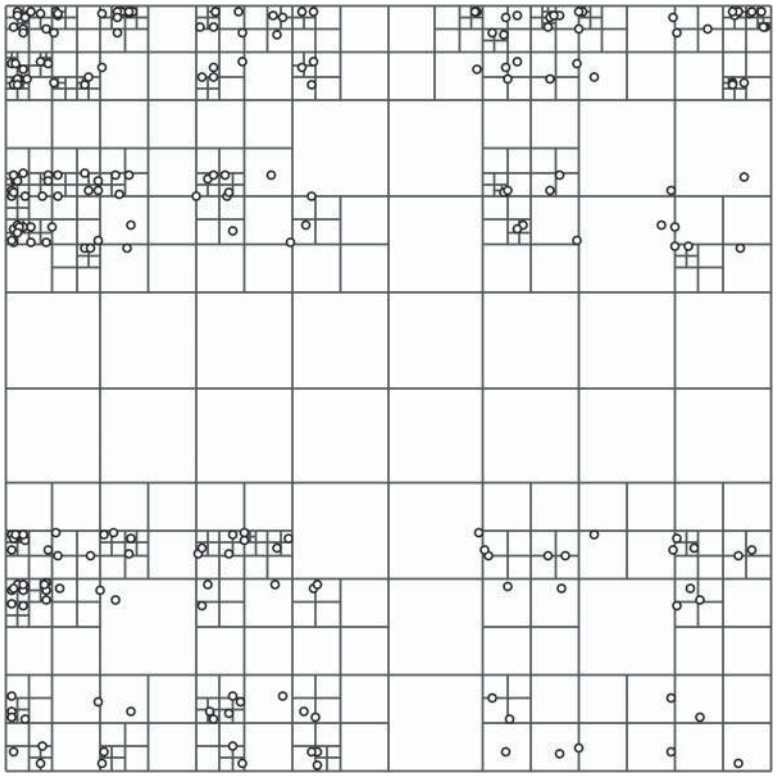
Chapter 21 Multidimensional Searching



A point quadtree, 2005. Credit: David Eppstein. Public domain image.

Contents

We live in a multidimensional geometric world. Physical space itself is three-dimensional, for we can use three coordinates, x, y, and z, to describe points in space. Completely describing the orientation of the tip of a robot arm actually requires six dimensions, for we use three dimensions to describe the position of the tip in space, plus three more dimensions to describe the angles the tip is in (which are typically called pitch, roll, and yaw). Describing the state of an airplane in flight takes at least nine dimensions, for we need six to describe its orientation in the same manner as for the tip of a robot arm, and we need three more to describe the plane's velocity. In fact, these physical representations are considered "low-dimensional," particularly in applications in machine learning or computational biology, where 100- and 1000-dimensional spaces are not unusual. For instance, a vector describing a collection of genes or movie ratings could easily have dimensionality in this range. This chapter is therefore directed at ***multidimensional searching***, which studies data structures for storing and querying multidimensional data sets.

Multi-dimensional data arise in a variety of applications, including statistics and robotics. The simplest type of multidimensional data are d-dimensional points, which can be represented by a sequence,

$$(x_0, x_1, \ldots, x_{d-1}),$$

of d numeric ***coordinates***. In business applications, a d-dimensional point may represent the various attributes of a product or an employee in a database. For example, televisions in an electronics catalog would probably have different attribute values for price, screen size, weight, height, width, and depth. Multi-dimensional data can also come from scientific applications, where each point represents attributes of individual experiments or observations. For example, heavenly objects in an astronomy sky survey would probably have different attribute values for brightness (or apparent magnitude), diameter, distance, and position in the sky (which is itself two-dimensional). Thus, these applications can benefit from efficient methods for storing and searching in multidimensional data sets.

There are actually a great number of different data structures and algorithms for processing multidimensional data, and it is beyond the scope of this chapter to discuss all of them. We provide instead an introduction to some of the more interesting ones in this chapter. We begin with a discussion of ***range trees***, which can store multidimensional points so as to support a special kind of query operation, called a ***range-searching query***, and we also include an interesting variant of the range tree called the ***priority search tree***. Finally, we discuss a class of data structures, called ***partition trees***, which partition space into cells, and focus on variants known as ***quadtrees*** and ***k-d trees***. These data structures are used, for instance, in computer graphics and computer gaming applications, where we need to find nearest neighbors to a query point or map out the trajectory of a ray through a virtual environment.

21.1 Range Trees

A natural query operation to perform on a set of multidimensional points is a ***range-search query***, which is a request to retrieve all points in a multidimensional collection whose coordinates fall within given ranges. For example, a consumer wishing to buy a new television may request, from an electronic store's catalog, all units that have a screensize between 24 and 27 inches, and have a price between \$200 and \$400. Alternately, an astronomer interested in studying asteroids may request all heavenly objects that are at a distance between 1.5 and 10 astronomical units, have an apparent magnitude between $+1$ and $+15$, and have a diameter between 0.5 and $1,000$ kilometers. The range tree data structure, which we discuss in this section, can be used to answer such queries.

Two-Dimensional Range-Search Queries

To keep the discussion simple, let us focus on two-dimensional range-searching queries. Exercise C-21.7 addresses how the corresponding two-dimensional range tree data structure can be extended to higher dimensions. A ***two-dimensional dictionary*** is a data structure for storing key-element items such that the key is a pair (x, y) of numbers, called the ***coordinates*** of the element. A two-dimensional dictionary D supports the following fundamental query operation:

findAllInRange(x_1, x_2, y_1, y_2): Return all the elements of D with coordinates (x, y) such that $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$.

Operation findAllInRange is the ***reporting*** version of the range-searching query, because it asks for all the items satisfying the range constraints. There is also a ***counting*** version of the range query, in which we are simply interested in the number of items in that range. We present data structures for answering two-dimensional range queries in the remainder of this section.

