SIAM J. COMPUT.
Vol. 14, No. 2, May 1985

PRIORITY SEARCH TREES*

EDWARD M. McCREIGHT†

Abstract. Let D be a dynamic set of ordered pairs $[x, y]$ over the set $0, 1, \cdots, k-1$ of integers. Consider the following operations applied to D:

(1) Insert (delete) a pair $[x, y]$ into (from) D.

(2) Given test integers $x0$, $x1$, and $y1$, among all pairs $[x, y]$ in D such that $x0 \leqq x \leqq x1$ and $y \leqq y1$, find a pair whose x is minimal (or maximal).

(3) Given test integers $x0$ and $x1$, among all pairs $[x, y]$ in D such that $x0 \leqq x \leqq x1$, find a pair whose y is minimal.

(4) Given test integers $x0$, $x1$, and $y1$, enumerate those pairs $[x, y]$ in D such that $x0 \leqq x \leqq x1$ and $y \leqq y1$.

Using a new data structure that we call a priority search tree, of which two variants are introduced, operations (1), (2), and (3) can be implemented in $O(\log n)$ time, where n is the cardinality of D. Operation (4) is performed in at most $O(\log n + s)$ time, where s is the number of pairs enumerated. The priority search tree occupies $O(n)$ space.

Priority seach tree algorithms can be used effectively as subroutines in diverse applications. With them one can answer questions of intersection or containment in a dynamic set of linear intervals. They can be used in combination with a well-known plane-sweep technique, to implement off-line algorithms for enumerating all intersecting pairs of rectangles. Priority search trees can also be used to implement best-/first-fit storage allocation.

Key words. computational geometry, search trees, priority queues, intersection, intervals, rectangles, storage allocation, concrete complexity

CR categories. 5.25, 3.74, 5.39

1. Introduction.

Efficient multi-dimensional searching is one of the persistent puzzles of computer science. Many lovely one-dimensional search structures with linear space requirements and guaranteed logarithmic-time maintenance and search algorithms have been discovered. But multi-dimensional structures with similar attractive properties continue to elude discovery.

We present here a new data structure, called a priority search tree, for representing a dynamic set D of ordered pairs $[x, y]$ over the set $0, 1, \cdots, k-1$ of integers, and a set of algorithms that operate on the priority search tree to implement the following operations:

InsertPair (x, y): Insert a pair $[x, y]$ into D.

DeletePair (x, y): Delete a pair $[x, y]$ from D.

MinXInRectangle (x0, x1, y1): Given test integers $x0$, $x1$, and $y1$, among all pairs $[x, y]$ in D such that $x0 \leqq x \leqq x1$ and $y \leqq y1$, find a pair whose x is minimal.

MaxXInRectangle (x0, x1, y1): Given test integers $x0$, $x1$, and $y1$, among all pairs $[x, y]$ in D such that $x0 \leqq x \leqq x1$ and $y \leqq y1$, find a pair whose x is maximal.

MinYInXRange (x0, x1): Given test integers $x0$ and $x1$, among all pairs $[x, y]$ in D such that $x0 \leqq x \leqq x1$, find a pair whose y is minimal.

EnumerateRectangle (x0, x1, y1): Given test integers $x0$, $x1$, and $y1$, enumerate those pairs $[x, y]$ in D such that $x0 \leqq x \leqq x1$ and $y \leqq y1$.

This searching might fairly be described as 1.5-dimensional. The data has two independent dimensions, but the priority search tree does not allow equally powerful searching operations on both. There is a major dimension (x) permitting arbitrary range queries, and a minor one (y) permitting only enumeration in increasing order.

* Received by the editors July 14, 1980, and in final revised form April 26, 1983.

† Xerox Corporation, Palo Alto Research Center, Palo Alto, California 94304.

All the search rectangles have only three sides free; the fourth side is anchored at $y0 = 0$.

In § 2 we present the simple radix priority search tree, and examine some of its properties. In § 3 we elaborate this to the balanced priority search tree. In § 4 we discuss a few of the applications to which these priority search trees can be put.

2. Radix priority search trees. First off, let us simplify the problem somewhat. In the following exposition we assume that the set D of pairs contains no duplicate x values. A restriction like this might or might not occur naturally in a real application. If not, we can work with a derived set D_π consisting of a pair $[F(x, y), y]$ for every pair $[x, y]$ in D. The function F is an invertible encoding function that maps pairs of integers into single integers with the property that differences in x are more significant than differences in y. For example, we might use the function $F(x, y) = j^*x + y$, which maps pairs of integers in the domain $0\,..\,j-1$ to single integers in the range $0\,..\,j^2 - 1 \leqq k - 1$. Other ways of implementing such a function F are left to the reader's imagination. If $[x, y]$ pairs are unduplicated in the original problem, then x_π values are unduplicated in the derived problem. (Going even further, we could accommodate duplicated $[x, y]$ pairs in the original problem by representing them as unduplicated pairs with associated counts.)

The simple idea that underlies priority search trees is most easily seen from a diagram. Suppose that you wanted to represent the set of pairs in Fig. 1 so that **EnumerateRectangle** could be executed efficiently on this representation. One good way to do this is to select the pair $[x^*, y^*]$ with minimum y, write it at the root of a binary data structure, divide the region in two with a line of constant x, and recursively represent the remaining points in the two subregions in the two subtrees of the root in the same manner.

If one divides the region along the line $x = x^*$, then the resulting data structure is the Cartesian tree of Vuillemin [13]. This structure allows very good performance in the average case, but its performance in the worst case is no better than a linear

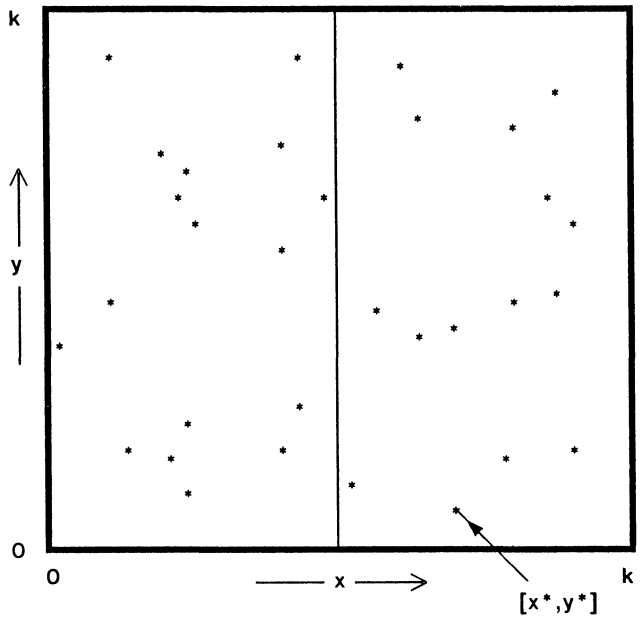

FIG. 1

list. For example, the set of pairs with $x = y$ forces the data structure to degenerate to a linear list.

