log n \rceil$.

Queries can be performed on k-d trees in a similar way as in quad-trees. Details are left to the reader (or consult e.g. [1,2,16,28]). As we can construct optimal k-d trees the query time is normally better than in quad-trees.

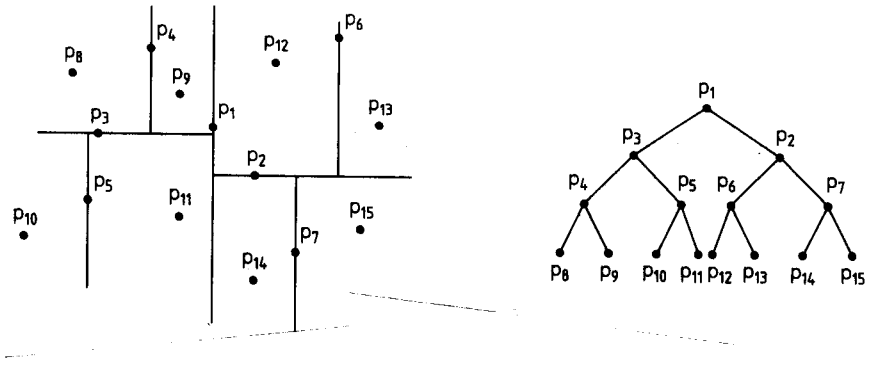

Figure 8: A k-d tree.

To make k-d trees dynamic, in [21] pseudo k-d trees were introduced. They are k-d trees in which the points of V are stored in the leaves only. Internal nodes store arbitrary points. They show the following result:

Theorem 2.8 *For any ϵ, $0 < \epsilon < 1$, there exists an algorithm for inserting and deleting points in a pseudo k-d tree in average time $O(\frac{1}{\epsilon}\log^2 n)$, such that the depth of the tree remains bounded by $\log_{2-\epsilon} n + O(1)$.*

Also d-dimensional k-d trees and pseudo k-d trees can be constructed.

3 Range trees.

In this section we study the range tree, a structure designed recently independently by a number of researchers (see e.g. [3,14,31,33]) for solving the range searching problem in two- and higher dimensional space.

Let us first consider the one-dimensional problem. So we are given a set of keys and we want to store them such that for a given range $[A_1 : B_1]$ we can efficiently determine the points lying in between A_1 and B_1. To this end we store the keys in the leaves of a balanced binary search tree, in sorted order. Moreover we link the leaves in this order in a double linked list. The structure clearly uses $O(n)$ storage. We call this structure a one-dimensional range tree.

To perform a query with range $[A_1 : B_1]$ we search with A_1 in the tree to locate the smallest key $\geq A_1$. Let this key be K. If K does not exist or $K > B_1$ we are done. Otherwise, we report K and look at the right neighbor of K in the tree and repeat the test here. It is clear that such a query takes time $O(\log n)$ to search with A_1 plus the number of answers found.

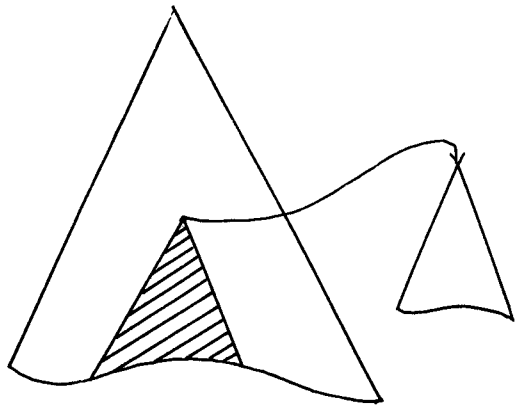

Figure 9: Associated trees.

Theorem 3.1 *Given a set of n keys, the one-dimensional range searching problem can be solved in a query time of $O(k + \log n)$ using $O(n)$ storage, where k is the number of answers found.*

In the next subsection we will use the one-dimensional range tree to obtain a two-dimensional range tree. In subsection 3.2 we generalise this result to multi-dimensional space. Finally in subsection 3.3 we consider some other structures for range searching.

3.1 Two-dimensional range trees.

Let V be a set of points in the plane. A two-dimensional range tree of V has the following form. We sort all point with respect to their x-coordinate and store them in the leaves of a balanced binary search tree in this order. To each internal node δ we associate a one-dimensional range tree T_δ of the points in the subtree rooted at δ, on their y-coordinate (see figure 9). Hence, in the main tree points are sorted with respect to their x-coordinate and in the associated structures point are sorted with respect to their y-coordinate.

