CMSC 425: Lecture 10
Geometric Data Structures for Games: Index Structures
Tuesday, Feb 26, 2013

Reading: Some of today's materials can be found in "Foundations of Multidimensional and Metric Data Structures," by H. Samet, 2006.

Index Structures: We continue our discussion of useful geometric data structure for computer games. So far we have discussed data structures for representing meshes (and more generally, cell complexes), geometric graphs (such as the visibility graph, θ-graph), and spatial subdivisions (such as the Delaunay triangulation). Usually, when we think of data structures for 1-dimensional data (such as hash maps, binary trees, and skip lists), the data structure is a means for efficiently accessing data based on a *key value*. A data structure that locates an object based on a key value is called an *index structure*.

In this lecture, we consider some useful index structures for storing spatial data sets. The topic is amazingly vast, and we will just provide a very short description of a few of the simplest spatial index structures. (See Samet's book above for more topics.)

Geometric Objects and Queries: What sorts of objects might be stored in a geometric data structure? The simplest object to store is a point, and it is common to introduce data structures under the assumption that the objects themselves are points. As needed, the data structure and associated algorithms can be enhanced to deal with more general types of objects.

What sorts of geometric queries might we be interested in asking? This depends a great deal about the application at hand. Queries typically involve determining what things are "close by." One reason is that nearby objects are more likely to have interesting interactions in a game (collisions or attacks). Of course, there are other sorts of interesting geometric properties. For example, in a shooting game, it may be of interest to know which other players have a line-of-sight to a given entity.

While we will focus on purely geometric data in this lecture, it is worth noting that geometry of an object may not be the only property of interest. For example, the query "locate all law enforcement vehicles within a half-mile radius of the player's car", might be quite reasonable for a car theft game. Such queries involve both geometric properties (half-mile radius) and nongeometric properties (law enforcement). Such hybrid queries may involve a combination of multiple data structures.

Bounding Enclosures: When storing complex objects in a spatial data structure, it is common to first approximate the object by a simple enclosing structure. Bounding enclosures are often handy as a means of approximating an object as a filter in collision detection. If the bounding enclosures do not collide, then the objects to not collide. If they do, then we strip away the enclosures and apply a more extensive intersection test to the actual objects. Examples of bounding structures include:

Axis-aligned bounding boxes: This is an enclosing rectangle whose sides are parallel to the coordinate axes (see Fig. 1(a)). They are commonly called *AABBs* (axis-aligned bounding boxes). They are very easy to compute. (The corners are based on the minimum and maximum x- and y-coordinates.) They are also very easy to intersect. (Just determine whether the intervals of x-coordinates intersect and the intervals of the y-coordinates intersect.)

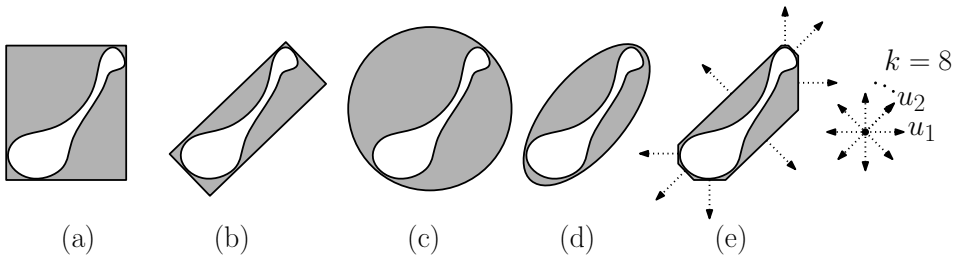

Fig. 1: Examples of common enclosures: (a) AABB, (b) general BB, (c) sphere, (d) ellipsoid, (e) 8-DOP.

General bounding boxes: The principal shortcoming of axis-parallel bounding boxes is that it is not possible to rotate the object without recomputing the entire bounding box. In contrast, general (arbitrarily-oriented) bounding boxes can be rotated without the need to recompute them (see Fig. 1(b)). Unfortunately, computing the minimum area or minimum volume bounding box is not a trivial problem. Determining whether two such boxes intersect is a more complex procedure, but it is likely to be much more efficient than computing the intersection of the two objects.

Bounding spheres: Given that arbitrary bounding boxes are harder to intersect, an alternative is to use bounding spheres (see Fig. 1(b)). Spheres are invariant under both translation and rotation. It is very easy to determine whether two spheres intersect one another. Just compute the distance between their centers, and check that this is smaller than the sum of their radii.