Fortunately there is no compelling reason to divide the region along the line $x = x^*$. We define a radix priority search tree such that at each recursive level the previous level's x-interval is cut exactly in half (geographically). This is called a radix bisection, and it has two important consequences.

The first is that after at most lg k levels of bisection in x, we encounter a stripe in y that is one unit wide in x. By our nonduplication assumption, there can be at most one pair in D within this stripe. Therefore, even though a radix priority search tree might contain k pairs, it is at most lg k levels deep.

The other important consequence of radix bisection is that a node in a given position in the radix search tree represents a fixed rectangle in x-y space. The only way a pair enters or leaves one of these fixed rectangles is through an updating operation involving that pair. Therefore no lateral data motion is ever required during updating operations; an updating operation only moves down a single spine in the tree. This fact enables a lg k time bound on updating operations.

2.1. Data structure. We can represent a radix priority search tree in Pascal as follows:

```
CONST
  k = 30000;
    (*Comfortable for a 16-bit 2's complement machine*)
  FirstKey = 0;
  LastKey = k - 1;
  FirstNonKey = LastKey + 1;
TYPE
  KeyRange = FirstKey .. LastKey;
  KeyBound = FirstKey .. FirstNonKey;
  Pair = RECORD x, y: KeyRange END;
  RPSTPtr = ↑RPST;
  RPST=RECORD
    p: Pair;
    left, right: RPSTPtr
    END;
```

A radix priority search tree is characterized by a fidelity condition and two data structural invariants. The fidelity condition asserts that if the tree is representing a set D of pairs, then each pair of D will appear in the p field of exactly one node (or RPST record) of the tree. Thus a tree representing a set of n pairs occupies $O(n)$ words of storage, where each word is $O(\log k)$ bits long. In conventional algorithm-analytic terms, a radix priority search tree is a linear-space data structure.

The first invariant is a priority queue invariant on y-values. It asserts that for any node **t** in a radix priority search tree, if **t.left** is not **NIL** then **t.p.y** $\leqq$ **t.left↑.p.y**, and if **t.right** is not **NIL** then **t.p.y** $\leqq$ **t.right↑.p.y**. This first invariant constrains only direct ancestor-descendant relations; it does not constrain sibling relations at all.

The second invariant is a radix search tree invariant on x-values. It asserts that associated with each node **t** in a radix priority search tree is an x-interval [**lower .. upper**) (this notation denotes all integers between **lower** and **upper**, including **lower** and excluding **upper**) within which **t.p.x** lies. The x-interval associated with the root of the radix search tree is the interval **KeyBound**. For any node **t** such that **t.left** is not

NIL, the x-interval associated with the node **t.left↑ is [lower . . floor((lower+upper)/2))**. For any node **t** such that **t.right** is not **NIL**, the x-interval associated with the node **t.right↑** is **[floor((lower+upper)/2) . . upper)**.

2.2. Algorithms. The complete radix search tree algorithms, represented in Pascal, are presented in Appendix A, and the reader is encouraged to read the following in parallel with Appendix A.

In all of these algorithms, two preconditions are assumed true, and their truth is maintained in recursive calls. The first precondition is that the interval **[lowerX . . upperX)** is nonempty; that is, that **lowerX < upperX**. To maintain the truth of this first precondition in recursive calls, the algorithms depend upon our assumption that no two pairs have the same x-value. The second precondition is that whenever a procedure takes an **[x0 . . x1]** argument range as a parameter, the procedure is only called if the interval **[x0 . . x1]** shares at least one integer in common with the interval **[lowerX . . upperX)**.

First consider the **InsertPair** procedure. To insert a new pair, we begin at the root of the priority search tree. First we discover whether the new pair "beats" the pair already sitting at the root, in the sense that its y-value is smaller. If not, then the new pair is inserted recursively into either the left or right subtree, determined by its x-value. Otherwise, the new pair belongs at the root, so the pair that originally lay at the root is saved, the new pair is put there instead, and the saved pair is inserted recursively into the subtree determined by its x-value.

DeletePair operates in two distinct phases. The first phase locates the pair to be deleted and it operates as a search in an ordered search tree. Once the pair to be deleted has been located its deletion leaves a hole, which is filled by a priority queue tree selection (or a "knock-out tournament") phase [1], in which a pair of brothers compete for the vacated spot formerly occupied by their father, and then the sons of the victor compete for his former spot, and so on. The second phase completes when the vacant spot has fewer than two sons.

Consider **MinXInRectangle** applied to a subtree rooted at node **t**. If **t.p.y** lies above the top of the constraint rectangle, then because a priority search tree is a priority queue in y, no pair in the subtree rooted at **t** lies within the constraint rectangle. Otherwise, the solution might be found in the left subtree. If no pair in the left subtree lies within the constraint rectangle, then (and only then) the solution might be found in the right subtree. This is because every constrained pair in the left subtree is better than any constrained pair in the right subtree. Finally, if **t.p** lies within the constraint rectangle, then it might or might not be the correct solution, depending on whether it is better than the best constrained pair found in a subtree. **MaxXInRectangle** is entirely symmetric in x.

Next consider **MinYInXRange** applied to a subtree rooted at node **t**. If **t.p** lies within the constraint x-interval, then because a priority search tree is a priority queue in y, **t.p** is the correct solution. Otherwise, the solution, if it exists, is the better of the solutions of the two subtrees. The tests of **middleX** against **x0** and **x1** simply guarantee that subtrees that are certain to be fruitless are not explored. These tests also maintain the truth of the second precondition in recursive calls.

Finally, **EnumerateRectangle** is a depth-first enumeration that calls the function **Report** whenever a pair is found within the constraint rectangle. **Report** returns TRUE if the enumeration should continue, and FALSE if it should terminate.

Each of these procedures is called at the top level with **t↑** being the root of a radix priority search tree, and with **lower** being **FirstKey** and **upper** being **FirstNonKey**.

2.3. Execution time analysis. The logarithmic time bounds can be seen from the recursive structure of the algorithms. Each recursive level of **InsertPair** is called on a node with a **[lower .. upper)** interval, and makes at most one recursive call on **InsertPair**, handing it a son node with a **[lower .. upper)** interval at most half as large. The recursion must stop before the size of this **[lower .. upper)** interval shrinks to zero. This implies a bound of $\lg k$ on the depth of recursion and the number of nodes visited, and the same order found on running time. An identical analysis applies to **DeletePair**.

