1

Interval, Segment, Range, and Priority Search Trees

D. T. Lee
Academia Sinica

1.1 Introduction

In this Chapter we introduce four basic data structures that are of fundamental importance and have many applications as we will briefly cover them in later sections. They are *interval trees*, *segment trees*, *range trees*, and *priority search trees*. Consider for example the following problems. Suppose we have a set of *iso-oriented rectangles* in the planes. A set of rectangles are said to be *iso-oriented* if their edges are parallel to the coordinate axes. The subset of iso-oriented rectangles define a *clique*, if their common intersection is nonempty. The *largest* subset of rectangles whose common intersection is non-empty is called a *maximum clique*. The problem of finding this largest subset with a non-empty common intersection is referred to as the *maximum clique problem* for a rectangle intersection graph[14, 16].[1] The k-dimensional, $k \geq 1$, analog of this problem is defined similarly. In 1-dimensional case we will have a set of *intervals* on the real line, and an *interval intersection graph*, or simply *interval graph*. The maximum clique problem for interval graphs is to find a largest subset of intervals whose common intersection is non-empty. The cardinality of the maximum clique is sometimes referred to as the *density* of the set of intervals.

The problem of finding a subset of objects that satisfy a certain property is often referred to as *searching problem*. For instance, given a set of numbers $S = \{x_1, x_2, \ldots, x_n\}$, where

[1] A rectangle intersection graph is a graph $G = (V, E)$, in which each vertex in V corresponds to a rectangle, and two vertices are connected by an edge in E, if the corresponding rectangles intersect.

0-8493-8597-0/01/$0.00+$1.50

$x_i \in \Re$, $i = 1, 2, \ldots, n$, the problem of finding the subset of numbers that lie between a range $[\ell, r]$, i.e., $F = \{x \in S | \ell \leq x \leq r\}$, is called a (1D) *range search* problem[5, 22].

To deal with this kind of geometric searching problem, we need to have appropriate data structures to support efficient searching algorithms. The data structure is assumed to be *static*, i.e., the input set of objects is given *a priori*, and *no* insertions or deletions of the objects are allowed. If the searching problem satisfies *decomposability property*, i.e., if they are *decomposable*[2], then there are general *dynamization* schemes available[21], that can be used to convert static data structures into *dynamic* ones, where *insertions* and *deletions* of objects are permitted. Examples of decomposable searching problems include the *membership* problem in which one queris if a point p in S. Let S be partitioned into two subsets S_1 and S_2, and Member(p, S) returns *yes*, if $p \in S$, and *no* otherwise. It is easy to see that Member(p, S)= OR(Member(p, S_1), Member(p, S_2)), where OR is a boolean operator.

1.2 Interval Trees

Consider a set S of intervals, $S = \{I_i | i = 1, 2, \ldots, n\}$, each of which is specified by an ordered pair, $I_i = [\ell_i, r_i], \ell_i, r_i \in \Re, \ell_i \leq r_i, i = 1, 2, \ldots, n$.

An *interval tree*[8, 9], *Interval_Tree*(S), for S is a rooted augmented binary search tree, in which each node v has a key value, $v.key$, two tree pointers $v.left$ and $v.right$ to the left and right subtrees, respectively, and an auxiliary pointer, $v.aux$ to an augmented data structure, and is recursively defined as follows:

- The root node v associated with the set S, denoted Interval_Tree_root(S), has key value $v.key$ equal to the median of the $2 \times |S|$ endpoints. This key value $v.key$ divides S into three subsets S_ℓ, S_r and S_m, consisting of sets of intervals lying totally to the *left* of $v.key$, lying totally to the *right* of $v.key$ and containing $v.key$ respectively. That is, $S_\ell = \{I_i | r_i < v.key\}$, $S_r = \{I_j | v.key < \ell_j\}$ and $S_m = \{I_k | \ell_k \leq v.key \leq r_k\}$.
- Tree pointer $v.left$ points to the left subtree rooted at Interval_Tree_root(S_ℓ), and tree pointer $v.right$ points to the right subtree rooted at Interval_Tree_root(S_r).
- Auxiliary pointer $v.aux$ points to an augmented data structure consisting of two sorted arrays, SA$(S_m.left)$ and SA$(S_m.right)$ of the set of left endpoints of the intervals in S_m and the set of right endpoints of the intervals in S_m respectively. That is, $S_m.left = \{\ell_i | I_i \in S_m\}$ and $S_m.right = \{r_i | I_i \in S_m\}$.

