DYNAMIC DATA STRUCTURES FOR ORTHOGONAL INTERSECTION QUERIES

by

H. Edelsbrunner *)

Abstract.

This paper investigates data structures supporting d-dimensional intersection queries involving orthogonal objects such as rectangles, line segments, points, etc. A d-dimensional orthogonal object is defined to consist of d properties, where each property belongs to a unique dimension and denotes either an interval or a single value.

We develope the so-called SRI-pyramid family whose members are multi-component trees composed of instances of the segment tree, the range tree, and the interval tree. This family allows us to efficiently attack arbitrary instances of orthogonal intersection queries such as rectangle intersection queries, line segment intersection queries, range queries, etc.

The new results obtained in this paper are first, a general attack on a great variety of different searching problems, and

second, the development of dynamic relatives of the well known segment tree and interval tree which lead together with the dynamic relative of the range tree to dynamic multi-component trees supporting arbitrary orthogonal intersection queries. Although we emphasize the new dynamic solutions, a bulk of the presented static data structures are described for the first time too.

*) Institut fuer Informationsverarbeitung,
Technical University Graz,
Steyrergasse 17,
A-8010 Graz, Austria.

1. Introduction.

This paper offers a general approach to multi-dimensional intersection problems involving orthogonal objects. A d-dimensional geometric object is called orthogonal if it consists of d properties, where each property belongs to a unique dimension and is either an interval or a single value. The orthogonal object stands for the Cartesian product of its properties. Two orthogonal objects A and B are said to intersect if they have at least one point in common.

Examples of orthogonal objects are rectilinearly oriented d-dimensional rectangles (for short d-recs), line segments, points in d-space, and others.

Applications of solutions to such problems can be found in areas like Computer Graphics, Databases, VLSI, etc.

The various problems involving objects of the above kind can be formulated in two fundamentally different manners (as is true of many standard problems in computational geometry):

(1) Given a set of equal typed objects and a (query-) subject which is another kind of object, a searching problem asks for all objects that intersect the subject.

(2) Given a set of equal typed objects, an all pairs problem asks for all pairs of intersecting objects.

Sometimes, one does not desire all intersecting objects or pairs but rather just the number of them, or one even only wants to know if any intersect at all. The structures described in this paper support the original folumulation of the problems, i.e. they serve to detect all intersecting elements. Nevertheless, the structures may be slightly modified to be fit for determining the number of intersecting elements or even to decide if there exists an intersection or not.

This paper will mainly deal with the searching problem variant of the various intersection problems. However, the developed data structures are well applicable to the all pairs problems.

There are a number of preliminary results concerning special cases of orthogonal intersection queries. Above all, the range searching problem caused much interest in the last few years. This problem involves a set of points in d-space as objects and asks for all points that lie in a query d-rec (which is equivalent to asking for all points that intersect the d-rec). The following five papers that deal with range searching have some relevance for our further discussion: Bentley, Maurer [2], Bentley [1], Lueker [12, 13], and Willard [27]. Recently, the inverse range searching problem (with d-recs as objects and a point in d-space as subject), the rectangle intersection searching problem (involving d-recs for objects and subjects), and the 2-dimensional line segment intersection searching problem (with, e.g., horizontal line segments as objects and a vertical one as subject) have been investigated. The respective results can be found in Vaishnavi [21], Six, Wood [18, 19], Edelsbrunner [6], and Vaishnavi, Wood

[23].

Searching problems are settled by initially building up a data structure accommodating the given objects. The structure has to be carefully designed that it allows for a query to be answored as fast as possible and by simultaneously requiring as little space as possible.

In static environments, the cost functions for query time, space, and preprocessing time needed to set up the structure, describe the behaviour of the solution.

In dynamic environments, one desires to intermix update commands and queries arbitrarily. Thus, the structure has to be designed flexible enough to allow for efficient insertion of new objects and deletion of old objects. Of course, the design of dynamic data structures is much more difficult, since the fast query time has to be maintained by simultaneously requiring as little space, insert, and delete time as possible.

There are two fundamentally different approaches for dynamizing static structures solving decomposable searching problems. (For a definition of ´decomposable´ consult for instance Bentley, Saxe [3].)

The first dynamization approach maintains a system of instances of the static structure solving the particular problem which implies an increase in query time, due to the necessity of searching each instance, and an improvement of the update time, since a single update command causes the change of only a small part of the system. The reader who desires more detailed information is referred to Bentley, Saxe [3], Willard [28], Overmars, van Leeuwen [17], and van Leeuwen, Maurer [8] that deal with the so called logarithmic block method, and to Maurer, Ottmann [14], van Leeuwen, Wood [10], and Edelsbrunner [5] that describe the so called equal block method.

The second approach works with tree structures and replaces optimal trees used in static environments by balanced trees. In multi-dimensional circumstances much care has to be taken to employ the best suited balanced tree scheme. The interested reader is referred to Willard [25, 26, 27], Lueker [12, 13], and Edelsbrunner [6] where examples of this approach are given. Our paper employs the second approach for the dynamization of a few additional static 1-dimensional data structures, namely the segment tree and the interval tree, and a whole family of multi-dimensional trees, the so called SRI-pyramids.

The paper is organized as follows: First, the basic data structures, namely the segment tree, the range tree, and the interval tree are recalled and dynamic versions of the segment and the interval tree are developed. Second, Section 3 shows how static and dynamic multi-component trees composed of the above types support the various intersection searching problems. Finally, Section 4 demonstrates some applications yielding new results for a number of special orthogonal intersection problems.

2. New Dynamic Tree Structures.

In this section we will provide a few new results in the design of dynamic data structures supporting multi-dimensional searching problems. Initially, recent results concerning the dynamic range tree developed by Willard [25, 26, 27] are reviewed. Then, the following two subsections will apply the same dynamization approach to the segment tree and the interval tree (called the rectangle tree in the original paper of Edelsbrunner [6]).

Most of our results concerning the dynamic behaviour of data structures will be given in average complexity. An update is carried out in average time $f(N)$ if a sequence C of update commands is executed in $|C|f(N)$ time in the worst case, where C consists of $|C|$ update commands and N denotes the maximal number of objects ever stored in the initially empty data structure. However, a bulk of the results carry over to worst-case complexity per update command due to a clever refinement of balancing algorithms developed by Willard [25, 26].

Before discussing Willard's dynamic range tree we briefly review the static shape of the range tree originally introduced by Bentley [1].

The 1-fold range tree is just an optimal, sorted, binary tree storing values (i.e. 1-dimensional points) in its leaves. The d-fold range tree consists of a 1-fold range tree storing the d-th coordinates of d-dimensional points. In addition, each inner node v is associated with the (d-1)-fold range tree accommodating all the points whose d-th coordinates are stored in leaves descending from v.

It is well known that the d-fold range tree representing n d-dimensional points requires $O(n\log^{d-1}n+n\log n)$ preprocessing and $O(n\log^{d-1}n)$ space. The significant property of the d-fold range tree is that it allows for d-dimensional range queries to be answered in $O(\log^{d}n+t)$ time when t points are found to lie in the query d-rec.

Next we recall Willard's approach for dynamizing the above static structure.

The optimal binary trees are replaced by trees of bounded balance, introduced by Nievergelt, Reingold [16] and improved by Willard [25, 26]. From an accounting argument it can be derived that the new structure supports dynamic range searching with the same bounds for searching and for space as the static tree, but with $O(\log^{d}N)$ update time on the average. (The particular argumentation will again be used and described in the following two subsections.)

A further improvement is obtained by a subtle refinement of the above procedure to assure $O(\log^{d}n)$ update time per command in the worst case. It should be mentioned that this refinement (Willard called the outcoming procedure a super-B-tree algorithm) can be employed to improve the subsequent results concerning the segment tree to worst-case complexity, too. The results for the interval

tree, however, can be improved to worst-case complexity only by increasing the update time by a factor O(logn).

