CHAPTER 2

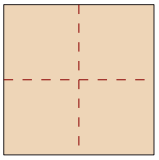
Quadtrees – Hierarchical Grids

In this chapter, we discuss quadtrees, arguably one of the simplest and most powerful geometric data-structures. We begin in Section 2.1 by defining quadtrees and giving a simple application. We also describe a clever way for performing point-location queries quickly in such quadtrees. In Section 2.2, we describe how such quadtrees can be compressed, constructed quickly, and be used for point-location queries. In Section 2.3, we show how to dynamically maintain a compressed quadtree under insertions and deletions of points.

2.1. Quadtrees – a simple point-location data-structure

Let P_{map} be a planar map. To be more concrete, let P_{map} be a partition of the unit square into polygons. The partition P_{map} can represent any planar map where a region in the map might be composed of several polygons (or triangles). For the sake of simplicity, assume that every vertex in P_{map} appears in a constant number of polygons.

We want to preprocess P_{map} for point-location queries, so that given a query point, we can figure out which polygon contains the query (see the figure on the right). Of course, there are numerous data-structures that can do this, but let us consider the following simple solution (which in the worst case, can be quite bad).

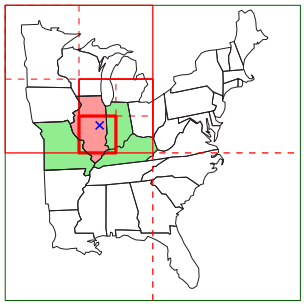


Build a tree $\mathcal{T}$, where every node $v \in T$ corresponds to a cell $\square_v$ (i.e., a square) and the root corresponds to the unit square. Each node has four children that correspond to the four equal sized squares formed by splitting $\square_v$ by horizontal and vertical cuts; see the figure on the right.

The construction is recursive, and we start from $v = \text{root}_T$. The ***conflict list*** of the square $\square_v$ (i.e., the square associated with v) is a list of all the polygons of P_{map} that intersect $\square_v$. If the current node's conflict list has more than, say, nine[①] polygons, we create its children nodes, and we call recursively on each child. We compute each child's conflict list from its parent list. As such, we stop at a leaf if its conflict list is of size at most nine. For each constructed leaf, we store in it its conflict list (but we do not store the conflict list for internal nodes).

Given a query point q, in the unit square, we can compute the polygon of P_{map} containing q by traversing down $\mathcal{T}$ from the root, repeatedly going into the child of the current node, whose square contains q. We stop as soon as we reach a leaf, and then we scan the leaf's conflict list and check which of the polygons in this list contains q.

[①] The constant here is arbitrary. It just has to be at least the number of polygons of P_{map} meeting in one common corner.



An example is depicted on the right, where the nodes on the search path of the point-location query (i.e., the nodes accessed during the query execution) are shown. The marked polygons are all the polygons in the conflict list of the leaf containing the query point.

Of course, in the worst case, if the polygons are long and skinny, this quadtree might have unbounded complexity. However, for reasonable inputs (say, the polygons are all fat triangles), then the quadtree would have linear complexity in the input size (see Exercise 2.2). The major advantage of quadtrees, of course, is their simplicity. In a lot of cases, quadtrees would be a sufficient solution, and seeing how to solve a problem using quadtrees might be a first insight into a problem.

2.1.1. Fast point-location in a quadtree. One possible interpretation of quadtrees is that they are a multi-grid representation of a point set.

DEFINITION 2.1 (Canonical squares and canonical grids). A square is a ***canonical square*** if it is contained inside the unit square, it is a cell in a grid G_r, and r is a power of two (i.e., it might correspond to a node in a quadtree). We will refer to such a grid G_r as a ***canonical grid***.

Consider a node v of a quadtree of depth i (the root has depth 0) and its associated square $\square_v$. The sidelength of $\square_v$ is 2^{-i}, and it is a canonical square in the canonical grid $\mathsf{G}_{2^{-i}}$. As such, we will refer to $\ell(v) = -i$ as the ***level*** of v. However, a cell in a grid has a unique ID made out of two integer numbers. Thus, a node v of a quadtree is uniquely defined by the triple $\mathrm{id}(v) = (\ell(v), \lfloor x/r \rfloor, \lfloor y/r \rfloor)$, where (x, y) is any point in $\square_v$ and $r = 2^{\ell(v)}$.

Furthermore, given a query point q and a desired level ℓ, we can compute the ID of the quadtree cell of this level that contains q in constant time. Thus, this suggests a natural algorithm for doing a point-location in a quadtree: Store all the IDs of nodes in the quadtree in a hash-table, and also compute the maximal depth h of the quadtree. Given a query point q, we now have access to any node along the point-location path of q in $\mathcal{T}$, in constant time. In particular, we want to find the point in $\mathcal{T}$ where this path "falls off" the quadtree (i.e., reaches the leaf). This we can find by performing a binary search for the leaf.

```
QTFastPLI(T, q, l, h).
    m ← ⌊(l + h)/2⌋
    v ← QTGetNode(T, q, m)
    if v = null then
        return QTFastPLI(T, q, l, m − 1).
    w ← Child(v, q)
          //w is the child of v containing q.
    If w = null then
        return v
    return QTFastPLI(T, q, m + 1, h)
```

FIGURE 2.1. One can perform point-location in a quadtree $\mathcal{T}$ by calling QTFastPLI$(T, q, 0, \mathrm{height}(T))$.

Let **QTGetNode**(T, q, d) denote the procedure that, in constant time, returns the node v of depth d in the quadtree $\mathcal{T}$ such that $\square_v$ contains the point q. Given a query point q, we can perform point-location in $\mathcal{T}$ (i.e., find the leaf containing the query) by calling **QTFastPLI**$(T, \mathsf{q}, 0, \mathrm{height}(T))$. See Figure 2.1 for the pseudo-code for **QTFastPLI**.

LEMMA 2.2. *Given a quadtree $\mathcal{T}$ of size n and of height h, one can preprocess it (using hashing), in linear time, such that one can perform a point-location query in $\mathcal{T}$ in $O(\log h)$*

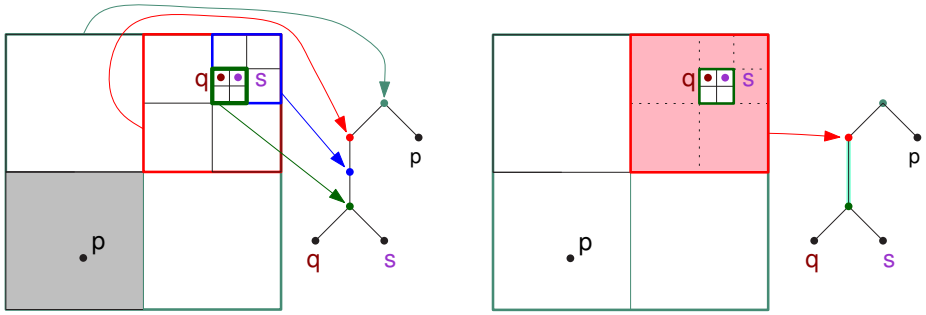


FIGURE 2.2. A point set, its quadtree and its compressed quadtree. Note that each node is associated with a canonical square. For example, the node in the above quadtree marked by p would store the gray square shown on the left and also would store the point p in this node.

time. In particular, if the quadtree has height $O(\log n)$ (i.e., it is "balanced"), then one can perform a point-location query in $\mathcal{T}$ in $O(\log \log n)$ time.