1.2.1 Construction of Interval Trees

The following is a pseudo code for the recursive construction of the interval tree of a set S of n intervals. Without loss of generality we shall assume that the endpoints of these n intervals are all distinct. See Fig. 1.1(a) for an illustration.

function Interval_Tree(S)
/* It returns a pointer v to the root, Interval_Tree_root(S), of the interval tree for a set S

[2] A searching problem is said to be *decomposable* if and only if $\forall x \in T_1, A, B \in 2^{T_2}, Q(x, A \cup B) = \bigcirc(Q(x, A), Q(x, B))$ for some efficiently computable associative operator $\bigcirc$ on the elements of T_3, where Q is a mapping from $T_1 \times 2^{T_2}$ to T_3.[1, 3]

of intervals. */

Input: A set S of n intervals, $S = \{I_i | i = 1, 2, \ldots, n\}$ and each interval $I_i = [\ell_i, r_i]$, where ℓ_i and r_i are the left and right endpoints, respectively of I_i, $\ell_i, r_i \in \Re$, and $\ell_i \leq r_i, i = 1, 2, \ldots, n$.

Output: An interval tree, rooted at Interval_Tree_root(S).

Method:

1. **if** $S = \emptyset$, **return nil**.
2. Create a node v such that $v.key$ equals x, where x is the middle point of the set of endpoints so that there are exactly $|S|/2$ endpoints less than x and greater than x respectively. Let $S_\ell = \{I_i | r_i < x\}$, $S_r = \{I_j | x < \ell_j\}$ and $S_m = \{I_k | \ell_k \leq x \leq r_k\}$.
3. Set $v.left$ equal to Interval_Tree(S_ℓ).
4. Set $v.right$ equal to Interval_Tree(S_r)
5. Create a node w which is the root node of an auxiliary data structure associated with the set S_m of intervals, such that $w.left$ and $w.right$ point to two sorted arrays, SA(S_m.*left*) and SA(S_m.*right*), respectively. SA(S_m.*left*) denotes an array of left endpoints of intervals in S_m in *ascending* order, and SA(S_m.*right*) an array of right endpoints of intervals in S_m in *descending* order.
6. Set $v.aux$ equal to node w.

Note that this recursively built interval tree structure requires $O(n)$ space, where n is the cardinality of S, since each interval is either in the left subtree, the right subtree or the middle augmented data structure.

1.2.2 Example and Its Applications

Fig. 1.1(b) illustrates an example of an interval tree of a set of intervals, spread out as shown in Fig. 1.1(a).

The interval trees can be used to handle quickly queries of the following form.

Enclosing Interval Searching Problem [11, 15] Given a set S of n intervals and a query point, q, report all those intervals containing q, i.e., find a subset $F \subseteq S$ such that $F = \{I_i | \ell_i \leq q \leq r_i\}$.

Overlapping Interval Searching Problem [4, 8, 9] Given a set S of n intervals and a query interval Q, report all those intervals in S overlapping Q, i.e., find a subset $F \subseteq S$ such that $F = \{I_i | I_i \cap Q \neq \emptyset\}$.

The following pseudo code solves the **Overlapping Interval Searching Problem** in $O(\log n) + |F|$ time. It is invoked by a call to Overlapping_Interval_Search(v, Q, F), where v is Interval_Tree_root(S), and F, initially set to be $\emptyset$, will contain the set of intervals overlapping query interval Q.

procedure Overlapping_Interval_Search(v, Q, F)

Input: A set S of n intervals, $S = \{I_i | i = 1, 2, \ldots, n\}$ and each interval $I_i = [\ell_i, r_i]$, where ℓ_i and r_i are the left and right endpoints, respectively of I_i, $\ell_i, r_i \in \Re$, and $\ell_i \leq r_i, i = 1, 2, \ldots, n$ and a query interval $Q = [\ell, r], \ell, r \in \Re$.

Output: A subset $F = \{I_i | I_i \cap Q \neq \emptyset\}$.

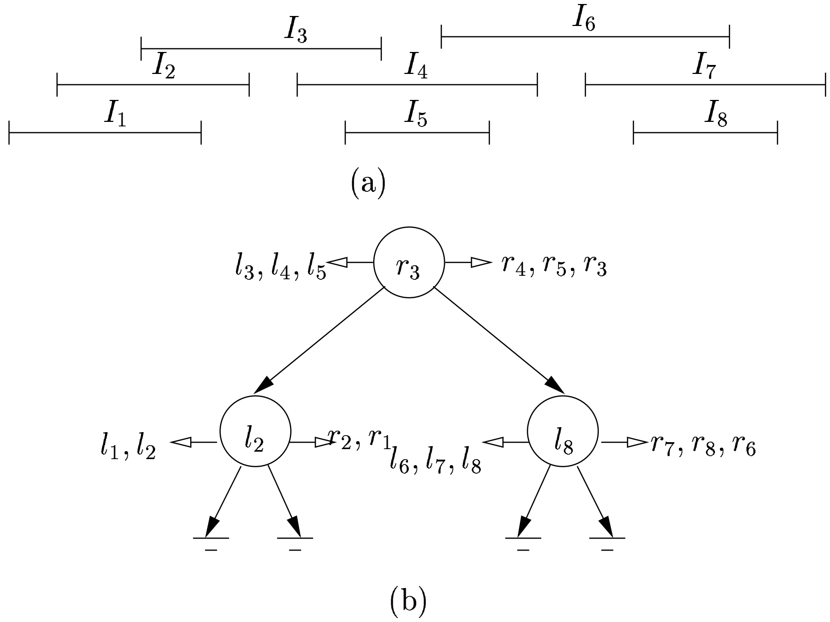


FIGURE 1.1: Interval tree for $S = \{I_1, I_2, \ldots, I_8\}$ and its interval models