Note that a two-dimensional range-search operation is directly analogous to a one-dimensional range search, which is discussed in Section 3.2. The main ideas from one-dimensional searching can inform our thoughts about how to answer a two-dimensional range-search query, but more work is needed. For instance, one way to answer is a one-dimensional range search is to use a balanced binary tree, with the keys stored in sorted order. Then we can search for the lower end of a range, x_1, and the upper end of a range, x_2, and answer the query by reporting or counting all the elements in the search tree that are between these two locations using the in-order ordering. If the points are two-dimensional, though, then this approach can only be used to report or count points in the vertical strip between the lines $x = x_1$ and $x = x_2$. Thus, for two-dimensional range searching, something more is needed.

21.1.1 Two-Dimensional Range Searching

The two-dimensional range tree is a data structure that can implement a two-dimensional dictionary. It consists of a ***primary*** structure, which is a balanced binary search tree T, together with a number of ***auxiliary*** structures. (See Figure 21.1.)

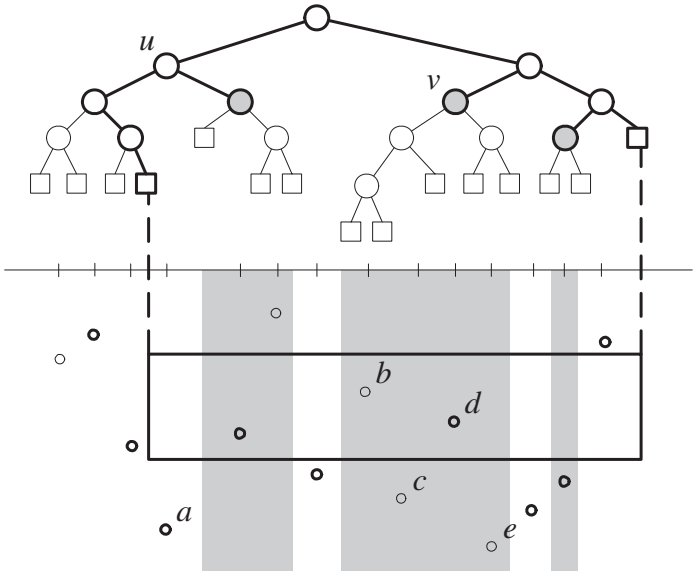


Figure 21.1: A set of items with two-dimensional keys represented by a two-dimensional range tree, and a range search on it. The primary structure T is shown. The nodes of T visited by the search algorithm are drawn with thick lines. The boundary nodes are white-filled, and the allocation nodes are gray-filled. Point a, stored at boundary node u, is outside the search range. The gray vertical strips cover the points stored at the auxiliary structures of the allocation nodes. For example, the auxiliary structure of node v stores points b, c, d, and e.

Specifically, each internal node in the primary structure T stores a reference to a related auxiliary structure. The function of the primary structure, T, is to support searching based on x-coordinates. To also support searching in terms of the y-coordinates, we use a collection of auxiliary data structures, each of which is a one-dimensional range tree that uses y-coordinates as its keys. The primary structure of T is a balanced binary search tree built using the x-coordinates of the items as the keys. An internal node v of T stores the following data:

- An item, whose coordinates are denoted by $x(v)$ and $y(v)$, and whose element is denoted by $\mathsf{element}(v)$.
- A one-dimensional range tree $T(v)$ that stores the same set of items as the subtree rooted at v in T (including v), but using the y-coordinates as keys.

We give the details of answering a two-dimensional range search with a range tree in Algorithm 21.2 (see also Figure 21.1).

Algorithm 2DTreeRangeSearch$(x_1, x_2, y_1, y_2, v, t)$:
Input: Search keys x_1, x_2, y_1, and y_2; node v in the primary structure T of a two-dimensional range tree; type t of node v
Output: The items in the subtree rooted at v whose coordinates are in the x-range $[x_1, x_2]$ and in the y-range $[y_1, y_2]$

```
if T.isExternal(v) then
    return ∅
if x1 ≤ x(v) ≤ x2 then
    if y1 ≤ y(v) ≤ y2 then
        M ← {element(v)}
    else
        M ← ∅
    if t = "left" then
        L ← 2DTreeRangeSearch(x1, x2, y1, y2, T.leftChild(v), "left")
        R ← 1DTreeRangeSearch(y1, y2, T.rightChild(v))
    else if t = "right" then
        L ← 1DTreeRangeSearch(y1, y2, T.leftChild(v))
        R ← 2DTreeRangeSearch(x1, x2, y1, y2, T.rightChild(v), "right")
    else
        // t = "middle"
        L ← 2DTreeRangeSearch(x1, x2, y1, y2, T.leftChild(v), "left")
        R ← 2DTreeRangeSearch(x1, x2, y1, y2, T.rightChild(v), "right")
else
    M ← ∅
    if x(v) < x1 then
        L ← ∅
        R ← 2DTreeRangeSearch(x1, x2, y1, y2, T.rightChild(v), t)
    else
        // x(v) > x2
        L ← 2DTreeRangeSearch(x1, x2, y1, y2, T.leftChild(v), t)
        R ← ∅
    return L ∪ M ∪ R
```

Algorithm 21.2: A recursive method for performing a two-dimensional range search in a two-dimensional range tree. The initial method call is 2DTreeRangeSearch$(x_1, x_2, y_1, y_2, T.\text{root}(), \text{"middle"})$. The algorithm is called recursively on all the boundary nodes with respect to the x-range $[x_1, x_2]$. Parameter t indicates whether v is a left, middle, or right boundary node. The method, 1DTreeRangeSearch, is the same as the method, RangeQuery, given in Algorithm 3.11.