To perform a query with range $([A_1 : B_1], [A_2 : B_2])$ we search with both A_1 and B_1 in the main tree. For some time A_1 and B_1 will follow the same search path but at some internal node δ A_1 will go to the left son and B_1 will go to the right son. Let $\beta_1 \ldots \beta_i$ be the nodes that are rightson of a node on the search path of A_1 below δ and do not lie on the search path themselves. Silmilar, let $\gamma_1 \ldots \gamma_j$ be the nodes that are leftson of a node on the search path of B_1 below δ that do not lie on the search path themselves. See figure 10 for the situation. All points with x-coordinate between A_1 and B_1 lie in exactly one of the subtrees rooted at the β and γ nodes. Of these points we have to find the ones whose y-coordinate

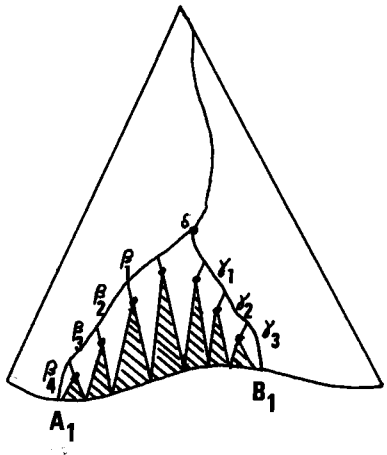

Figure 10: The β and γ nodes.

lies between A_2 and B_2. For this we use the structures $T_{\beta_1} \ldots T_{\beta_i}$ and $T_{\gamma_1} \ldots T_{\gamma_j}$. All points in the x-range are stored in exactly one such structure and all points in these structures lie in the x-range. On each of these T-structures we perform a one-dimensional range query with range $[A_2 : B_2]$.

Theorem 3.2 *Given a set of n points in the plane, the two-dimensional range searching problem can be solved in a query time of $O(k + \log^2 n)$ using $O(n \log n)$ storage, where k is the number of answers found.*

Proof. Let us first consider the amount of storage required. The main tree uses only $O(n)$ storage. On each level of the main tree each point in V occurs exacly once in an associated structures. As an associated structure of n' points uses storage $O(n')$, the associated structures on one level together use $O(n)$ storage. As the tree has $O(\log n)$ level the storage bound follows.

To perform a query we first seach in the main tree with A_1 and B_1 which takes time $O(\log n)$. In this way we locate $O(\log n)$ T_β and T_γ structures that each have a query time of at most $O(\log n)$ plus the number of answers found. Hence, the total query time is bounded by $O(\log^2 n)$ plus the total number of answers found. □

The two-dimensional range tree can be made dynamic but the update algorithms are quite complicated (see e.g [31,33]). See also Section 5.

3.2 Multi-dimensional range trees.

The two-dimensional range tree can easily be generalised to a d-dimensional range tree for any constant $d > 2$. Let V be a set of points in d-dimensional space. We

again construct a balanced binary search tree that stores all points from V in its leaves, ordered with respect to their first coordinate. With each internal node δ we now store a $d-1$-dimensional range tree T_δ of all point in the subtree rooted at δ, restricted to their last $d-1$ coordinates.

To perform a query with a range $([A_1 : B_1], [A_2 : B_2], \ldots, [A_d : B_d])$ we search with both A_1 and B_1 in the main tree, locating nodes $\beta_1 \ldots \beta_i$ and $\gamma_1 \ldots \gamma_j$ in the same way as in the two-dimensional case. For each of the structures $T_{\beta_1} \ldots T_{\beta_i}$ and $T_{\gamma_1} \ldots T_{\gamma_j}$ we perform a $d-1$-dimensional range query with range $([A_2 : B_2], \ldots, [A_d : B_d])$. It is easy to see that this yields the right answers.

Theorem 3.3 *Given a set of n points in d-dimensional space, the d-dimensional range searching problem can be solved in a query time of $O(k + \log^d n)$ using $O(n \log^{d-1} n)$ storage, where k is the number of answers found.*

Proof. Let $M(d, n)$ denote the amount of memory required for a d-dimensional range tree of n points. The T-structures associated to the nodes on one level of the main tree together require at most $M(d-1, n)$ storage. Hence, we know

$$M(d, n) \leq \log n M(d-1, n)$$
$$M(2, n) = O(n \log n).$$

This results in $M(d, n) = O(n \log^{d-1} n)$.

The bound on the query time can be proven in exactly the same way. □

Again d-dimensional range trees can be made dynamic but the update algorithms are very complicated.

3.3 Related results.

Many other results have been obtained for range searching. See for example [4,5,27,32]. We will shortly describe an interesting approach by Edelsbrunner[9] based on the so-called *priority search tree* by McCreight[15].

A priority search tree is a structure for solving half-infinite range queries in 2-dimensional space, i.e., queries of the form $([A_1 : B_1], [A_2 : \infty])$. Let V be a set of n points in the plane. For convenience, we assume that no two points have an equal value for some coordinate. We store the points, ordered with respect to their first coordinate in a balanced binary search tree T. With each internal node we associate a point of the set in the following way: With the root we associate the point in T with largest y-coordinate. With each internal node δ we store the point in T_δ (the subtree rooted at δ) with largest y-coordinate that has not been stored higher up in the tree. See figure 11 for an example of a priority search tree. It is clear that this structure uses $O(n)$ storage.