Method:

1. Set $F = \emptyset$ initially.
2. **if** v is **nil return**.
3. **if** ($v.key \in Q$) **then**
   for each interval I_i in the augmented data structure pointed to by $v.aux$
   do $F = F \cup \{I_i\}$
   Overlapping_Interval_Search($v.left, Q, F$)
   Overlapping_Interval_Search($v.right, Q, F$)
4. **if** ($r < v.key$) **then**
   for each left endpoint ℓ_i in the sorted array pointed to by $v.aux.left$ such that $\ell_i \geq r$ **do** $F = F \cup \{I_i\}$
   Overlapping_Interval_Search($v.left, Q, F$)
5. **if** ($\ell > v.key$) **then**
   for each right endpoint r_i in the sorted array pointed to by $v.aux.right$ such that $r_i \geq \ell$ **do** $F = F \cup \{I_i\}$
   Overlapping_Interval_Search($v.right, Q, F$)

It is obvious to see that an interval I in S overlaps a query interval $Q = [\ell, r]$ if (i) Q contains the left endpoint of I, (ii) Q contains the right endpoint of I, or (iii) Q is totally contained in I. Step 3 reports those intervals I that contain a point $v.key$ which is also contained in Q. Step 4 reports intervals in either case (i) or (iii) and Step 5 reports intervals in either case (ii) or (iii).

Note the special case of **procedure** Overlapping_Interval_Search(v, Q, F) when we set the query interval $Q = [\ell, r]$ so that its left and right endpoints coincide, i.e., $\ell = r$ will report all the intervals in S containing a query point, solving the *Enclosing Interval Searching Problem.*

However, if one is interested in the problem of finding a special type of overlapping intervals, *e.g.*, all intervals containing or contained in a given query interval[11, 15], the interval tree data structure does not necessarily yield an efficient solution. Similarly, the interval tree does not provide an effective method to handle queries about the set of intervals, *e.g.,* the maximum clique, or the *measure*, the total length of the union of the intervals[10, 17].

We conclude with the following theorem.

THEOREM 1.1 *The Enclosing Interval Searching Problem and Overlapping Interval Searching Problem for a set S of n intervals can both be solved in $O(\log n)$ time (plus time for output) and in linear space.*

1.3 Segment Trees

The segment tree structure, originally introduced by Bentley[5, 22], is a data structure for intervals whose endpoints are *fixed* or *known a priori.* The set $S = \{I_1, I_2, \ldots, I_n\}$ of n intervals, each of which represented by $I_i = [\ell_i, r_i]$, $\ell_i, r_i \in \Re$, $\ell_i \leq r_i$, is represented by a data array, Data_Array(S), whose entries correspond to the endpoints, ℓ_i or r_i, and are sorted in non-decreasing order. This sorted array is denoted SA[1..N], $N = 2n$. That is, $SA[1] \leq SA[2] \leq \ldots \leq SA[N]$, $N = 2n$. We will in this section use the indexes in the range $[1, N]$ to refer to the entries in the sorted array SA[1..N]. For convenience we will be working in the transformed domain using indexes, and a comparison involving a point $q \in \Re$ and an index $i \in \aleph$, unless otherwise specified, is performed in the original domain in $\Re$. For instance, $q < i$ is interpreted as $q <$SA[i].

The segment tree structure, as will be domonstrated later, can be useful in finding the *measure* of a set of intervals. That is, the length of the union of a set of intervals. It can also be used to find the maximum clique of a set of intervals. This structure can be generalized to higher dimensions.

1.3.1 Construction of Segment Trees

The segment tree, as the interval tree discussed in Section 1.2 is a rooted augment binary search tree, in which each node v is associated with a range of integers $v.range = [v.B, v.E]$, $v.B, v.E \in \aleph$, $v.B < v.E$, representing a range of indexes from $v.B$ to $v.E$, a key, $v.key$ that split $v.range$ into two subranges, each of which is associated with each child of v, two tree pointers $v.left$ and $v.right$ to the left and right subtrees, respectively, and an auxiliary pointer, $v.aux$ to an augmented data structure. Given integers s and t, with $1 \leq s < t \leq N$, the segment tree, denoted Segment_Tree(s, t), is recursively described as follows.