The analyses of **MinXInRectangle** (and, symmetrically, **MaxXInRectangle**), **MinYInXRange**, and **EnumerateRectangle** are more complicated because each sometimes calls itself recursively on both sons of a tree node. How many nodes can these procedures visit? We begin to answer this question by classifying tree nodes according to how their **[lower .. upper)** intervals, denoted by $\langle\rangle$, compared with the interval **[x0 .. x1]**, denoted by $\{\}$. There are six such classes:

$$1\colon \langle\rangle\{\},\quad 2\colon \{\}\langle\rangle,\quad 3\colon \langle\{\}\rangle,\quad 4\colon \langle\{\}\rangle,\quad 5\colon \{\langle\}\rangle,\quad \text{and}\quad 6\colon \{\langle\rangle\}.$$

Neither **MinXInRectangle** nor **MinYInXRange** ever visits nodes in classes 1 or 2; this is prevented by the second precondition. The second data structure invariant guarantees at most $\lg k$ nodes in each of classes 3, 4, and 5, so every one of these nodes could be visited without violating a logarithmic time bound. Finally, there can be a very large number of nodes in class 6, but these nodes can be grouped into maximal subtrees that are sons of nodes in classes 4 or 5, at most one such son each, so there are at most $2 \lg k$ of these maximal subtrees. Within each of these class-6 subtrees, any **t.p.x** lies within the interval **[x0 .. x1]**, so **MinYInXRange** will be prevented by its second **IF** statement from exploring beyond the roots of these maximal subtrees. This shows a time bound for **MinYInXRange** that is logarithmic in k.

An identical argument applies to **EnumerateRectangle** on all node classes except 6. In each of the class-6 maximal subtrees, beyond that subtree's root level **EnumerateRectangle** can visit a node only if the pair in the node's father was **Report**'ed. It follows that if **EnumerateRectangle** in fact enumerates s pairs, it runs in a time bound of $\lg k + s$. This is true whether or not the enumeration is terminated by the **Report** function.

The operation of **MinXInRectangle** is more subtle, because **MinXInRectangle** might find its answer deep in a subtree of class-6 nodes. The key observation is that once a single recursive instance of **MinXInRectangle** succeeds (in the sense that it returns a **valid CondPair**), there will be no further recursive calls of **MinXInRectangle**, and all currently active recursive invocations (of which there can be at most $\lg k$, the length of the longest path in the tree) will succeed. In other words, a top-level call to **MinXInRectangle** will generate some number of recursive invocations applied to various nodes of the tree that will fail, plus at most $\lg k$ invocations applied to other nodes that will succeed. Now how many recursive invocations might fail? We observe that whenever **MinXInRectangle** will fail when applied to a class-6 node, then it will fail on its second **IF** statement, in constant time and without making any further recursive calls. Thus a loose count concludes that **MinXInRectangle** can encounter at most $3 \lg k$ nodes of classes 3, 4, and 5, and at most $\lg k$ successful nodes, and therefore at most $4 \lg k$ other (failed) class-6 nodes. This confirms a time bound for **MinYInXRange** that is logarithmic in k. Closer reasoning tightens the constants considerably.

I have tried to code the procedures in Appendix A for clarity. There are several straightforward program transformations that would improve execution time by significant constant factors. The most important of these replace recursion with iteration and division by 2 with a binary shift.

Another important optimization reduces the number of unary (nonbinary) nodes within the tree in many applications. One way of thinking about the algorithms in this section is that a search for x is steered left or right through the tree by the sequence of bits in the binary representation of x, one bit per left/right decision. The definition of **RPST** can be augmented with a bit count field to allow a single left/right decision to consume several bits of the binary representation of x, thereby eliminating unary nodes for the intermediate bits. Depending on the application, this optimization can result in large reductions of average path length, with attendant improvements in speed.

3. Balanced priority search trees. Careful consideration of the literature on search structures suggests that when a radix structure permits certain opeations to be performed in certain asymptotic time bounds, there almost always exists a parallel balanced comparative structure (that is, a structure within which order may be inferred only by comparing with keys that are present in the structure) that permits the same operations to be performed in the same asymptotic time bounds. It would be a surprise and a disappointment if this observation did not also hold true for priority search trees.

Fortunately, it does hold true. The structure of a balanced priority search tree node can be expressed in Pascal as follows:

```
TYPE
  BPSTPtr = ↑BPST;
  BPST=RECORD
    p, q: Pair;
    p, q: Pair;
    validP, duplQ: BOOLEAN;
    left, right: BPSTPtr;
    balance: BalanceInfo (*appropriate to the
      underlying tree form chosen*)
    END;
```

A balanced priority search tree is characterized by a fidelity condition and four data structural invariants. The fidelity condition asserts that if the tree is representing a set D of pairs, then each pair of D will appear in the **q** field of exactly one node of the tree. Thus a balanced priority search tree is also a linear-space data structure.

In a **BPST** node, unlike a **RPST** node, two pairs can be recorded: the pair **q** is chosen for its near-median x-value, while the pair **p** is chosen for its minimal y-value. Two pairs are necessary because, as we saw in § 2, for some sets of pairs it is impossible to satisfy both criteria with the same pair. Any pair $[x, y]$ in D appears as the **q** field of exactly one node **t**, and may also appear as the **p** field of at most one ancestor node of **t**.

The first structural invariant is a standard search tree invariant on **q.x**. It asserts that with each node **t** in a balanced priority search tree is associated a search key interval **[x0 . . x1)** containing **t.q.x**, and also containing **t.p.x** if **t.validP** is true. The x-interval associated with the root of the search tree is **KeyBound**. For any node **t**, if **t.left** is not **NIL**, then the x-interval associated with the node **t.left**↑ is **[x0 . . t.q.x)**. Similarly, if **t.right** is not **NIL**, then the x-interval associated with the node **t.right**↑ is **[t.q.x . . x1)**.

The second structural invariant is a priority queue invariant on **p.y**. Let **t** be any node in a balanced priority search tree, and let **a** be a proper ancestor of **t**, and **d** a proper descendant of **t**. If **{a.p}**$\supseteq${**d.q**} then **t.validP** is **FALSE**. Otherwise **t.validP** is **TRUE**, and **t.p** is a pair chosen from **{d.q}-{a.p}** so that **t.p.y** is minimal. In other words,

t.p. is a pair with minimal y that appears as the **q** field of one of **t**'s descendants and does not appear as the **p** field of any of **t**'s ancestors. If no such pair exists, **t.validP** is **FALSE**. It is easy to show that if **validP** is **FALSE** at **t**, it is **FALSE** at all of **t**'s descendants as well. Conversely, if **validP** is **TRUE** at **t**, it is **TRUE** at all of **t**'s ancestors as well.

The third structural invariant specifies the **duplQ** field. It asserts that the field **t.duplQ** is **TRUE** if and only if there is some proper ancestor node **a** of **t** such that **a.validP** = **TRUE** and **a.p** = **t.q**. This field allows our algorithms easily to avoid duplicate enumeration of pairs.