Lemma 21.1: *A two-dimensional range tree storing n items uses $O(n \log n)$ space and can be constructed in $O(n \log n)$ time.*

Proof: The primary structure uses $O(n)$ space. There are n secondary structures. The size of an auxiliary structure is proportional to the number of items stored in it. An item stored at node v of the primary structure T is also stored at each auxiliary structure $T(u)$ such that u is an ancestor of v. Since tree T is balanced, node v has $O(\log n)$ ancestors. Hence, there are $O(\log n)$ copies of the item in the auxiliary structures. Thus, the total space used is $O(n \log n)$. The construction algorithm is left as an exercise (C-21.3). ■

The algorithm for operation findAllInRange(x_1, x_2, y_1, y_2) begins by performing what is essentially a one-dimensional range search on the primary structure T for the range $[x_1, x_2]$. Namely, we traverse down tree T in search of inside nodes. We make one important modification, however: when we reach an inside node v, instead of recursively visiting the subtree rooted at v, we perform a one-dimensional range search for the interval $[y_1, y_2]$ in the auxiliary structure of v.

We call ***allocation nodes*** the inside nodes of T that are children of boundary nodes. The algorithm visits the boundary nodes and the allocation nodes of T, but not the other inside nodes. Each boundary node v is classified by the algorithm as a ***left node***, ***middle node***, or ***right node***. A middle node is in the intersection of the search paths P_1 for x_1 and P_2 for x_2. A left node is in P_1 but not in P_2. A right node is in P_2 but not in P_1. At each allocation node v, the algorithm executes a one-dimensional range search on the auxiliary structure $T(v)$ for the y-range $[y_1, y_2]$.

Theorem 21.2: *A two-dimensional range tree T for a set of n items with two-dimensional keys uses $O(n \log n)$ space and can be constructed in $O(n \log n)$ time. Using T, a two-dimensional range-search query takes time $O(\log^2 n + s)$, where s is the number of elements reported.*

Proof: The space requirement and construction time follow from Lemma 21.1. We now analyze the running time of a range-search query performed with Algorithm 21.2 (2DTreeRangeSearch). We account for the time spent at each boundary node and allocation node of the primary structure T. The algorithm spends a constant amount of time at each boundary node. Since there are $O(\log n)$ boundary nodes, the overall time spent at the boundary nodes is $O(\log n)$. For each allocation node v, the algorithm spends $O(\log n_v + s_v)$ time doing a one-dimensional range search in auxiliary structure $T(v)$, where n_v is the number of items stored in $T(v)$ and s_v is the number of elements returned by the range search in $T(v)$. Denoting with A the set of allocation nodes, we have that the total time spent at the allocation nodes is proportional to $\sum_{v \in A}(\log n_v + s_v)$. Since $|A|$ is $O(\log n)$, $n_v \leq n$ and $\sum_{v \in A} \leq s$, we have that the overall time spent at the allocation nodes is $O(\log^2 n + s)$. We conclude that a two-dimensional range search takes $O(\log^2 n + s)$ time. ■

21.2 Priority Search Trees

In this section, we present the ***priority search tree*** structure, which can answer ***three-sided*** range queries on a set S of items with two-dimensional keys:

findAllInRange(x_1, x_2, y_1): Return all the items of S with coordinates (x, y) such that $x_1 \leq x \leq x_2$ and $y_1 \leq y$.

Geometrically, this query asks us to return all points between two vertical lines ($x = x_1$ and $x = x_2$) and above a horizontal line ($y = y_1$).

A ***priority search tree*** for set S is a binary tree storing the items of S that behaves like a binary search tree with respect to the x-coordinates, and like a heap with respect to the y-coordinates. For simplicity, let us assume that all the items of S have distinct x and y-coordinates. If set S is empty, T consists of a single external node. Otherwise, let $\bar{p}$ be the topmost item of S, that is, the item with maximum y-coordinate. We denote with $\hat{x}$ the median x-coordinate of the items in $S - \{\bar{p}\}$, and with S_L and S_R the subsets of $S - \{\bar{p}\}$ with items having x-coordinate less than or equal to $\hat{x}$ and greater than $\hat{x}$, respectively. We recursively define the priority search tree T for S as follows:

- The root T stores item $\bar{p}$ and the median x-coordinate $\hat{x}$.
- The left subtree of T is a priority search tree for S_L.
- The right subtree of T is a priority search tree for S_R.

For each internal node v of T, we denote with $\bar{p}(v)$, $\bar{x}(v)$, and $\bar{y}(v)$ the topmost item stored at v and its coordinates. Also, we denote with $\hat{x}(v)$ the median x-coordinate stored at v. An example of a priority search tree is shown in Figure 21.3.

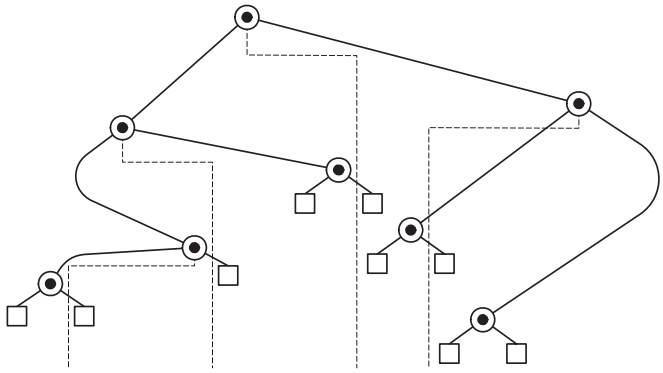


Figure 21.3: A set S of items with two-dimensional keys and a priority search tree T for S. Each internal node v of T is drawn as a circle around point $\bar{p}(v)$. The median x-coordinate $\hat{x}(v)$ is represented by a dashed line below node v that separates the items stored in the left subtree of v from those stored in the right subtree.