To perform a query with range $([A_1 : B_1], [A_2 : \infty])$ we search with both A_1 and B_1 in the tree. For each node on the search path we check whether the associated

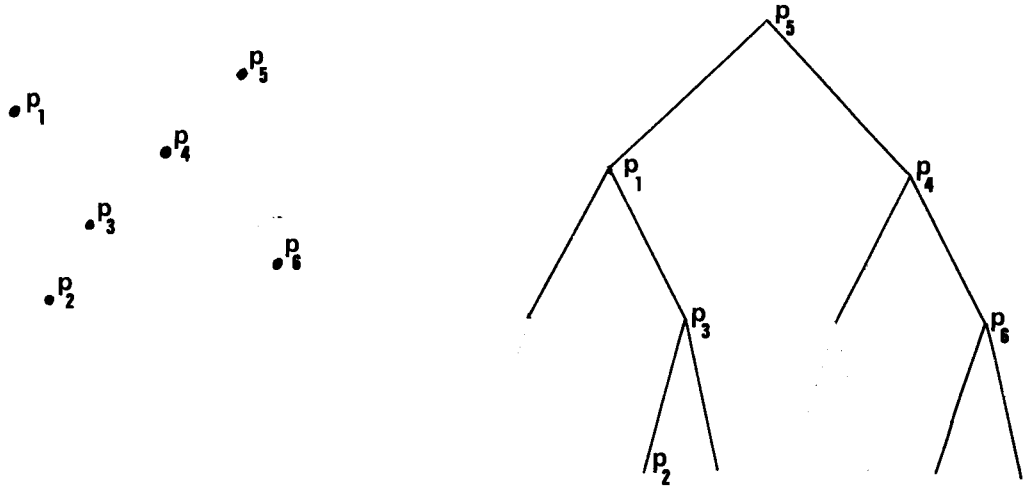

Figure 11: A priority search tree.

point lies in the range and, if so, report it. All other answers must lie in between the two search paths. Let δ be a node that lies in between the two search paths, whose father lies on the search path. We will show how to report all answers in T_δ. All points in T_δ have first coordinate in the range. Hence, we only have to test whether the second coordinate is larger than A_2. δ contains the point in T_δ with largest y-coordinate. Hence, if this point does not lie in the range, no point in T_δ does and we are done. If the point does lie in the range we report it and continue looking at the two sons of δ. It is easily shown that the amount of work for finding all the answers in T_δ is bounded by O(1) plus the number of answers found. Hence, the total query time is $O(\log n)$ for searching with A_1 and B_1, another $O(\log n)$ because we have to spend O(1) time in $O(\log n)$ subtrees plus the total number of answers. For details see [15].

Theorem 3.4 *Given a set of n points in the plane, they can be stored, using $O(n)$ storage, such that half-infinite range queries can be answered in time $O(k + \log n)$ where k is the number of reported answers.*

Edelsbrunner[9] uses this structure to solve general 2-dimensional range queries in the following way. We construct a balanced binary search tree T in which we store all points, sorted by y-coordinate, in the leaves. With an internal node δ that is left son of its father we store a priority search tree P_δ of all points in T_δ. If δ is right son of its father we store an inverse priority search tree, i.e., a priority search tree for ranges that extend downwards to infinity. The structure takes $O(n \log n)$ storage.

Now assume we want to perform a query with range $([A_1 : B_1], [A_2 : B_2])$. We search with both A_2 and B_2 in T. For some time both search paths will be the same but at some internal node δ A_2 will lie in the left subtree and B_2 will lie

in the right subtree. Let δ_l be the left son of δ and δ_r the right son. Clearly all answers lie in T_{δ_l} or T_{δ_r}. All points in T_{δ_l} have second coordinate $< B_2$. Hence, we can as well perform a query on them with range $([A_1 : B_1], [A_2 : \infty])$. For this we can use the structure P_{δ_l}. Similar, for the answers in T_{δ_r} we can as well perform a half-infinite range query with $([A_1 : B_1], [A_2 : -\infty])$ on P_{δ_r}. Hence, the total query time is bounded by $O(k + \log n)$.

Theorem 3.5 *Given a set of n points in the plane, they can be stored, using $O(n \log n)$ storage, such that range queries can be answered in time $O(k + \log n)$ where k is the number of reported answers.*

4 Segment trees.

Range trees are only capable of storing points. In many applications, especially in computer graphics, objects are not points but, for example, rectangles, line segments, etc. In this section we will treat a second data structure, the segment tree, that can, in combination with e.g. range trees, be used for storing other geometric objects. We will first describe the segment tree. Next we will show how it can be used to store rectangles and line segments and we solve some searching problems using it. Finally, we shortly describe some related results like e.g. the interval tree.