We feel it important to indicate that the dynamic versions of the segment tree and the interval tree are not only relevant to our results concerning orthogonal intersection queries. For instance, Edelsbrunner, Maurer [7] described data structures based on the segment, resp. interval tree, that support general point locating and may gain by our dynamic tree structures.

2.1. The Dynamic Segment Tree.

This section is devoted to the dynamization of the segment tree whose static shape caused much attention in the last few years. For example, it proved useful in circumstances involving orthogonal objects such as recs or line segments parallel to one coordinate axis.

Before dynamizing this useful static data structure we give a succinct description of what we call the static segment tree. For more detailed exhibition consult e.g. van Leeuwen, Wood [10].

Let us assume that there is given a set M of n intervals on the only coordinate axis. Each interval I of M is specified by a left end l_I and a right end r_I, and I stands for all points in between l_I and r_I inclusive the two ends. The intervals may intersect or enclose one another in an arbitrary manner, but for the time being we do not allow two equal ends. This agreement will simplify the subsequent discussion and, in this way, intensify the clearness of our presentation.

Let E be the sorted list of all left and right ends. Since E comprises exactly 2n different values, E induces a partitioning of the coordinate-axis into 2n+1 atomic intervals, henceforth termed <u>fragments</u>. Notice that the leftmost and the rightmost fragment extents to -∞, resp. +∞. The segment tree T accomodating the set M of intervals consists of an optimal tree structure, whose every leaf is in one-to-one correspondence with a fragment, more precisely, the i-th leaf from left corresponds with the i-th fragment from left, for all i from 1 to 2n+1. Each inner node v of T is associated with a standard interval, termed <u>segment</u>, which is the set-theoretical union of the fragments associated with the leaves descending from v. In addition, each node v is assigned a two-way linear list, called v's node list, housing all those intervals of M that cover the associated segment but do not cover the one of v's father.

Figure 2-1. depicts the segment tree for the interals A=[0,5], B=[2,9], and C=[3,7].

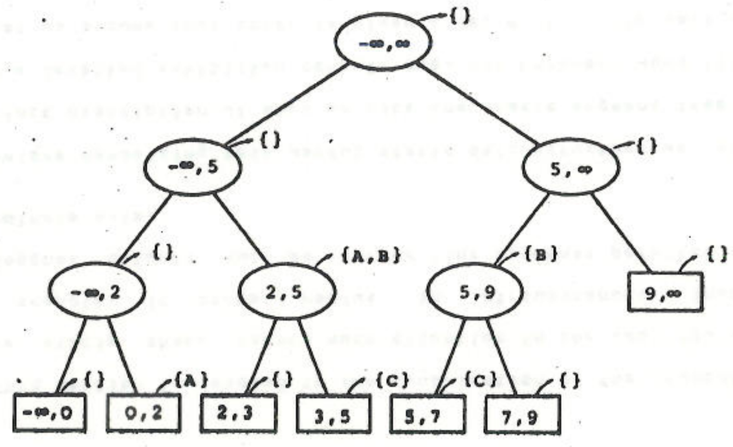


Figure 2-1. The segment tree for three intervals.

The segment tree for n intervals requires O(nlogn) space and can be built in O(nlogn) time. The space requirements are due to the fact that each interval belongs to O(logn) node lists. The construction of the tree is carried out in two phases: first, the optimal tree with 2n+1 leaves is built, and second, the n intervals are inserted into the particular node lists.

The significance of the segment tree is due to the fact that it allows for a 1-dimensional inverse range query to be answered in O(logn+t) time, when t intervals contain the query value. The query for a value is carried out by descending the segment tree from the root to the leaf that corresponds with the fragment that contains the value. The intervals stored in the node lists of the visited nodes are the desired ones. Note that none of the intervals found is detected more than once, since the node lists associated with ancestors of a fixed leaf are throughout disjoint.

After having introduced the notion of a static segment tree, we will focus on a dynamic version of the tree that is based on trees of bounded balance, introduced by Nievergelt, Reingold [16], and on a clever refinement of these trees, recently developed by Willard [25, 26].

In the sequel, we will describe the mechanism for balancing the segment tree before discussing the update strategies used for inserting and deleting intervals. Finally, we will give an analysis of the average dynamic behaviour.

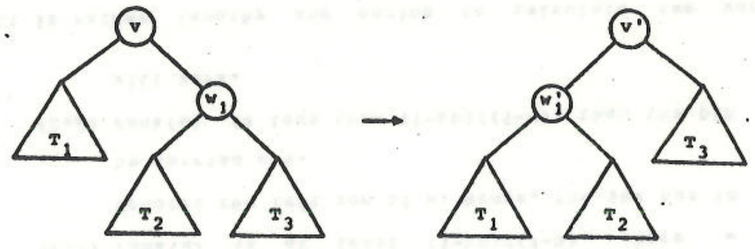
The main idea behind our dynamization of the segment tree is the same as behind the standard dynamization of simple binary trees: We do not insist on the optimal balance but on a more relaxed one.

Each node v of the segment tree T is assigned an integer, termed rank(v), which denotes the number of leaves descending from v. Further, we call b(v):=rank(w)/rank(v) the balance of v, whereby w denotes the left son of v. The segment tree T is called α-balanced if the balance of each of its inner nodes lies in the [α,1-α] range, for an α in (0,1/2].

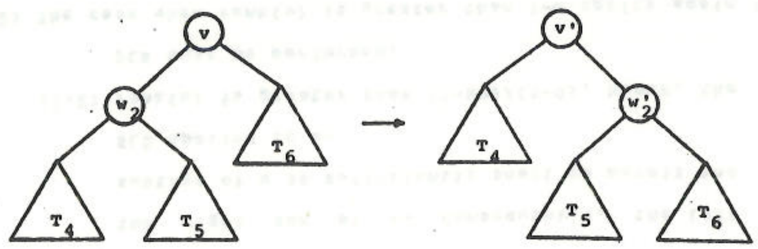
Our balanced segment tree will be α-balanced for a fixed α in

$(0,1-\sqrt{k^2-k}/k]$, where k is a fixed real greater than 2. It can be shown that one of four types of restructurings suffice to force a node v back into the desired range when its balance has moved outside of [α,1-α]. These restructurings are the well known single and double rotations. It has to be noted that these restructurings will be effective only if the rank of the rebalanced node is big enough. Nodes with small ranks can be treated by completely rebuilding their subtrees.

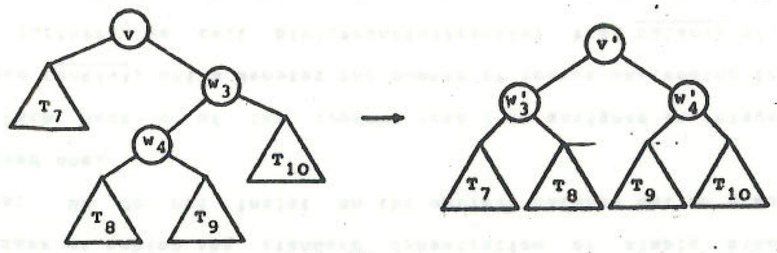
(1) Single left rotation (SLR).



(2) Single right rotation (SRR).



(3) Double left rotation (DLR).



(4) Double right rotation (DRR).

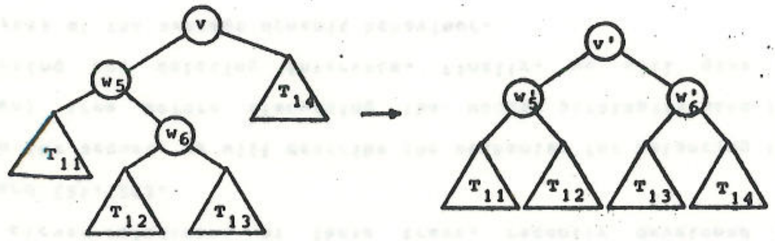


Figure 2-2. The four types of restructurings.