- The root node v, denoted Segment_Tree_root(s, t), is associated with the range $[s, t]$, and $v.B = s$ and $v.E = t$.
- If $s + 1 = t$ then we have a leaf node v with $v.B = s, v.E = t$ and $v.key =$ nil.
- Otherwise (*i.e.,* $s + 1 < t$), let m be the mid-point of s and t, or $m = \lfloor \frac{(v.B+v.E)}{2} \rfloor$. Set $v.key = m$.
- Tree pointer $v.left$ points to the left subtree rooted at Segment_Tree_root(s, m),

and tree pointer $v.right$ points to the right subtree rooted at Segment_Tree_root(m,t).

- Auxiliary pointer $v.aux$ points to an augmented data structure, associated with the range $[s,t]$, whose content depends on the usage of the segment tree.

The following is a pseudo code for the construction of a segment tree for a range $[s,t]$ $s < t, s, t \in \aleph$, and the construction of a set of n intervals whose endpoints are indexed by an array of integers in the range $[1,N], N = 2n$ can be done by a call to Segment_Tree$(1,N)$. See Fig. 1.2(b) for an illustration.

function Segment_Tree(s,t)
/* It returns a pointer v to the root, Segment_Tree_root(s,t), of the segment tree for the range $[s,t]$.*/

Input: A set $\mathcal{N}$ of integers, $\{s, s+1, \ldots, t\}$ representing the indexes of the endpoints of a subset of intervals.

Output: A segment tree, rooted at Segment_Tree_root(s,t).

Method:

1. Let v be a node, $v.B = s, v.E = t$, $v.left = v.right = nil$, and $v.aux$ to be determined.
2. **if** $s+1 = t$ **then return**.
3. Let $v.key = m = \lfloor \frac{(v.B+v.E)}{2} \rfloor$.
4. $v.left$ = Segment_Tree_root(s,m)
5. $v.right$ = Segment_Tree_root(m,t)

The parameters $v.B$ and $v.E$ associated with node v define a range $[v.B, v.E]$, called a *standard range* associated with v. The standard range associated with a leaf node is also called an *elementary range*. It is straightforward to see that Segment_Tree(s,t) constructed in **function** Segment_Tree(s,t) described above is balanced, and has height, denoted Segment_Tree.height, equal to $\lceil \log_2(t-s) \rceil$.

We now introduce the notion of *canonical covering* of a range $[s,t]$, where $s,t \in \aleph$ and $1 \leq s < t \leq N$. A node v in Segment_Tree$(1,N)$ is said to be in the *canonical covering* of $[s,t]$ if its associated standard range satisfies this property $[v.B, v.E] \subseteq [s,t]$, while that of its parent node does not. It is obvious that if a node v is in the canonical covering, then its *sibling node*, *i.e.*, the node with the same parent node as the present one, is not, for otherwise the common parent node would have been in the canonical covering. Thus at each level there are *at most* two nodes that belong to the canonical covering of $[s,t]$.

Thus, for each range $[s,t]$ the number of nodes in its canonical covering is at most $\lceil \log_2(t-s) \rceil + \lfloor \log_2(t-s) \rfloor - 2$. In other words, a range $[s,t]$ (or respectively an interval $[s,t]$) can be decomposed into at most $\lceil \log_2(t-s) \rceil + \lfloor \log_2(t-s) \rfloor - 2$ standard ranges (or respectively subintervals)[5, 22].

To identify the nodes in a segment tree T that are in the canonical covering of an interval $I = [b,e]$, representing a range $[b,e]$, we perform a call to Interval_Insertion(v,b,e,Q), where v is Segment_Tree_root(S). The procedure Interval_Insertion(v,b,e,Q) is defined below.

procedure Interval_Insertion(v,b,e,Q)
/* It returns a queue Q of nodes $q \in T$ such that $[v.B, v.E] \subseteq [b,e]$ and its parent node u whose $[u.B, u.E] \not\subseteq [b,e]$.*/

Input: A segment tree T pointed to by its root node, v = Segment_Tree_root$(1,N)$, for a set S of intervals.

Output: A queue Q of nodes in T that are in the canonical covering of $[b, e]$
Method:

1. Initialize an output queue Q, which supports insertion ($\Rightarrow Q$) and deletion ($\Leftarrow Q$) in constant time.
2. **if** $([v.B, v.E] \subseteq [b, e])$ **then append** $[b, e]$ to v, $v \Rightarrow Q$, and **return**.
3. **if** $(b < v.key)$ **then** Interval_Insertion$(v.left, b, e, Q)$
4. **if** $(v.key < e)$ **then** Interval_Insertion$(v.right, b, e, Q)$

To **append** $[b, e]$ to a node v means to insert interval $I = [b, e]$ into the auxiliary structure associated with node v to indicate that node v whose standard range is totally contained in I is in the canonical covering of I. If the auxiliary structure $v.aux$ associated with node v is an array, the operation **append** $[b, e]$ to a node v can be implemented as $v.aux[j{+}{+}] = I$,

procedure Interval_Insertion(v, b, e, Q) described above can be used to represent a set S of n intervals in a segment tree by performing the insertion operation n times, one for each interval. As each interval I can have at most $O(\log n)$ nodes in its canonical covering, and hence we perform at most $O(\log n)$ **append** operations for each insertion, the total amount of space required in the auxiliary data structrues reflecting all the nodes in the canonical covering is $O(n \log n)$.