4.1 The data structure.

The *segment tree* was introduced by Bentley and Wood[8] to solve some rectangle problems. Although used for soving many two-dimensional problems, the segment tree is a one-dimensional data structure.

Let $V = \{[a_1 : b_1], [a_2 : b_2], \ldots, [a_n : b_n]\}$ be a set of n intervals (segments) on the real line. (Intervals are allowed to intersect, overlap, etc.) All left and right endpoints of the intervals we sort. Let the sorted list be $x_1, x_2, \ldots, x_{2n}$. These endpoints split the real line in a number of so-called elementary intervals $(-\infty : x_1), [x_1 : x_2), [x_2 : x_3), \ldots, [x_{2n} : \infty)$. These elementary intervals we store in the leaves of a balanced binary search tree. With each internal node δ we associate the interval I_δ that is the union of the intervals associated to the sons of δ. (In other words, I_δ in the interval spanned by the subtree rooted at δ.) With each node δ we store a structure T_δ containing all intervals in V that contain I_δ but do not contain $I_{\text{father}(\delta)}$. The form of T_δ depends on the problem we want to solve using the segment tree. See figure 12 for an example of a segment tree. With each node δ the intervals stored in T_δ are indicated.

The segment tree is mostly used for so-called *stabbing queries*. Let p be a point on the line, we want to report all intervals that contain p. To solve this problem we structure T_δ as a simple list of the intervals. To find all intervals containing p, we search with p in the tree, starting at the root. Assume we are at some node δ.

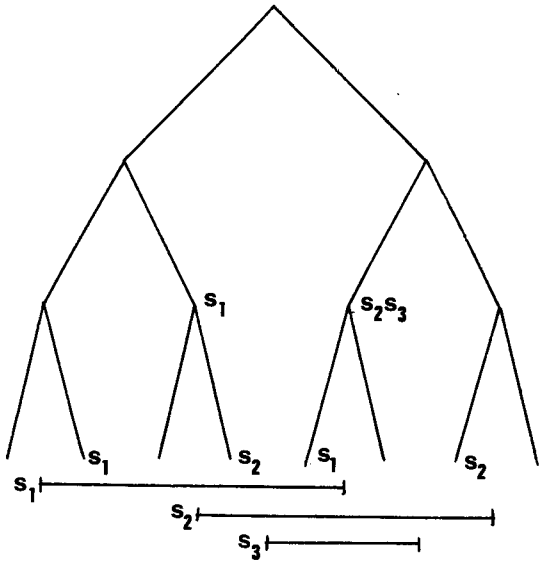

Figure 12: A segment tree.

We report all intervals stored in T_δ. Next we look at the two sons δ_1 and δ_2 of δ. When p lies in I_{δ_1} we continue the search in δ_1. Otherwise we continue the search in δ_2.

The correctness of the algorithm can be seen as follows. Clearly all intervals reported contain p, because for each node δ visited, p lies in I_δ and the reported intervals contain I_δ. No interval is reported more than once, because on each path from the root to a leaf an interval can occur at most once. Finally, if interval i contains p it clearly contains the elementary interval stored in the leaf p ends in during the search. As i does not contain I_{root} there must be some node δ on the search path towards this leaf such that i contains I_δ but not $I_{\text{father}(\delta)}$. i will, hence, be reported when the search passes this node. (In fact, this is not completely correct. When p is the right endpoint of i, i will not be found. This case can easily be solved by either treating endpoints in a special way, or by adding some extra information in the segment tree.)

Theorem 4.1 *Given a set of n intervals, they can be stored in a segment tree, using $O(n \log n)$ storage, such that stabbing queries can be answered in time $O(k + \log n)$ where k is the number of reported answers.*

Proof. The query time follows immediately from the above discussion. It remains to prove the bound on storage required. On each level of the segment tree, each interval can be stored at at most 2 nodes. (This follows from the definition.) Hence, at each level the T-structures together use at most $O(n)$ storage. The bound follows. □

In some applications we are only interested in the number of intervals that contain p. This problem is often called a *stabbing counting query.* In this case the

structure can be made much simpler. As T_δ we just store the number of intervals rather than the intervals themselves. This reduces the storage to $O(n)$. When searching with p we simply add up the numbers stored on the search path. This lead to:

Theorem 4.2 *Given a set of n intervals, they can be stored in a segment tree, using $O(n)$ storage, such that stabbing counting queries can be answered in time $O(\log n)$.*

In the next two subsections we will give some more examples of choices for T_δ to solve different types of searching problems using segment trees.

Segment trees are (depending on the choice of T_δ) dynamic. See e.g. [8] for details.