21.2.1 Constructing a Priority Search Tree

The y-coordinates of the items stored at the nodes of a priority search tree T satisfy the heap-order property (Section 5.3). That is, if u is the parent of v, then $\bar{y}(u) > \bar{y}(v)$. Also, the median x-coordinates stored at the nodes of T define a binary search tree (Section 3.1.1). These two facts motivate the term "priority search tree." Let us therefore explain how to construct a priority search tree from a set S of n two-dimensional items. We begin by sorting S by increasing x-coordinate, and then call the recursive method buildPST(S) shown in Algorithm 21.4.

Algorithm buildPST(S):
Input: A sequence S of n two-dimensional items, sorted by x-coordinate
Output: A priority search tree T for S

Create an elementary binary tree T consisting of single external node v
if !S.isEmpty() **then**
  Traverse sequence S to find the item $\bar{p}$ of S with highest y-coordinate
  Remove $\bar{p}$ from S
  $\bar{p}(v) \leftarrow \bar{p}$
  $\hat{p} \leftarrow S$.get($\lceil S$.size()$/2 \rceil$)
  $\hat{x}(v) \leftarrow x(\hat{p})$
  Split S into two subsequences, S_L and S_R, where S_L contains the items up to $\hat{p}$ (included), and S_R contains the remaining items
  $T_L \leftarrow$ buildPST(S_L)
  $T_R \leftarrow$ buildPST(S_R)
  T.expandExternal(v)
  Replace the left child of v with T_L
  Replace the right child of v with T_R
return T

Algorithm 21.4: Recursive construction of a priority search tree.

Lemma 21.3: *Given a set S of n two-dimensional items, a priority search tree for S uses $O(n)$ space, has height $O(\log n)$, and can be built in $O(n \log n)$ time.*

Proof: The $O(n)$ space requirement follows from the fact that every internal node of the priority search tree T stores a distinct item of S. The height of T follows from the halving of the number of nodes at each level. The preliminary sorting of the items of S by x-coordinate can be done in $O(n \log n)$ time using an asymptotically optimal sorting algorithm, such as heap sort or merge sort. The running time $T(n)$ of method buildPST (Algorithm 21.4) is characterized by the recurrence, $T(n) = 2T(n/2) + bn$, for some constant $b > 0$. Therefore, by the Master Theorem (11.4), $T(n)$ is $O(n \log n)$. ∎

21.2.2 Searching in a Priority Search Tree

We now show how to perform a three-sided range query findAllInRange(x_1, x_2, y_1) on a priority search tree T. We traverse down T in a fashion similar to that of a one-dimensional range search for the range $[x_1, x_2]$. One important difference, however, is that we only continue searching in the subtree of a node v if $y(v) \geq y_1$. We give the details of the algorithm for three-sided range searching in Algorithm 21.5 (PSTSearch) and we illustrate the execution of the algorithm in Figure 21.6.

Algorithm PSTSearch(x_1, x_2, y_1, v):
Input: Three-sided range, defined by x_1, x_2, and y_1, and a node v of a priority search tree T
Output: The items stored in the subtree rooted at v with coordinates (x, y), such that $x_1 \leq x \leq x_2$ and $y_1 \leq y$

```
if ȳ(v) < y1 then
    return ∅
if x1 ≤ x̄(v) ≤ x2 then
    M ← {p̄(v)}        // we should output p̄(v)
else
    M ← ∅
if x1 ≤ x̂(v) then
    L ← PSTSearch(x1, x2, y1, T.leftChild(v))
else
    L ← ∅
if x̂(v) ≤ x2 then
    R ← PSTSearch(x1, x2, y1, T.rightChild(v))
else
    R ← ∅
return L ∪ M ∪ R
```

Algorithm 21.5: Three-sided range searching in a priority search tree T. The algorithm is initially called with PSTSearch$(x_1, x_2, y_1, T.\text{root}())$.

Note that we have defined three-sided ranges to have a left, right, and bottom side, and to be unbounded at the top. This restriction was made without loss of generality, however, for we could have defined our three-sided range queries using any three sides of a rectangle. The priority search tree from such an alternate definition is similar to the one defined above, but "turned on its side."

Let us analyze the running time of method PSTSearch for answering a three-sided range-search query on a priority search tree T storing a set of n items with two-dimensional keys. We denote with s the number of items reported. Since we spend $O(1)$ time for each node we visit, the running time of method PSTSearch

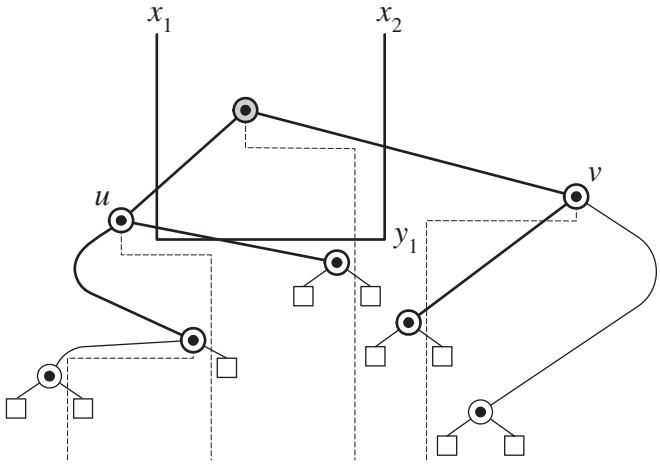


Figure 21.6: Three-sided range-searching in a priority search tree. The nodes visited are drawn with thick lines. The nodes storing reported items are gray-filled.

is proportional to the number of visited nodes.

Each node v visited by method PSTSearch is classified as follows:

- Node v is a ***boundary node*** if it is on the search path for x_1 or x_2 when viewing T as a binary search tree on the median x-coordinate stored at its nodes. The item stored at an internal boundary node may be inside or outside the three-sided range. By Lemma 21.3, the height of T is $O(\log n)$. Thus, there are $O(\log n)$ boundary nodes.
- Node v is an ***inside node*** if it is internal, it is not a boundary node, and $\bar{y}(v) \geq y_1$. The item stored at an internal node is inside the three-sided range. The number of inside nodes is no more than the number s of items reported.
- Node v is a ***terminal node*** if it is not a boundary node and, if internal, $\bar{y}(v) < y_1$. The item stored at an internal terminal node is outside the three-sided range. Each terminal node is the child of a boundary node or an inside node. Thus, the number of terminal nodes is at most twice the number of boundary nodes plus inside nodes. Thus, there are $O(\log n + s)$ terminal nodes.

