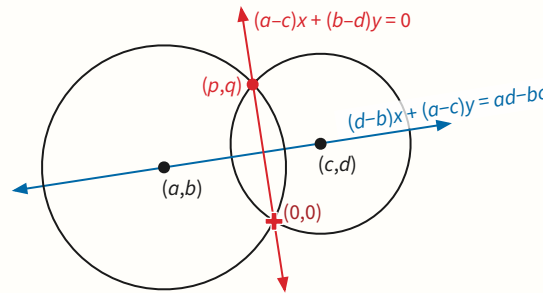


1. Suppose we are given a set D of n circular disks in the plane, each with the origin $(0, 0)$ on its boundary, and we are asked to compute the boundary of their union. Each disk is represented by the x - and y -coordinates of its center.
 - (a) Describe how to compute the intersection points of the boundaries of two disks in D in $O(1)$ time. (The origin is always one of these intersection points; compute the other one.)

Solution: Let (a, b) and (c, d) are the centers of two circles that intersect at the origin. The other intersection point is

$$\left(\frac{2(b-d)(ad-bc)}{(a-c)^2 + (b-d)^2}, \frac{2(a-c)(ad-bc)}{(a-c)^2 + (b-d)^2} \right)$$

Proof: The following figure summarizes the construction:



The boundary circles are defined by the following equations:

$$(x - a)^2 + (y - b)^2 = a^2 + b^2 \quad (x - c)^2 + (y - d)^2 = c^2 + d^2$$

Thus, the intersection points must satisfy the equations

$$(x - a)^2 + (y - b)^2 - (a^2 + b^2) = (x - c)^2 + (y - d)^2 - (c^2 + d^2) = 0.$$

Straightforward but tedious algebra simplifies the first equation to

$$(a - c)x + (b - d)y = 0.$$

This equation describes the line through the two intersection points. (This line, drawn in red in the figure above, is called the **radical axis** of the two circles.)

The coefficients α and β of the line $\alpha x + \beta y = 1$ through the two centers (a, b) and (c, d) (the blue line in the figure above) satisfy the linear system

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

which by Cramer's rule implies^a

$$\alpha = \frac{d - b}{ab - cd} \quad \beta = \frac{a - c}{ad - bc}.$$

So the line through the two centers is $(d - b)x + (a - c)y = ad - bc$.

Finally, let (p, q) denote the second intersection point of the two circles. By symmetry, the line through the centers and the line through the intersection points intersect at the point $(p/2, q/2)$. Thus, the coordinates p and q satisfy the linear system

$$\begin{bmatrix} a - c & b - d \\ d - b & a - c \end{bmatrix} \begin{bmatrix} p/2 \\ q/2 \end{bmatrix} = \begin{bmatrix} 0 \\ ad - bc \end{bmatrix},$$

and the claimed solution follows from Cramer's rule. \square

^aI am implicitly assuming here that $ad - bc \neq 0$, or equivalently, that the two centers do not lie on a common line through the origin. If the centers *are* collinear with the origin, then both circles are tangent to their radical axis, and thus to each other, at the origin $(0, 0)$.

- (b) Describe an algorithm to compute (a discrete description of) the boundary of the union of D in $O(n \log h)$ time, where h is the number of boundary arcs. [Hint: Reduce to a problem you've already seen. Turn the input inside out. Or don't.]

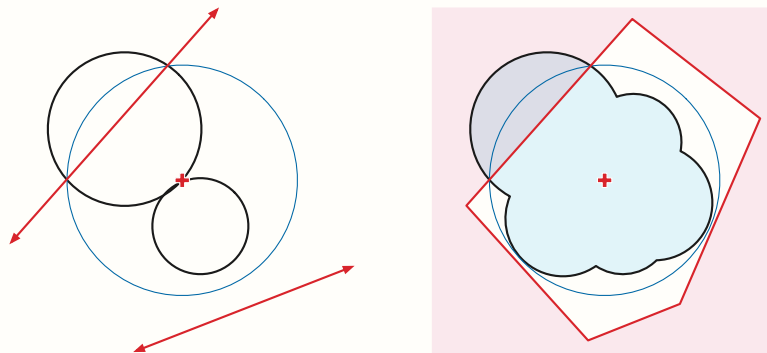
Solution (inversion): Let's apply a geometric transformation to the input called *inversion* defined by the function

$$\text{Inv}(x, y) = \left(\frac{x}{x^2 + y^2}, \frac{y}{x^2 + y^2} \right).$$

The inversion function Inv maps any point at distance r from the origin to the point at distance $1/r$ in the same direction; in particular, Inv maps every point on the unit circle to itself, and maps the origin to a special point "at infinity".