4.2 Rectangle problems.

In the same way as in range trees, we can construct two- and multi-dimensional segment trees. Let V be a set n of axis-parallel rectangles in the plane. (For example, bounding boxes of a set of graphical objects.) To store them, we project all rectangles on the x-axis. In this way we obtain a set of intervals. These intervals we store in a segment tree. For each node δ we have a number of intervals that have to be stored in T_δ. These intervals correspond to rectangles. These corresponding rectangles we project on the y-axis and we make T_δ into a segment tree that stores these y-intervals. In T_δ we store at each internal node a list of rectangles to which the y-intervals, that should be stored there, belong. It is easy to see that the structure obtained requires $O(n \log^2 n)$ storage.

A two-dimensional stabbing query asks for all rectangles that contain a given point p. In graphics, this is for example important when picking an object. To perform a stabbing query we use the two-dimensional segment tree. Let $p = (p_1, p_2)$. We search with p_1 in the main segment tree. The nodes on the search path together contain all rectangles that contain p_1 in the x-projection. Among them we have to find those whose y-projection contains p_2. To this end we use the T-structures. For each node δ on the search path of p_1 we search with p_2 in T_δ, reporting all rectangles stored at nodes on the search path of p_2. As we have to query $O(\log n)$ T-structures, each requiring time $O(\log n)$ plus the number of answers found, we obtain the following result:

Theorem 4.3 *Given a set of n axis-parallel rectangles, they can be stored in a two-dimensional segment tree, using $O(n \log^2 n)$ storage, such that stabbing queries can be answered in time $O(k + \log^2 n)$ where k is the number of answers found.*

The result can easily be generalised to d-dimensional space, resulting in a structure that uses $O(n \log^d n)$ storage that solves d-dimensional stabbing queries in time $O(k + \log^d n)$.

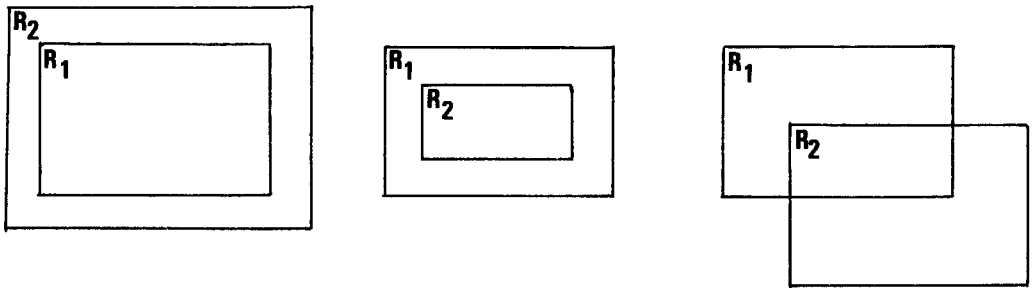

Figure 13: Intersections between rectangles.

As a second example we consider the *axis-parallel line segment intersection problem:* Let V be a set of n vertical line segments and let s be a horizontal line segment, report all line segments in V that are intersected by s.

All line segments in V we project on the y-axis. Of the intervals obtained we construct a segment tree. At each internal node δ we store in T_δ the line segments, ordered by x-coordinate, in a one-dimensional range tree. The structure uses $O(n \log n)$ storage.

Let $s = \overline{(x_1, y)(x_2, y)}$. We search with y in the segment tree. For each node δ on the search path we perform a one-dimensional range query with $[x_1 : x_2]$ on the x-coordinates of the line segments in T_δ. It is easy to see that in this way the correct line segments will be reported.

Theorem 4.4 *Let V be a set of n vertical line segments in the plane. We can store V in a segment tree, using $O(n \log n)$ storage, such that those k line segments intersecting a given horizontal line segment can be found in time $O(k + \log^2 n)$.*

Finally, let us consider the *rectangle intersection problem:* Let V be a set of n axis-parallel rectangles and R another axis-parallel rectangle, report all rectangles in V that intersect R. The problem is important in graphics applications that work with bounding boxes (for example in windowing).

Let R_1 and R_2 be two axis-parallel rectangles. If R_1 and R_2 intersect each other one of the following three cases occur: (i) R_1 is completely contained in R_2. (ii) R_2 is completely contained in R_1. (iii) R_1 intersects R_2 in the boundary. (See figure 13 for the cases.)

Now assume R_1 is in the set V and R_2 is the query rectangle. Case (i) can be solved by constructing a two-dimensional range tree containing for each rectangle in V its left bottom point and performing a range query with R_2. Case (ii) can be solved by constructing a two-dimensional segment tree of V and performing a stabbing query with the left bottom point of R_2. Finally, case (iii) can be

solved by two instances of the axis-parallel line segment intersection problem. One instance stores all vertical boundaries of the rectangles in V and performs a query with the two horizontal boundaries of R_2. The other instance stores all horizontal boundaries of V and performs a query with the vertical bondaries of R_2. (Rectangles might be reported more than once in this way. By checking before reporting this can be avoided without increasing the runtime.)