2.2. Compressed quadtrees

2.2.1. Definition.

DEFINITION 2.3 (Spread). For a set P of n points in a metric space, let

$$\Phi(\mathsf{P}) = \frac{\max_{\mathsf{p},\mathsf{q}\in\mathsf{P}} \|\mathsf{p}-\mathsf{q}\|}{\min_{\mathsf{p},\mathsf{q}\in\mathsf{P},\mathsf{p}\neq\mathsf{q}} \|\mathsf{p}-\mathsf{q}\|} \tag{2.1}$$

be the ***spread*** of P. In words, the spread of P is the ratio between the diameter of P and the distance between the two closest points in P. Intuitively, the spread tells us the range of distances that P possesses.

One can build a quadtree $\mathcal{T}$ for P, storing the points of P in the leaves of $\mathcal{T}$, where one keeps splitting a node as long as it contains more than one point of P. During this recursive construction, if a leaf contains no points of P, we save space by not creating this leaf and instead creating a null pointer in the parent node for this child.

LEMMA 2.4. *Let P be a set of n points contained in the unit square, such that* $\operatorname{diam}(\mathsf{P}) = \max_{\mathsf{p},\mathsf{q}\in\mathsf{P}} \|\mathsf{p}-\mathsf{q}\| \geq 1/2$. *Let $\mathcal{T}$ be a quadtree of P constructed over the unit square, where no leaf contains more than one point of P. Then, the depth of $\mathcal{T}$ is bounded by $O(\log \Phi)$, it can be constructed in $O(n \log \Phi)$ time, and the total size of $\mathcal{T}$ is $O(n \log \Phi)$, where $\Phi = \Phi(\mathsf{P})$.*

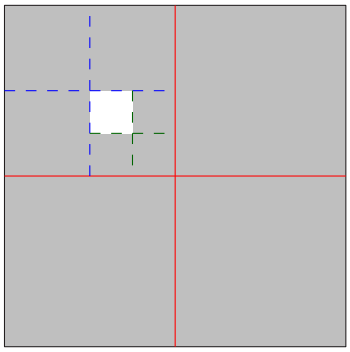
PROOF. The construction is done by a straightforward recursive algorithm as described above.

Let us bound the depth of $\mathcal{T}$. Consider any two points $\mathsf{p}, \mathsf{q} \in \mathsf{P}$, and observe that a node v of $\mathcal{T}$ of level $u = \lfloor \lg \|\mathsf{p}-\mathsf{q}\| \rfloor - 1$ containing p must not contain q (we remind the reader that $\lg n = \log_2 n$). Indeed, the diameter of $\square_v$ is smaller than $\sqrt{1^2+1^2}2^u = \sqrt{2}2^u \leq \sqrt{2}\|\mathsf{p}-\mathsf{q}\|/2 < \|\mathsf{p}-\mathsf{q}\|$. Thus, $\square_v$ cannot contain both p and q. In particular, any node u of $\mathcal{T}$ of level $r = -\lceil \lg \Phi \rceil - 2$ can contain at most one point of P. Since a node of the quadtree containing a single point of P is a leaf of the quadtree, it follows that all the nodes of $\mathcal{T}$ are of depth $O(\log \Phi)$.

Since the construction algorithm spends $O(n)$ time at each level of $\mathcal{T}$, it follows that the construction time is $O(n \log \Phi)$, and this also bounds the size of the quadtree $\mathcal{T}$. ■

The bounds of Lemma 2.4 are tight, as one can easily verify; see Exercise 2.3. But in fact, if you inspect a quadtree generated by Lemma 2.4, you would realize that there are a lot of nodes of $\mathcal{T}$ which are of degree one (the *degree* of a node is the number of children it has). See Figure 2.2 for an example. Indeed, a node v of $\mathcal{T}$ has more than one child only if it has at least two children x and y, such that both $\square_x$ and $\square_y$ contain points of P. Let P_v be the subset of points of P stored in the subtree of v, and observe that $\mathsf{P}_x \cup \mathsf{P}_y \subseteq \mathsf{P}_v$ and $\mathsf{P}_x \cap \mathsf{P}_y = \emptyset$. Namely, such a node v splits P_v into at least two non-empty subsets and globally there can be only $n-1$ such splitting nodes. Thus, a regular quadtree might have a large number of "useless" nodes that one should be able to get rid of and get a more compact data-structure.

If you compress them, they will fit in memory. We can replace such a sequence of edges by a single edge. To this end, we will store inside each quadtree node v its square $\square_v$ and its level $\ell(v)$. Given a path of vertices in the quadtree that are all of degree one, we will replace them with a single vertex v that corresponds to the first vertex in this path, and its only child would be the last vertex in this path (this is the first node of degree larger than one). This ***compressed node*** v has a single child, and the region rg_v that v "controls" is an annulus, seen as the gray area in the figure on the right.



Otherwise, if v is not compressed, the ***region*** that v is in charge of is a square $\mathsf{rg}_v = \square_v$. Specifically, an uncompressed node v is either a leaf or a node that splits the point set into two (or more) non-empty subsets (i.e., two or more of the children of v have points stored in their subtrees). For a compressed node v, its sole child corresponds to the inner square, which contains all the points of P_v. We call the resulting tree a ***compressed quadtree***.

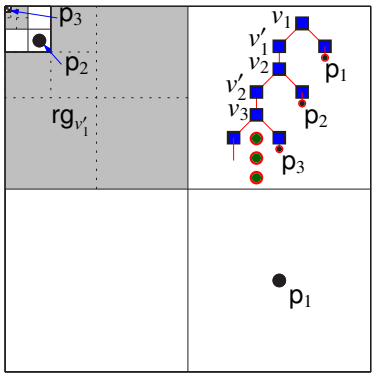


FIGURE 2.3

Observe that the only nodes in a compressed quadtree with a single child are compressed (they might be trivially compressed, the child being one of the four subsquares of the parent). In particular, since any node that has only a single child is compressed, we can charge it to its parent, which has two (or more) children. There are at most $n-1$ internal nodes in the new compressed quadtree that have degree larger than one, since one can split a set of size n at most $n-1$ times till ending up with singletons. As such, a compressed quadtree has linear size (however, it still can have linear depth in the worst case).