Inversion maps every circle either to another circle or to a straight line (a circle with infinite radius). In particular, inverting the closed disk with center (a, b) that touches the origin yields the halfplane $ax + by \geq 1/2$. We can see this most easily using the fact that Inv is an involution, meaning $\text{Inv}(\text{Inv}(p)) = p$ for every point $p \neq (0, 0)$. Inverting the halfplane $ax + by \geq 1/2$ gives us

$$\begin{aligned} 1/2 \leq \frac{ax}{x^2 + y^2} + \frac{by}{x^2 + y^2} &\iff x^2 + y^2 \leq 2ax + 2by \\ &\iff x^2 + y^2 - 2ax - 2by \leq 0 \\ &\iff x^2 + y^2 - 2ax - 2by + a^2 + b^2 \leq a^2 + b^2 \\ &\iff (x - a)^2 + (y - b)^2 \leq a^2 + b^2 \end{aligned}$$



Left: Inverting two circles through the origin yields two straight lines.

Right: Inverting the union of disks touching the origin yields the union of halfplanes.

In both figures, the thin blue circle is the unit circle.

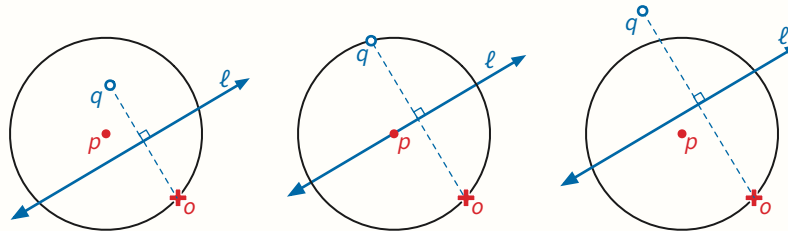
It follows that inverting the union of the n given disks yields the union of n halfplanes. The complement of this union is the *intersection* of the complementary halfplanes. Finally, we can compute this halfplane intersection in $O(n \log h)$ time using duality and Chan's convex hull algorithm. ■

Solution (duality): Let P denote the given set of points, let o denote the origin $(0, 0)$, and let $P' = P \cup \{o\}$. For any point p , let $\odot p$ denote the circle centered at p that passes through o , and let $\odot P$ denote the resulting set of circles. Let U denote the union of disks bounded by the circles $\odot p$; this is the shape we are trying to compute.

Lemma 1.1. *Let o and p be an arbitrary distinct points, let ℓ be an arbitrary line, and let q be the reflection of o across ℓ . Then p lies on ℓ if and only if q lies on $\odot p$, and p and o lie on the same side of ℓ if and only if q lies outside $\odot p$.*

Proof: Let $p = (a, b)$ and $q = (c, d)$. Line ℓ is the perpendicular bisector of oq , so it is defined by the equation $(x, y) \cdot (c, d) = \frac{1}{2}(c, d) \cdot (c, d)$, or equivalently, $2xc + 2yd = c^2 + d^2$. Thus, p lies on ℓ if and only if $2ac + 2bd = c^2 + d^2$. Moreover, p and o lie on the same side of ℓ if and only if $(a, b) \cdot (c, d) < \frac{1}{2}(c, d) \cdot (c, d)$, or equivalently, $2ac + 2bd < c^2 + d^2$.

On the other hand, $\odot p$ is defined by the equation $(x - a)^2 + (y - b)^2 = a^2 + b^2$, or equivalently, $x^2 + y^2 = 2ax + 2by$. Thus, q lies on $\odot p$ if and only if $2ac + 2bd = c^2 + d^2$. Moreover, q lies outside $\odot p$ if and only if $(c - a)^2 + (d - b)^2 > a^2 + b^2$, or equivalently, $2ac + 2bd < c^2 + d^2$. □