Bounding ellipsoids: The main problem with spheres (and problem that also exists with axis-parallel bounding boxes) is that skinny objects are no well approximated by a sphere. A ellipse (or generally, an ellipsoid in higher dimensions) is just the image of a sphere under a linear transformations (see Fig. 1(b)). As with boxes, ellipsoids may either be axis-parallel (meaning that the principal axes of the ellipse are parallel to the coordinate axes) or arbitrary.

k-DOPs: People like objects bounded by flat sides, because the mathematics involved is linear. (There is no need to solve algebraic equations.) Unfortunately, an axis-aligned bounding box may not be a very close approximation to an object. (Imagine a skinny diagonal line segment.) As mentioned above, computing the minimum general bounding box may also be quite complex. A k-DOP is a compromise between these two. Given an integer parameter $k \geq 3$ (or generally $k \geq d+1$), we generate k directional vectors that are roughly equally spaced and span the entire space. For example, in two-dimensional space, we might consider a set of unit vectors at angles $2\pi i/k$, for $0 \leq i < k$. Let $\{u_1, \ldots, u_k\}$ be the resulting vectors. We then compute extreme point of the object along each of these directions. We then put an orthogonal hyperplane through this point. The intersection of these hyperplanes defines a convex polygon with k sides (generally, a convex polytope with k facets) that encloses the objects. This is called a *k-discrete oriented polytope*, or *k-DOP* (see Fig. 1(e)).

Hierarchies of bounding boxes: A natural generalization of the concept of a bounding box is that of constructing a hierarchy consisting of multiple levels of bounding boxes, where the boxes at a given level enclose a constant number of boxes at the next lower level. A common approach is to specify that the number of boxes contained within a given node lies between some minimum and maximum value. The result is called an *R-tree*. In Fig. 2 we given an example, where the input boxes are shown in (a), the hierarchy (allowing between 2–3 boxes per group) is shown in (b) and the final tree is shown in

(c).

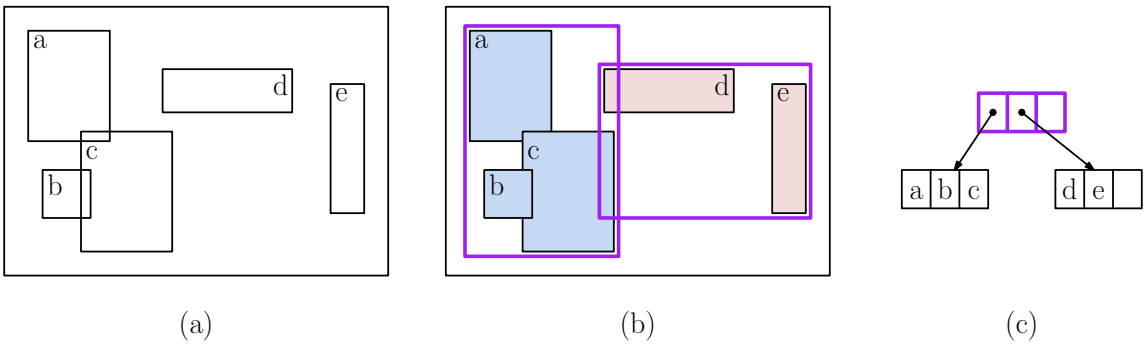

Fig. 2: A hierarchy of bounding boxes: (a) the input boxes, (b) the hierarchy of boxes, (c) the associated R-tree structure.

There are a number of messy technical issues in the design of R-trees. For example, what is the best way to cluster smaller boxes together to form larger boxes. How do you minimize the wasted space within each box and how do you minimize the overlap between boxes at a given level of the tree.

This structure is widely used in the field of spatial databases, since the number of boxes contained within another box can be adjusted so that each node corresponds to a single disk block. (In this respect, it is analogous to the B-tree data structure for 1-dimensional data sets.)

Grids: One virtue of simple data structures is that are usually the easiest to implement, and (if you are fortunate in your choice of geometric model) they may work well. An example of a very simple data structure that often performs quite well is a simple rectangular grid. For simplicity, suppose that want to store a collection of objects. We will assume that the distribution of objects is fairly uniform. In particular, we will assume that there exists a positive real Δ such that almost all the objects are of diameter at most $c\Delta$ for some small constant c, and the number of objects that intersect any cube of side length Δ is also fairly small.