See Figure 2.2 for an example of a compressed quadtree.

EXAMPLE 2.5. As an application for compressed quadtrees, consider the problem of reporting the points that are inside a query rectangle r. We start from the root of the quadtree and recursively traverse it, going down a node only if its region intersects the query rectangle. Clearly, we will report all the points contained inside r. Of course, we have no guarantee about the query time, but in practice, this might be fast enough.

Note that one can perform the same task using a regular quadtree. However, in this case, such a quadtree might require unbounded space. Indeed, the spread of the point set might be arbitrarily large, and as such the depth (and thus size) of the quadtree can be arbitrarily large. On the other hand, a compressed quadtree would use only $O(n)$ space, where n is the number of points stored in it.

EXAMPLE 2.6. Consider the point set $\mathsf{P} = \{\mathsf{p}_1, \ldots, \mathsf{p}_n\}$, where $\mathsf{p}_i = (3/4, 3/4)/8^{i-1}$, for $i = 1, \ldots, n$. The compressed quadtree for this point set is depicted in Figure 2.3.

Here, we have $\square_{v_i} = [0,1]^2/8^{i-1}$ and $\square_{v'_i} = [0,1]^2/(2 \cdot 8^{i-1})$. As such, v'_i is a compressed node, where

$$\mathsf{rg}_{v'_i} = \square_{v'_i} \setminus \square_{v_{i+1}}.$$

Note that this compressed quadtree has depth and size $\Theta(n)$. In particular, in this case, the compressed quadtree looks like a linked list storing the points.

2.2.2. Efficient construction of compressed quadtrees.

2.2.2.1. *Bit twiddling and compressed quadtrees.* Unfortunately, to be able to efficiently build compressed quadtrees, one requires a somewhat bizarre computational model. We are assuming implicitly the unit RAM model, where one can store and manipulate arbitrarily large real numbers in constant time. To work with grids efficiently, we need to be able to compute quickly (i.e., constant time) $\lg(x)$ (i.e., $\log_2$), 2^x, and $\lfloor x \rfloor$. Strangely, computing a compressed quadtree efficiently is equivalent to the following operation.

DEFINITION 2.7 (Bit index). Let $\alpha, \beta \in [0,1)$ be two real numbers. Assume these numbers in base two are written as $\alpha = 0.\alpha_1\alpha_2\ldots$ and $\beta = 0.\beta_1\beta_2\ldots$. Let $\mathrm{bit}_\Delta(\alpha,\beta)$ be the index of the first bit after the period in which they differ.

For example, $\mathrm{bit}_\Delta(1/4 = 0.01_2, 3/4 = 0.11_2) = 1$ and $\mathrm{bit}_\Delta(7/8 = 0.111_2, 3/4 = 0.110_2) = 3$.

LEMMA 2.8. *If one can compute a compressed quadtree of two points in constant time, then one can compute* $\mathrm{bit}_\Delta(\alpha,\beta)$ *in constant time.*

PROOF. Given α and β, we need to compute $\mathrm{bit}_\Delta(\alpha,\beta)$. Now, if $|\alpha - \beta| > 1/128$, then we can compute $\mathrm{bit}_\Delta(\alpha,\beta)$ in constant time. Otherwise, build a one-dimensional compressed quadtree (i.e., a compressed trie) for the set $\{\alpha,\beta\}$. The root is a compressed node in this tree, and its only child[②] v has sidelength 2^{-i}; that is, $\ell(v) = -i$. A quadtree node stores the canonical grid cell it corresponds to, and as such $\ell(v)$ is available from the compressed quadtree. As such, α and β are identical in the first i bits of their binary representation, but clearly they differ at the $(i+1)$st bit, as the next level of the quadtree splits them into two different subtrees. As such, if one can compute a compressed trie of two numbers in constant time, then one can compute $\mathrm{bit}_\Delta(\alpha,\beta)$ in constant time.

If the reader is uncomfortable with building a one-dimensional compressed quadtree, then use the point set $\mathsf{P} = \{(\alpha, 1/3), (\beta, 1/3)\}$, and compute a compressed quadtree $\mathcal{T}$ for P having the unit square as the root. Similar argumentation would apply in this case. ■

Interestingly, once one has such an operation at hand, it is quite easy to compute a compressed quadtree efficiently via "linearization". The idea is to define an order on the nodes of a compressed quadtree and maintain the points sorted in this order; see Section 2.3 below for details. Given the points sorted in this order, one can build the compressed quadtree in linear time using, essentially, scanning.

[②]The root is Chinese and is not allowed to have more children at this point in time.

However, the resulting algorithm is somewhat counterintuitive. As a first step, we suggest a direct construction algorithm.

2.2.2.2. *A construction algorithm.* Let P be a set of n points in the unit square, with unbounded spread. We are interested in computing the compressed quadtree of P. The regular algorithm for computing a quadtree when applied to P might require unbounded time (but in practice it might be fast enough). Modifying it so that it requires only quadratic time is an easy exercise. Getting down to $O(n \log n)$ time requires some cleverness.

THEOREM 2.9. *Given a set P of n points in the plane, one can compute a compressed quadtree of P in $O(n \log n)$ deterministic time.*

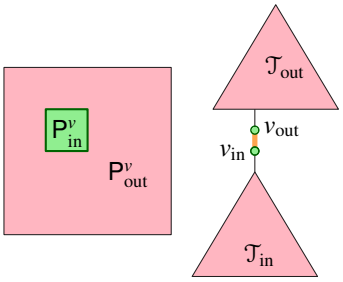
PROOF. Compute, in linear time, a disk D of radius r, which contains at least $n/10$ of the points of P, such that $r \leq 2r_{\mathrm{opt}}(\mathsf{P}, n/10)$, where $r_{\mathrm{opt}}(\mathsf{P}, n/10)$ denotes the radius of the smallest disk containing $n/10$ points of P. Computing D can be done in linear time, by a rather simple algorithm (see Lemma 1.16_{p6}).

Let $\alpha = 2^{\lfloor \lg r \rfloor}$. Consider the grid G_α. It has a cell that contains at least $(n/10)/25$ points of P (since D is covered by $5 \times 5 = 25$ grid cells of G_α and $\alpha \geq r/2$), and no grid cell contains more than $5(n/10)$ points, by Lemma 1.19_{p7}. Thus, compute $\mathsf{G}_\alpha(\mathsf{P})$, and find the cell $\square$ containing the largest number of points. Let P_{in} be the set of points inside this cell $\square$, and let $\mathsf{P}_{\mathrm{out}}$ be the set of points outside this cell. Specifically, we have

$$\mathsf{P}_{\mathrm{in}} = \mathsf{P} \cap \square \quad \text{and} \quad \mathsf{P}_{\mathrm{out}} = \mathsf{P} \setminus \square = \mathsf{P} \setminus \mathsf{P}_{\mathrm{in}}.$$

We know that $\left|\mathsf{P}_{\mathrm{in}}\right| \geq n/250$ and $\left|\mathsf{P}_{\mathrm{out}}\right| \geq n/2$.