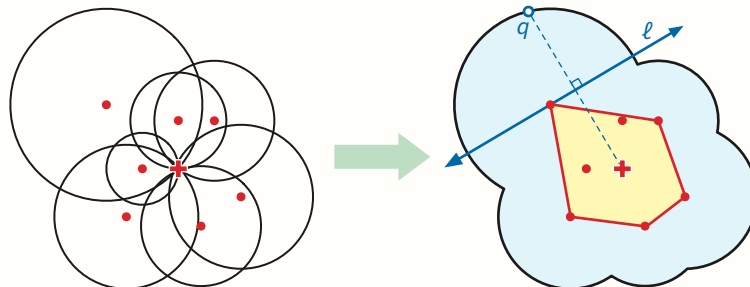


Theorem 1.2. *For any point $p \in P$, the boundary of U contains a non-empty arc of $\odot p$ if and only if p is a vertex of the convex hull of P' .*

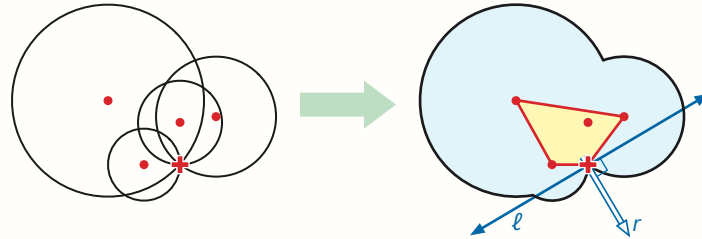
Proof: The following observations follow immediately of the definitions of $\text{conv}(P')$ and U .

- Point $p \in P$ is a vertex of $\text{conv}(P')$ if and only if there is a line ℓ through p , such that all other points in P' , including the origin, lie on one side of ℓ .
- The boundary of U contains a non-empty arc of $\odot p$ if and only if some point q on $\odot p$ lies outside all other circles in $\odot P$.

Lemma 1.1 implies that these two conditions are identical. □



Let me emphasize that it is not enough to compute the convex hull of the original centers P ; we must include the origin in the convex hull computation. Arguments similar to Lemma 1.1 imply that a line ℓ through the origin has all points in P in one halfplane if and only if the ray r orthogonal to ℓ in the opposite halfplane lies entirely outside U .



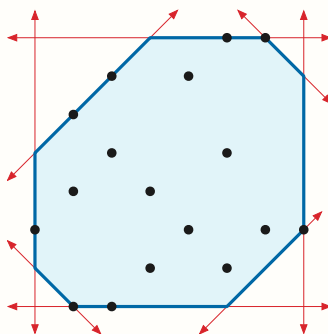
Theorem 1.2 implies that we can compute a reasonable representation of U by computing the convex hull of P' in $O(n \log h)$ time, using Chan's algorithm. The counterclockwise sequence of indices of boundary arcs of U is identical to the sequence of indices of vertices of $\text{conv } P'$ (except the origin o). If we need the coordinates of any vertex of U , we can compute them in $O(1)$ time using the algorithm from part (a). ■

2. Suppose we are given a set P of n points on the integer grid. Describe an efficient algorithm that computes an octilinear polygon with *minimum perimeter* that encloses P .

Solution: I claim that the required enclosure is the intersection of eight halfplanes:

$$\begin{array}{ll} x \geq \min_i x_i & x \leq \max_i x_i \\ x + y \geq \min_i (x_i + y_i) & x + y \leq \max_i (x_i + y_i) \\ y \geq \min_i y_i & y \leq \max_i y_i \\ x - y \geq \min_i (y_i - x_i) & x - y \leq \max_i (y_i - x_i) \end{array}$$

(Here (x_i, y_i) is the i th point in P .) Let's call the resulting convex polygon the *bounding octagon* of P , denoted $Oct(P)$, even though it could have less than eight edges. By definition, $Oct(P)$ is an octilinear polygon that encloses P .



We need to prove that $Oct(P)$ is a *minimum-perimeter* octilinear enclosing polygon. The main idea of the proof is relatively simple, but the details are surprisingly delicate.

Recall that a polygon is *x-monotone* if its intersection with any vertical line is connected or empty. Similarly, a polygon is *y-monotone*, *(x + y)-monotone*, or *(x - y)-monotone* if its intersection with any horizontal line, any line with slope -1 , or any line with slope 1 , respectively, is connected or empty.

Lemma 2.1. *Every minimum-perimeter octilinear enclosure of P is x -monotone, y -monotone, $(x + y)$ -monotone, and $(x - y)$ -monotone.*

Proof: Let E be any octilinear enclosure of P that is not x -monotone. Then there must be a line ℓ whose intersection with E has more than one component. Let s and t be two consecutive points in $\partial E \cap \ell$ such that the segment st is outside E . Replacing a subpath of E from s to t with the line segment st yields another octilinear enclosure with smaller perimeter than E . We conclude that E is not a minimum-perimeter octilinear enclosure of P .