The fourth and final structural invariant is a balance invariant inherited from whatever underlying form of balanced tree is chosen. This invariant is usually a relation between weights or path lengths in the left and right subtrees of a node, or on the sequence of "colors" on arcs leading to the node.

The operation necessary for maintaining balance in all known forms of balanced comparative tree is some form of "rotation." [1, p. 454] This is a way of moving some "weight" from the "heavier" subtree of a node to the "lighter" one, thereby preserving the balance invariant that guarantees a maximal path length at most logarithmic in the number of nodes. The **BalanceInfo** field in the type of definition allows determination of when and where to do these rotations.

Figure 2 shows the standard picture of a single rotation. Lower-case letters indicate points along the x-value line, and also tree nodes whose **q.x** fields contain those points. Upper-case letters indicate intervals on this line, and also subtrees containing nodes whose **p.x** and **q.x** fields lie within those intervals. All intervals are assumed to include their lower endpoint, and exclude their upper one.

During the priority search tree rotation, the **q** fields remain unchanged, just as they would in an ordinary balanced tree rotation. The interesting question is, what happens to the **p** fields? First of all, it is clear that node **c** can use node **e**'s original **p** field, because it represents the "best" pair in the **[a . . g)** interval that is not represented higher in the tree. Now, what happens to node **c**'s original **p** field, and where does the

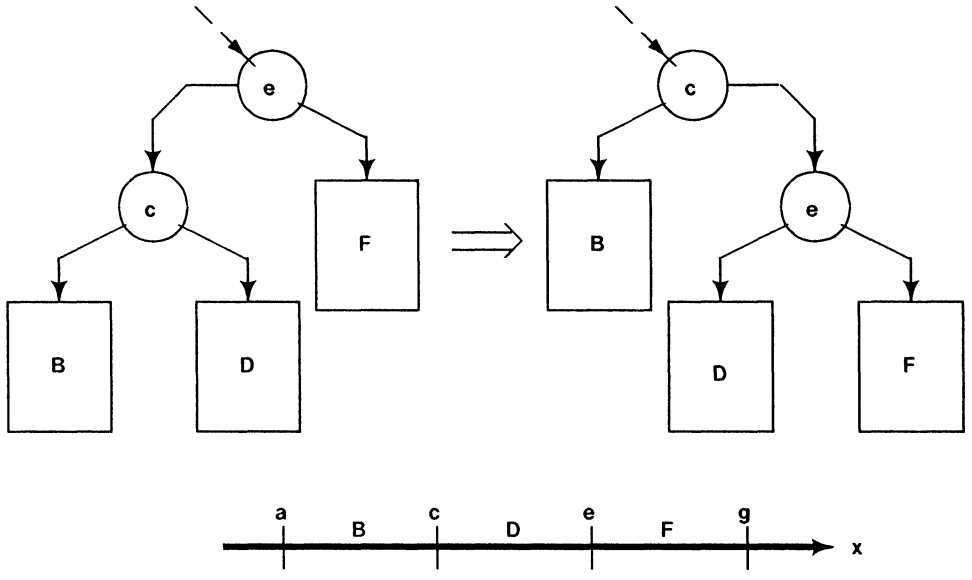

FIG. 2

new **p** field for node **e** come from? As luck might have it, **c**'s original **p** field might lie in the interval **D**, and if so it would be a candidate for **e**'s new **p** field. But in the general case we need to dispose of **c**'s old **p** field in one of **c**'s old subtrees, and to extract a new **p** field for node **e** from one of its new subtrees.

Appendix B contains Pascal procedures for doing just that, along with skeleton procedures for the balanced forms of **InsertPair** and **DeletePair** that manipulate the **p**, **q**, **validP** and **duplQ** fields correctly but ignore re-balancing. The balanced forms of the other algorithms are left as exercises for the interested reader.

The **DisposeP** and **ExtractP** procedures are each recursive down at most one path in the tree, so their execution time is at most linear in the length of the longest path in the tree. In a balanced tree, this longest path length is at most logarithmic in the number of nodes in the tree. This means that in a balanced priority search tree **BalancedInsertPair** and **BalancedDeletePair** each run in logarithmic time, except for rotations, and that at most logarithmic extra time is needed for a single rotation.

To attain an overall logarithmic time bound, the number of rotations per updating operation must be bounded by a constant. The usual families of balanced tree, such as AVL [1, pp. 451–458], weight-balanced [1, p. 468], and *B*-tree [1, pp. 471–479], do not guarantee such a bound. Fortunately there are at least two families that do guarantee at most a constant (in fact, three) rotations per updating operation: 2-3-4 trees [14], [15], [16], and the half-balanced binary trees of Olivie [2].

4. Applications. There are many real computer applications involving a large number of data items where the size of the computer to be used mandates a linear-space data structure. All of the applications below require space only linear in the number of data items. In several cases algorithms are known with smaller asymptotic time bounds (generally by a factor of $\log n$) but larger space requirements (generally by the same factor) [10].

In all of the two-dimensional applications below, line segments are taken to be parallel to the x- or y-axis. In my own applications this restriction is not a serious one. But for others it may be, and it is not known in general how essential this restriction is for the existence of fast algorithms.

4.1. On-line intersections in a dynamic set of linear intervals. We can use a priority search tree to represent a dynamic set of linear intervals, letting the x-value of the pair represent the upper endpoint of an interval, and letting its y-value represent the lower endpoint. To enumerate all intervals that share at least one point with a test interval **[u . . v]**, we use the **EnumerateRectangle** algorithm to enumerate all pairs whose x-value lies in the interval **[u . . LastKey]** and whose y-value lies in the interval **[FirstKey . . v]**. If the set contains n intervals, then the structure to represent it requires $O(n)$ space, a new interval can be added or an old one deleted in $O(\log n)$ time, and all s intervals in the set that intersect a test interval can be enumerated in $O(\log n + s)$ time. Results almost as good as this have been discovered previously by McCreight [3] and Guibas and Saxe [4], and independently by Edelsbrunner [5], [6]. The improvement over [3] and [5] is that here the parameter k can be ignored because balanced priority search trees are used. The improvement over [4] and [6] is that here the time bound applies to each operation individually, rather than to an average taken over a sequence of operations.

4.2. On-line containments in a dynamic set of linear intervals. With exactly the same data structure we can enumerate all intervals that completely contain a test interval **[u . . v]** by using the **EnumerateRectangle** algorithm to enumerate all pairs whose

x-value lies in the interval **[v . . LastKey]** and whose y-value lies in the interval **[FirstKey . . u]**. If the set contains n intervals, then all s intervals in the set that contain a test interval can be enumerated in $O(\log n + s)$ time. This is thought to be a new insult.