Deletion of an interval represented by a range $[b, e]$ can be done similarly, except that the **append** operation will be replaced by its corresponding inverse operation **remove** that removes the node from the list of canonical covering nodes.

THEOREM 1.2 *The segment tree for a set S of n intervals can be contructed in $O(n \log n)$ time, and if the auxiliary structure for each node v contains a list of intervals containing v in the canonical covering, then the space required is $O(n \log n)$.*

1.3.2 Examples and Its Applications

Fig. 1.2(b) illustrates an example of a segment tree of the set of intervals, as shown in Fig. 1.2(a). The integers, if any, under each node v represent the indexes of intervals that contain the node in its cononical covering. For example, Interval I_2 contains nodes labeled by standard ranges $[2, 4]$ and $[4, 7]$.

We now describe how segment trees can be used to solve the *Enclosing Interval Searching Problem* defined before and the *Maximum Clique Problem* of a set of intervals, which is defined below.

Maximum Density or Maximum Clique of a set of Intervals [12, 16, 23] Given a set S of n intervals, find a maximum subset $C \subseteq S$ such that the common intersection of intervals in C is non-empty. That is, $\bigcap_{I_i \in C \subseteq S} I_i \neq \emptyset$ and $|C|$ is maximized. $|C|$ is called the *density* of the set.

The following pseudo code solves the **Enclosing Interval Searching Problem** in $O(\log n) + |F|$ time, where F is the output set. It is invoked by a call Point_in_Interval_Search(v, q, F), where v is Segment_Tree_root(S), and F initially set to be $\emptyset$, will contain the set of intervals containing a query point q.

procedure Point_in_Interval_Search(v, q, F)
/* It returns in F the list of intervals stored in the segment tree pointed to by v and containing query point q */

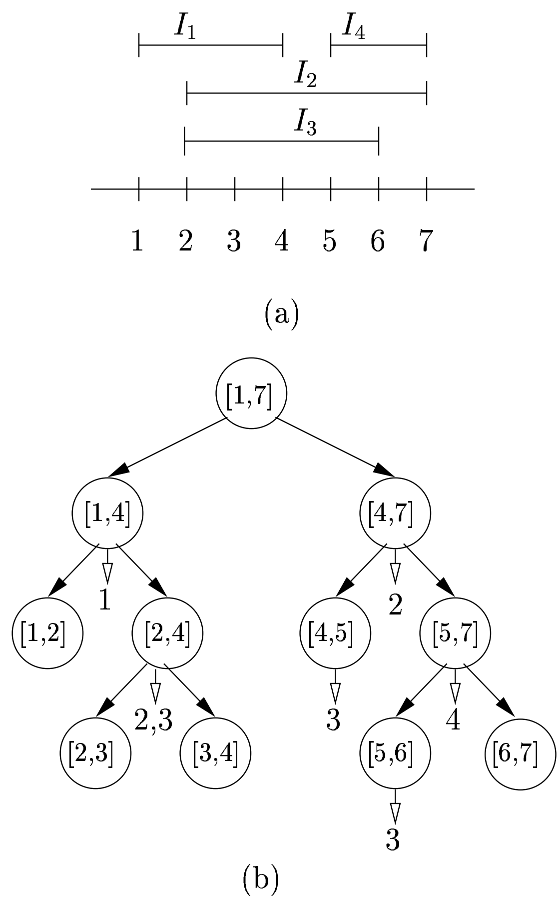


FIGURE 1.2: Segment tree of $S = \{I_1, I_2, I_3, I_4\}$ and its interval models

Input: A segment tree representing a set S of n intervals, $S = \{I_i | i = 1, 2, \ldots, n\}$ and a query point $q \in \Re$. The auxiliary structure $v.aux$ associated with each node v is a list of intervals $I \in S$ that contain v in their canonical covering.

Output: A subset $F = \{I_i | \ell_i \leq q \leq r_i\}$.