Similar arguments apply to the other three directions. □

Lemma 2.2. *Every minimum-perimeter octilinear enclosure of P is a subset of $Oct(P)$.*

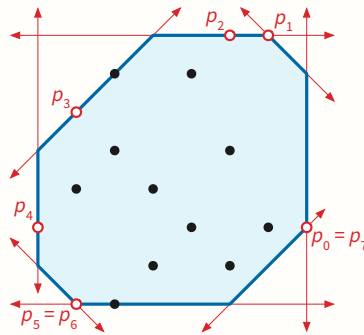
Proof: Let E be any octilinear enclosure of P that contains any point outside the halfplane $y \leq \max_i y_i$. The previous lemma implies the intersection of E and the

bounding line $y = \max_i y_i$ is a single line segment st . Replacing a subpath of E from s to t with st yields another octilinear enclosure with smaller perimeter than E . We conclude that E is not a minimum-perimeter octilinear enclosure of P .

Similar arguments apply to the other seven halfplanes defining $Oct(P)$. \square

Theorem 2.3. $BOct(P)$ is a minimum-perimeter octilinear enclosure of P .

Proof: Choose arbitrary points $p_0, p_1, \dots, p_7 \in P$ that lie on the eight bounding lines of $Oct(P)$ in counterclockwise order. Specifically, let p_0 be any point of P on the line $x = \max_i x_i$, let p_1 be any point of P on the line $x + y = \max_i(x_i + y_i)$, let p_2 be any point of P on the line $y = \max_i y_i$, and so on. Some of these points may coincide. Finally, let $p_8 = p_0$.



The previous lemmas imply that every minimum-perimeter octilinear enclosure of P passes through the points p_i in cyclic index order. The boundary of $Oct(P)$ contains a shortest octilinear path from each point p_i to its successor p_{i+1} , which consists of a single point (if $p_i = p_{i+1}$), a single octilinear segment (if $p_i p_{i+1}$ is octilinear), or two octilinear segments meeting at 135° . \square

Computing $Oct(P)$ in $O(n)$ time is straightforward. \blacksquare

3. (a) Describe and analyze a variant of Graham’s scan that begins by sorting the points by their x -coordinates; the rest of the algorithm should take $O(n)$ time. In particular, prove that the algorithm actually works. What invariant does the "three-penny" algorithm maintain that guarantees that the final resulting polygon is convex?

Solution: First sort the input points P by increasing x -coordinate. We can construct the lower convex hull of P using the following variant of the three-penny algorithm.

```

MONOTHREEPENNY( $P[1..n]$ ):
  initialize a stack
  PUSH(0)
  PUSH(1)
  next ← 2
  while next ≤  $n$ 
    curr ← top index on the stack
    prev ← second index on the stack
    if  $i = 0$  or  $\Delta(P[prev], P[curr], P[next]) > 0$ 
      PUSH(next)
      next ← next + 1
    else
      POP()  <<discard curr>>[0.5ex]
    
```

When this algorithm ends, the stack contains the vertices of the lower convex hull, in order from right (at the top of the stack) to left.

If we sort P by *decreasing* x -coordinate, the same algorithm computes the *upper* convex hull of P .

Intuitively, we are starting with a degenerate polygon that traverses the points twice—first from left to right and then from right to left—and then repeatedly removing reflex vertices. At all times, the polygon is x -monotone, meaning its intersection with any vertical line is either empty, a single point, or a single line segment. A polygon is convex if and only if it is x -monotone and has no reflex vertices. ■

- (b) Suppose we can magically sort the n given x -coordinates in $sort(n) = o(n \log n)$ time. Describe how to modify Chan’s algorithm so that it runs in $O((n/h) \cdot sort(h))$ time. Your algorithm must use the magical sorting algorithm as a black box; however, you can assume that the function $sort(n)$ is known in advance.

Solution: We make three modifications to Chan’s algorithm. First, we use the algorithm from part (a) instead of Graham’s scan. Second, we partition the input points into fewer larger subsets, so that dependence on h in the running time *only* comes from sorting. Finally, we use a different series of estimates for h that depends on the *sorting* time.

First, to simplify notation, let $overhead(n) = sort(n)/n$. We immediately have $overhead(n) = O(\log n)$. We can also assume that $overhead$ is a non-constant, monotonically non-decreasing function of n . (If $sort(n) = O(n)$, we can just run

the algorithm from part (a).)