4.3. On-line visibility in a dynamic set of semi-infinite line segments. Suppose one has a dynamic set of semi-infinite line segments beginning at points $[x, y]$ and extending upward in y. From a given point $[x', y']$, which of these line segments would be visible along a line of increasing x? To solve this problem one can represent the endpoints of the semi-infinite lines as $[x, y]$ pairs in a priority search tree. One could either think of the line segments as being translucent or opaque. In the former case, the solution is all pairs in the rectangle bounded on the left by $x = x'$ and above by $y = y'$, which can be enumerated by **EnumerateRectangle** in $O(\log n + s)$ time. In the latter case, the solution is the single pair within that rectangle whose x is minimal, which can be produced by **MinXInRectangle** in $O(\log n)$ time.

4.4. On-line visibility in a dynamic set of line segments. Relaxing the restriction in the previous problem that the line segments be semi-infinite gives the problem an additional degree of freedom. To deal with this extra degree of freedom we adapt a previous technique [3], [5] that recursively bisects the y-space, dividing the line segments at each level into three classes: segments that lie entirely above the bisector, segments that lie entirely below it, and segments that are cut by it. Segments that are not cut by the bisector are represented in deeper recursive levels. Segments that are cut by the bisector are represented in two priority search trees: one representing the pieces of the cut segments below the bisector, and one representing those above. In each of these two priority search trees the line segments are now semi-infinite, because they extend to the limit of the (reduced) y space. Therefore the solution from § 4.3 carries over for each recursive level, and there are $\log k$ such levels, so the translucent (opaque) problem can be solved in $O(\log n \log k(+s))$ time. This is also thought to be a new result.

4.5. On-line point containment in a dynamic set of rectangles. We apply recursive bisection one more time, this time in x. For those rectangles cut by the bisector, we now consider their left and right bisected pieces. These are symmetric, so we describe only how to deal with the right-hand pieces. Each of these rectangular pieces is completely described by the line segment of constant x that is its right-hand side, because its left-hand side is the bisector. The set of these right-hand sides can be handled as in § 4.4 above. A point is in one of these rectangular pieces whenever the right-hand side of the piece is visible along a line of increasing x from the point. Therefore the solution is simply $\log k$ iterations of the solution of § 4.4 above, and one can enumerate the set of all rectangles in a dynamic set of rectangles that contain a test point in $O(\log n \log^2 k + s)$ time, a further new result. By now the reader can see how to extend this indefinitely, so we shall next consider a different class of applications.

4.6. Off-line intersections among a set of rectangles. We can enumerate all intersecting pairs among a set of axis-aligned rectangles (rectangles with sides parallel to the axes) by using the plane sweep technique first proposed by Shamos and Hoey [7]. This technique simulates the motion of a horizontal line across a plane from bottom to top, and considers the sequence of cross-sectional slices induced by this line.

For aligned rectangles a cross-sectional slice is a set of horizontal intervals. As the sweep line moves upward onto a new rectangle, that rectangle's horizontal interval is added to the set. As the sweep line moves upward off a rectangle, that rectangle's

horizontal interval is removed from the set. Every time a new horizontal interval is added to the set, an enumeration is made of all other intervals in the set that touch the new interval.

We now analyze the performance of the rectangle intersection algorithm somewhat more carefully. The sweep technique requires that the rectangles be sorted by their bottom edges, and that their top edges be maintained in a priority queue. For n rectangles this takes $O(n)$ space and $O(n \log n)$ time. The priority search tree operations can be done in $O(n)$ space and $O(\log n + s)$ time apiece. Each rectangle causes one **InsertPair**, one **DeletePair**, and one **EnumerateRectangle** operation. Moreover each rectangle intersection is discovered by exactly one **EnumerateRectangle** operation, and therefore contributes to the s of that operation. Thus the overall time performance is $O(n \log n + s)$. This performance, which has been achieved before with more complex data structures [5], [10], is within a constant factor of the best possible worst-case performance.

An application like circuit extraction from IC masks might involve rectangles of several colors, and be concerned only with intersections between rectangles of different colors. For this purpose one could have a different priority search tree for each color. As the sweep line passes the bottom edge of a red rectangle, for example, the corresponding red horizontal interval would be inserted into the red priority search tree, while the same interval would be used in a **EnumerateRectangle** operation on every nonred priority search tree. Now arbitrary intersection patterns of red with red rectangles do not increase the time beyond $O(n \log n)$. The s term counts only the number of *inter-color* intersections.

4.7. Memory allocation. Many computer operating systems satisfy dynamic requests for memory according to a first-fit (use the free block of adequate size at the smallest address) or a best-fit (use the smallest free block of adequate size) policy. One might imagine that a synthesis of these two policies could perform better than either one separately, but at first glance it is not apparent how to organize the free blocks into a single space-efficient structure that will allow the time-efficient implementation of either policy.

Now consider a priority search tree (of either kind, but one would probably want to use the radix kind), where the x dimension is an encoding of [free block length, free block address] for uniqueness (see § 2) and the y dimension is the free block address. Best-fit can be implemented using only the search tree part of the radix search tree in the obvious way. A first-fit implementation uses **MinYInXRange** on an x range of **[neededBlockSize .. largestPossibleBlockSize]**. Each of these operations, as well as insertion or removal of free blocks in the structure requires at most logarithmic time. The extra space requirements are minimal: a radix priority search tree for this purpose requires that each free block contain two pointers and a field to hold the length of the free block.

A result similar to this is attributed to McCreight in [2], and that earlier data structure bears a striking resemblance to a priority search tree. The difference between them is that in the earlier structure, the pair whose y-value is minimal in a subtree not only appears at the root of the subtree, but might also be repeated on a spine all the way down to a leaf of the subtree. In a priority search tree, as in a proper priority queue, pairs are not repeated. The effect of this is that the earlier structure and the priority search tree perform equally well (within constant factors) for all operations except **EnumerateRectangle**, but the $O(\text{long } n + s)$ time bound for **EnumerateRectangle** cannot be attained with the earlier structure. This is because in the earlier structure

enumerations can encounter the same pairs over and over again, often enough to ruin linearity in s.

5. Open questions. The question that led me from tile trees [3] to priority search trees remains unanswered. I still do not know whether it is possible, for an arbitrary set of n aligned rectangles, to enumerate all s pairs that totally contain one another in linear space and time $O(n \log n + s)$. The methods in this paper allow one to determine containment on three edges, but alas, three edges do not a rectangle make.

Priority search trees are one small step closer to the ultimate goal of general two-dimensional range searching in linear space and logarithmic worst-case time. Is that ultimate goal attainable? If not, or if we cannot discover how, are there further small but useful steps?

Priority search trees are an interesting case of two data structures (a search tree and a priority queue) in symbiosis, defined as "the living together of two dissimilar organisms, especially when this association is mutually beneficial." Are there other pairs of data structures that also benefit from symbiosis?