Assume that the inner node v has moved out of balance, i.e. b(v) is either less than α or greater than $1-\alpha$. Assume further that the rank of v is big enough, i.e. rank(v) is greater than $1/\alpha^2$.

In order to force b(v) back into the $[\alpha,1-\alpha]$ range, one of the four restructurings above has to be chosen as follows:

(1) rank(v) is less than α. It is now clear that a left rotation must be performed.

(1.1) rank(w) is at most $(1-k\alpha)/(1-\alpha)$, where w denotes

the right son of v. Consequently, the left subtree of w is sufficiently small to permit the SLR applied to v.

(1.2) rank(w) is greater than $(1-k\alpha)/(1-\alpha)$, hence, the DLR must be performed.

(2) The case when rank(v) is greater than $1-\alpha$ splits again in two subcases.

(2.1) rank(w) is at least $(1-k\alpha)/(1-\alpha)$, where w denotes the left son of v. Hence, the SRR has to be carried out.

(2.2) rank(w) is less than $(1-k\alpha)/(1-\alpha)$, then the DRR will work.

It is rather lengthy and boring to calculate the various inequalities in order to verify that the application of one of the four restructurings in the above manner indeed forces the balance of v back into the desired range. This work is left to the interested reader. Note that a rotation forcing the balance of v back into $[\alpha,1-\alpha]$ does not affect the balance of any other node.

It should be clear that the application of a restructuring of the above type makes it necessary to reconstruct the node lists for a few nodes involved. This aspect will be examined for the SLR and DLR only, since the SRR and DRR are just the mirror images of the SLR and DLR.

Let us consider the SLR first and let v, w_1', and w_1'' denote the nodes depicted in Figure 2-2.(1). In addition, let t_1, t_2, and t_3

denote the roots of T_1, T_2, and T_3 before the SLR is carried out, and let t_1', t_2', and t_3' denote the roots of T_1, T_2, and T_3 after having performed the SLR. From trivial reflections one can derive that w_1, t_1, t_2, and t_3 have to be replaced by the new nodes w_1', t_1', t_2', and t_3' which are associated with new node lists that must be constructed when restructuring v's subtree. (We talk about new nodes if the corresponding node lists must be reconstructed. In practice, one will not replace the nodes but only the node lists and the segments if necessary.)

Next, we present a rather detailed table laying open how the new node lists have to reflect the application of the SLR to node v, see Figure 2-2.(1). (The new node lists are displayed by using set notations freely.)

First, the nodes w_1, t_1, t_2, and t_3 have to be replace by w_1', t_1', t_2', and t_3', respectively. For convenience, let NL(x) denote the node list of node x and mean the set of intervals stored in it. Then, the new node lists can be described as follows:

$NL(w_1') = NL(t_1) \cap NL(t_2)$,

$NL(t_1') = NL(t_1) - NL(t_2)$,

$NL(t_2') = NL(w_1) \cup (NL(t_2) - NL(t_1))$, and

$NL(t_3') = NL(w_1) \cup NL(t_3)$.

For the exhibition of the new node lists to be constructed when applying the DLR to node v, we will talk about v, w_3, w_4, t_7, t_8, t_9, and t_{10} as well as about w_3', w_4', t_7', t_8', t_9', and t_{10}' in

accordance to Figure 2-2.(3) and to the definition of primed t-nodes used above.

Again, the nodes w_3, w_4, t_7, t_8, t_9, and t_{10} have to be replaced by the new nodes w_3', w_4', t_7', t_8', t_9', and t_{10}', and the associated node lists must reflect the following equalities:

$NL(w_3') = NL(t_7) \cap (NL(w_4) \cup NL(t_8))$,

$NL(w_4') = NL(w_3) \cup (NL(t_9) \cap NL(t_{10}))$,

$NL(t_7') = NL(t_7) - (NL(t_8) \cup NL(w_4))$,

$NL(t_8') = NL(w_3) \cup (NL(w_4) - NL(t_7)) \cup (NL(t_8) - NL(t_7))$,

$NL(t_9') = NL(w_4) \cup (NL(t_9) - NL(t_{10}))$, and

$NL(t_{10}') = NL(t_{10}) - NL(t_9)$.

It has to be mentioned that the construction of the various new node lists can be carried out in time proportional to the number of intervals involved. However, in some circumstances, one may wish to arrange the node lists not just as unsorted linked list but in some other fashion, e.g. as interval tree or anything else. The following analysis will accommodate this slightly more general notion by assuming that such a node list comprising m elements allows for updates to be carried out in f(m) time on the average.

After having demonstrated the performance of restructurings, we are ready to present the update strategies and to analyze the time requirements for carrying out a sequence of update commands.

Here, the procedure that inserts an interval is described in appropriate detail whereas the procedure that deletes an interval will be skimmed only.

Let T be an α-balanced segment tree. Let n intervals be stored in T and assume that no two ends are the same. Hence, T comprises 2n+1 leaves and the leftmost and the rightmost leaf is associated with an infinite fragment. Two steps have to be performed in order to insert an interval I into T.

<u>Step</u> 1: Descend T and look for the leaf l associated with the fragment f that contains the left end l_I of I. Change l into an inner node and assign two new leaves l_1 and l_2 as left and right son of l. (l_1 correspondens to that part of f that lies to the left of l_I, l_2 correspondens to the part right of l_I.)

Ascend T and adjust the ranks of the nodes visited. Rebalance all nodes whose balances move outside $[\alpha,1-\alpha]$.

Repeat the above actions for the right end r_I of I.

<u>Step</u> 2: Insert I into the various node lists starting at the root of T.

Let v denote the current node whose segment is considered. If I covers the segment of v then I must be added to v's node list, resp. v must be inserted into the structure representing v's node list. On the other hand, if I does not cover the segment of v, the sons of v whose segments intersect I have to be considered in the same manner.

The strategy for deleting an interval resembles the one displayed above. The only differences are that two leaves have to be merged which can be accomplished by removing one and replacing the other by the new leaf, and that I has to be deleted from the appropriate node lists. A dictionary that associates with each interval its appearances in the various node lists enables fast deletion for the case that the node lists are arranged as unsorted two-way list. This dictionary can be maintained without deteriorating the time bounds for insertion and deletion.

It can be readily seen that all actions except the ones employed for constructing new node lists can be carried out in O(logn) time on the average. In what follows, the average time needed for building the new node lists will be analyzed.

<u>Lemma</u> 1: Let v be an inner node of an α-balanced segment tree T and let r be the rank of v. Then the number of intervals in v's node list is upper bounded by $r(1-\alpha)/\alpha$.

<u>Proof:</u> Let f denote v's father and w v's brother. Note that the inequality pair responsible for the balance of v

$$\alpha \leq r/\mathrm{rank}(f) \leq 1-\alpha$$

can be reformulated as

$$\alpha/(1-\alpha) \leq \mathrm{rank}(w)/r \leq (1-\alpha)/\alpha.$$

Consequently, the number of leaves descending from w is at most equal to $r(1-\alpha)/\alpha$.

W.l.g. let v denote the left son of f. Each interval in v's node list must not appear in the node list of w. Hence, if the m leftmost (and none more) leaves descending from w are contained in an interval I in v's node list, m must be an integer at least equal to 0 and less than rank(w). From our assumption that no two equal ends exist, we can derive that each interval in v's node list correspondens to a unique m. Thus, the number of intervals in the node list of v is at most equal to the number of leaves descending from v's brother which in turn is at most r(1-α)/α.