Combining the results for range trees, two-dimensional segment trees and the line segment intersection problem we obtain:

Theorem 4.5 *Let V be a set of n axis-parallel rectangles in the plane. They can be stored using $O(n \log^2 n)$ storage, such that those k rectangles intersecting a given axis-parallel query rectangle can be reported in time $O(k + \log^2 n)$.*

For more results on rectangle intersection problems see Edelsbrunner[10].

4.3 Windowing in a set of line segments.

As another example of the use of the segment tree we will describe a solution to the so-called *windowing problem*: given a large picture, built out of non-intersecting line segments, compute that part of the picture visible in an axis-parallel rectangle. We will show how to store the picture in a two-dimensional data structure based on the segment tree such that for each given window we can efficiently determine what part of the picture is visible in the window. This is particular useful when the picture is large (compared to the size of the window) and does not change often. For example, a large map stored in a database. The results in this section are based on work by Overmars[18] (see also [19]).

To solve the windowing problem, we split it in two parts. The first part asks for all line segments that lie completely in the window. The second part asks for all line segments that intersect the boundary of the window.

To solve the first problem we can use the solution for range searching described in the previous section. When a line segment s lies completely in the window surely its left endpoint lies in the window. So we can store all left endpoints in a range tree and perform a range query with the window. This will take time $O(\log^2 n)$. The structure uses $O(n \log n)$ storage. Clearly, in this way, we also report some line segments that only partially lie in the window. But we have to find them anyway.

So we only have to consider the second subproblem: find those line segments that intersect the boundary of the window. We will only look at the top boundary. The other boundaries follow in a similar way.

First we project all line segments on a vertical line. In this way each line segment becomes an interval. Of these intervals we construct a segment tree. All intervals stored at some internal node δ completely cover the interval I_δ associated to δ. I_δ corresponds to a horizontal slab in the plane. All the line segment stored

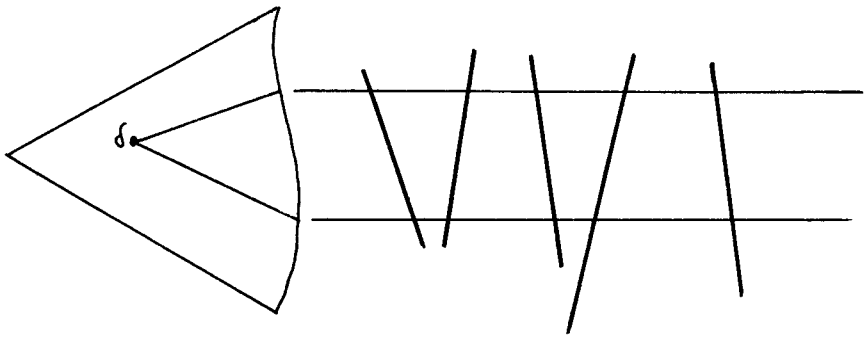

Figure 14: Line segments stored at δ.

at δ intersect the slab completely and, because they do not intersect each other, appear ordered in the slab. See figure 14. We store them in this order in the internal nodes of a balanced binary search tree T_δ. It is easy to see that the structure requires $O(n \log n)$ storage.

To perform a query with the top boundary $\overline{(x_1, y)(x_2, y)}$ we search with y in the segment tree. Clearly, line segments intersecting the top boundary must be stored at one of the nodes on the search path. For each such node δ we search with x_1 and x_2 in T_δ, locating all the segments in between. It is easy to see that this can be done in time $O(\log n)$ plus the number of answers found. As we have to query $O(\log n)$ structures T_δ the total query time is bounded by $O(\log^2 n)$ plus the total number of answers. For details see [17]. It is also shown there that insertions and deletion can be performed in time $O(\log^2 n)$. Combining this with the result for range queries we obtain:

Theorem 4.6 *Given a set of n non-intersecting line segments in the plane, we can store them using $O(n \log n)$ storage such that those k segments visible in a given window can be determined in $O(k + \log^2 n)$ time. The structure is dynamic and can be updated in $O(\log^2 n)$ time.*

Using this method, some of the line segments might be reported more than once. This can easily be avoided by checking whether segments have already been reported before. This can be done in $O(1)$ time per reported segment.

4.4 Related results.

Many other results have been obtained using segment trees and related structures. In this subsection we will shortly discuss a variation of segment trees, called *interval trees*, devellopedby Edelsbrunner[10].