Next, compute (recursively) the compressed quadtrees for P_{in} and $\mathsf{P}_{\mathrm{out}}$, respectively, and let $\mathcal{T}_{\mathrm{in}}$ and $\mathcal{T}_{\mathrm{out}}$ denote the respective quadtrees. Create a node in both quadtrees that corresponds to $\square$. For $\mathcal{T}_{\mathrm{in}}$ this would just be the root node v_{in}, since $\mathsf{P}_{\mathrm{in}} \subseteq \square$. For $\mathcal{T}_{\mathrm{out}}$ this would be a new leaf node v_{out}, since $\mathsf{P}_{\mathrm{out}} \cap \square = \emptyset$. Note that inserting this new node might require linear time, but it requires only changing a constant number of nodes and pointers in both quadtrees. Now, hang v_{out} on v_{in} creating a compressed quadtree for P; see the figure on the right. Observe that the new glued node (i.e., v_{out} and v_{in}) might be redundant because of compression, and it can be removed if necessary in constant time.

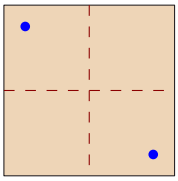


The overall construction time is $T(n) = O(n) + T\left(\left|\mathsf{P}_{\mathrm{in}}\right|\right) + T\left(\left|\mathsf{P}_{\mathrm{out}}\right|\right) = O(n \log n)$. ■

REMARK 2.10. The reader might wonder where the Tweedledee and Tweedledum operation (i.e., $\mathrm{bit}_\Delta(\cdot,\cdot)$) gets used in the algorithm of Theorem 2.9. Observe that the hanging stage might require such an operation if we hang $\mathcal{T}_{\mathrm{in}}$ from a compressed node of $\mathcal{T}_{\mathrm{out}}$, as computing the new compressed node requires exactly this kind of operation.

Compressed quadtree from squares. It is sometime useful to be able to construct a compressed quadtree from a list of squares that must appear in it as nodes.

LEMMA 2.11. *Given a list $\mathcal{C}$ of n canonical squares, all lying inside the unit square, one can construct a (minimal) compressed quadtree $\mathcal{T}$ such that for any square $c \in \mathcal{C}$, there exists a node $v \in T$, such that $\square_v = c$. The construction time is $O(n \log n)$.*



PROOF. For every *canonical* square $\square \in C$, we place two points into a set P, such that any quadtree for P must have $\square$ in it. This is done by putting two points in $\square$ such that they belong to two different subsquares of $\square$. See the figure on the right.

The resulting point set P has $2n$ points, and we can compute its compressed quadtree $\mathcal{T}$ in $O(n \log n)$ time using Theorem 2.9. Observe that any cell of C is an internal node of $\mathcal{T}$. Thus trimming away all the leaves of the quadtree results in a minimal quadtree that contains all the cells of C as nodes. ■

2.2.3. Fingering a compressed quadtree – fast point-location. Let $\mathcal{T}$ be a compressed quadtree of size n. We would like to preprocess it so that given a query point, we can find the lowest node of $\mathcal{T}$ whose cell contains a query point q. As before, we can perform this by traversing down the quadtree, but this might require $\Omega(n)$ time. Since the range of levels of the quadtree nodes is unbounded, we can no longer use a binary search on the levels of $\mathcal{T}$ to answer the query.

Instead, we are going to use a rebalancing technique on $\mathcal{T}$. Namely, we are going to build a balanced tree $\mathcal{T}'$, which would have cross pointers (i.e., fingers) into $\mathcal{T}$. The search would be performed on $\mathcal{T}'$ instead of on $\mathcal{T}$. In the literature, the tree $\mathcal{T}'$ is known as a ***finger tree***.

DEFINITION 2.12. Let $\mathcal{T}$ be a tree with n nodes. A ***separator*** in $\mathcal{T}$ is a node v, such that if we remove v from $\mathcal{T}$, we remain with a forest, such that every tree in the forest has at most $\lceil n/2 \rceil$ vertices.

LEMMA 2.13. *Every tree has a separator, and it can be computed in linear time.*

PROOF. Consider $\mathcal{T}$ to be a rooted tree. We precompute for every node in the tree the number of nodes in its subtree by a bottom-up computation that takes linear time. Set v to be the lowest node in $\mathcal{T}$ such that its subtree has $\geq \lceil n/2 \rceil$ nodes in it, where n is the number of nodes of $\mathcal{T}$. This node can be found by performing a walk from the root of $\mathcal{T}$ down to the child with a sufficiently large subtree till this walk gets "stuck". Indeed, let v_1 be the root of $\mathcal{T}$, and let v_i be the child of v_{i-1} with the largest number of nodes in its subtree. Let $s(v_i)$ be the number of nodes in the subtree rooted at v_i. Clearly, there exists a k such that $s(v_k) \geq n/2$ and $s(v_{k+1}) < n/2$.

Clearly, all the subtrees of the children of v_k have size at most $n/2$. Similarly, if we remove v_k and its subtree from $\mathcal{T}$, we remain with a tree with at most $n/2$ nodes. As such, v_k is the required separator. ■

This suggests a natural way for processing a compressed quadtree for point-location queries. Find a separator $v \in T$, and create a root node f_v for $\mathcal{T}'$ which has a pointer to v; now recursively build a finger tree for each tree of $T \setminus \{v\}$. Hang the resulting finger trees on f_v. The resulting tree is the required finger tree $\mathcal{T}'$.

Given a query point q, we traverse $\mathcal{T}'$, where at node $f_v \in \mathcal{T}'$, we check whether the query point $\mathsf{q} \in \square_v$, where v is the corresponding node of $\mathcal{T}$. If $\mathsf{q} \notin \square_v$, we continue the search into the child of f_v, which corresponds to the connected component outside $\square_v$ that was hung on f_v. Otherwise, we continue into the child that contains q; naturally, we have to check out the $O(1)$ children of v to decide which one we should continue the search into. This takes constant time per node. As for the depth for the finger tree $\mathcal{T}'$, observe $D(n) \leq 1 + D(\lceil n/2 \rceil) = O(\log n)$. Thus, a point-location query in $\mathcal{T}'$ takes logarithmic time.