[]

Of course, our argument holds only for sets of intervals meeting our restrictive assumptions that no two ends coincide. If this is not true, things can be handled as follows:

Assume first that two intervals I and J share the same right end. This fact causes two modifications in the above discussion. First, the update procedures must be designed to detect such cases and treat them appropriately. Second, it implies that I and J correspond to the same m in the proof of Lemma 1. Both difficulties may be eliminated by assigning ranks different from 1 to the leaves in question. For our case this would mean that the leaf in correspondence with the fragment immediately to the left of the shared right end is assigned the rank 2. The case that I has a left end equal to the right end of J can be treated by adding 1/2 to the ranks of the leaves immediately to the left and right.

<u>Lemma</u> 2: The time required to carry out a single or double rotation including the construction of new node lists is proportional with a factor f(n) to the decrease in leaf depth caused by the rotation. (The <u>leaf depth</u> of tree T is defined as the sum of the distances of all leaves from the root. The <u>distance</u> of a leaf l from the root of T is equal to the number of inner nodes visited when descending to l.)

<u>Proof:</u> Let us examine the case of performing the SLR. The other cases are very similare and are left to the reader.

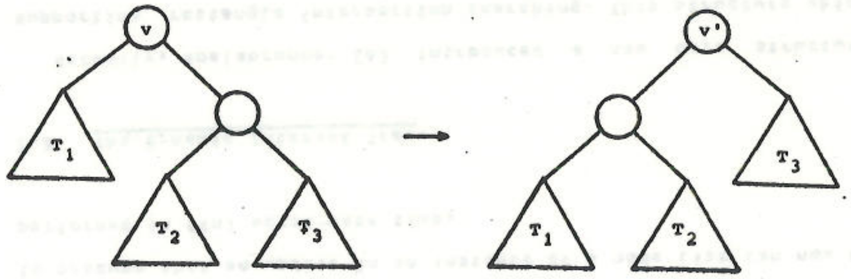


Figure 2-3. The SLR applied to v.

This type of restructuring is carried out when the number of leaves in T_1 (denoted by $|T_1|$) is less than αrank(v) and when $|T_3|$ is at least (k-1) α rank(v).

The depth of each leaf in T_1 increases by 1 and the depth of each leaf in T_3 decreases by 1. Hence, the decrease in total leaf depth is at least (k-2) α rank(v) and thus proportional to rank(v). (Note that k is chosen greater than 2 and therefore the above factor $(k-2)\alpha$ is positive.)

The performance of the SLR forcing the balance of a node v into $[\alpha,1-\alpha]$ takes O(f(n)rank(v)) time, since the only time costing action is the construction of the node lists for w' and the roots of T_1, T_2, and T_3. From Lemma 1 we know that there will be at most O(rank(v)) intervals in the node lists in question.

[]

Theorem 1: The above algorithm performs a sequence C of insert and delete commands in O(|C|f(n)logN) time, where |C| denotes the length of C (i.e. the number of update commands in C) and where N is the maximal number of stored intervals, while executing C.

Proof: We assume to start with the empty segment tree. From the definition of N we can derive that the tree never exeeds the height O(logN). This implies that all actions except for the construction of new node lists can be performed in O(|C|logN) time.

Moreover, since the increase in leaf depth caused by the |C| updates is at most O(|C|logN) the building of new node list can be carried out in O(|C|f(n)logN) time.

[]

As concluding remark of this subsection devoted to the dynamization of the segment tree, we note that the strategy for updating may be refined to worst-case complexity. This refinement

consists of an almost straightforward application of Willard's super-B-tree algorithm. The only modifications that have to be considered are the growing number of anticipated nodes (now each node is assigned 20 anticipated nodes) and that a single update causes changes along to root-leaf ways in the tree.

Thus, we postulate that the segment tree can be dynamized such that its height is bounded by O(logn) and each update command can be carried out in O(f(n)logn) worst-case time. Notice that we have to presume that an update in an instance of a node list can now be performed in f(n) worst-case time.

2.2. The Dynamic Interval Tree.

Recently, Edelsbrunner [6] introduced a new data structure supporting rectangle intersection searching. This structure which he called the d-fold rectangle tree is based on a ternary tree structure (the 1-fold rectangle tree) which we will call the interval tree. In this paper we will describe a slightly modified version of the interval tree that leads to a better dynamic structure than developed by Edelsbrunner [6].

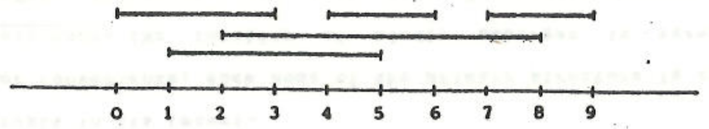
As was done for the segment and the range tree we give, first of all, a succinct description of the static shape of the interval tree. It is a ternary tree structure supporting interval intersection searching in O(logn+t) time, where n intervals are stored and t of them intersect the query interval. The optimal query time is achieved simultaneously with the optimal space requirements (O(n) space is needed) by augmenting inner nodes with fingers (i.e. pointers to leaves) and arranging the leaves in a two-way linear list.

Let M be the given set of n intervals where each interval I is specified by its left end l_I and its right end r_I. For the time being assume that there are no two intervals with the same left end. Let L be the sorted array of the left ends of all intervals.

The interval tree T storing the intervals of M consists of two parts: the primary structure depending on L, and the secondary structure depending on the primary structure and on M. The primary structure is an optimal, sorted, binary tree storing the values of L in its leaves. Every inner node is assigned a value that lies between the greatest value of any node in its left subtree and the smallest value of any node in its right subtree. Each node v of the primary structure is associated with a middle subtree accommodating all intervals in M that contain the value assigned to v and that do not contain any value assigned to an ancestor of v. The middle subtree representing a set of intervals is an optimal, sorted, binary tree housing the ends of the intervals in its leaves.

For convenience, each node of the primary structure is called a primary node, the totality of middle subtrees is termed the secondary structure, and each node of a middle subtree is also termed secondary.

In addition to the primary and secondary structure we need a so called auxiliary information comprising two kinds of pointers: First, the totality of secondary leaves is arranged as two-way linear list. A leaf l_1 appears to the left of another leaf l_2 in the two-way list if either l_1 belongs to a middle subtree whose primary node is assigned a smaller value than the one of l_2's middle subtree, or l_1 and l_2 belong to the same middle subtree and l_1 is associated with a lower end than l_2. Second, each primary node and each secondary root (i.e. a middle son of a primary node) is augmented with two fingers, i.e. pointers to the leftmost and the rightmost secondary leaf in its subtree, respectively.



L = (0,1,2,4,7)

M = ([0,3],[1,5],[2,8],[4,6],[7,9])

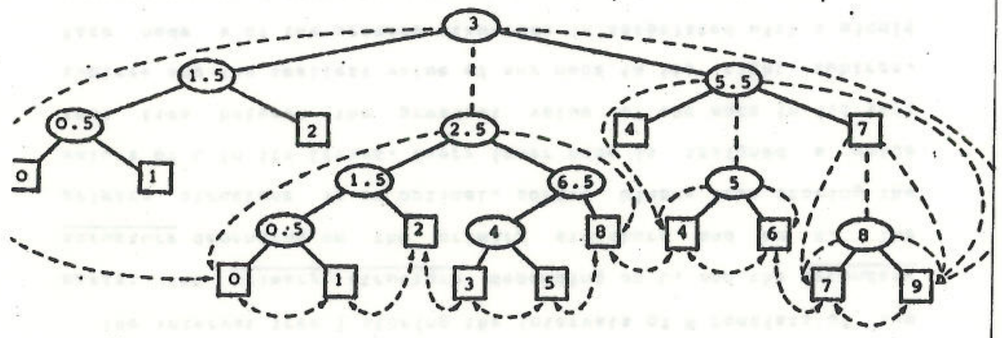


Figure 2-4. An interval tree accomodating five intervals.