If these two conditions are met, then a square grid of side length Δ may be a good way to store your data. Here is how this works. First, for each of your objects, compute its axis-aligned bounding box. We can efficiently determine which cells of the grid are overlapped by this bounding box as follows. Let p and q denote the bounding box's lower-left and upper-right corners (see Fig. 3(a)). Compute the cells of the grid that contain these points (see Fig. 3(b)). Then store a pointer to the object in all the grid cells that lie in the rectangle defined by these two cells (see Fig. 3(c)). Note that this is not perfect, since the object may be associated with grid cells that it does not actually intersect. This increases the space of the data structure, but it does not compromose the data structure's correctness.

Computing the indices of the grid cell that contain a given point is a simple exercise in integer arithmetic. For example, if $p = (p_x, p_y)$, then let

$$j = \left\lfloor \frac{p_x}{\Delta} \right\rfloor \qquad \text{and} \qquad i = \left\lfloor \frac{p_y}{\Delta} \right\rfloor.$$

Then, the point p lies within the grid cell $G[i, j]$.

If the diameter of most of the objects is not significantly larger than Δ, then each object will only be associated with a constant number of grid cells. If the density of objects is not too high, then each

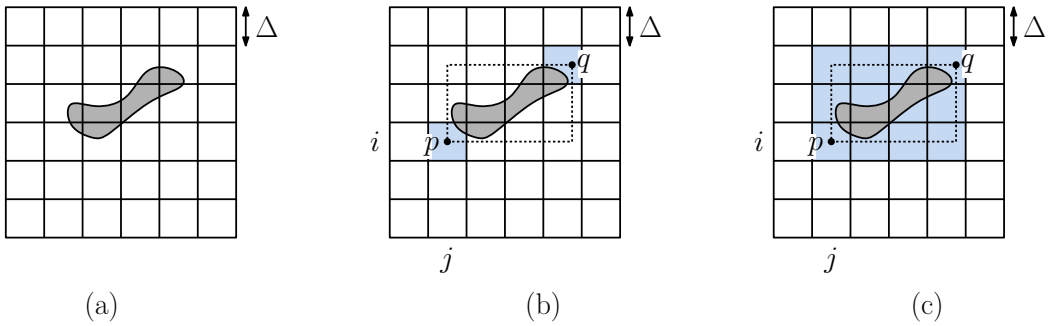

Fig. 3: Storing an object in a grid.

grid square will only need to store a constant number of pointers. Thus, if the above assumptions are satisfied then the data structure will be relatively space efficient.

Storing a Grid: As we have seen, and grid consists of a collection of cells, where each cell stores a set of pointers to the objects that overlap this cell (or at least might overlap this cell). How do we store these cells. Here are a few ideas.

d-dimensional array: The simplest implementation is to allocate a d-dimensional array that is sufficiently large to handle all the cells of your grid. If the distribution of objects is relatively uniform, then it is reasonable to believe that a sizable fraction of the cells will be nonempty. On the other hand, if the density is highly variable, there may be many empty cells. This approach will waste space.

Hash map: Each cell is identified by its indices, (i, j) for a 2-dimensional grid or (i, j, k) for the 3-dimensional grid. Treat these indices like a key into a hash map. Whenever a new object o is entered into some cell (i, j), we access the hash map to see whether this cell exists. If not, we generate a new cell object and add it to the hash map under the key (i, j). If so, we add a pointer to o into this hash map entry.

Linear allocation: Suppose that we decide to adopt an array allocation. A straightforward implementation of the d-dimensional array will result in a memory layout according to how your compiler chooses to allocate arrays, typically in what is called *row-major order* (see Fig. 4(a)). For example, if there are N columns, then the element (i, j) is mapped to index $i \cdot N + j$ in row-major order.

Why should you care? Well, the most common operation to perform on a grid is to access the cells that surround a given grid square. Memory access tends to be most efficient when accessed elements are close to one another in memory (since they are likely to reside within the same cache lines). The problem with row-major order is that entries in successive rows are likely to be far apart in physical memory.

A cute trick for avoiding this problem is to adopt a method of mapping cells to physical memory that is based on a *space filling curve*. There are many different space-filling curves. We show two examples in Figs. 4(b) and (c). The first is called the *Hilbert curve* and the second is known as the *Morton order* (also called the *Z-order*).