THEOREM 2.14. *Given a compressed quadtree $\mathcal{T}$ of size n, one can preprocess it, in time $O(n \log n)$, such that given a query point q, one can return the lowest node in $\mathcal{T}$ whose region contains q in $O(\log n)$ time.*

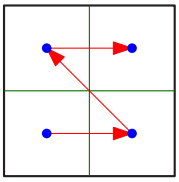
PROOF. We just need to bound the preprocessing time. Observe that it is $T(n) = O(n) + \sum_{i=1}^{t} T(n_i)$, where $n_1, \ldots, n_t$ are the sizes of the subtrees formed by removing the separator node from $\mathcal{T}$. We know that $t = O(1)$ and $n_i \leq \lceil n/2 \rceil$, for all i. As such, $T(n) = O(n \log n)$. ■

2.3. Dynamic quadtrees

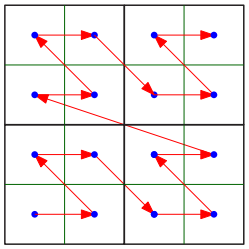
Here we show how to store compressed quadtrees using any data-structure for ordered sets. The idea is to define an order on the compressed quadtree nodes and then store the compressed nodes in a data-structure for ordered sets. We show how to perform basic operations using this representation. In particular, we show how to perform insertions and deletions. This provides us with a simple implementation of dynamic compressed quadtrees.

We start our discussion with regular quadtrees and later on discuss how to handle compressed quadtrees.

2.3.1. Ordering of nodes and points. Consider a regular quadtree $\mathcal{T}$ and a **DFS** traversal of $\mathcal{T}$, where the **DFS** always traverse the children of a node in the same relative order (i.e., say, first the bottom-left child, then the bottom-right child, top-left child, and top-right child).

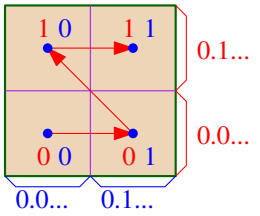


Consider any two canonical squares $\square$ and $\widehat{\square}$, and imagine a quadtree $\mathcal{T}$ that contains both squares (i.e., there are nodes in $\mathcal{T}$ with these squares as their cells). Notice that the above **DFS** would always visit these two nodes in a specific order, independent of the structure of the rest of the quadtree. Thus, if $\square$ gets visited before $\widehat{\square}$, we denote this fact by $\square \prec \widehat{\square}$. This defines a total ordering over all canonical squares. It is useful to extend this ordering to also include points. Thus, consider a point p and a canonical square $\square$. If $\mathsf{p} \in \square$, then we will say that $\square \prec \mathsf{p}$. Otherwise, if $\square \in \mathsf{G}_i$, let $\widehat{\square}$ be the cell in G_i that contains p. We have that $\square \prec p$ if and only if $\square \prec \widehat{\square}$. Next, consider two points p and q, and let G_i be a grid fine enough such that p and q lie in two different cells, say, $\square_{\mathsf{p}}$ and $\square_{\mathsf{q}}$, respectively. Then $\mathsf{p} \prec \mathsf{q}$ if and only if $\square_{\mathsf{p}} \prec \square_{\mathsf{q}}$.



We will refer to the ordering induced by $\prec$ as the ***Q-order***.

The ordering $\prec$ when restricted only to points is the ordering along a space filling mapping that is induced by the quadtree **DFS**. This ordering is know as the ***Z-order***. Specifically, there is a bijection z from the unit interval $[0, 1)$ to the unit square $[0, 1)^2$ such that the ordering along the resulting "curve" is the Z-order. Note, however, that since we allow comparing cells to cells and cells to points, the Q-order no longer has this exact interpretation. Unfortunately, unlike the Peano or Hilbert space filling curves, the Z-order mapping z is not continuous. Nevertheless, the Z-order mapping has the advantage of being easy to define. Indeed, given a real number $\alpha \in [0, 1)$, with the binary expansion $\alpha = 0.x_1x_2x_3\ldots$ (i.e., $\alpha = \sum_{i=1}^{\infty} x_i 2^{-i}$), the Z-order mapping of α is the point $z(\alpha) = (0.x_2x_4x_6\ldots, 0.x_1x_3x_5\ldots)$.



To see that this property indeed holds, think about performing a point-location query in an infinite quadtree for a point $\mathsf{p} \in [0,1)^2$. In the top level of the quadtree we have four possibilities to continue the search. These four possibilities can be encoded as a binary string of length 2; see the figure on the right. Now, if $y(\mathsf{p}) \in [0, 1/2)$, then the first bit in the encoding is 0, and if $y(\mathsf{p}) \in [1/2, 1)$, the first bit output is 1.

Similarly, the second bit in the encoding is the first bit in the binary representation of $x(\mathsf{p})$. Now, after resolving the point-location query in the top level, we continue this search in the tree. Every level in this traversal generates two bits, and these two bits corresponds to the relevant two bits in the binary representation of $y(\mathsf{p})$ and $x(\mathsf{p})$. In particular, the $2i+1$ and $2i+2$ bits in the encoding of p as a single real number (in binary), are the ith bits of (the binary representation of) $y(\mathsf{p})$ and $x(\mathsf{p})$, respectively. In particular, for a point $\mathsf{p} \in [0,1)^d$, let $\mathrm{enc}_{\prec}(\mathsf{p})$ denote the number in the range $[0,1)$ encoded by this process.

Claim 2.15. *For any two points $\mathsf{p}, \mathsf{q} \in [0,1)^2$, we have that $\mathsf{p} \prec \mathsf{q}$ if and only if $\mathrm{enc}_{\prec}(\mathsf{p}) < \mathrm{enc}_{\prec}(\mathsf{q})$.*

2.3.1.1. *Computing the $\mathcal{Q}$-order quickly.* For our algorithmic applications, we need to be able to find the ordering according to $\prec$ between any two given cells/points quickly.

Definition 2.16. For any two points $\mathsf{p}, \mathsf{q} \in [0,1]^2$, let $\mathrm{lca}(\mathsf{p}, \mathsf{q})$ denote the smallest canonical square that contains both p and q. It intuitively corresponds to the node that must be in any quadtree storing p and q.

To compute $\mathrm{lca}(\mathsf{p}, \mathsf{q})$, we revisit the Tweedledee and Tweedledum operation $\mathrm{bit}_\Delta(\alpha, \beta)$ (see Definition 2.7) that for two real numbers $\alpha, \beta \in [0,1)$ returns the index of the first bit in which α and β differ (in base two). The level ℓ of $\square = \mathrm{lca}(\mathsf{p}, \mathsf{q})$ is equal to

$$\ell = 1 - \min(\mathrm{bit}_\Delta(x_\mathsf{p}, x_\mathsf{q}), \mathrm{bit}_\Delta(y_\mathsf{p}, y_\mathsf{q})),$$

where x_p and y_p denote the x and y coordinates of p, respectively. Thus, the sidelength of $\square = \mathrm{lca}(\mathsf{p}, \mathsf{q})$ is $\Delta = 2^\ell$. Let $x' = \Delta \lfloor x/\Delta \rfloor$ and $y' = \Delta \lfloor y/\Delta \rfloor$. Thus,

$$\mathrm{lca}(\mathsf{p}, \mathsf{q}) = [x', x' + \Delta) \times [y', y' + \Delta).$$

We also define the lca of two cells to be the lca of their centers.