The space requirements are upper bounded by O(n), since there are exactly 2n-1 primary nodes and each interval is stored only once in only one middle subtree, consequently, there are at most O(n) secondary nodes. It is clear that only O(n) auxiliary pointers have to be established, hence, the interval tree needs only O(n) space. It can be built up in O(nlogn) time by a more or less obvious preprocessing algorithm and an interval intersection query can be answered in O(logn+t) time by exploiting the pointer

mechanism supported by the auxiliary information. For a more detailed description see Edelsbrunner [6].

Now, we are ready for the discussion of an analogous dynamization as undertaken in Section 2.1. That is, the primary structure (originally an optimal, binary tree) is replaced by an α-balanced binary tree as introduced by Nievergelt, Reingold [16], improved by Willard [25, 26], and reviewed in Section 2.1. The primary structure is called α-balanced if the balance b(v)=rank(w)/rank(v) of any primary node v is in [α,1-α], whereby w denotes v's left son. In addition, each middle subtree is replaced by an arbitrary balanced tree scheme, e.g. again an α-balanced binary tree. The auxiliary information is defined as for the static interval tree.

Next we present the update procedures which will be analyzed in sequence.

Let us consider the case of inserting an interval I into an α-balanced interval tree T storing n intervals at the moment. The algorithm that inserts I into T performs this task in two steps as follows:

<u>Step</u> 1: Perform a binary search in the primary structure of T in order to detect the position where the left end of I has to be stored. Let l denote the new leaf representing I's left end.

Ascend the primary structure from l to the root and adjust the particular ranks. For each node v whose balance moves outside the $[\alpha,1-\alpha]$ range perform one of the four restructing depicted in Figure 2-2. A single rotation requires the building of two new middle subtrees, a double rotation the building of three new middle subtrees.

Step 2: Descend T from the root until the first node v is found whose value lies inside of I. Insert the left and right end of I into the middle subtree of v.

Some care has to be taken for correctly adjusting the auxiliary information serving for quick searching. First, the leaves representing the left and right end of I have to be integrated into the two-way list comprising the totality of secondary leaves and second, some fingers associated with primary nodes and secondary roots have to be adjusted. Since only the fingers of ancestors of the two new secondary leaves may need a change, this task can be done by two walks from the new secondary leaves to the root of T.

What concerns our first problem, we distinguish two cases: If the middle subtree into which I was inserted had not been empty then the integration into the two-way list is no problem at all. On the other hand, assume that the middle subtree had been empty and now accomodates I only. The actual task is to detect the predecessor and successor of the leaves representing I's ends in the two-way list. This can be accomplished by ascending the tree from the two new leaves and exploiting the fingers assigned to the

primary nodes on the path or to sons of primary nodes on the path. The rather lengthy examination of all the different cases is left to the interested reader.

It can be readily seen that all actions except for the rebuilding of middle subtrees can be performed in O(logn) time and that these actions guarantee that the height of the tree never exceeds an upper bound of O(logn) which implies that at any time a query can be answered in O(logn+t) time, where t denotes the number of intersecting intervals found. The following lemmas are established to assure that the time required for rebuilding new middle subtrees is bounded by O(logN) on the average, where N is the maximal number of intervals in the tree while executing a sequence C of update commands.

Lemma 3: Assume that T_1 and T_2 denote the middle subtrees that have to be changed into T_3 and T_4, because a single rotation is applied to node v. If T_1 and T_2 contain m_1 and m_2 leaves respectively this task can be performed in $O(m_1+m_2)$ time.

Proof: W.l.g. let T_1 denote the middle subtree of node v (to which the single rotation is applied) and let T_2 denote the middle subtree of the particular son w of v. Then it is clear that T_3 must contain all those intervals stored in T_1 that do not contain the value associated with w. Similarly, T_4 must contain all intervals of T_2 and the remaining intervals of T_1.

By carefully maintaining the sorted order of the left and right ends of the intervals involved the task of constructing T and T can be done in $O(m_1+m_2)$ time.

[]

Lemma 4: The number of leaves in the middle subtree associated with the primary node v is upper bounded by the rank of v, i.e. the number of primary leaves descending from v.

Proof: Let a_r denote the ancestor closest to v whose left son is either v or another ancestor of v. Similarly, let a_l denote the ancestor closest to v whose right son is either v or another ancestor of v. Note that the number of primary leaves whose values lie between the ones of a_r and a_l is exactly equal to rank(v). Note also that an interval I may potentially be in v's middle subtree only if its left end is between the values assigned to a_r and a_l. Since the left ends of intervals correspond one-to-one with primary leaves, the number of intervals in v's middle subtree is upper bounded by rank(v).

[]

Now, we are able to establish an analogous assertion to Lemma 2 that relates the decrease in leaf depth due to a restructuring with the time required for building the new middle subtrees.

Lemma 5: The time required to carry out a single or double rotation including the construction of the new middle subtrees is proportional to the decrease in leaf depth caused by the rotation.

The proof is similar to the one of Lemma 2 and omitted.

Theorem 2: There exists an algorithm that executed a sequence C of insert and delete commands in an interval tree in O(|C|logN) time, where C consists of |C| update commands and where N is equal to the maximal number of intervals ever present in the tree, while processing C.

The proof follows immediately from Lemma 5 analogous the argumentation presented in the proof of Theorem 1.

It has to be mentioned that the method of constructing new middle subtrees by deriving them directly from the old ones is not improveable by Willard's super-B-tree algorithm. However, there exists a strategy that leads to worst-case bounds per update command: Let v be an unbalanced inner node that has to be rebalanced by a rotation. At most three new middle subtrees have to be constructed and this is done by succesively inserting the intervals involved. If Willard's super-B-tree algorithm is employed to supervise the rebalancings and the constructions of the new middle subtrees a worst-case update time of $O(\log^2 n)$ can be guaranteed.

We do not know if $O(\log^2 n)$ worst-case update time is the best one can do. Maybe a slight modification of the super-B-tree strategy will work even in O(logn) time. Or, perhaps, there exists a refinement of the method presented above that yields a O(logn) update time in the worst case.

The reason that we do not consider augmented interval trees (i.e. each inner node v is associated with an additional structure accomodating all intervals stored in secondary leaves descending from v) is simply that there is no need of augmented interval trees as will be seen in the next section. The main reason is that a query in an augmented interval tree implies $O(\log^2 n)$ separate queries in augmented structures. As will be shown later we can do better by employing the segment and the range tree instead.

3. SRI-Pyramids.

In 2- and higher-dimensional circumstances one often is not satisfied with 1-component trees like the 1-fold segment tree, the 1-fold range tree, or the interval tree. The d-dimensional range searching problem, for instance, may be well solved with the d-fold range tree which is a d-component tree in our terminology. (A d-component tree is a 1-component tree whose every node is associated with a (d-1)-component tree.) Another example is the 2-dimensional orthogonal line segment intersection searching problem recently investigated by Vaishnavi, Wood [23]. They developed the 2-component segment-range tree for this specific problem.

For convenience, we will abbreviate the three 1-component trees that will build the multi-component trees, for short called pyramids, by their initial letters. Thus, the 3-component segment-range-interval tree will be written as SRI-tree. Additionally, we agree upon using customary formal language notation for expressing the various tree names. E.g. the d-fold segment tree is thus designated by S^d-tree.

Our first aim is to explain by means of an example which objects may be stored in a Δ-tree and what kind of subjects can be used for posing queries, where Δ denotes a fixed word in $\{S,R,I\}^+$. For our purpose, Δ equal to SRI will do.

Remember that an S-tree stores intervals and can be used to detect all those intervals containing a certain value. An R-tree stores values and supports the detection of those values that lie in a certain interval. Finally, an I-tree stores intervals and helps to find all those intervals that intersect another query interval. Notice that the tasks supported by the S-tree and the R-tree are special cases of the one supported by the I-tree.