There is experimental evidence that shows that altering the physical allocation of cells can improve running times moderately. Unfortunately, the code that maps an index (i, j) to the corre-

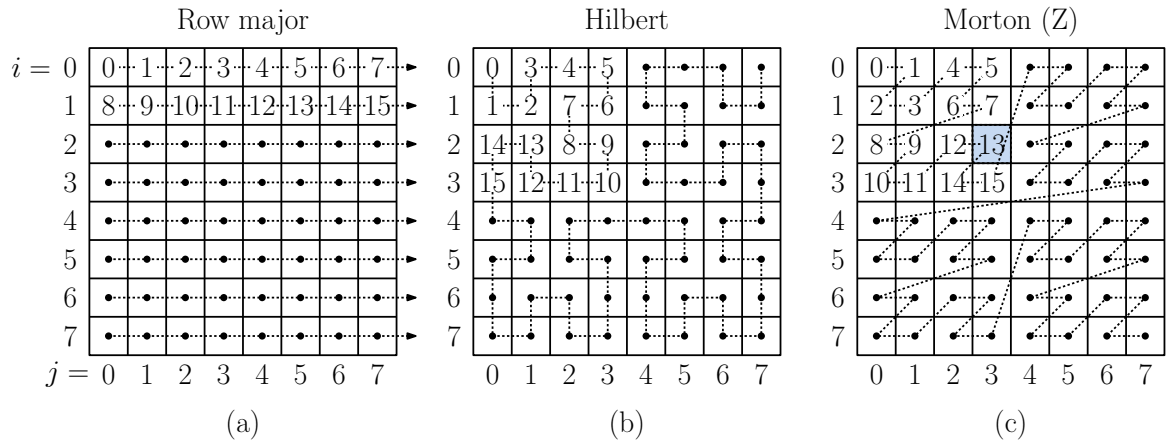

Fig. 4: Linear allocations to improve reference locality of neighboring cells: (a) row-major order, (b) the Hilbert curve, (c) the Morton order (or Z-order).

sponding address in physical memory becomes something of a brain teaser.

Computing the Morton Order: Between the Hilbert order and the Morton order, the Morton order is by far the more commonly use. One reason for this is that there are some nifty tricks for computing the this order. To make this easier to see, let us assume that we are working in two-dimensional space and that the grid of size $2^m \times 2^m$. The trick we will show applies to any dimension. If your grid is not of this size, you can embed it within the smallest grid that has this property.

Next, since the grid has 2^m rows and 2^m columns, we can view each row and column index as an m-element bit vector, call them $i = \langle i_1, \ldots, i_m \rangle_2$ and $j = \langle j_1, \ldots, j_m \rangle_2$, in order from most significant bit (i_1) to the least significant bit (i_m). Next, we take these bit vectors and interleave them as if we were shuffling a deck of cards:

$$k \;=\; \langle i_1, j_1, i_2, j_2, \ldots, i_m, j_m \rangle_2.$$

If you have not seen this trick before, it is rather remarkable that it works. As an example, consider the cell at index $(i, j) = (2, 3)$, which is labeled as 13 in Fig. 4(c). Expressing i and j as 3-element bit vectors we have $i = \langle 0, 1, 0 \rangle_2$ and $j = \langle 0, 1, 1 \rangle_2$. Next, we interleave these bits to obtain

$$k \;=\; \langle 0, 0, 1, 1, 0, 1 \rangle_2 \;=\; 13,$$

just as we expected.

This may seem like a lot of bit manipulation, particularly if m is large. It is possible, however, to speed this up. For example, rather than processing one bit at a time, we could break i and j up into 8-bit bytes, and then for each byte, we could access a 256-element look-up table to convert its bit representation to one where the bits have been "spread out." (For example, suppose that you have the 8-element bit vector $\langle b_0, b_1, \ldots, b_7 \rangle_2$. The table look-up would return the 16-element bit vector $\langle b_0, 0, b_1, 0, \ldots, b_7, 0 \rangle_2$.) You repeat this for each byte, applying a 16-bit shift in each case. Finally, you apply an addition right shift of the j bit vector by a single position and bitwise "or" the two spread-out bit vectors for i and j together to obtain the final shuffled

bit vector. By interpreting this bit vector as an integer we obtain the desired *Morton code* for the pair (i, j).

Quadtrees: Grids are fine if the density of objects is fairly regular. If there is considerable variation in the density, a quadtree is a practical alternative. You have probably seen quadtrees in your data structures course, so I'll just summarize the main points, and point to a few useful tips.

First off, the term "quadtree" is officially reserved for 2-dimensional space and "octree" for three dimensional space. However, it is too hard to figure out what the name should be when you get to 13-dimensional space, so I will just use the term "d-dimensional quadtree" for all dimensions.