For the moment, suppose we know the number of hull edges h . Instead of partitioning the input points into subsets of size about h , we partition into $O(n/h^2)$ subsets of size $O(h^2)$.^a Then, as usual, we proceed in two stages.

- First we compute the convex hull of each subset using our algorithm for part (a), in $O(\text{sort}(h^2)) = O(h^2 \cdot \text{overhead}(h^2))$ time. Because $\text{overhead}(n) = O(\log n)$, we have $\text{overhead}(h^2) = O(\text{overhead}(h))$. Thus, the overall time for the first phase is $O(n \cdot \text{overhead}(h))$.
- Then for each convex hull edge, and for each of the $O(n/h^2)$ subhulls, we spend $O(\log h^2) = O(\log h)$ time computing a tangent line via modified binary search. The overall time for the second phase is $h \cdot O(n/h^2) \cdot O(\log h) = O((n/h) \log h) = O(n)$. Notice that this improvement does *not* depend on faster sorting.

Finally, we need to modify the schedule of estimates for h , so that the times for each guess define an increasing geometric series. In the i th iteration, we use the estimate

$$h_i = \min \{k \mid \text{overhead}(k) = 2^i\}.$$

For example (writing $a \uparrow b$ instead of a^b to avoid nested superscripts):

- If $\text{overhead}(n) = \lg n$, then $h_i = 2^{2 \uparrow i}$. This is the schedule used by Chan's original algorithm.
- If $\text{overhead}(n) = \sqrt{\lg n}$, then $h_i = 2^{4 \uparrow i}$.
- If $\text{overhead}(n) = \lg \lg n$, then $h_i = 2^{2 \uparrow 2 \uparrow i}$.
- If $\text{overhead}(n) = \sqrt{\lg \lg n}$, then $h_i = 2^{2 \uparrow 4 \uparrow i}$.

In the i th iteration of the main loop, we run the known- h algorithm using the estimate h_i , for $O(n \cdot \text{overhead}(h_i))$ time.^b The algorithm ends after t iterations, where $t = \min\{i \mid h_i \leq h\}$. The overall running time is

$$\sum_{i=1}^t O(n \cdot \text{overhead}(h_i)) = \sum_{i=1}^t O(n \cdot 2^i) = O(n \cdot 2^t) = O(n \cdot \text{overhead}(h_t))$$

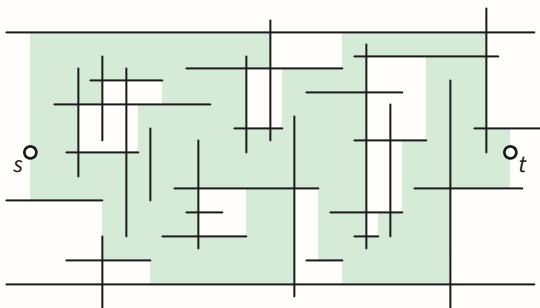
Because $h_{t-1} < h \leq h_t$, we have $2^{t-1} < \text{overhead}(h) \leq 2^t$, and therefore $\text{overhead}(h_t) < 2 \cdot \text{overhead}(h)$. We conclude that our modified algorithm runs in $O(n \cdot \text{overhead}(h))$ time, as required. ■

^aChan's original paper suggests using subsets of size $O(h \log h)$, which is the smallest size that makes the analysis work.

^bI am implicitly assuming that the *sort* function is well-behaved enough that we can compute h_i quickly. For all of the listed examples, we can compute h_i in $O(\log h_i)$ time by repeated squaring.

4. Suppose we are given a set S of n horizontal and vertical line segments in the plane, some of which may intersect, along with two points s and t . Describe an algorithm that determines whether there is an x -monotone path from s to t that does not intersect any segment in S .

Solution: First, let's call a point p in the plane *reachable* if there is an x -monotone path from s to p that does not intersect any input segments. The following figure shows an example input with reachable points colored green.



We use a plane-sweep algorithm to determine whether t is reachable. Let H and V denote the subsets of horizontal and vertical input segments, respectively. We sweep a vertical line ℓ from left to right, maintaining the sorted sequence of y -coordinates of intersection points $H \cap \ell$; for each interval between y -coordinates, we also maintain whether the corresponding segment of the sweep-line ℓ is reachable.