Method:

1. **if** v is **nil or** $(q < v.B$ **or** $q > v.E)$ **then return**.
2. **if** $(v.B \leq q \leq v.E)$ **then**
   for each interval I_i in the auxiliary data structure pointed to by $v.aux$ **do**
   $F = F \cup \{I_i\}$.
3. **if** $(q \leq v.key)$ **then** Point_in_Interval_Search($v.left, q, F$)
4. **else** (/* *i.e.*, $q > v.key$ */) Point_in_Interval_Search($v.right, q, F$)

We now address the problem of finding the maximum clique of a set S of intervals, $S = \{I_1, I_2, \ldots, I_n\}$, where each interval $I_i = [\ell_i, r_i]$, and $\ell_i \leq r_i, \ell_i, r_i \in \Re, i = 1, 2, \ldots, n$. There are other approaches, such as plane-sweep [12, 16, 22, 23] that solve this problem within the same complexity.

For this problem we introduce an auxiliary data structure to be stored at each node v. $v.aux$ will contain two pieces of information: one is the *number* of intervals containing v

in the canonical covering, denoted $v.\sharp$, and the other is the *clique size*, denoted $v.clq$. The *clique size* of a node v is the size of the maximum clique whose common intersection is contained in the standard range associated with v. It is defined to be equal to the *larger* of the two numbers: $v.left.\sharp + v.left.clq$ and $v.right.\sharp + v.right.clq$. For a leaf node v, $v.clq = 0$. The size of the maximum clique for the set of intervals will then be stored at the root node Segment_Tree_root(S) and is equal to the sum of $v.\sharp$ and $v.clq$, where $v =$ Segment_Tree_root(S). It is obvious that the space needed for this segment tree is linear.

As this data structure supports insertion of intervals incrementally, it can be used to answer the maximum clique of the current set of intervals as the intervals are inserted into (or deleted from) the segment tree T. The following pseudo code finds the size of the maximum clique of a set of intervals.

function Maximum_Clique(S)
/* It returns the size of the maximum clique of a set S of intervals. */

Input: A set S of n intervals and the segment tree T rooted at Segment_Tree_root(S).

Output: An integer, which is the size of the maximum clique of S.

Method: Assume that $S = \{I_1, I_2, \ldots, I_n\}$ and that the endpoints of the intervals are represented by the indexes of a sorted array containing these endpoints.

1. Initialize $v.clq = v.\sharp = 0$ for all nodes v in T.
2. **for** each interval $I_i = [\ell_i, r_i] \in S$, $i = 1, 2, \ldots, n$ **do**
   /* Insert I_i into the tree and update $v.\sharp$ and $v.clq$ for all visited nodes and those nodes in the canonical covering of I_i */
3. **begin**
4. $s =$ Find_split-node(v, ℓ_i, r_i), where v is Segment_Tree_root(S). (See below)
   Let the root-to-split-node(s)-path be denoted P.
5. /* Find all the canonical covering nodes in the left subtree of s */
   Traverse along the left subtree from s following the *left* tree pointer, and find a *leftward* path, $s_{\ell_1}, s_{\ell_2}, \ldots$ till node s_{ℓ_L} such that $s_{\ell_1} = s.left$, $s_{\ell_k} = s_{\ell_{k-1}}.left$, for $k = 2, \ldots, L$. Note that the standard ranges of all these nodes overlap I_i, but the standard range associated with $s_{\ell_L}.left$ is totally disjoint from I_i. s_{ℓ_L} is s_{ℓ_1} only if the standard range of s_{ℓ_1} is totally contained in I_i, *i.e.*, s_{ℓ_1} is in the canonical covering of I_i. Other than this, the right child of each node on this *leftward* path belongs to the canonical covering of I_i.
6. Increment $u.\sharp$ for all nodes u that belong to the canonical covering of I_i.
7. Update $s_{\ell_j}.clq$ according to the definition of *clique size* for all nodes on the *leftward* path in reverse order, *i.e.,* starting from node s_L to s_{ℓ_1}.
8. /* Find all the canonical covering nodes in the right subtree of s */
   Similarly we traverse along the right subtree from s along the *right* tree pointer, and find a *rightward* path. Perform **Steps** 5 to 7.
9. Update $s.clq$ for the split node s after the clique sizes of both left and right child of node s have been updated.
10. Update $u.clq$ for all the nodes u on the root-to-split-node-path P in reverse order, starting from node s to the root.
11. **end**
12. **return** $(v.\sharp + v.clq)$, where $v =$ Segment_Tree_root(S).

function Find_split-node(v, b, e)
/* Given a segment tree T rooted at v and an interval $I = [b, e] \subseteq [v.B, v.E]$, this procedure returns the *split-node* s such that either $[s.B, s.E] = [b, e]$ or $[s_\ell.B, s_\ell.E] \cap [b, e] \neq \emptyset$ and $[s_r.B, s_r.E] \cap [b, e] \neq \emptyset$, where s_ℓ and s_r are the left child and right child of s respectively. */

1. **if** $[v.B, v.E] = [b, e]$ **then return** v.
2. **if** $(b < v.key)$ **and** $(e > v.key)$ **then return** v.
3. **if** $(e \leq v.key)$ **then return** Find_split-node$(v.\mathit{left}, b, e)$
4. **if** $(b \geq v.key)$ **then return** Find_split-node$(v.\mathit{right}, b, e)$

Note that in **procedure** Maximum_Clique(S) it takes $O(\log n)$ time to process each interval. We conclude with the following theorem.