Now, given two cells $\square$ and $\widehat{\square}$, we would like to determine their $\mathcal{Q}$-order. If $\square \subseteq \widehat{\square}$, then $\widehat{\square} \prec \square$. If $\widehat{\square} \subseteq \square$, then $\square \prec \widehat{\square}$. Otherwise, let $\widetilde{\square} = \mathrm{lca}(\square, \widehat{\square})$. We can now determine which children of $\widetilde{\square}$ contain these two cells, and since we know the traversal ordering among children of a node in a quadtree, we can now resolve this query in constant time.

Corollary 2.17. *Assuming that the bit_Δ operation and the $\lfloor \cdot \rfloor$ operation can be performed in constant time, then one can compute the* lca *of two points (or cells) in constant time. Similarly, their $\mathcal{Q}$-order can be resolved in constant time.*

Computing bit_Δ efficiently. It seems somewhat suspicious that one assumes that the bit_Δ operations can be done in constant time on a classical RAM machine. However, it is a reasonable assumption on a real world computer. Indeed, in floating point representation, once you are given a number, it is easy to access its mantissa and exponent in constant time. If the exponents are different, then bit_Δ can be computed in constant time. Otherwise, we can easily $\oplus_{\mathrm{xor}}$ the mantissas of both numbers and compute the most significant bit that is one. This can be done in constant time by converting the resulting mantissa into a floating point number and computing its $\log_2$ (some CPUs have this command built in). Observe

that all these operations are implemented in hardware in the CPU and require only constant time.

2.3.2. Performing operations on a (regular) quadtree stored using $\mathcal{Q}$-order. Let $\mathcal{T}$ be a given (regular) quadtree, with its nodes stored in an ordered-set data-structure (for example, using a red-black tree or a skip-list), using the $\mathcal{Q}$-order over the cells. We next describe how to implement some basic operations on this quadtree.

2.3.2.1. *Performing a point-location in a quadtree.* Given a query point $\mathsf{q} \in [0,1]^2$, we would like to find the leaf v of $\mathcal{T}$ such that its cell contains q.

To answer the query, we first find the two consecutive cells in this ordered list such that q lies between them. Formally, let $\square$ be the last cell in this list such that $\square \prec \mathsf{q}$. It is now easy to verify that $\square$ must be the quadtree leaf containing q. Indeed, let $\square_{\mathsf{q}}$ be the leaf of $\mathcal{T}$ whose cell contains q. By definition, we have that $\square_{\mathsf{q}} \prec \mathsf{q}$. Thus, the only bad scenario is that $\square_{\mathsf{q}} \prec \square \prec \mathsf{q}$. But this implies, by the definition of the $\mathcal{Q}$-order, that $\square$ must be contained inside $\square_{\mathsf{q}}$, contradicting our assumption that $\square_{\mathsf{q}}$ is a leaf of the quadtree.

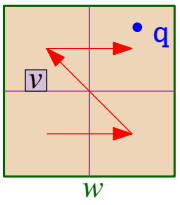
LEMMA 2.18. *Given a quadtree $\mathcal{T}$ of size n, with its leaves stored in an ordered-set data-structure $\mathcal{D}$ according to the $\mathcal{Q}$-order, then one can perform a point-location query in $O(Q(n))$ time, where $Q(n)$ is the time to perform a search query in $\mathcal{D}$.*

2.3.2.2. *Overlaying two quadtrees.* Given two quadtrees $\mathcal{T}'$ and $\mathcal{T}''$, we would like to overlay them to compute their combined quadtree. This is the minimal quadtree such that every cell of $\mathcal{T}'$ and $\mathcal{T}''$ appears in it. Observe that if the two quadtrees are given as sorted lists of their cells (ordered by the $\mathcal{Q}$-order), then their overlay is just the merged list, with replication removed.

LEMMA 2.19. *Given two quadtrees $\mathcal{T}'$ and $\mathcal{T}''$, given as sorted lists of their nodes, one can compute the merged quadtree in linear time (in the total size of the lists representing them) by merging the two sorted lists and removing duplicates.*

2.3.3. Performing operations on a compressed quadtree stored using the $\mathcal{Q}$-order. Let $\mathcal{T}$ be a compressed quadtree whose nodes are stored in an ordered-set data-structure using the $\mathcal{Q}$-order. Note that the nodes are sorted according to the canonical square that they correspond to. As such, a node $v \in \mathcal{T}$ is sorted according to $\square_v$. The subtlety here is that a compressed node v is stored according to the square $\square_v$ in this representation, but in fact it is in charge of the compressed region rg_v. This will make things slightly more complicated.

2.3.3.1. *Point-location in a compressed quadtree.* Performing a point-location query is a bit subtle in this case. Indeed, if the query point q is contained inside a leaf of $\mathcal{T}$, then a simple binary search for the predecessor of q in the $\mathcal{Q}$-order sorted list of the cells of $\mathcal{T}$ would return this leaf.



However, if the query is contained inside the region of a compressed node, the predecessor to q in this list (sorted by the $\mathcal{Q}$-order) might be some arbitrary leaf that is contained inside the compressed node of interest. As a concrete example, consider the figure on the right of a compressed node w. Because of the $\mathcal{Q}$-order, the predecessor query on q would return a leaf v that is stored in the subtree of w but does not contain the query point.

In such a case, we need to find the LCA of the query point and the leaf returned. This returned cell $\square$ could be in the quadtree itself. In this case, we are done as $\square$ corresponds to the compressed node that contains q. The other possibility is that $\square$ is contained inside

the cell $\square_w$ of a compressed node w that is the parent of v. Again, we can now find this parent node using a single predecessor query.

The code for the point-location procedure is depicted on the right. The query point is q. The query point is not necessarily stored in the quadtree, and as such the cell that contains it might be a compressed node (as described above). As such, the required node v has $\mathsf{q} \in \mathsf{rg}_v$ and is either a leaf of the quadtree or a compressed node.

```
algPntLoc_Qorder(𝒯, q).
  □_v = predecessor_{Q−order}(𝒯, q).
  // □_v last node in 𝒯
  // s.t.  □_v ⪯ q.
  if q ∈ □_v then
      return v
  □ = lca(□_v, q)
  □_w = predecessor_{Q−order}(𝒯, □).
  return w
```

Lemma 2.20. *We have that: (i) if $\mathsf{q} \in \square_v$, then $\mathsf{q} \in \mathsf{rg}_v$ and (ii) if the region of $\mathcal{T}$ containing the query point q is a leaf, then v is the required leaf.*

Proof. (i) Otherwise, there exists a node $x \in \mathcal{T}$ such that $\mathsf{q} \in \mathsf{rg}_x \subseteq \square_x$ and $\square_x \subsetneq \square_v$, but that would imply that $\square_v \prec \square_x \prec \mathsf{q}$, contradicting how $\square_v$ was computed.