Theorem 21.4: *A priority search tree T storing n items with two-dimensional keys uses $O(n)$ space and can be constructed in $O(n \log n)$ time, to answer three-sided range queries in $O(\log n + s)$ time, where s is the number of items reported.*

Of course, three-sided range queries are not as constrained as four-sided range queries, which can be answered in $O(\log^2 n + k)$ time using the range tree data structure discussed above. Still, priority search trees can be used to speed up the running time of answering standard four-sided, two-dimensional range queries. The resulting data structure, which is known as the ***priority range tree***, uses priority search trees as auxiliary structures in a way that achieves the same space bound as traditional range trees. We discuss this data structure next.

21.2.3 Priority Range Trees

Let T be a balanced binary search tree storing n items with two-dimensional keys, ordered according to their x-coordinates. We show how to augment T with priority search trees as auxiliary structures to answer (four-sided) range queries. The resulting data structure is called a ***priority range tree***.

To convert T into a priority range tree, we visit each internal node v of T other than the root and construct, as an auxiliary structure, a priority search tree $T(v)$ for the items stored in the subtree of T rooted at v. If v is a left child, $T(v)$ answers range queries for three-side ranges unbounded on the right. If v is a right child, $T(v)$ answers range queries for three-side ranges unbounded on the left. By Lemmas 21.1 and 21.3, a priority range tree uses $O(n \log n)$ space and can be constructed in $O(n \log n)$ time. The method for performing a two-dimensional range query in a priority range tree is given in Algorithm 21.7 (PSTRangeSearch).

Algorithm PSTRangeSearch(x_1, x_2, y_1, y_2, v):
Input: Search keys x_1, x_2, y_1, and y_2; node v in the primary structure T of a priority range tree
Output: The items in the subtree rooted at v whose coordinates are in the x-range $[x_1, x_2]$ and in the y-range $[y_1, y_2]$

```
if T.isExternal(v) then
    return ∅
if x1 ≤ x(v) ≤ x2 then
    if y1 ≤ y(v) ≤ y2 then
        M ← {element(v)}
    else
        M ← ∅
    L ← PSTSearch(x1, y1, y2, T(leftChild(v)).root())
    R ← PSTSearch(x2, y1, y2, T(rightChild(v)).root())
    return L ∪ M ∪ R
else if x(v) < x1 then
    return PSTRangeSearch(x1, x2, y1, y2, T.rightChild(v))
else
    // x2 < x(v)
    return PSTRangeSearch(x1, x2, y1, y2, T.leftChild(v))
```

Algorithm 21.7: Range searching in a priority range tree T. The algorithm is initially called with PSTRangeSearch$(x_1, x_2, y_1, y_2, T.\text{root}())$.

Theorem 21.5: *A priority range tree T for a set of n items with two-dimensional keys uses $O(n \log n)$ space and can be constructed in $O(n \log n)$ time. Using T, a two-dimensional range-search query takes time $O(\log n + s)$, where s is the number of elements reported.*

21.3 Quadtrees and k-d Trees

Multi-dimensional data sets often come from large applications; hence, we often desire linear-space structures for storing them. A general framework for designing such linear-space structures for d-dimensional data, where the dimensionality d is assumed to be a fixed constant, is based on an approach called the partition tree.

A ***partition tree*** is a rooted tree T that has at most n external nodes, where n is the number of d-dimensional points in our given set S. Each external node of a partition tree T stores a different small subset from S. Each internal node v in a partition tree T corresponds to a region of d-dimensional space, which is then divided into some number c of different cells or regions associated with v's children. For each region R associated with a child u of v, we require that all the points in u's subtree fall inside the region R. Ideally, the c different cells for v's children should easily be distinguished using a constant number of operations.

21.3.1 Quadtrees

The first partition tree data structure we discuss is the ***quadtree***. The main application for quadtrees is for sets of points that come from images, where x- and y-coordinates are integers, because the data points come from image pixels. In addition, they exhibit their best properties if the distributions of points is fairly nonuniform, with some areas being mostly empty and others being dense.

Suppose we are given a set S of n points in the plane. In addition, let R denote a square region that contains all the points of S (for example, R could be a bounding box of a 2048×2048 image that produced the set S). The quadtree data structure is a partition tree T such that the root r of T is associated with the region R. To get to the next level in T, we subdivide R into four equal-sized squares R_1, R_2, R_3, and R_4, and we associate each square R_i with a potential child of the root r. Specifically, we create a child v_i of r, if the square R_i contains a point in S. If a square R_i contains no points of S, then we create no child of r for it. This process of refining R into the squares R_1, R_2, R_3, and R_4 is called a ***split***.

The ***quadtree*** T is defined by recursively performing a split at each child v of r if necessary. That is, each child v of r has a square region R_i associated with it, and if the region R_i for v contains more than one point of S, then we perform a split at v, subdividing R_i into four equal-sized squares and repeating the above subdivision process at v. We continue in this manner, splitting squares that contain more than one point into four subsquares, and recursing on the nonempty subsquares, until we have separated all the points of S into individual squares. We then store each point p in S at the external node of T that corresponds to the smallest square in the subdivision process that contains p. We store at each internal node, v, a concise representation of the split that we performed for v.