Thus, we derive that the exemplary SRI-tree requires objects consisting of two intervals and a value. Let the intervals stored in the S-tree be the z-intervals of the objects, let the values stored in the R-tree component be their y-values, and let the intervals stored in the I-tree component be their x-intervals. Hence, the SRI-tree may be used to accommodate 2-recs in 3-space whose edges are parallel to the x- and z-coordinate axes. Let M be a set of such objects. The SRI-tree accommodates M by storing the z-intervals of the objects in an S-tree, arranging each node list as R-tree storing the y-values of the objects originally in the node list, and associating with each inner node v of each R-tree an I-tree storing the x-intervals of objects whose y-values correspond with leaves descending from v.

A subject for posing a query in the SRI-tree of the above kind must consist of a z-value, a y-interval, and an x-interval. Thus, a query subject is nothing else than a 2-rec in 3-space whose edges are parallel to the x- and y-coordinate axes, respectively.

A query asks for all stored objects that intersect the subject. The desired objects can be found by descending the first-component S-tree with the z-value of the subject. For each visited node the

associated R-tree has to be searched with the y-intervals of the subject. Finally, the I-trees associated with appropriate nodes of the investigated R-trees have to be examined with the subject's x-interval.

It should be clear that the family of pyramids whose components consist of S-, R-, and I-trees (we will term it the SRI-pyramid family) is well suited to support the various intersection queries involving orthogonal objects and subjects.

The following lemmas will show that we need not the whole variety of $(S,R,I)^+$-trees but may confine to S^*R^*I-trees and S^*R^*-trees only.

Lemma 6: Let Δ be an arbitrary word in $(S,R,I)^*$. Assume that the letter S appears s times in Δ, R appears r times, and I appears i times, for non-negative integer s, r, and i.

We claim that the intersection searching problem supported by the Δ-tree can be solved as well by the $S^sR^rI^i$-tree.

Proof: Let e be an arbitrary object eligible for being stored in a Δ-tree. Then, e must have s+r+i properties, more precisely, e must consist of s+i intervals to be stored in the S- and I-tree components, and of r values to be stored in the R-tree components.

The word $S^sR^rI^i$ can be viewed as a permutation of Δ. In order to store e in an $S^sR^rI^i$-tree, the properties of e have to be permuted in the same way. Also the properties of the query subjects have to be permuted as above, however, the sequence of the properies to be tested for intersection does not mind at all.

[]

Note that the original Δ-tree accommodating n objects requires $O(n\log^{s+r+1}n)$ space if S is the final letter of Δ, and $O(n\log^{s+r+i-1}n)$ space otherwise. The $S^sR^rI^i$-tree requires $O(n\log^{s+r+1}n)$ space only if r+i is equal to 0. Hence, the new tree requires at most as much space as the old one - in a few cases even less.

Note also that the query time needed for searching in a Δ-tree runs up to $O(\log^{s+r+2i}n+t)$ if I is unlike the final letter of Δ. In the other case, i.e. if Δ ends with I, $O(\log^{s+r+2i-1}n+t)$ time suffices for answering a query. The $S^sR^rI^i$-tree allows for a query to be answered in $O(\log^{s+r+2i-1}n+t)$ time if i is at least 1. Consequently, the new tree saves query time in many cases - in the other cases it remains the same as for the Δ-tree.

<u>Lemma</u> 7: Let Δ be equal to $S^sR^rI^i$ where s, r, and i denote non-negative integers and i is no less than 1.

The intersection searching problem supported by the Δ-tree can be solved by a pair comprising the $S^{s+1}R^rI^{i-1}$-tree and the $S^sR^{r+1}I^{i-1}$-tree as well.

<u>Proof</u>: An I-tree is designed to support interval intersection searching. The same problem can be settled by a pair composed of an S-tree and an R-tree as follows:

Let M be the set of intervals. An element e_j of M is specified by its left end l_j and its right end r_j, for j running from 1 to n. Let $e_0=[l_0,r_0]$ designate the query interval. The intersection query may be answered by detecting all left ends of intervals in M that lie in e_0 and all intervals in M that contain the left end l_0 of e_0. The two tasks can be solved by establishing an S-tree for all intervals in M and an R-tree for all left ends of intervals in M. This method for solving interval intersection queries is described in more detail by Six, Wood [18, 19].

The above substitution may be carried out for multi-component structures as well. An I-tree component is replaced by a pair consisting of an S-tree and an R-tree component which yields a pair of pyramids as stated in the theorem.

[]

Of course, one can save nominal space by storing the identical s S-tree components only once and augmenting the s-th component S-tree twice, with an SR^rI^{i-1}-tree and an $R^{r+1}I^{i-1}$-tree.

Note that the transformation formulated in Lemma 7 implies, for no less than 2, only a nominal increase in space. On the other hand, the query time is improved by a factor O(logn) due to the replacement of an I-tree.

Since the query time is improved only if i is at least 2 (the same is true for the only nominal increase in space), the above transformation is advantageous only i-1 times. Hence, we will use tupels of SRI-pyramids whose names end with the only I rather than

SRI-pyramids involving more than one I-tree component.

Even the S^d-tree may be replaced by the S^{d-1}I-tree supporting d-dimensional inverse range searching with the same time-complexity for answering a query but with a factor O(logn) less space. However, the S^d-tree dominates the S^{d-1}I-tree in respect to the update time in worst-case complexity. Another disadvantage of I-trees will become obvious in this section when we discuss improvements for static SRI-pyramids.

In dynamic environments, one wants to execute an arbitrary sequence of update commands and queries. Therefore, the components of a dynamic SRI-pyramid are supplied by the dynamic relatives of the S- and I-tree as introduced in the previous sections, and by the dynamic relative of the R-tree, as developed by Willard [25, 26, 27].

Theorem 3: (a) Let T be a dynamic S^sR^rI-tree, with non-negative integers s and r. If T accommodates n eligible objects then it requires $O(n\log^{s+r}n)$ space, allows for an appropriate intersection query to be answered in $O(\log^{s+r+1}n+t)$ time, and supports updating in $O(\log^{s+r+1}N)$ time on the average, or $O(\log^{s+r+2}n)$ time in the worst case, where N denotes the maximal number of objects stored in T, while processing the sequence of update commands.

(b) Let U be a dynamic S^sR^r-tree representing n eligible objects, where s and r denote non-negative integers and s+r is at least 1. Then U requires $O(n\log^{s+r-1}n)$ space if r is

at least 1, and $O(n\log^s n)$ otherwise. An intersection query in U can be answered in $O(\log^{s+r}n+t)$ time, and a single update requires $O(\log^{s+r}N)$ time on the average, or $O(\log^{s+r}n)$ time in the worst case. The alternative update time depends on the procedure used which is much more difficult for the worst-case complexity.

Proof: When we subsequently talk about S-, R-, or I-trees, we will always mean the dynamic versions sufficiently described in Section 2.

Note that, for s no less than 1, T is nothing else than an S-tree whose every node v is augmented with an $S^{s-1}R^r$I-tree that represents v's node list. A query can be answered by descending the first-component S-tree and further investigating the $S^{s-1}R^r$I-trees of the nodes visited. Hence, we may conclude that the query time in T amounts O(logn) times the query time in an $S^{s-1}R^r$I-tree that represents n objects. From repeating the above step s-1 times we derive that the query time amounts $O(\log^s n)$ times the one required for answering a query in an R^rI-tree.

Similarly, we may calculate the latter query time being $O(\log^r n)$ times the one needed in an I-tree. Since an I-tree allows for the set of intersecting intervals to be located in O(logn) time, we finally obtain $O(\log^{s+r+1}n+t)$ as upper bound for a query in T.

Notice that the query time in U as well as in T for the case of s equal to 0, has already been derived in the above explanation.

Using similar inductive argumentations one can deduct the asserted bounds for space and update time, too.