(ii) Since q is contained in a leaf, then the last square in the $\mathcal{Q}$-order that is before q is the cell of this leaf, that is, the node v. ■

Now, if $\mathsf{q} \notin \square_v$, then our search "failed" to find a node that contains the query point. So, consider the node w that should be returned. It must be a compressed node, and by the above, the search returned a node v that is a descendant of w. As a first step, we compute the square $\square = \mathrm{lca}(\square_v, \mathsf{q})$. Next, we compute the last cell $\square_u$ such that $\square_u \preceq \square$.

Now, if $\square_u = \square$, then we are done as $\mathsf{q} \in \mathsf{rg}_u$. Otherwise, if $\square$ is not in the tree, then u is a compressed node, and consider its child u'. We have that $\square$ is contained inside $\square_u$ and contains $\square_{u'}$. Namely, $\mathsf{q} \in \square_u \setminus \square_{u'} = \mathsf{rg}_u$. Now, u' is the only child of u (since it is a compressed node), which implies that $\square_u$ and $\square_{u'}$ are consecutive in $\mathcal{T}$ according to the $\mathcal{Q}$-order, and $\square$ lies in between these two cells in this order. As such, $u \prec \square \prec u'$ and as such u is the required node that contains the query point.

We get the following result.

Lemma 2.21. *Given a compressed quadtree $\mathcal{T}$ of size n, with its nodes stored in an ordered-set data-structure $\mathcal{D}$ according to the $\mathcal{Q}$-order, then point-location queries can be performed using $\mathcal{T}$ in $O(Q(n))$ time, where $Q(n)$ is the time to perform a search query in $\mathcal{D}$.*

2.3.3.2. *Insertions and deletions.* Let q be a point to be inserted into the quadtree, and let w be the node of the compressed quadtree such that $\mathsf{q} \in \mathsf{rg}_w$. There are several possibilities:

(1) The node w is a leaf, and there is no point associated with it. Then store p at w, and return.
(2) The node w is a leaf, and there is a point p already stored in w. In this case, let $\square = \mathrm{lca}(\mathsf{p}, \mathsf{q})$, and insert $\square$ into the compressed quadtree. This is done by creating a new node z in the quadtree, with the cell of z being $\square$. We hang z below w if $\square \neq \square_w$ (this turns w into a compressed node). (If $\square = \square_w$, we do not need to introduce a new cell and set $z = w$.) Furthermore, split $\square$ into its children, and also insert the children into the compressed quadtree. Finally, associate p with the new leaf that contains it, and associate q with the leaf that contains it. Note that because of the insertion w becomes a compressed node if $\square_w \neq \square$, and it becomes a regular internal node otherwise.

(3) The node w is a compressed node. Let z be the child of w, and consider $\square = \mathrm{lca}(\square_z, \mathsf{q})$. Insert $\square$ into the compressed quadtree if $\square \neq \square_w$ (note that in this case w would still be a compressed node, but with a larger "hole"). Also insert all the children of $\square$ into the quadtree, and store p in the appropriate child. Hang $\square_z$ from the appropriate child, and turn this child into a compressed node.

In all three cases, the insertion requires a constant number of search/insert operations on the ordered-set data-structure.

Deletion is done in a similar fashion. We delete the point from the node that contains it, and then we trim away nodes that are no longer necessary.

THEOREM 2.22. *Assuming one can compute the $\mathcal{Q}$-order in constant time, then one can maintain a compressed quadtree of a set of points in $O(\log n)$ time per operation, where insertions, deletions and point-location queries are supported. Furthermore, this can implemented using any data-structure for ordered-set that supports an operation in logarithmic time.*

2.3.4. Compressed quadtrees in high dimension. Naively, the constants used in the compressed quadtree are exponential in the dimension d. However, one can be more careful in the implementation. The first problem, for a node v in the compressed quadtree, is to store all the children of v in an efficient way so that we can access them efficiently. To this end, each child of v can be encoded as a binary string of length d, and we build a trie inside v for storing all the strings defining the children of v. It is easy, given a point, to figure out what the binary string encoding the child containing this point is. As such, in $O(\mathsf{d})$ time, one can retrieve the relevant child. (One can also use hashing to this end, but it is not necessary here.)

Now, we build the compressed quadtree using the algorithm described above using the $\mathcal{Q}$-order. It is not too hard to verify that all the basic operations can be computed in $O(\mathsf{d})$ time. Specifically, comparing two points in the $\mathcal{Q}$-order takes $O(\mathsf{d})$ time. One technicality that matters when d is large (but this issue can be ignored in low dimensions) is that if a node has only a few children that are not empty, we create only these nodes and not the other children. Putting everything together, we get the following result.

THEOREM 2.23. *Assuming one can compute the $\mathcal{Q}$-order in $O(\mathsf{d})$ time for two points $\mathbb{R}^{\mathsf{d}}$, then one can maintain a compressed quadtree of a set of points in $\mathbb{R}^{\mathsf{d}}$, in $O(\mathsf{d} \log n)$ time per operation. The operations of insertion, deletion, and point-location query are supported. Furthermore, this can be implemented using any data-structure for ordered-set that supports insertion/deletion/predecessor operations in logarithmic time.*

In particular, one can construct a compressed quadtree of a set of n points in $\mathbb{R}^{\mathsf{d}}$ in $O(\mathsf{d}n \log n)$ time.

2.4. Bibliographical notes

The authoritative text on quadtrees is the book by Samet [**Sam89**]. He also has a more recent book that provides a comprehensive survey of various tree-like data-structures [**Sam05**] (our treatment is naturally more theoretically oriented than his). The idea of using hashing in quadtrees is a variant of an idea due to Van Emde Boas and is also used in performing fast lookup in IP routing (using PATRICIA tries which are one-dimensional quadtrees [**WVTP97**]), among a lot of other applications.

The algorithm described in Section 2.2.2 for the efficient construction of compressed quadtrees is new, as far as the author knows. The classical algorithms for computing compressed quadtrees efficiently achieve the same running time but require considerably more

careful implementation and paying careful attention to details [**CK95, AMN+98**]. The idea of fingering is used in [**AMN+98**] (although their presentation is different than ours) but the idea is probably much older.

The idea of storing a quadtree in an ordered set by using the $\mathcal{Q}$-order on the nodes (or even only on the leaves) is due to Gargantini [**Gar82**], and it is referred to as ***linear quadtrees*** in the literature. The idea was used repeatedly for getting good performance in practice from quadtrees.

Our presentation of the dynamic quadtrees (i.e., Section 2.3) follows (very roughly) the work de Berg et al. [**dHTT07**].