Acknowledgments. The ideas leading to this paper have developed over several years, and I have many people to thank. I especially thank Jon Bentley for pointing out the gaping hole in my original attempt to solve this problem, and Howard Sturgis for pointing out that within the rococo walls of one of my intermediate formulations lay the elegant machinery of § 2. Jan van Leeuwen first made me aware of Olivie's balanced trees with guaranteed constant rotations per update, thereby saving the reader a very tortuous and otherwise useless new data structure in § 3, and Bob Tarjan recently observed that 2-3-4 trees can have the same property. I also thank John Warnock, Leo Guibas, Bob Sedgewick, Mark Brown, Jurg Nievergelt, Herbert Edelsbrunner, Jean Vuillemin, the referees, and others for discussions that led to the present algorithms and presentation.

Appendix A. Pascal procedures for radix priority search trees.

```
PROCEDURE InsertPair(VAR t: RPSTPtr; newPr: Pair;
   lowerX: KeyRange; upperX: KeyBound);
   VAR
      p: Pair;
      middleX: KeyRange;
   BEGIN
   IF t = NIL THEN
      BEGIN
      NEW(t); (* add a new leaf node *)
      t↑.p := newPr;
      t↑.left := NIL;
      t↑.right := NIL;
      END
   ELSE IF t↑.p.x <> newPr.x (* assumes unique x values *)
      THEN
      BEGIN
      IF newPr.y < t↑.p.y THEN (* new pair beats existing one *)
         BEGIN p := t↑.p; t↑.p := newPr END
            (* exchange new/existing *)
      ELSE p := newPr;
      middleX := (lowerX+upperX) DIV 2;
      IF p.x < middleX
         THEN InsertPair(t↑.left, p, lowerX, middleX)
         ELSE InsertPair(t↑.right, p, middleX, upperX);
      END;
   (* ELSE this pair already present, so don't insert it *)
   END;  (* of InsertPair *)
```

```
PROCEDURE DeletePair(VAR t: RPSTPtr; oldPr: Pair;
   lowerX: KeyRange; upperX: KeyBound);
   VAR
      middleX: KeyRange;
   BEGIN
   IF t <> NIL THEN
      BEGIN
      IF t↑.p.x = oldPr.x  (* assumes unique x values *)
         THEN
         BEGIN (* have located pair to delete *)
         IF t↑.left <> NIL THEN
            BEGIN
            IF t↑.right <> NIL THEN
               BEGIN (* node has both left and right subtrees *)
               IF t↑.left↑.p.y < t↑.right↑.p.y THEN
                  BEGIN (* left beats right *)
                  t↑.p := t↑.left↑.p;
                  DeletePair(t↑.left, t↑.p, lowerX, upperX);
                  END
               ELSE
                  BEGIN (* right beats left *)
                  t↑.p := t↑.right↑.p;
                  DeletePair(t↑.right, t↑.p, lowerX, upperX);
                  END;
               END
            ELSE
               BEGIN (* node has only left subtree *)
               t↑.p := t↑.left↑.p;
               DeletePair(t↑.left, t↑.p, lowerX, upperX);
               END;
            END
         ELSE
            BEGIN
            IF t↑.right <> NIL THEN
               BEGIN (* node has only right subtree *)
               t↑.p := t↑.right↑.p;
               DeletePair(t↑.right, t↑.p, lowerX, upperX);
               END
            ELSE
               BEGIN (* node has no subtrees *)
               DISPOSE(t);
               t := NIL;
               END;
            END;
         END
      ELSE
         BEGIN (* pair to delete is in a subtree *)
         middleX := (lowerX+upperX) DIV 2;
         IF oldPr.x < middleX
            THEN DeletePair(t↑.left, oldPr, lowerX, middleX)
            ELSE DeletePair(t↑.right, oldPr, middleX, upperX);
         END;
      END;
   (* ELSE this pair wasn't in the tree so it can't be deleted *)
   END;  (* of DeletePair *)
```

```
TYPE CondPair = RECORD
   valid: BOOLEAN;
   p: Pair;
   END;

FUNCTION MinXInRectangle(t: RPSTPtr; x0, x1, y1: KeyRange;
   lowerX: KeyRange; upperX: KeyBound): CondPair;
   VAR
      c: CondPair;
      middleX: KeyRange;
   BEGIN
   IF t <> NIL THEN
      BEGIN
      IF t↑.p.y > y1 THEN
         (* No nodes in this subtree lie in the search
         rectangle, because they all have y-values that are
         too large. *)
         c.valid := FALSE
      ELSE
         BEGIN
         middleX := (lowerX+upperX) DIV 2;

         IF x0 < middleX THEN
            (* The answer can only lie in the left subtree
            if some point in the search rectangle
            could lie in the left subtree. *)
            c := MinXInRectangle(t↑.left, x0, x1, y1,
               lowerX, middleX)
         ELSE c.valid := FALSE;

         IF (NOT c.valid) AND (middleX <= x1) THEN
            (* The answer can only lie in the right subtree
            if no point in the left subtree lies in the search
            rectangle, but some point in the search rectangle
            could lie in the right subtree. *)
            c := MinXInRectangle(t↑.right, x0, x1, y1,
               middleX, upperX);

         IF (x0 <= t↑.p.x) AND (t↑.p.x <= x1) AND
            ((NOT c.valid) OR (t↑.p.x < c.p.x)) THEN
            (* t↑.p is best of all in the search rectangle *)
            BEGIN
            c.valid := TRUE;
            c.p := t↑.p;
            END;
         END
      END
   ELSE c.valid := FALSE;   (* empty subtree *)
   MinXInRectangle := c;
   END;  (* of MinXInRectangle *)
```

```
FUNCTION MaxXInRectangle(t: RPSTPtr; x0, x1, y1: KeyRange;
   lowerX: KeyRange; upperX: KeyBound): CondPair;
   VAR
      c: CondPair;
      middleX: KeyRange;
   BEGIN
   IF t <> NIL THEN
      BEGIN
      IF t↑.p.y > y1 THEN
         (* No nodes in this subtree lie in the search
         rectangle, because they all have y-values that are
         too large. *)
         c.valid := FALSE
      ELSE
         BEGIN
         middleX := (lowerX+upperX) DIV 2;

         IF middleX < x1 THEN
            (* The answer can only lie in the right subtree
            if some point in the search rectangle
            could lie in the right subtree. *)
            c := MaxXInRectangle(t↑.right, x0, x1, y1,
               middleX, upperX)
         ELSE c.valid := FALSE;

         IF (NOT c.valid) AND (x0 <= middleX) THEN
            (* The answer can only lie in the left subtree
            if no point in the right subtree lies in the search
            rectangle, but some point in the search rectangle
            could lie in the left subtree. *)
            c := MaxXInRectangle(t↑.left, x0, x1, y1,
               lowerX, middleX);

         IF (x0 <= t↑.p.x) AND (t↑.p.x <= x1) AND
            ((NOT c.valid) OR (c.p.x < t↑.p.x)) THEN
            (* t↑.p is best of all in the search rectangle *)
            BEGIN
            c.valid := TRUE;
            c.p := t↑.p;
            END;
         END
      END
   ELSE c.valid := FALSE;   (* empty subtree *)
   MaxXInRectangle := c;
   END;  (* of MaxXInRectangle *)
```

```
FUNCTION MinYInXRange(t: RPSTPtr; x0, x1: KeyRange;
   lowerX: KeyRange; upperX: KeyBound): CondPair;
   VAR
      c, cRight: CondPair;
      middleX: KeyRange;
   BEGIN
   IF t <> NIL THEN
      IF (x0 <= t↑.p.x) AND (t↑.p.x <= x1) THEN
         (* This node's p pair lies in the x range, and it must
         be the min y in its subtree because of the priority
         queue invariant on y. *)
         BEGIN
         c.valid := TRUE;
         c.p := t↑.p;
         END
      ELSE
         BEGIN
         middleX := (lowerX+upperX) DIV 2;

         IF x0 < middleX THEN c :=
            MinYInXRange(t↑.left, x0, x1, lowerX, middleX)
         ELSE c.valid := FALSE;

         IF middleX <= x1 THEN cRight :=
            MinYInXRange(t↑.right, x0, x1, middleX, upperX)
         ELSE cRight.valid := FALSE;

         IF NOT c.valid OR
            (cRight.valid AND (cRight.p.y < c.p.y)) THEN
               c := cRight;
         END
   ELSE c.valid := FALSE;   (* empty subtree *)
   MinYInXRange := c;
   END;  (* of MinYInXRange *)
```

```
FUNCTION EnumerateRectangle(t: RPSTPtr; x0, x1, y1: KeyRange;
   FUNCTION Report(Pair): BOOLEAN;
   lowerX: KeyRange; upperX: KeyBound): BOOLEAN;
   VAR
      continue: BOOLEAN;
      middleX: KeyRange;
   BEGIN
   IF t <> NIL THEN
      IF t↑.p.y <= y1 THEN
         BEGIN  (* node passes y test *)
         IF (x0 <= t↑.p.x) AND (t↑.p.x <= x1) THEN
            continue := Report(t↑.p)
         ELSE continue := TRUE;
         middleX := (lowerX+upperX) DIV 2;
         IF continue AND (x0 < middleX)
            THEN
            continue := EnumerateRectangle(t↑.left, x0, x1, y1,
               Report, lowerX, middleX);
         IF continue AND (middleX <= x1)
            THEN
            continue := EnumerateRectangle(t↑.right, x0, x1, y1,
               Report, middleX, upperX);
         EnumerateRectangle := continue;
         END
      ELSE EnumerateRectangle := TRUE  (* node fails y test *)
   ELSE EnumerateRectangle := TRUE;   (* empty subtree *)
   END;  (* of EnumerateRectangle *)
```