THEOREM 1.3 *Given a set $S = \{I_1, I_2, \ldots, I_n\}$ of n intervals, the maximum clique of $S_i = \{I_1, I_2, \ldots, I_i\}$ can be found in $O(i \log i)$ time and linear space, for each $i = 1, 2, \ldots, n$, by using a segment tree.*

We note that the above procedure can be adapted to find the maximum clique of a set of *hyperrectangles* in k-dimensions for $k > 2$ in time $O(n^k)$.[16]

1.4 Range Trees

Consider a set S of points in k-dimensional space $\Re^k$. A range tree for this set S of points is a data structure that supports general range queries of the form $[x_\ell^1, x_r^1] \times [x_\ell^2, x_r^2] \times \ldots \times [x_\ell^k, x_r^k]$, where each range $[x_\ell^i, x_r^i], x_\ell^i, x_r^i \in \Re, x_\ell^i \leq x_r^i$ for all $i = 1, 2, \ldots, k$, denotes an interval in $\Re$. The cartesian product of these k ranges is referred to as a kD range. In 2-dimensional space, a 2D range is simply an axes-parallel rectangle in $\Re^2$. The *range search problem* is to find all the points in S that satisfy any range query. In 1-dimension, the range search problem can be easily solved in logarithmic time using a sorted array or a balanced binary search tree. The 1D range is simply an interval $[x_\ell, x_r]$. We first do a binary search using x_ℓ as searched key to find the first node v whose key is no less than x_ℓ. Once v is located, the rest is simply to retrieve the nodes, one at a time, until the node u whose key is greater than x_r. We shall in this section describe an augmented binary search tree which is easily generalized to higher dimensions.

1.4.1 Construction of Range Trees

A range tree is primarily a binary search tree augmented with an auxiliary data structure. The root node v, denoted Range_Tree_root(S), of a kD-range tree[5, 18, 22, 24] for a set S of points in k-dimensional space $\Re^k$, *i.e.*, $S = \{p_i = (x_i^1, x_i^2, \ldots, x_i^k), i = 1, 2, \ldots, n\}$, where $p_i.x^j = x_i^j \in \Re$ is the jth-coordinate value of point p_i, for $j = 1, 2, \ldots, k$, is associated with the entire set S. The key stored in $v.key$ is to partition S into two approximately equal subsets S_ℓ and S_r, such that all the points in S_ℓ and in S_r lie to the left and to the right, respectively of the hyperplane $H^k : x^k = v.key$. That is, we will store the median of the kth coordinate values of all the points in S in $v.key$ of the root node v, *i.e.*, $v.key = p_j.x^k$ for some point p_j such that S_ℓ contains points p_ℓ, $p_\ell.x^k \leq v.key$, and S_r contains points p_r, $p_r.x^k > v.key$. Each node v in the kD-range tree, as before, has two tree pointers, $v.\mathit{left}$ and $v.\mathit{right}$, to the roots of its left and right subtrees respectively. The node pointed to by *v.left* will be associated with the set S_ℓ and the node pointed to by *v.right* will be associated with the set S_r. The auxiliary pointer *v.aux* will point to an augmented data structure, in our case a $(k-1)$D-range tree.

A $1D$-range tree is a sorted array of all the points $p_i \in S$ such that the entries are drawn from the set $\{x_i^1 | i = 1, 2, \ldots, n\}$ sorted in nondecreasing order. This 1D-range tree supports the 1D range search in logarithmic time.

The following is a pseudo code for a kD-range tree for a set S of n points in k-dimensional space. See Fig. 1.3(a) and (b) for an illustration. Fig. 1.4(c) is a schematic representation of a kD-range tree.

function kD_Range_Tree(k, S)
/* It returns a pointer v to the root, kD_Range_Tree_root(k, S), of the kD-range tree for a set $S \subseteq \Re^k$ of points, $k \geq 1$. */

Input: A set S of n points in $\Re^k$, $S = \{p_i = (x_i^1, x_i^2, \ldots, x_i^k), i = 1, 2, \ldots, n\}$, where $x_i^j \in \Re$ is the jth-coordinate value of point p_i, for $j = 1, 2, \ldots, k$.

Output: A kD-range tree, rooted at kD_Range_Tree_root(k, S).