We begin by assuming that the domain of interest has been enclosed within a large bounding square (or generally a hypercube in d-dimensional space). Let's call this Q_0. Let us suppose that we have applied a uniform scaling factor so that Q_0 is mapped to the d-dimensional unit hypercube $[0, 1]^d$. A *quadtree box* is defined recursively as follows:

- Q_0 is a quadtree box
- If Q is any quadtree box, then the 2^d boxes that result by subdividing Q through its midpoint by axis aligned hyperplanes is also a quadtree box.

This definition naturally defines a hierarchical subdivision process, which subdivides Q_0 into a collection of quadtree boxes. This defines a tree, in which each node is associated with a quadtree box, and each box that is split is associated with the 2^d sub-boxes as its children (see Fig. 5). The root of the tree is associated with Q_0. Because Q_0 has a side length of 1, it follows directly that the quadtree boxes at level k of the tree have side length $1/2^k$.

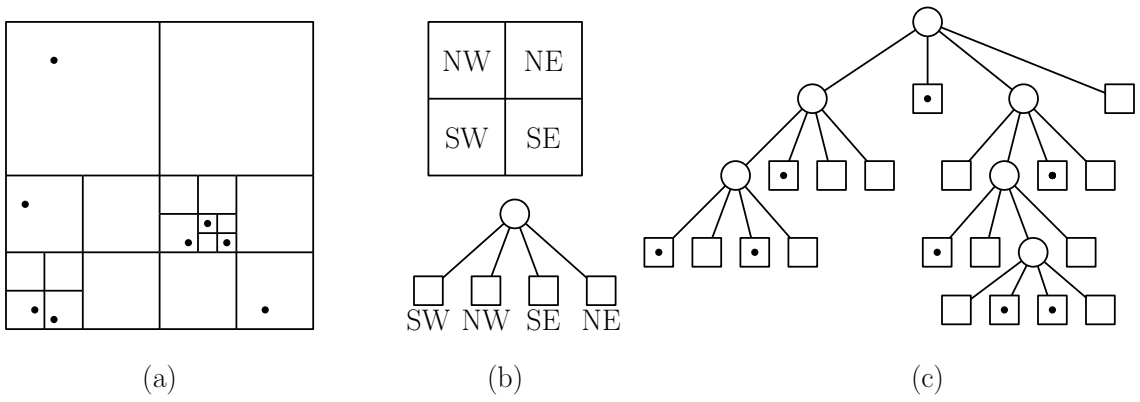

Fig. 5: A quadtree decomposition and the associated tree.

Here are a few tips to making quadtrees somewhat more manageable.

Binary Quadtrees: In dimension 3 and higher, having to allocate 2^d children for every internal node can be quite wasteful. Unless the points are uniformly distributed, it is often the case that only a couple of these nodes contain points. An alternative is rely only on binary splits. First, split along the midpoint x-coordinate, then the midpoint y-coordinate, and so forth, cycling through the axes (see Fig. 6).

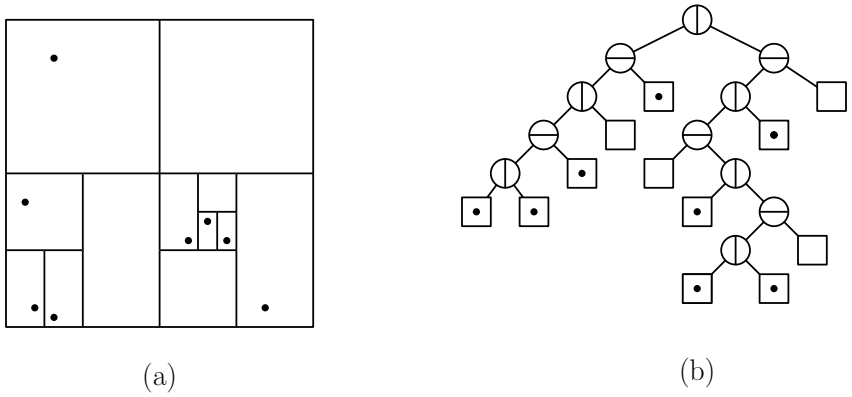

Fig. 6: A binary quadtree.

Compressed Quadtree: If the objects are very highly clustered, it is possible that the quadtree may actually much larger than the number of objects. Consider the case of two very close points in Fig. 7. Separating these two points would involve a long string of splits, most of which are *trivial* in the sense that each node has only one nonempty child. One way to deal with this is to compress out such trivial paths by a single edge that jumps, in one hop, to the first node that has two or more nonempty children. For example, you could introduce a new type of node, called a *compression node*, that has a single child, which points to the last node of the trivial path. The result is called a *compressed quadtree*. A nice feature of the compressed quadtree is that a compressed quadtree with n points has $O(n)$ nodes.