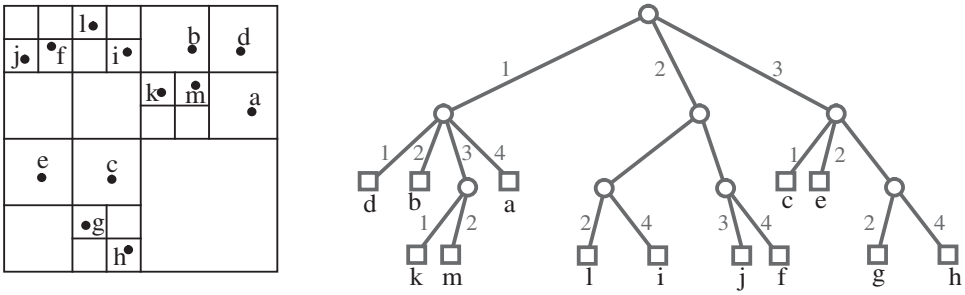


Figure 21.8: A quadtree. We illustrate an example point set and its corresponding quadtree data structure.

We illustrate an example point set and an associated quadtree in Figure 21.8. Note, however, that, contrary to the illustration, there is potentially no upper bound on the depth of a quadtree, as we have previously defined. For example, our point set S could contain two points that are very close to one another, and it may take a long sequence of splits before we separate these two points. Thus, it is customary for quadtree designers to specify some upper bound D on the depth of T. Given a set S of n points in the plane, we can construct a quadtree T for S so as to spend $O(n)$ time building each level of T. Thus, in the worst case, constructing such a depth-bounded quadtree takes $O(Dn)$ time.

Answering Range Queries with a Quadtree

One of the queries that quadtrees are often used to answer is range searching. Suppose that we are given a rectangle A aligned with the coordinate axes, and are asked to use a quadtree T to return all the points in S that are contained in A. The method for answering this query is quite simple. We start with the root r of T, and we compare the region R for r to A. If A and R do not intersect at all, then we are done—there are no points in the subtree rooted at r that fall inside A. Alternatively, if A completely contains R, then we simply enumerate all the external-node descendants of r. These are two simple cases. If instead R and A intersect, but A does not completely contain R, then we recursively perform this search on each child v of r.

In performing such a range-searching query, we can traverse the entire tree T and not produce any output in the worst case. Thus, the worst-case running time for performing a range query in a depth D quadtree, with n external nodes is $O(Dn)$. From a worst-case point of view, answering a range-searching query with a quadtree is actually worse than a brute-force search through the set S, which would take $O(n)$ time to answer a two-dimensional range query. In practice, however, the quadtree typically allows for range-searching queries to be processed faster than this.

21.3.2 k-d Trees

There is a drawback to quadtrees, which is that they do not generalize well to higher dimensions. In particular, each node in a four-dimensional analogue of a quadtree can have as many as 16 children. Each internal node in a d-dimensional quadtree can have as many as 2^d children. To overcome the out-degree drawback for storing data from dimensions higher than three, data structure designers often consider alternative partition tree structures that are binary.

Another kind of partition data structure is the k-d tree, which is similar to quadtree structure, but is binary. The k-d tree data structure is actually a family of partition tree data structures, all of which are binary partition trees for storing multidimensional data. Like the quadtree data structure, each node v in a k-d tree is associated with a rectangular region R, although in the case of k-d trees this region is not necessarily square. The difference is that when it comes time to perform a split operation for a node v in a k-d tree, it is done with a single line that is perpendicular to one of the coordinate axes. For three- or higher-dimensional data sets, this "line" is an axis-aligned hyperplane. Thus, no matter the dimensionality, a k-d tree is a binary tree, for we resolve a split by associating the part of v's region R to the "left" of the splitting line with v's left child, and associating the part of v's region R to the "right" of the splitting line with v's right child. As with the quadtree structure, we stop performing splits if the number of points in a region falls below some fixed constant threshold. (See Figure 21.9.)

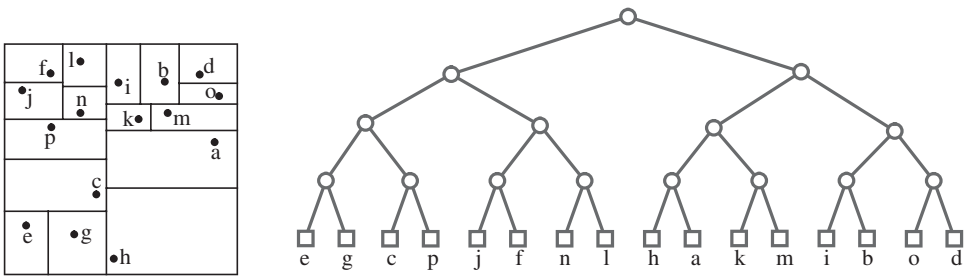


Figure 21.9: An example k-d tree.

There are two fundamentally different kinds of k-d trees, ***region-based*** k-d trees and ***point-based*** k-d trees. Region-based k-d trees are essentially binary versions of quadtrees. Each time a rectangular region R needs to be split in a region-based k-d tree, the region R is divided exactly in half by a line perpendicular to the longest side of R. If there is more than one longest side of R, then they are split in a "round-robin" fashion. On the other hand, point-based k-d trees, perform splits based on the distribution of points inside a rectangular region. The k-d tree of Figure 21.9 is point-based.

The method for splitting a rectangle R containing a subset $S' \subseteq S$ in a point-based k-d tree involves two steps. In the first step, we determine the dimension i that has the largest variation in dimension i from among those points in S'. This can be done, for example, by finding, for each dimension j, the points in S' with minimum and maximum dimension j values, and taking i to be the dimension with the largest gap between these two values. In the second step, we determine the median dimension i value from among all those points in S', and we split R with a line going through this median perpendicular to the dimension i axis. Thus, the split for R divides the set of points in S' in half, but may not divide the region R itself very evenly. Using a linear-time median-finding method (Section 9.2), this splitting step can be performed in $O(k|S'|)$ time. Therefore, the running time for building a k-d tree for a set of n points can be characterized by the following recurrence equation: $T(n) = 2T(n/2) + kn$, which is $O(kn \log n)$. Moreover, since we divide the size of the set of points associated with a node in two with each split, the height of T is $\lceil \log n \rceil$. Figure 21.9 illustrates a point-based k-d tree built using this algorithm.