Appendix B. Pascal procedures for balanced priority search trees.

```
PROCEDURE BalancedInsertPair(VAR t: BPSTPtr; newPr: Pair;
   useAsP: BOOLEAN);
   BEGIN (* Top-level call has useAsP = TRUE *)
   IF t = NIL THEN
      BEGIN (* put newPr in the q field of a new leaf node *)
      NEW(t);
      t↑.q := newPr;
      t↑.left := NIL;
      t↑.right := NIL;
      t↑.validP := FALSE;
      t↑.duplQ := NOT useAsP;
      t↑.balance := leafBalance
         (* depends on tree family *)
      END
   ELSE
      BEGIN
      IF useAsP AND ((NOT t↑.validP) OR (newPr.y < t↑.p.y)) THEN
         BEGIN (* newPr belongs in t↑.p *)
         DisposeP(t↑);
         t↑.p := newPr;
         t↑.validP := TRUE;
         useAsP := FALSE;
         END;
      IF newPr.x < t↑.q.x
         THEN BalancedInsertPair(t↑.left, newPr, useAsP)
         ELSE BalancedInsertPair(t↑.right, newPr, useAsP);
      AdjustBalanceForInsert(t);
         (* implementation varies by tree family *)
      END;
   END;  (* of BalancedInsertPair *)
```

```
PROCEDURE BalancedDeletePair(VAR t: BPSTPtr; oldPr: Pair);

   TYPE
      ExtractedPair = RECORD
         q: Pair;
         duplAsP: BOOLEAN (* q appears as p field higher in tree *)
         END;
   VAR
      n: ExtractedPair;
      d: BPSTPtr;

   FUNCTION ExtractMaxQX(VAR t: BPSTPtr): ExtractedPair;
      VAR
         d: BPSTPtr;
      BEGIN (* extract the pair with maximal q.x in t *)
      IF t↑.right = NIL THEN
         BEGIN
         ExtractMaxQX.q := t↑.q;
         ExtractMaxQX.duplAsP := t↑.duplQ;
         DisposeP(t↑);
         d := t;
         t := t↑.left;
         DISPOSE(d);
         END
      ELSE
```

```
      BEGIN
      ExtractMaxQX := ExtractMaxQX(t↑.right);
      IF ExtractMaxQX.duplAsP AND t↑.validP
         AND (t↑.p = ExtractMaxQX.q) THEN
         BEGIN
         ExtractMaxQX.duplAsP := FALSE;
         ExtractP(t↑);
            (* re-fill invalidated p field if possible *)
         END;
      AdjustBalanceForNeighborExtract(t);
         (* implementation varies by tree family *)
      END
   END;  (* of ExtractMaxQX *)

BEGIN
IF t <> NIL THEN
   BEGIN
   IF t↑.q = oldPr THEN
      (* have located node whose .q field is oldPr *)
      BEGIN
      DisposeP(t↑);
      IF (t↑.left = NIL) OR (t↑.right = NIL) THEN
         (* t↑ has at most one subtree and can
            be bypassed *)
         BEGIN
         d := t;
         IF t↑.left = NIL
            THEN t := t↑.right
            ELSE t := t↑.left;
         DISPOSE(d);
         END
      ELSE

         (* t↑ has both subtrees.  We must find
         a neighboring pair n.q that can
         replace t↑.q without violating x-order. *)
         BEGIN
         n := ExtractMaxQX(t↑.left);
         t↑.q := n.q;
         t↑.duplQ := n.duplAsP;
         ExtractP(t↑);
            (* re-fill invalidated p field if possible *)
         END;
      END
   ELSE
      BEGIN (* oldPr must be a .q field in a subtree *)
      IF oldPr.x < t↑.q.x
         THEN BalancedDeletePair(t↑.left, oldPr)
         ELSE BalancedDeletePair(t↑.right, oldPr);
      IF t↑.validP AND (t↑.p = oldPr) THEN
         ExtractP(t↑);
            (* re-fill invalidated p field if possible *)
      END;
   AdjustBalanceForDelete(t↑);
   (* implementation varies by tree family *)
   END
(* ELSE this pair wasn't in the tree so it can't be deleted *);
END;  (* of BalancedDeletePair *)
```

```
PROCEDURE RotateRight(VAR t: BPSTPtr);
   VAR
      e, c: BPSTPtr;
   BEGIN (* implements the rotation in Figure 2 *)
   e := t;
   DisposeP(e↑);
   c := e↑.left;
   DisposeP(c↑);
   e↑.left := c↑.right;
   ExtractP(e↑);
   c↑.right := e;
   ExtractP(c↑);
   AdjustBalanceForRotateRight(c);
      (* implementation varies by tree family *)
   t := c;
   END;  (* of RotateRight *)
```

```
PROCEDURE DisposeP(VAR t: BPST);
   BEGIN (* DisposeP can cause a temporary violation of the second
      structural invariant, leaving a node in the middle of the tree
      with an invalid p field while one of its children has a
      valid p field.  This violation is usually repaired by a
      subsequent invocation of ExtractP *)
   IF t.validP THEN
      BEGIN
      IF t.p.x < t.q.x THEN
         BEGIN (* dispose into left subtree *)
         IF t.p = t.left↑.q THEN
            (* maintains third invariant, depends on
            uniqueness of pairs in t *)
            t.left↑.duplQ := FALSE
         ELSE
            BEGIN
            DisposeP(t.left↑);
            t.left↑.p := t.p;
            t.left↑.validP := TRUE;
            END
         END
      ELSE
         BEGIN (* dispose into right subtree *)
         IF t.p = t.right↑.q THEN
            (* maintains third invariant, depends on
            uniqueness of pairs in t *)
            t.right↑.duplQ := FALSE
         ELSE
            BEGIN
            DisposeP(t.right↑);
            t.right↑.p := t.p;
            t.right↑.validP := TRUE;
            END
         END;
      t.validP := FALSE;
      END
   (* ELSE no p field to dispose *);
   END;  (* of DisposeP *)
```

```
PROCEDURE ExtractP(VAR t: BPST);
   CONST Worst = LastKey+1;
   VAR
      leftY,rightY: FirstKey..Worst;
   BEGIN
   leftY := Worst;
   IF t.left <> NIL THEN
      BEGIN
      IF t.left↑.validP THEN leftY := t.left↑.p.y;
      IF NOT t.left↑.duplQ THEN leftY := MIN(leftY, t.left↑.q.y);
      END;
   rightY := Worst;
   IF t.right <> NIL THEN
      BEGIN
      IF t.right↑.validP THEN rightY := t.right↑.p.y;
      IF NOT t.right↑.duplQ THEN rightY := MIN(rightY, t.right↑.q.y);
      END;
   IF leftY < rightY THEN
      BEGIN (* best is left *)
      IF t.left↑.validP AND (leftY = t.left↑.p.y) THEN
         BEGIN (* steal his p field *)
         t.p := t.left↑.p;
         ExtractP(t.left↑);
         END
      ELSE
         BEGIN (* his q field is unduplicated and better,
            so duplicate it *)
         t.p := t.left↑.q;
         t.left↑.duplQ := TRUE;
         END;
      t.validP := TRUE;
      END
   ELSE IF rightY <> Worst THEN
      BEGIN (* best is right *)
      IF t.right↑.validP AND (rightY = t.right↑.p.y) THEN
         BEGIN (* steal his p field *)
         t.p := t.right↑.p;
         ExtractP(t.right↑);
         END
      ELSE
         BEGIN (* his q field is unduplicated and better,
            so duplicate it *)
         t.p := t.right↑.q;
         t.right↑.duplQ := TRUE;
         END;
      t.validP := TRUE;
      END
   ELSE t.validP := FALSE; (* no candidates *)
   END;  (* of ExtractP *)
```