An interval tree again stores a number of intervals on a real line. It consists of a binary tree with the left and right endpoints of the intervals, in sorted order,

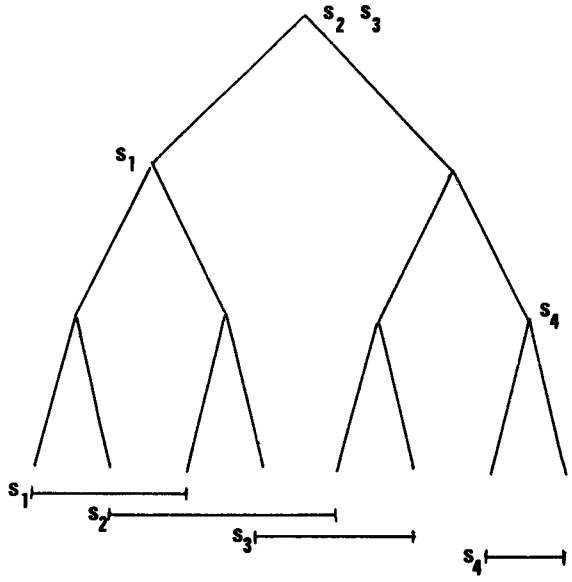

Figure 15: An interval tree.

stored in the leaves. In internal nodes we store splitting values inbetween the endpoints to guide the search, like in a normal search tree. Each interval we store at the highest internal node δ such that the splitting value of δ is contained in the interval. It is obvious that each interval is stored at exactly one internal node. For each internal node δ we store the intervals sorted by left and sorted by right endpoint. See figure 15 for an example of an interval tree.

To perform a stabbing query with a point p we search with p in the interval tree. Each interval containing p must be stored at some node δ on the search path. For each node δ we proceed in the following way: Let s be the splitting value at δ. Assume $p < s$. Then all intervals stored at δ have their right endpoint to the right of p. Hence, we have to report those that have their left endpoint to the left of p. Because we have the intervals sorted by left endpoint, we can just walk along this sorted list, reporting all intervals, until we find an interval with left endpoint to the right of p. This takes time O(1) plus the number of answers found. Similar, when $p \geq s$ we walk along the list sorted by right endpoint. This leads to the following result:

Theorem 4.7 *Given a set of n intervals, they can be stored in an interval tree, using $O(n)$ storage, such that stabbing queries can be answered in time $O(k+\log n)$ where k is the number of reported answers.*

Hence, we reduced the amount of storage from $O(n \log n)$ to $O(n)$. Unfortunately, the interval tree cannot be used for some other searching problems. For example, it cannot solve the stabbing counting problem efficiently. But we can use it as the T_δ structures in the two-dimensional segment tree. This leads to the following improved versions of theorems 4.3 and 4.5:

Theorem 4.8 *Let V be a set of n axis-parallel rectangles in the plane. They can be stored using $O(n \log n)$ storage, such that stabbing queries can be answered in time $O(k + \log^2 n)$ where k is the number of reported answers.*

Theorem 4.9 *Let V be a set of n axis-parallel rectangles in the plane. They can be stored using $O(n \log n)$ storage, such that those k rectangles intersecting a given axis-parallel query rectangle can be reported in time $O(k + \log^2 n)$.*

5 Dynamisation.

We will end this paper with a discussion on some general methods for turning static data structures that do not allow for efficient insertions and deletions into structures that do allow for updates. Because many data structures for geometric problems tend to be static (or the update algorithms are very complicated) such general methods might be very useful. We will only outline some main ideas. For a detailed study of dynamisation methods, the reader is referred to Overmars[17].

5.1 Decomposable searching problems.

The most important dynamisation methods are based on some property of the problem for which the data structure is used. Bentley[3] defined the class of so-called *decomposable searching problems*. He observed that one does not need to keep all objects of a set in one massive data structure as long as it is possible to obtain the answer to a query over the total set out of the answers to the same query over some partition of the set. Let $PR(x, V)$ denote the answer to a query problem PR with query object x over a set V.

Definition 5.1 *A searching problem PR is called* decomposable *if and only if for any partition $A \cup B$ of the set V and any query object x,*

$$PR(x, V) = \square(PR(x, A), PR(x, B))$$

for some constant time computable operator $\square$.

For example, when we know whether $x \in A$ and whether $x \in B$ we can compute in O(1) time whether $x \in V$ using $\vee$ (or). Hence, the membership problem is decomposable. Another example is the nearest neighbor problem that asks for the point in a planar point set that is nearest to a given query point x. When p is nearest to x in A and p' is nearest to x in B than either p or p' is nearest in V. This can of course be determined in O(1) time. Also the range searching problem is decomposable using $\square = \cup$. (Note that, as A and B are disjoint, the answer sets over A and B are disjoint and, hence, can be merged in constant time.) Not all searching problems are decomposable. For example, the convex hull problem

that asks whether a query point x lies inside the convex hull of a set of points is not decomposable.

For decomposable searching problems one need not store all objects in one large (static) data structure. Instead, the set can be maintained as a "dynamic system" of disjoint subsets. Each subset is stored in a (static) data structure, called a *block*. Queries are performed by querying the blocks separately and combining the answers using the operation $\square$. The time required for combining the answers is bounded by the number of blocks and is clearly overshadowed by the time required for querying the blocks. Hence, we can neglect it.

Insertions are in general performed by rebuilding some small blocks with the new object included. A deletion is performed by rebuilding the block the object belongs to. Observe that (i) maintaining a large number of small blocks will lead to low update times and high query times, while (ii) maintaining a small number of large blocks will lead to high update times and low query times.