The advantage of point-based k-d trees is that they are guaranteed to have nice depth and construction times. The drawback of point-based schemes is that they may give rise to "long-and-skinny" rectangular regions, which are usually considered bad for most k-d tree query methods.

Using k-d Trees for Nearest Neighbor Searching

Let us discuss how k-d trees can be used to do nearest-neighbor searching, where we are given a query point p and asked to find the point in S that is closest to p. A good way to use a k-d tree T to answer such a query is as follows. We first search down the tree T to locate the external node v with smallest rectangular region R that contains p. Any points of S that fall in R or in the region associated with v's sibling are then compared to find a current closest neighbor, q. We can then define a sphere centered at p and containing q as a current nearest-neighbor sphere, s. Given this sphere, we then perform a traversal of T (with a bottom-up traversal being preferred) to find any regions associated with external nodes of T that intersect s. If during this traversal we find a point closer than q, then we update the reference q to refer to this new point and we update the sphere s to contain this new point. We do not visit any nodes that have regions not intersecting s. When we have exhausted all possible alternatives, we output the current point q as the nearest neighbor of p. In the worst case, this method may take $O(n)$ time, but there are many different analytic and experimental analyses that suggest that the average running time is more like $O(\log n)$, using some reasonable assumptions about the distribution of points in S. In addition, there are a number of useful heuristics for speeding up this search in practice, with one of the best being the ***priority*** searching strategy, which says that we should explore subtrees of T in order of the distance of their associated regions to p.

21.4 Exercises

Reinforcement

R-21.1 What would be the worst-case space usage of a range tree, if the primary structure were not required to have $O(\log n)$ height?

R-21.2 Given a binary search tree, T, built on the x-coordinates of a set of n objects, describe an $O(n)$-time method for computing $\min_x(v)$ and $\max_x(v)$ for every node, v, in T.

R-21.3 Show that the high_y values in a priority search tree satisfy the heap-order property.

R-21.4 Argue why the algorithm for answering three-sided range-searching queries with a priority search tree is correct.

R-21.5 What is the worst-case depth of a k-d tree defined on n points in the plane? What about in higher dimensions?

R-21.6 Suppose a set S contains n two-dimensional points whose coordinates are all integers in the range $[0, N]$. What is the worst-case depth of a quadtree defined on S?

R-21.7 Draw a quadtree for the following set of points, assuming a 16×16 bounding box:

$$\{(1,2),(4,10),(14,3),(6,6),(3,15),(2,2),(3,12),(9,4),(12,14)\}.$$

R-21.8 Construct a k-d tree for the point set of Exercise R-21.7.

R-21.9 Construct a priority search tree for the point set of Exercise R-21.7.

Creativity

C-21.1 The $\min_x(v)$ and $\max_x(v)$ labels used in the two-dimensional range tree are not strictly needed. Describe an algorithm for performing a two-dimensional range-searching query in a two-dimensional range tree where each internal node of the primary structure only stores a key(v) label (which is the x-coordinate of its element). What is the running time of your method?

C-21.2 Let D be an ordered dictionary with n items implemented with a balanced search tree. Show how to implement the following method for D in time $O(\log n)$:

countAllInRange(k_1, k_2): Compute and return the number of items in D with key k such that $k_1 \leq k \leq k_2$.

Note that this method returns a single integer.

C-21.3 Give a pseudocode description of an algorithm for constructing a range tree from a set of n points in the plane in $O(n \log n)$ time.

C-21.4 Describe an efficient data structure for storing a set S of n items with ordered keys, so as to support a rankRange(a, b) method, which enumerates all the items with keys whose ***rank*** in S is in the range $[a, b]$, where a and b are integers in the interval $[0, n-1]$. Describe methods for object insertions and deletion, and characterize the running times for these and the rankRange method.

C-21.5 Design a static data structure (which does not support insertions and deletions) that stores a two-dimensional set S of n points and can answer, in $O(\log^2 n)$ time, queries of the form countAllInRange(a, b, c, d), which return the number of points in S with x-coordinates in the range $[a, b]$ and y-coordinates in the range $[c, d]$. What is the space used by this structure?

C-21.6 Design a data structure for answering countAllInRange queries (as defined in the previous exercise) in $O(\log n)$ time.

Hint: Think of storing auxiliary structures at each node that are "linked" to the structures at neighboring nodes.

C-21.7 Show how to extend the two-dimensional range tree so as to answer d-dimensional range-searching queries in $O(\log^d n)$ time for a set of d-dimensional points, where $d \geq 2$ is a constant.

Hint: Design a recursive data structure that builds a d-dimensional structure using $(d-1)$-dimensional structures.

C-21.8 Suppose we are given a range-searching data structure D that can answer range-searching queries for a set of n points in d-dimensional space for any fixed dimension d (like 8, 10, or 20) in time that is $O(\log^d n + k)$, where k is the number of answers. Show how to use D to answer the following queries for a set S of n rectangles in the plane:

- findAllContaining(x, y): Return an enumeration of all rectangles in S that contain the point (x, y).
- findAllIntersecting(a, b, c, d): Return an enumeration of all rectangles that intersect the rectangle with x-range $[a, b]$ and y-range $[c, d]$.

What is the running time needed to answer each of these queries?

C-21.9 Let S be a set of n intervals of the form $[a, b]$, where $a < b$. Design an efficient data structure that can answer, in $O(\log n + k)$ time, queries of the form contains(x), which asks for an enumeration of all intervals in S that contain x, where k is the number of such intervals. What is the space usage of your data structure?