[]

In the next section, some applications of the dynamic SRI-pyramids will be demonstated, such as rectangle intersection searching, inverse range searching, and a few other related problems. However, before applying the developed results we recall improvements of certain static SRI-pyramids due to a layer technique originally introduced by Willard [27], and later employed by Vaishnavi [21], and Vaishnavi, Wood [23].

Theorem 4: The static S^2-tree, R^2-tree, and SR-tree can be improved to allow for queries to be answered in

$O(\log n)$ time,

without affecting the asymptotic bounds for preprocessing and space.

The particular improved trees will be designated by $\overline{S^2}$-tree, $\overline{R^2}$-tree, and $\overline{SR}$-tree. It should be noted that Theorem 4 implies an improvement of the static Δ-tree if Δ is a word that ends with S^2, R^2, or SR.

We do not intend to give a proof of the above theorem. The interested reader is referred to Willard [27] for a discussion of the $\overline{R^2}$-tree, to Vaishnavi [21] where the $\overline{S^2}$-tree is considered, and to Vaishnavi, Wood [23] where the SR-tree is improved to the $\overline{SR}$-tree.

So far, we have considered intervals and values as properties of objects and have examined the cases that in each dimension either the objects or the subjects or even all of them have an interval as property. What remains is the case that in any dimension the properties of the objects and the subjects are single values. We call it the trivial case, since intersections only appear if there are objects with exactly the same value as the subject.

This case can be handled by employing a simple binary tree storing the values in its nodes (or maybe in its leaves only) and augmenting each node (or each leaf) with a structure representing those objects that share the value of the node (or the leaf).

The problem of searching objects with a specified value is customary called the exact match searching problem and there exist quite pleasant data structures supporting it.

Assume that the trivial case appears in k dimensions, then we are confronted with a k-dimensional exact match query and a (d-k)-dimensional non-trivial query. The so called D-trees introduced by Mehlhorn [15] allow us to locate those objects with the k desired values in $O(\log n+k)$ time in the worst case. An update can be carried out in $O(\log n+k)$ time.

Consequently, the family of DSRI-pyramids (i.e. multi-component trees composed of instances of the D-, S-, R-, and I-tree) is well suited to solve orthogonal intersection queries, even when trivial cases are involved.

As concluding remarks of this section we recall that the developed family of SRI-pyramids constitutes a powerful instrument to attack intersection searching problems involving arbitrary orthogonal objects in d-space. That these pyramids can be used in dynamic environments as well as in static ones is due to the dynamization results presented in Section 2. It has to be stated that the dynamizations are the best known today and that there is some indication of their asymptotic optimality, except for the worst-case update time of the I-tree.

4. Applications of the Above Results.

This section will demonstrate some applications of static and dynamic SRI-pyramids to problems that caused much attention in the last few years.

At first, we show how members of the SRI-pyramid family improve recent results concerning rectangle intersection searching, inverse range searching, and line segment intersection searching. Then, a few words will be said about all pairs problems (like the all intersecting rectangles problem considered by Bentley, Wood [4], Vaishnavi [20], Vaishnavi, Kiegel, Wood [22], Six, Wood [18, 17], and Edelsbrunner [6]) that are solved by employing the line sweeping technique and off-line dynamic relatives of SRI-pyramid structures.

4.1. Rectangle Intersection Searching.

The d-dimensional rectangle intersection searching problem involves a set M of d-recs that must be stored such that a query asking for all d-recs that intersect a query d-rec can be answered efficiently.

This problem has been investigated by Edelsbunner [6] who introduced the I^d-trees and by Six, Wood [18, 19] who implicitly developed the 2^d-tupel of possible $(S,R)^d$-trees. For instance, the 2-dimensional problem is solved by a 4-tupel comprising the S^2-tree, the SR-tree, the RS-tree, and the R^2-tree.

Subsequently, we will combine both methods and thus obtain new static and dynamic structures better than all preliminary ones.

Note that the d-dimensional rectangle intersection searching problem involves objects and subjects with d intervals. Consequently, this problem can be solved by establishing the I^d-tree for the objects.

We transform the I^d-tree utilizing Lemma 7 to a 2^{d-1}-tuple consisting of all possible $(S,R)^{d-1}$ I-trees. Next, each of the trees belonging to the 2^{d-1}-tuple consisting of s S-tree components and r R-tree components (for non-negative integers s and r whereby the sum s+r equals d-1) is transformed to the S^sR^rI-tree by appropriately permutating the particular properties of the objects, see Lemma 6.

Let us consider the outcoming structure for d equal to 3 more closely. With the above approach, this problem is solved by establishing four SRI-pyramids, namely

the S^2I-tree,
two instances of the SRI-tree, and
the R^2I-tree.

<u>Theorem</u> 5: The d-dimensional rectangle intersection searching problem involving n d-recs can be solved by a dynamic data structure with

$O(\log^d n+t)$ query time,
$O(\log^d N)$ update time on the average, and
$O(n\log^{d-1} n)$ space,

where N denotes the maximal number of d-recs stored in the structure, while processing the updates, and where t designates the number of d-recs intersecting the subject.

Note that the update time may be improved to worst-case complexity by either increasing the update time or the required space by a factor O(logn). (The latter result is obtained if the I-tree component is replaced by an S-tree and an R-tree component.)

It is trivial to derive the complexities from combining the S^sR^rI-trees approach with the results stated in Theorem 3, hence, we omit the proof.

In static environments, one can improve the query time by simultaneously deteriorating the space.

<u>Theorem</u> 6: The d-dimensional rectangle intersection searching problem can be solved by a static data structure with

$O(\log^{d-1} n+t)$ query time,
$O(n\log^d n)$ preprocessing, and
$O(n\log^d n)$ space,

where n and t denote the same quantities as in Theorem 5, and d is no less than 2.

<u>Proof:</u> The asserted complexities can be obtained by replacing the I-tree component by an S- and an R-tree component yielding S^sR^r-trees, with s+r equal to d. The last two components (either

S^2-, SR-, or R^2-trees) are then replaced by the static $\overline{S^2}$-, $\overline{SR}$-, and $\overline{R^2}$-trees, mentioned in Theorem 4. The time and space bounds are now obvious.

[]

It should be noted that the layered trees constitute the best static structures known today for 2- and higher-dimensional spaces. However, the I-tree combines optimal query time and space requirements for the 1-dimensional case.

4.2. Inverse Range Searching.

The d-dimensional inverse range searching problem asks for all d-recs of a set M that intersect a d-dimensional query point. This problem has recently been investigated by Vaishnavi [21] who called this searching problem the point enclosure problem, because a d-rec intersects a point if and only if it encloses the point.

Vaishnavi employed S^d-trees to solve the d-dimensional problem and he improved these trees by developing the $\overline{S^2}$-tree (see Theorem 4) and replacing S^d-trees by $S^{d-2}\overline{S^2}$-trees. We may contribute to static structures supporting the d-dimensional problem by mentioning that the S^{d-1}I-tree solves the problem with query time a factor O(logn) worse than the $S^{d-2}\overline{S^2}$-tree but with a factor O(logn) less space.

Vaishnavi was not able to design efficient dynamic data structures for this problem. We will remedy this situation by

employing either the dynamic S^{d-1}I-tree or the dynamic S^d-tree.

Theorem 7: The d-dimensional inverse range searching problem can be solved by a dynamic structure with

$O(\log^d n+t)$ query time,

$O(\log^d N)$ update time on the average, and

$O(n\log^{d-1} n)$ space,

or by another structure with

$O(\log^d n+t)$ query time,

$O(\log^d n)$ worst-case update time, and

$O(n\log^d n)$ space,

where n, N, and t denote the quantities as used above.

We will dispense with a proof of this theorem, since all bounds are obviously derived from the previous discussion of SRI-pyramids.

4.3. Line Segments Intersection Searching.

From the various other problems supported by structures of the SRI-pyramid family, we briefly mention the 2-dimensional line segment intersection searching problem, recently considered by Vaishnavi, Wood [23].