The algorithm begins with the sweep-line ℓ passing through s . We can compute the initial intersection points $H \cap \ell$ by brute force in $O(n)$ time, and only the segment containing s is reachable. As we move ℓ from left to right, the sweep-dictionary changes at three types of events:

- At the left endpoint p of a horizontal segment, we insert y_p into the sweep dictionary, splitting one interval into two smaller intervals. If the interval containing y_p was reachable, then both subintervals are reachable; otherwise both subintervals are unreachable.
- At the right endpoint q of a horizontal segment, we delete y_q from the sweep dictionary, merging two intervals into one. If at least one of the two intervals was reachable, their union is reachable; otherwise it is unreachable.
- At any vertical segment pq , every sweep interval that is completely contained in the interval $[y_p, y_q]$ becomes unreachable. The reachability of the interval(s) containing the endpoints y_p and y_q does not change.

Finally, when the sweep line reaches t , we return TRUE if and only if the interval containing y_t is reachable.

To process these changes quickly, we maintain the y -coordinates of $\ell \cap H$ in a balanced binary search tree T , where each node v stores a y -coordinate $v.y$, pointers $v.left$ and $v.right$ to its children, and a boolean $v.reach$ indicating whether the interval immediately above $v.y$ is reachable. (To handle the bottom interval, we can either

maintain its status separately or add a sentinel at $y = -\infty$.) We process the three event types as follows:

<u>LEFTEND(y):</u> $v_{old} \leftarrow \text{PRED}(T, y)$ $v_{new} \leftarrow \text{INSERT}(T, y)$ $v_{new}.reach \leftarrow v_{old}.reach$	<u>RIGHTEND(y):</u> $v_{dn} \leftarrow \text{PRED}(T, y)$ $v_{up} \leftarrow \text{FIND}(T, y)$ $v_{dn}.reach \leftarrow v_{dn}.reach \vee v_{up}.reach$ $\text{DELETE}(T, v_{up})$
-------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<u>VERTICAL(y^-, y^+):</u> $v_{bot} \leftarrow \text{PRED}(T, y^-)$ $v_{top} \leftarrow \text{PRED}(T, y^+)$ $v \leftarrow \text{PRED}(T, v_{top}.y)$ while $v \neq v_{bot}$ $v.reach \leftarrow \text{FALSE}$ $v \leftarrow \text{PRED}(T, v)$

The LEFTEND and RIGHTEND algorithms each run in $O(\log n)$ time, and VERTICAL runs in $O((k + 1) \log n)$ time, where k is the number of intersection points on that vertical segment. Summing over all events gives us a total running time of $O((n + K) \log n)$, where K is the total number of intersection points.

We can remove the dependence on K using a more sophisticated sweep dictionary. We add a bit $v.poison$ to each node v of the BST, which is FALSE by default. Setting $v.poison \leftarrow \text{TRUE}$ indicates that *all* intervals in the subtree rooted at v are unreachable, regardless of how their *reach* bits are set. Whenever we touch any node v for any reason, we first run the following algorithm:

<u>CLEAN(v):</u> if $v.poison = \text{TRUE}$ $v.reach \leftarrow \text{FALSE}$ $v.poison \leftarrow \text{FALSE}$ if $v.left \neq \text{NULL}$ $v.left.poison \leftarrow \text{TRUE}$ if $v.right \neq \text{NULL}$ $v.right.poison \leftarrow \text{TRUE}$

This cleaning phase allows every other algorithm that searches or updates the BST to act as though the poison bit doesn't exist, with only $O(1)$ time overhead.

Now we modify VERTICAL using splits and merges, as we did in the faster variant of the Bentley-Ottmann algorithm in class. Here SPLIT(T, y) splits T into two smaller BSTs T^- and T^+ , respectively containing all keys in T less than y and greater than y , and JOIN(T^-, T^+) is the inverse of SPLIT. Almost all balanced BSTs support these operations in $O(\log n)$ (possibly amortized or expected) time.

<u>NEWVERTICAL(y^-, y^+):</u> $T^-, T \leftarrow \text{SPLIT}(T, y^-)$ $T, T^+ \leftarrow \text{SPLIT}(T, y^+)$ $T.root.poison \leftarrow \text{TRUE}$ $T \leftarrow \text{JOIN}(T, T^+)$ $T \leftarrow \text{JOIN}(T^-, T)$

The resulting orthogonal reachability algorithm runs in $O(n \log n)$ time. ■