C-21.10 Describe an efficient method for inserting an object into a (balanced) priority search tree. What is the running time of this method?

Applications

A-21.1 Quadtrees are often used for geometric environments defined by images, so the points involved can be normalized to be represented as pairs of fixed-precision binary numbers in the interval $[0, 1]$. Such environments also allow for quadtrees to be representable using one-dimensional search structures, as is explored in this exercise. So let (x, y) be such a point, with

$$x = 0.x_1x_2 \ldots x_k \text{ and } y = 0.y_1y_2 \ldots y_k,$$

in binary. Define interleave(x, y) to be the binary number, z, formed by interleaving of the binary numbers x and y, so

$$z = 0.x_1y_1x_2y_2 \ldots x_ky_k,$$

in binary. Assume we have a set, S, of n such points in the plane and we have constructed a quadtree, T, for this set of points. Let Z be an array that contains the points of S in lexicographical order by the results of calling interleave on each point. Show that for any node, v, in T, the descendants of v are stored in a contiguous subarray of Z and there is no point in this subarray that is not a descendant of v in T.

A-21.2 In some computer graphics and computer gaming applications, in order to save space, we might like to store a set of two-dimensional points in a single data structure that can be used for both nearest-neighbor queries and range searching. We have already discussed above how a k-d tree can be used to answer nearest-neighbor queries with good expected-time behavior. Show that a (round-robin) k-d tree defined on n two-dimensional points can also be used to answer range-search queries in $O(\sqrt{n} + s)$ time, where s is the number of points output by the query.

A-21.3 In some applications, such as in computer vision, an input set of two-dimensional points can be assumed to be given as pairs of integers, rather than arbitrary real numbers. Suppose, then, that you are given a set of n two-dimensional points such that each coordinate is in the range $[0, 4n]$. Show that you can construct a priority search tree for this set of points in $O(n)$ time.

A-21.4 In applications involving the use of quadtrees in memory-constrained devices, such as smartphones, we often want to optimize the data structure to make it more space efficient. One obvious improvement is to take a standard quadtree, T, and replace each chain of nodes having only one child with a single edge. This gives us a data structure known as the ***compressed quadtree***. Describe an efficient method for constructing a compressed quadtree for a set of n two-dimensional points. What is the time and space complexity for your algorithm?

A-21.5 A higher-dimensional version of a quadtree is known as an ***octree***, since, in three dimensions it divides each cube into 8 subcubes and recursively constructs an octree for each nonempty subcube as a child. Suppose we are given such a three-dimensional structure, but we are only interested in two of the three dimensions, since we can only display two-dimensional images on a computer screen. Describe an efficient method for converting an octree into a quadtree defined on two of the dimensions used to construct the octree.

A-21.6 When the Sload Digital Sky Survey decided to create a data structure for storing the objects identified by their projects, they needed a method for searching for two-dimensional points that are on a sphere rather than being in a rectangle. So they started with a sphere, cut it into quarters by two perpendicular great circles going through the poles and then into eight pieces by one more cut using a great circle through the equator. This divided the sphere into eight regions that "almost" equilateral triangles. Viewing each region as a perfect equilateral triangle, describe a recursive way to subdivide each one of these triangles that results in a set of children that are also equilaterial triangles, in a fashion suggestive of a quadtree. Describe how your structure could be used to effectively answer circular range-search queries to find all the points inside a given circle on this sphere.

A-21.7 The quadtree is often used by people who never worry about its worst-case depth being high. There is a good reason for this belief, if one can assume some randomness exists for the input set of points. Use a Chernoff bound (Section 19.5) to show that the height of a quadtree defined on n points in the unit square chosen uniformly and independently at random is $O(\log n)$ with high probability.

Hint: If we divide a square into four equal-sized squares, and assign n points uniformly and independently at random to the square, consider the probability that any subsquare has more than $n/2$ of the points.

A-21.8 In computer graphics and computer gaming environments, a common heuristic is to approximate a complex two-dimensional object by a smallest enclosing rectangle whose sides are parallel to the coordinate axes, which is known as a ***bounding box***. Using this technique allows system designers to generalize range searching over points to more complex objects. So, suppose you are given a collection of n complex two-dimensional objects, together with a set of their bounding boxes,

$$\mathcal{S} = \{R_1, R_2, \ldots, R_n\}.$$

Suppose further that for some reason you have a data structure, D, that can store n four-dimensional points so as to answer four-dimensional range-search queries in $O(\log^3 n + s)$ time, where s is the number of points in the query range. Explain how you can use D to answer two-dimensional range queries for the rectangles in $\mathcal{S}$, given a query rectangle, R, would return every bounding box, R_i, in $\mathcal{S}$, such that R_i is completely contained inside R. Your query should run in $O(\log^3 n + s)$ time, where s is the number of bounding boxes that are output as being completely inside the query range, R.

A-21.9 Consider the previous exercise, but instead of explaining how to use D to answer box-containment range queries, now explain how to build an efficient data structure, D, that can store four-dimensional range queries for four-dimensional points in $O(\log^3 n + s)$ time, where s is the number of answers. What is the space complexity of your data structure, D?

Chapter Notes

Multi-dimensional search trees are discussed in books by Mehlhorn [159], Samet [181, 182], and Wood [217]. Please see these books for an extensive discussion of the history of multidimensional search trees, including various data structures for solving range queries. Priority search trees are due to McCreight [149], although Vuillemin [213] introduced this structure earlier under the name "Cartesian trees." They are also known as "treaps," as described by McCreight [149] and Seidel and Aragon [191]. Edelsbrunner [61] shows how priority search trees can be used to answer two-dimensional range queries. Arya *et al.* [15, 16] present the balanced box decomposition tree and show it can be used for approximate nearest-neighbor and range searching. The reader interested in recent developments for range-searching data structures is referred to the book chapters by Agarwal [3, 4] or the survey paper by Matoušek [147].