7 Geometric data structures

There is a rich landscape of data structures for handling geometric objects. Let us perform a small survey. First, there are different objects which can be stored in the structure: points in $\mathbb{R}^d$, lines in $\mathbb{R}^d$, or even more complex shapes like polygons, polytopes, conic sections or splines.

Queries are asked about all objects which lie within a given *region* (or intersect its boundary). Usually, the region is one of:

- *a single object* — test if the given object is stored in the structure
- *a range* — a d-dimensional "box" $[a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_d, b_d]$; some of the intervals can extend to infinity
- *partial match* — some parameters of the object are specified precisely, the remaining ones can be arbitrary. E.g., report all points in $\mathbb{R}^3$ for which $x = 3$ and $z = 5$. This is a special case of a range, where each interval is either a single point or the whole $\mathbb{R}$. Partial match queries frequently occur in database indices, even in otherwise non-geometric situations.
- *a polygon*

We might want to *enumerate* all objects within the region, or just *count* them without enumerating them one by one. If there is a value associated with each object, we can ask for a sum or a maximum of values of all objects within the range — this is generally called an *aggregation* query.

In this chapter, we will consider the simple case of range queries on points in $\mathbb{R}^d$.

7.1 Range queries in one dimension

We show the basic techniques on one-dimensional range queries. The simplest solution is to use a sorted array. Whenever we are given a query interval $[a, b]$, we can locate its endpoints in the array by binary search. Then we either enumerate the items between the endpoints or we count them by subtracting indices of the endpoints. We only have to be careful and check whether the endpoints lies at an item or in a gap between items. Overall, we can answer any query in $\mathcal{O}(\log n + p)$, where n is the number of items and p the number of points enumerated. The structure can be built in time $\mathcal{O}(n \log n)$ by sorting the items and it is stored in $\mathcal{O}(n)$ space.

Another solution uses binary search trees. It is more complicated, but more flexible. It can be made dynamic and it can also answer aggregation queries.

Definition: Let T be a binary search tree with real-valued keys. For each node v, we define the set $\text{int}(v)$ called the *interval of* v. It contains all real numbers whose search visits v.

Observation: We can see that the sets have the following properties:

- $\text{int}(\textit{root})$ is the whole set $\mathbb{R}$.
- If v is a node with key $\text{key}(v)$, left child $\ell(v)$ and right child $r(v)$, then:
  - $\text{int}(\ell(v)) = \text{int}(v) \cap (-\infty, \text{key}(v))$
  - $\text{int}(r(v)) = \text{int}(v) \cap (\text{key}(v), +\infty)$
- By induction, $\text{int}(v)$ is always an open interval.
- All keys in the subtree of v lie in $\text{int}(v)$.
- The definition of $\text{int}(v)$ applies to external nodes, too. The intervals obtained by cutting the real line to parts at the keys in internal nodes are exactly the intervals assigned to external nodes.

This suggests a straightforward recursive algorithm for answering range queries.

Procedure RangeQuery(v, Q)
Input: a root of a subtree v, query range Q

1. If v is external, return.
2. If $\text{int}(v) \subseteq Q$, report the whole subtree rooted at v and return.
3. If $\text{key}(v) \in Q$, report the item at v.
4. $Q_\ell \leftarrow Q \cap \text{int}(\ell(v))$, $Q_r \leftarrow Q \cap \text{int}(r(v))$
5. If $Q_\ell \neq \emptyset$: call RangeQuery($\ell(v), Q_\ell$)
6. If $Q_r \neq \emptyset$: call RangeQuery($r(v), Q_r$)

Let us analyze time complexity of this algorithm now.

Lemma: If the tree is balanced, RangeQuery called on its root visits $\mathcal{O}(\log n)$ nodes and subtrees.

Proof: Let $Q = [\alpha, \beta]$ be the query interval. Let a and b the tree nodes (internal or external) where search for α and β ends. We denote the lowest common ancestor of a and b by p.

Whenever we enter a node v with some interval $\text{int}(v)$, the key $\text{key}(v)$ splits the interval to two parts, corresponding to $\text{int}(\ell(v))$ and $\text{int}(r(v))$.

On the path from the root to p, Q always lies in one of these parts and we recurse on one child. In some cases, the current key lies in Q, so we report it.

When we enter the common ancestor p, the range Q lives in both parts, so we report key(p) and recurse on both parts.

On the ''left path'' from p to a, we encounter two situations. Either Q lies solely in the right part, so we recurse on it. Or Q crosses key(v), so we recurse on the left part and report the whole right subtree. Again, we report key(v) if it lies in Q.