Numerous methods have been proposed to decompose the set into blocks and to maintain the partition when objects are inserted or deleted, to obtain an optimal balance between query time and update time (see [17] for an overview of the methods). There are two main different approaches. The *equal block method* splits the set in about equal sized blocks. The *logarithmic method* splits the set in blocks of exponentially increasing sizes. We will only describe the logarithmic method here.

We split the set V in disjoint subsets V_0, V_1, V_2, ..., such that for each i $|V_i| = 2^i$ or V_i is empty. For example, when $|V| = 26$, V_1, V_3 and V_4 are filled. For each non-empty V_i we build a data structure S_i containing the 2^i objects.

To insert an object p, we act the following way:

1. Locate the smallest i with V_i empty.
2. Discard $S_0 \ldots S_{i-1}$.
3. Build S_i out of p and the objects from $S_0 \ldots S_{i-1}$.

Note that the new S_i will contain exactly 2^i objects. See figure 16 for an example of how data structures are constructed and discarded.

Sometimes an insertion will take a lot of time (when i is big we have to rebuild a large structure) but, as after such an insertion all preceeding S-structures are empty again, the average insertion time will remain low.

Let the structure S have a query time of $Q(n)$, a construction time of $B(n)$ and let it use $M(n)$ storage. The complexity of the new data structure is expressed by the following theorem:

Theorem 5.1 *Given a (static) data structure S for a decomposable searching problem, the logarithmic method yield a new data structure with a query time of $O(\log n)Q(n)$, an average insertion time of $O(\log n)B(n)/n$, using $O(M(n))$ storage.*

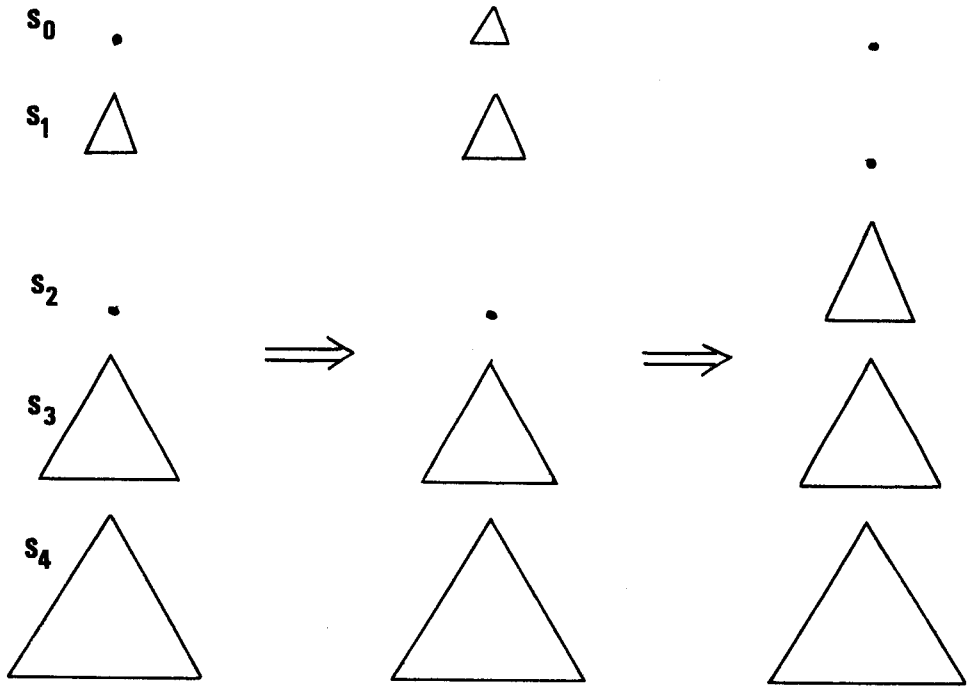

Figure 16: The logarithmic method.

Proof.

When the size of the set is n, the biggest i for which S_i is filled is $i = \lfloor \log n \rfloor$. Hence, to answer a query we have to perform the query on $O(\log n)$ S-structures. The query time bound follows.

The amount of storage required is clearly bounded by

$$\sum_{j=0}^{\lfloor \log n \rfloor} M(2^j)$$

which is bounded by $O(M(n))$.

To estimate the average insertion time, notice that when we build a block S_j all its objects come from lower indexed blocks. Hence, each object in the set has to be built at most once into each block B_j with $0 \leq j \leq \lfloor \log n \rfloor$. Dividing the costs for building a block S_j over the objects it is built from makes for $B(2^j)/2^j$ per object. Hence, the total amount of time charged to an insertion is bounded by

$$\sum_{j=0}^{\lfloor \log n \rfloor} B(2^j)/2^j$$

which is bounded by $O(\log n)B(n)/n$. □

The average bounds can be turned into worst-case bounds. It is also possible to perform deletions in some general way. See [17] for details.