REFERENCES

[1] D. E. KNUTH, *The Art of Computer Programming, Vol. 3, Sorting and Searching*, Addison-Wesley, Reading, MA, 1973, pp. 142–145.

[2] H. J. OLIVIE, *A new class of balanced search trees; half-balanced binary search trees*, RAIRO Theor. Inform., 16 (1982), pp. 51–71.

[3] E. MCCREIGHT, *Efficient algorithms for enumerating intersecting intervals and rectangles*, Xerox Palo Alto Research Center Technical Report CSL-80-9, Xerox Corporation, Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA, 1980.

[4] L. GUIBAS AND J. SAXE, Private communication, 1980.

[5] H. EDELSBRUNNER, *Dynamic rectangle intersection searching*, Report 47, Information Processing Institute, Technical University of Graz, Graz, Austria, February, 1980.

[6] ——— *Dynamic data structures for orthogonal intersection queries*, Report 59, Information Processing Institute, Technical University of Graz, Graz, Austria, October, 1980.

[7] M. I. SHAMOS AND D. HOEY, *Geometric instersection problems*, 17th Annual IEEE Symposium on Foundations of Computer Science, 1975, pp. 208–215.

[8] H. EDELSBRUNNER, *A time- and space-optimal solution for the planar all intersecting rectangles problem*, Report 50, Information Processing Institute, Technical University of Graz, Graz, Austria, April, 1980.

[9] J. BENTLEY, *Priority queues with range restrictions*, Bulletin of the European Association of Theoretical Computer Science, #9, H. Maurer, ed., Technical University of Graz, Graz, Austria, October, 1979, pp. 7–8.

[10] J. L. BENTLEY AND D. WOOD, *An optimal worst-case algorithm for reporting intersections of rectangles*, IEEE Trans. Comp., C-29 (1980), pp. 571–577.

[11] R. P. BRENT, *Efficient implementation of the first-fit strategy for dynamic storage allocation*, TR-CS-81-05, Dept. Computer Science, Australian National University, Canberra, ACT 2600, Australia, 1981, Australian Computer Science Communications, to appear.

[12] J. NIEVERGELT AND F. P. PREPARATA, *Plane-sweep algorithms for interesting geometric figures*, Technical Report in preparation, Institut für Informatik, ETH, Zurich.

[13] J. VUILLEMIN, *A unifying look at data structures*, Comm. ACM, 23 (1980), pp. 229–239.

[14] R. BAYER, *Symmetric binary B-trees: data structure and maintenance algorithms*, Acta Inform., 1 (1972), pp. 290–306.

[15] L. J. GUIBAS AND R. SEDGEWICK, *A dichromatic framework for balanced trees*, 19th Annual IEEE Symposium on Foundations of Computer Science, IEEE publication 78CH1397-9, 1978, pp. 8–21.

[16] R. TARJAN, Private communication, September, 1982.