The ''right path'' from p to b behaves symetrically: we recurse on the right part and possibly report the whole left subtree and/or the current key.

Since all paths contain $\mathcal{O}(\log n)$ nodes together, we visit $\mathcal{O}(\log n)$ nodes and report $\mathcal{O}(\log n)$ nodes and $\mathcal{O}(\log n)$ subtrees. □

Corollary: An enumeration query is answered in time $\mathcal{O}(\log n + p)$, where p is the number of items reported. If we precompute sizes of all subtrees, a counting query takes $\mathcal{O}(\log n)$ time. Aggregate queries can be answered if we precompute aggregate answers for all subtrees and combine them later. The structure can be built in $\mathcal{O}(n \log n)$ time using the algorithm for constructing perfectly balanced trees. It takes $\mathcal{O}(n)$ memory.

This query algorithm is compatible with most methods for balancing binary search trees. The interval int(v) need not be stored in the nodes — it can be computed on the fly when traversing the tree from the root. The subtree sizes or aggregate answers can be easily updated when rotating an edge: only the values in the endpoints of the edge are affected and they can be recomputed in constant time from values in their children. This way we can obtain a dynamic range tree with INSERT and DELETE in $\mathcal{O}(\log n)$ time.

7.2 Multi-dimensional search trees (k-d trees)

Binary search trees can be extended to k-dimensional trees, or simply k-d trees. Keys stored in nodes are points in $\mathbb{R}^d$, nodes at the ℓ-th level compare the ($\ell \bmod d$)-th coordinate. We will temporarily assume that no two points share the same value of any coordinate, so every point different from the node's key belongs either to the left subtree, or the right one.

Let us show a static 2-dimensional example. The root compares the x coordinate, so it splits the plane by a vertical line to two half-planes. The children of the root split their half-planes by horizontal lines to quarter-planes, and so on.

We can build a perfectly balanced k-d tree recursively. We place the point with median x coordinate in the root. We split the points to the two half-planes and recurse on each half-plane, which constructs the left and right subtree. During the recursion, we alternate

coordinates. As the number of points in the current subproblem decreases by a factor of two in every recursive call, we obtain a tree of height $\lceil \log n \rceil$.

Time complexity of building can be analyzed using the recursion tree: since we can find a median of m items in time $\mathcal{O}(m)$, we spend $\mathcal{O}(n)$ time per tree level. We have $\mathcal{O}(\log n)$ levels, so the whole construction runs in $\mathcal{O}(n \log n)$ time. Since every node of the 2-d tree contains a different point, the whole tree consumes $\mathcal{O}(n)$ space.

We can answer 2-d range queries similarly to the 1-d case. To each node v of the tree, we can assign a 2-d interval $\text{int}(v)$ recursively. This again generates a hierarchy of nested intervals, so the RangeQuery algorithm works there, too. However, 2-d range queries can be very slow in the worst case:

Lemma: Worst-case time complexity of range queries in a 2-d tree is $\Omega(\sqrt{n})$.

Proof: Consider a tree built for the set of points $\{(i, i) \mid 1 \leq i \leq n\}$ for $n = 2^t - 1$. It is a complete binary tree with t levels. Let us observe what happens when we query a range $\{0\} \times \mathbb{R}$ (unbounded intervals are not necessary for our construction, a very narrow and very high box would work, too).

On levels where the x coordinate is compared, we always go to the left. On levels comparing y, both subtrees lie in the query range, so we recurse on both of them. This means that the number of visited nodes doubles at every other level, so at level t we visit $2^{t/2} \approx \sqrt{n}$ nodes. □

Note: This is the worst which can happen, so query complexity is $\Theta(\sqrt{n})$. General k-d trees for arbitrary k can be built in time $\mathcal{O}(n \log n)$, they require space $\mathcal{O}(n)$, and they answer range queries in time $\mathcal{O}(n^{1-1/d})^{\langle 1 \rangle}$ This is quite bad in high dimension, but there is a matching lower bound for structures which fit in linear space.

Repeated values of coordinates can be handled by allowing a third child of every node, which is the root of a $(k-1)$-d subtree. There we store all points which share the compared coordinate with the parent's key. This extension does not influence asymptotic complexity of operations.

Dynamization is non-trivial and we will not show it.

$^{\langle 1 \rangle}$ Constants hidden in the $\mathcal{O}$ depend on the dimension.

7.3 Multi-dimensional range trees

The k-dimensional search trees were simple, but slow in the worst case. There is a more efficient date structure: the *multi-dimensional range tree,* which has poly-logarithmic query complexity, if we are willing to use super-linear space.