It is maybe beneficial to emphasize that if one does not require the internal nodes of the compressed quadtree for the application, then one can avoid storing them in the data-structure. If only the points are needed, then one can even skip storing the leaves themselves, and then the compressed quadtree just becomes a data-structure that stores the points according to their $\mathcal{Z}$-order. This approach can be used for example to construct a data-structure for approximate nearest neighbor [**Cha02**] (however, this data-structure is still inferior, in practice, to the more optimized but more complicated data-structure of Arya et al. [**AMN+98**]). The author finds that thinking about such data-structures as compressed quadtrees (with the whole additional unnecessary information) is more intuitive, but the reader might disagree[③].

$\mathcal{Z}$-order and space filling curves. The idea of using $\mathcal{Z}$-order for speeding up spatial data-structures can be traced back to the above work of Gargantini [**Gar82**], and it is widely used in databases and seems to improve performance in practice [**KF93**]. The $\mathcal{Z}$-order can be viewed as a mapping from the unit interval to the unit square, by splitting the sequence of bits representing a real number $\alpha \in [0,1)$, where the odd bits are the x-coordinate and the even bits are the y-coordinate of the mapped point. While this mapping is simple to define, it is not continuous. Somewhat surprisingly, one can find a continuous mapping that maps the unit interval to the unit square; see Exercise 2.5. A large family of such mappings is known by now; see Sagan [**Sag94**] for an accessible book on the topic.

But is it really practical? Quadtrees seem to be widely used in practice and perform quite well. Compressed quadtrees seem to be less widely used, but they have the benefit of being much simpler than their relatives which seems to be more practical but theoretically equivalent.

Compressed quadtrees require strange operations. Lemma 2.8 might be new, although it seems natural to assume that it was known before. It implies that computing compressed quadtrees requires at least one "strange" operation in the computation model. Once one comes to term with this imperfect situation, the use of the $\mathcal{Q}$-order seems natural and yields a reasonably simple algorithm for the dynamic maintenance of quadtrees. For example, if we maintain such a compressed quadtree by using skip-list on the $\mathcal{Q}$-order, we will essentially get the skip-quadtree of Eppstein et al. [**EGS05**].

Generalized compressed quadtrees. Har-Peled and Mendel [**HM06**] have shown how to extend compressed quadtrees to the more general settings of doubling metrics. Note that this variant of compressed quadtrees no longer requires strange bit operations. However, in the process, one loses the canonical grid structures that a compressed quadtree has, which is such a useful property.

[③]The author reserves the right to disagree with himself on this topic in the future if the need arises.

Why compressed quadtrees? The reader might wonder why we are presenting compressed quadtrees when in practice people use different data-structures (that can also be analyzed). Our main motivation is that compressed quadtrees seem to be a universal data-structure, in the sense that they can be used for many different tasks, they are conceptually and algorithmically simple, and they provide a clear route to solving a problem: Solve your problem initially on a quadtree for a point set with bounded spread. If this works, try to solve it on a compressed quadtree. If you want something practical, try some more practical variants like *kd*-trees [**dBCvKO08**].

2.5. Exercises

EXERCISE 2.1 (Geometric obesity is good). For a constant $\alpha \geq 1$, a planar convex region T is ***α-fat*** if $R(T)/r(T) \leq \alpha$, where $R(T)$ and $r(T)$ are the radii of the smallest disk containing T and the largest disk contained in T, respectively.

(A) Prove that if a triangle $\triangle$ has all angles larger than β, then $\triangle$ is $\dfrac{1}{\sin(\beta/2)}$ fat. We will refer to such a triangle as being ***fat***.

(B) Let S be a set of interior disjoint α-fat shapes all intersecting a common square $\square$. Furthermore, for every $\triangle \in \mathsf{S}$ we have that $\operatorname{diam}(\triangle) \geq c \cdot \operatorname{diam}(\square)$, where c is some constant. Prove that $|\mathsf{S}| = O(1)$. Here, the constant depends on c and α, and the reader is encouraged to be less lazy than the author and to figure out the exact value of this constant as a function of α and c.

EXERCISE 2.2 (Quadtree for fat triangles). Let P be a triangular planar map of the unit square (i.e., each face is a triangle but it is not necessarily a triangulation), where all the triangles are fat and the total number of triangles is n.

(A) Show how to build a compressed quadtree storing the triangles, such that every node of the quadtree stores only a constant number of triangles (the same triangle might be stored in several nodes) and given a query point p, the triangle containing p is stored somewhere along the point-location query path of p in the quadtree.

(Hint: Think about what is the right resolution to store each triangle.)

(B) Show how to build a compressed quadtree for P that stores triangles only in the leaves, and such that every leaf contains only a constant number of triangles and the total size of the quadtree is $O(n)$.

Hint: Using the construction from the previous part, push down the triangles to the leaves, storing in every leaf all the triangles that intersect it. Use Exercise 2.1 to argue that no leaf stores more than a constant number of triangles.

(C) (Tricky but not hard) Show that one must use a compressed quadtree in the worst case if one wants linear space.

(D) (Hard) We remind the reader that a ***triangulation*** is a triangular planar map that is compatible. That is, the intersection of triangles in the triangulation is either empty, a vertex of both triangles, or an edge of both triangles (this is also known as a ***simplicial complex***). Prove that a fat triangulation with n triangles can be stored in a regular (i.e., non-compressed) quadtree of size $O(n)$.

EXERCISE 2.3 (Quadtree construction is tight). Prove that the bounds of Lemma 2.4 are tight. Namely, show that for any $r > 2$ and any positive integer $n > 2$, there exists a set of n points with diameter $\Omega(1)$ and spread $\Phi(\mathsf{P}) = \Theta(r)$ and such that its quadtree has size $\Omega(n \log \Phi(\mathsf{P}))$.

Exercise 2.4 (Cell queries). Let $\widehat{\square}$ be a canonical grid cell. Given a compressed quadtree $\widehat{T}$, we would like to find the *single* node $v \in \widehat{T}$, such that $\mathsf{P} \cap \widehat{\square} = \mathsf{P}_v$. We will refer to such a query as a ***cell query***. Show how to support cell queries in a compressed quadtree in logarithmic time per query.

Exercise 2.5 (Space filling curve). The ***Peano curve*** $\sigma : [0, 1) \to [0, 1)^2$ maps a number $\alpha = 0.t_1t_2t_3 \ldots$ (the expansion is in base 3) to the point $\sigma(\alpha) = (0.x_1x_2x_3 \ldots,\ 0.y_1y_2 \ldots)$, where $x_1 = t_1$, $x_i = \phi(t_{2i-1}, t_2 + t_4 + \cdots + t_{2i-2})$, for $i \geq 1$. Here, $\phi(a, b) = a$ if b is even and $\phi(a, b) = 2 - a$ if b is odd. Similarly, $y_i = \phi(t_{2i}, t_1 + t_3 + \cdots + t_{2i-1})$, for $i \geq 1$.

(A) Prove that the mapping σ covers all the points in the open square $[0, 1)^2$, and it is one-to-one.

(B) Prove that σ is continuous.