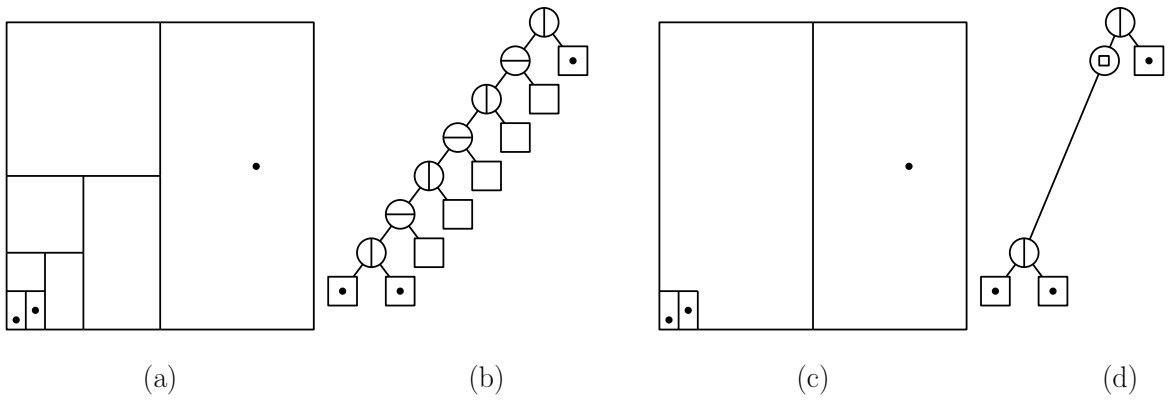

Fig. 7: A compressed quadtree.

Linear Quadtree: A very clever and succinct method for storing quadtrees for point sets involves no tree at all! Recall the Morton order, described earlier in this lecture. A point (x, y) is mapped to a point in a 1-dimensional space by shuffling the bits of x and y together. This maps all the points of your set onto a space filling curve.

What does this curve have to do with quadtrees? It turns out that the curve visits the cells of the quadtree (either the standard, binary, or compressed versions) according to an in-order traversal of the tree (see Fig. 8).

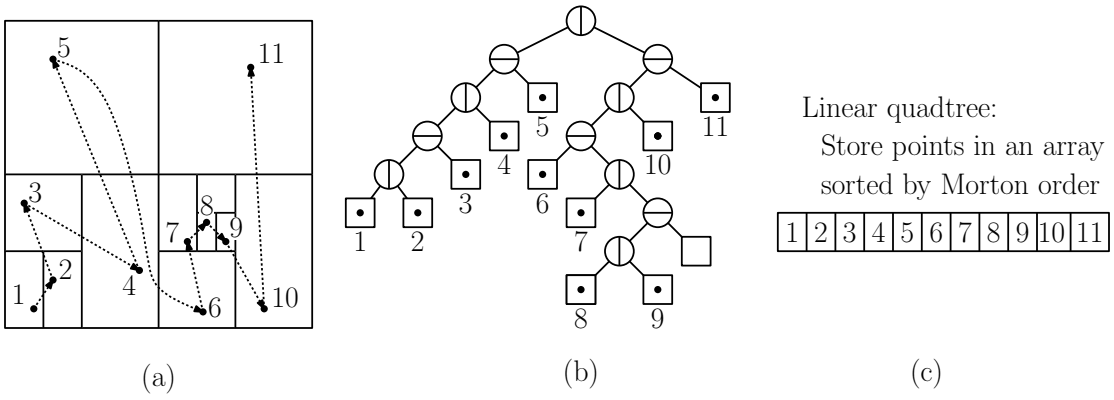

Fig. 8: A linear quadtree.

How can you exploit this fact? It seems almost unbelievable that this would work, but you sort all the points of your set by the Morton order and store them in an array (or any 1-dimensional data structure). While this would seem to provide very little useful structure, it is remarkable that many of the things that can be computed efficiently using a quadtree can (with some additional modifications) be computed directly from this sorted list. Indeed, the sorted list can be viewed as a highly compressed encoding of the quadtree.

The advantage of this representation is that it requires zero additional storage, just the points themselves. Even though the access algorithms are a bit more complicated and run a bit more slowly, this is a very good representation to use when dealing with very large data sets.