2-dimensional range trees

For simplicity, we start with a static 2-dimensional version and we will assume that no two points have the same x coordinate.

The 2-d range tree will consist of multiple instances of a 1-d range tree, which we built in section 7.1 — it can be a binary search tree with range queries, but in the static case even a sorted array suffices.

First we create an x-tree, which is a 1-d range over the x coordinates of all points stored in the structure. Each node contains a single point. Its subtree corresponds to an open interval of x coordinates, that is a *band* in $\mathbb{R}^2$ (an open rectangle which is vertically unbounded). For every band, we construct a y-tree containing all points in the band ordered by the y coordinate.

If the x-tree is balanced, every node lies in $\mathcal{O}(\log n)$ subtrees. So every point lies in $\mathcal{O}(\log n)$ bands and the whole structure takes $\mathcal{O}(n \log n)$ space: $\mathcal{O}(n)$ for the x-tree, $\mathcal{O}(n \log n)$ for all y-trees.

We can build the 2-d tree recursively. First we create two lists of points: one sorted by the x coordinate, one by y. Then we construct the x-tree. We find the point with median x coordinate in constant time. This point becomes the root of the x-tree. We recursively construct the left subtree from points with less than median x coordinate — we can construct the corresponding sub-lists of both the x and y list in $\mathcal{O}(n)$ time. We construct the right subtree similarly. Finally, we build the y-tree for the root: it contains all the points and we can build it from the y-sorted array in $\mathcal{O}(n)$ time.

The whole building algorithm requires linear time per sub-problem, which sums to $\mathcal{O}(n)$ over one level of the x-tree. Since the x-tree is logarithmically high, it makes $\mathcal{O}(n \log n)$ for the whole construction.

Now we describe how to answer a range query for $[x_1, x_2] \times [y_1, y_2]$. First we let the x-tree answer a range query for $[x_1, x_2]$. This gives us a union of $\mathcal{O}(\log n)$ points and bands which disjointly cover $[x_1, x_2]$. For each point, we test if its y coordinate lies in $[y_1, y_2]$. For each band, we ask the corresponding y-tree for points in the range $[y_1, y_2]$. We spend $\mathcal{O}(\log n)$ time in the x-tree, $\mathcal{O}(\log n)$ time to process the individual points, $\mathcal{O}(\log n)$ in each y-tree, and $\mathcal{O}(1)$ per point reported. Put together, this is $\mathcal{O}(\log^2 n + p)$ if p points are reported ($p = 0$ for a counting query).

Handling repeated coordinates

We left aside the case of multiple points with the same x coordinate. This can be handled by attaching another y-tree to each x-tree node, which contains nodes sharing the same x coordinate. That is, x-tree nodes correspond to distinct x coordinates and each has two y-trees: one for its own x coordinate, one for the open interval of x coordinates covering its subtree. This way, we can perform range queries for both open and closed ranges.

Time complexity of BUILD stays asymptotically the same: the maximum number of y-trees containing a given point increases twice, so it is still $\mathcal{O}(\log n)$. Similarly for range queries: we query at most twice as much y-trees.

Multi-dimensional generalization

The 2-d range tree is easily generalized to multiple dimensions. We build a 1-d tree on the first coordinate (usually called the *primary tree*). Each node defines a hyperplane of all points sharing the given first coordinate and a d-dimensional band (the Cartesian product of an open interval and $\mathbb{R}^{d-1}$). For each hyperplane and each point, we build a secondary tree, which is a $(d-1)$-dimensional range tree.

Since every point lies in $\mathcal{O}(\log n)$ secondary trees, memory consumption is $O(\log n)$ times larger than for a $(d-1)$-dimensional tree. Therefore the d-dimensional tree requires $\mathcal{O}(n \log^{d-1} n)$ space, where constants hidden in the $\mathcal{O}$ depend on d.

The data structure can be built by induction on dimension. We keep all points in d separate list, each sorted by a different coordinate. We build the primary tree as a perfectly balanced tree. In each node, we construct the $(d-1)$-dimensional secondary trees recursively. Since each point participates in $\mathcal{O}(\log n)$ secondary trees, each additional dimension multiplies time complexity by $\mathcal{O}(\log n)$. So a d-dimensional tree can be built in $\mathcal{O}(n \log^{d-1} n)$ time.

Range queries are again performed by recursion on dimension. Given a d-dimensional range $[a_1, b_1] \times \ldots \times [a_d, b_d]$, we first query the primary tree for the range $[a_1, b_1]$. This results in $\mathcal{O}(\log n)$ nodes and subtrees, whose secondary trees we ask for $[a_2, b_2] \times \ldots \times [a_d, b_d]$. Again, each additional dimensions slows queries down by a factor of $\mathcal{O}(\log n)$. Therefore a d-dimensional query requires $\mathcal{O}(\log^d n)$ time.