This problem involves a set of line segments parallel to, e.g., the x-coordinate axis. A query asks for all horizontal line segments intersecting a vertical one. Vaishnavi, Wood developed the $\overline{SR}$-tree to solve the static problem and, thereafter, considered some dynamizations. However, the dynamic SR-tree developed in the previous sections is better than their dynamic structures by solving the problem with

$O(\log^2 n+t)$ query time,

$O(\log^2 n)$ update time in the worst case, and

$O(n\log n)$ space.

4.4. All Pairs Problems.

All intersecting pairs problems ask for all pairs of intersecting objects of a set M. For a great deal of such problems the pairs can efficiently be detected by utilizing the line (resp. (d-1)-dimensional hyperplane) sweeping technique. It has become customary to associate appropriate off-line dynamic structures with the hyperplane.

An off-line dynamic structure is built up after examining the objects that will be inserted into it in future. Such an off-line dynamic structure initially is a skeletal structure representing no object at all. While sweeping the hyperplane from left to right, certain objects are inserted into or deleted from the structure without changing the skeletal. The skeletal supports a relatively simple mechanism for inserting or deleting objects. The updates need not be accompanied by restructrings, since the skeletal itself maintains the balance of the structure that assures the fast query time.

The reason for utilizing off-line dynamic structures was the

lack of comparably efficient on-line dynamic structures. The main disadvantage of off-line dynamic structures is the requirement of nearly as much time and space in situations when they represent only a few objects as in situations when they store almost the whole set of objects.

The on-line dynamic structures developed in the previous sections combine both advantages, they achieve the same efficiency as the off-line structures in respect to query and update time when they represent almost the whole set of objects, and they do not require more time and space as necessary in situations when they have to handle only a few objects.

We will single out the d-dimensional all intersecting rectangles problem which asks for all intersecting pairs of a given set of n d-recs.

Our approach is similar to the ones undertaken by Six, Wood [18, 19] and Edelsbrunner [6]. The d-dimensional space is swept by a (d-1)-dimensional hyperplane L normal to the x-coordinate axis from left to right. Associated with L is the 2^{d-2}-tuple of dynamic $S^s R^r I$-trees, where s+r is equal to d-2, see Section 3.

The problem is solved by sweeping L from left to right and performing some actions at each moment L passes an x-value of a d-rec. Initially, the associated structure T is empty. When L passes the left end of an object e, a query is performed to detect all objects stored in T that intersect e, and the projection of e onto L is inserted into T. If L passes the right end of an object e, the projection of e is deleted from T.

This algorithm solves the problem for n d-recs in $O(n\log^{d-1} n+t)$ time and $O(n\log^{d-2} n)$ space, wherby t denotes the number of intersecting pairs. Note that this solution (which was suggested by D. Wood) dominates the preliminary ones even in unfavourable distributions of the d-recs.

Certain distributions of the objects may imply that at any moment only a few objects are stored in T. For instance, if the number of objects stored in T never exeeds O(logn) the n queries, n insertions, and n deletions can be performed in $O(n(\log\log n)^{d-1})$ time and T requires no more than $O(\log n(\log\log n)^{d-2})$ space.

The replacement of off-line dynamic structures by on-line ones may lead to similar improvements in a number of related problems investigated by Vitanyi, Wood [24], van Leeuwen, Wood [9], Lipski, Preparata [11] and others.

References.

[1] Bentley,J.L. Multidimensional Divide-and-Conquer. Communications of the ACM 23 (1980), 214-229.

[2] Bentley,J.L., Maurer,H.A. Efficient Worst-Case Data Structures for Range Searching. Acta Informatica 13 (1980), 155-168.

[3] Bentley,J.L., Saxe,J.B. Decomposable Searching Problems I: Static-to-Dynamic Transformations. To appear in the Journal of Algorithms.

[4] Bentley,J.L., Wood,D. An Optimal Worst-Case Algorithm for Reporting Intersections on Rectangles. To appear in IEEE Transactions on Computers.

[5] Edelsbrunner,H. Optimizing the Dynamization of Decomposable Searching Problems. TU Graz, Institut fuer Informationsverarbeitung (1979), Report 35.

[6] Edelsbrunner,H. A New Approach to Rectangle Intersections. Submitted for publication.

[7] Edelsbrunner,H., Maurer,H.A. On Planar Point Location. TU Graz, Institut fuer Informationsverarbeitung (1980), Report 52.

[8] Leeuwen,J.v., Maurer,H.A. Dynamic Systems of Static

Data-Structures. TU Graz, Institut fuer Informationsverarbeitung (1980), Report 42.

[9] Leeuwen,J.v., Wood,D. The Measure Problem for Rectangular Ranges in d-Space. University of Utrecht, Department of Computer Science (1979), Report RUU-CS-79-6.

[10] Leeuwen,J.v., Wood,D. Dynamization of Decomposable Searching Problems. Information Processing Letters 10 (1980), 51-56.

[11] Lipski,W.Jun., Preparata,F.P. Finding the Contour of a Union of Iso-Oriented Rectangles. University of Illinois at Urbana-Champaign, Coordinated Science Laboratory (1979), Report R-853.

[12] Lueker,G. A Data Structure for Orthogonal Range Queries. Proceedings of the 19th FOCS Symposium (1978), 28-34.

[13] Lueker,G. A Transformation for Adding Range Restriction Capability to Dynamic Data Structures for Decomposable Searching Problems. University of California, Irvine Department of Information and Computer Science (1979), Report 129.

[14] Maurer,H.A., Ottmann,Th. Dynamic Solutions of Decomposable Searching Problems. In: Pape,U. (Ed.): Discrete Structures and Algorithms, Carl Hanser (1979), 19-24.

[15] Mehlhorn,K. Dynamic Binary Search. SIAM Journal on Computing 8 (1979), 175-198.

[16] Nievergelt,J., Reingold,E.M. Binary Search Trees of Bounded Balance. SIAM Journal on Computing 2 (1973), 33-43.

[17] Overmars,M.H., Leeuwen,J.v. Two General Methods for Dynamizing Decomposable Searching Problems. University of Utrecht, Department of Computer Science (1979), Report RUU-CS-79-10.

[18] Six,H.-W., Wood,D. The Rectangle Intersection Problem Revisited. McMaster University, Unit for Computer Science (1979), Report 79-CS-24.

[19] Six,H.-W., Wood,D. Counting and Reporting Intersections of d-Ranges. McMaster University, Unit for Computer Science (1980), Report 80-CS-6.

[20] Vaishnavi,V.K. Optimal Worst-Case Algortihms for Rectangle Intersection and Batched Range Seaching Problems. McMaster University, Unit for Computer Science (1979), Report 79-CS-12.

[21] Vaishnavi,V.K. Computing Point Enclosures. Submitted for publication.

[22] Vaishnavi,V.K., Kriegel,H.P., Wood,D. Space and Time Optimal Algorithms for a Class of Rectangle Intersection

Problems. To appear in Information Sciences.

[23] Vaishnavi,V.K., Wood,D. Rectilinear Line Segment Intersection, Layered Segment Trees and Dynamization. McMaster University, Unit for Computer Science (1980), Report 80-CS-8.

[24] Vitanyi,P.M.B., Wood,D. Computing the Perimeter of a Set o Rectangles. McMaster University, Unit for Computer Science (1979), Report 79-CS-23.

[25] Willard,D.E. Predicate Oriented Database Search Algorithms. Harvard Universtiy, Aiken Computer Laboratory (1978), Report TR-20-78.

[26] Willard,D.E. An Introduction to Super-B-Trees. University of Iowa, Department of Computer Science (1979).

[27] Willard,D.E. New Data Structures for Orthogonal Queries. To appear in Communications of the ACM.

[28] Willard,D.E. K-d Trees in Dynamic Environment. To appear in Communications of the ACM.