Dynamization

What if we want to make d-dimensional range trees dynamic? As usual, let us consider the 2-d case first. We could try using one of the many dynamic balanced binary search trees for 1-d trees. Alas, this is doomed to fail: if we rotate an edge in the x-tree, almost all points can change their bands, so in the worst case we need $\Theta(n)$ time to rebuild the corresponding y-trees.

However, rebuilding of subtrees comes relatively cheap: we can rebuild a single y-tree in linear time (we can collect the points in sorted order by recursion) and a m-node subtree of the x-tree in $\mathcal{O}(m \log m)$ time (we collect points in x order from the x-tree, in y order from the topmost y-tree and re-run Build).

We will use local rebuilds to keep the whole 2-d tree (and all its constituent 1-d trees) balanced. We will simply use the 1-d lazy weight-balanced trees of section **??** for all x-trees and y-trees.

Suppose that we want to Insert a new point. This causes up to one insertion to the x-tree (if the x coordinate was not used yet) and $\mathcal{O}(\log n)$ insertions to the y-trees. Every insertion to a y-tree can cause rebalancing of that tree, which costs $\mathcal{O}(\log n)$ amortized per tree, so $\mathcal{O}(\log^2 n)$ altogether. Insertion to the x-tree can cause rebalancing of the x-tree, which is more costly since rebuilding a subtree requires $\mathcal{O}(n \log n)$ time instead of $\mathcal{O}(n)$. So we have to spend $\mathcal{O}(\log^2 n)$ amortized to pre-pay this cost.

We have therefore shown that saving $\mathcal{O}(\log^2 n)$ time per Insert pays for all rebalancing of the constituent 1-d trees. The same argument applies to Delete. The whole argument can be extended by induction to d-d trees.

Let us summarize time complexity of all operations:

Build	$\mathcal{O}(n \log^{d-1} n)$
RangeQuery	$\mathcal{O}(\log^d n + p)$ for p reported points
Insert	$\mathcal{O}(\log^d n)$ amortized
Delete	$\mathcal{O}(\log^d n)$ amortized

The whole structure fits in $\mathcal{O}(n \log^{d-1} n)$ words of memory.

Fractional cascading

Time complexity of 2-d queries can be improved to $\mathcal{O}(\log n + p)$ by precomputing a little bit more. This translates to $\mathcal{O}(\log^{d-1} n + p)$ time per query in d-d case. We will show a static version, leaving dynamization as an exercise.

In the static case, y-trees can be simply sorted arrays. If we want to search for the range $[x_1, x_2] \times [y_1, y_2]$, we search the x-tree for $[x_1, x_2]$, which tells us to binary search for $[y_1, y_2]$ in $\mathcal{O}(\log n)$ sorted arrays. Each binary search per se requires $\mathcal{O}(\log n)$ time, but we can observe that these searches are somewhat correlated: each list is a sub-list of the list belonging to the parent node in the x-tree.

Whenever we have an array A in some node and its sub-array B in a child node, we precompute auxiliary pointers from A to B. For each element $a \in A$, we store its *predecessor*

in B: the index of the largest element $b \in B$ such that $b \leq a$. If a itself is in B, the predecessor of a is simply its occurrence in b; otherwise, it is the next smaller element. To make this always defined, we will place $-\infty$ at the beginning of every array. It is obvious that the predecessors can be computed in time linear with the length of A and B, so their construction does not slow down BUILD asymptotically.

Now, consider a query for $[x_1, x_2] \times [y_1, y_2]$. We walk the x-tree from the root and visit nodes and/or bands which together cover $[x_1, x_2]$. For each visited node, we calculate the indices in the node's y-array which correspond to the boundary of $[y_1, y_2]$ (if the boundary values are not present, we take the next value towards the interior of the interval).

As we know the positions of the boundary in the parent array, we can use the pre-computed predecessor pointers to update the positions to the child array in constant time. We simply use the predecessor pointers and shift the index by one position in the right direction if needed.

We therefore need $\mathcal{O}(\log n)$ for the initial binary search in the root y-array and $\mathcal{O}(\log n)$ steps per $\mathcal{O}(1)$ time in the subsequent y-arrays. This is $\mathcal{O}(\log n)$ altogether.

This trick is an instance of a more general technique called *fractional cascading,* which is generally used to find a given value in a collection of sorted lists, which need not be sub-lists of one another.

Exercises

1.* Show how to dynamize 2-d range trees with fractional cascading.