Method:

1. **if** $S = \emptyset$, **return nil**.
2. **if** $(k = 1)$ create a sorted array SA(S) pointed to by a node v containing the set of the 1st coordinate values of all the points in S, *i.e.,* SA$(1, S)$ has $\{p_i.x^1 | i = 1, 2, \ldots, n\}$ in nondecreasing order. **return** (v).
3. Create a node v such that *v.key* equals the *median* of the set $\{p_i.x^k |$ kth coordinate value of $p_i \in S, i = 1, 2, \ldots, n\}$. Let S_ℓ and S_r denote the subset of points whose kth coordinate values are not greater than and are greater than *v.key* respectively. That is, $S_\ell = \{p_i \in S\} | p_i.x^k \leq v.key\}$ and $S_r = \{p_j \in S\} | p_j.x^k > v.key\}$.
4. $v.left$ = kD_Range_Tree(k, S_ℓ)
5. $v.right$ = kD_Range_Tree(k, S_r)
6. $v.aux$ = kD_Range_Tree$(k-1, S)$

As this is a recursive algorithm with two parameters, k and $|S|$, that determine the recursion depth, it is not immediately obvious how much time and how much space are needed to construct a kD-range tree for a set of n points in k-dimensional space.

Let $T(n, k)$ denote the time taken and $S(n, k)$ denote the space required to build a kD-range tree of a set of n points in $\Re^k$. The following are recurrence relations for $T(n, k)$ and $S(n, k)$ respectively.

$$T(n,k) = \begin{cases} O(1) & \text{if } n = 1 \\ O(n \log n) & \text{if } k = 2 \\ 2T(n/2, k) + T(n, k-1) + O(n) & \text{otherwise} \end{cases}$$

$$S(n,k) = \begin{cases} O(1) & \text{if } n = 1 \\ O(n) & \text{if } k = 1 \\ 2S(n/2, k) + S(n, k-1) + O(1) & \text{otherwise} \end{cases}$$

Note that in 1-dimension, we need to have the points sorted and stored in a sorted array, and thus $T(n, 1) = O(n \log n)$ and $S(n, 1) = O(n)$. The solutions of $T(n, k)$ and $S(n, k)$ to the above recurrence relations are $T(n, k) = O(n \log^{k-1} n + n \log n)$ for $k \geq 1$ and $S(n, k) = O(n \log^{k-1} n)$ for $k \geq 1$. For a general multidimensional divide-and-conquer scheme, and solutions to the recurrence relation, please refer to Bentley[2] and Monier[20] respectivley.

We conclude that

THEOREM 1.4 *The kD-range tree for a set of n points in k-dimensional space can be constructed in* $O(n \log^{k-1} n + n \log n)$ *time and* $O(n \log^{k-1} n)$ *space for* $k \geq 1$.

1.4.2 Examples and Its Applications

Fig. 1.3(b) illustrates an example of a range tree for a set of points in 2-dimensions shown in Fig. 1.3(a). This list of integers under each node represents the indexes of points in ascending x-coordinates. Fig. 1.4 illustrates a general schematic representation of a kD-range tree, which is a *layered* structure[5, 22].

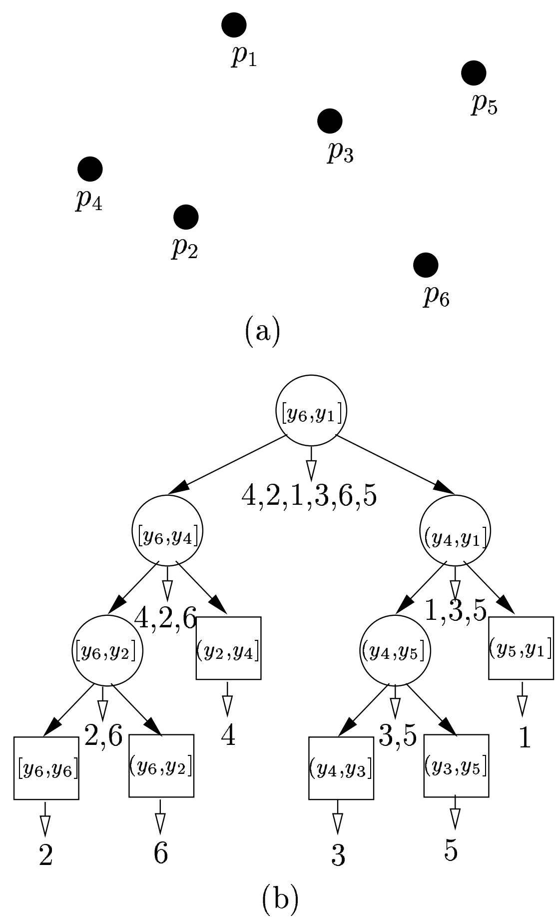


FIGURE 1.3: 2D-range tree of $S = \{p_1, p_2, \ldots, p_6\}$, where $p_i = (x_i, y_i)$.

We now discuss how we make use of a range tree to solve the range search problem. We shall use 2D-range tree as an example for illustration purposes. It is rather obvious

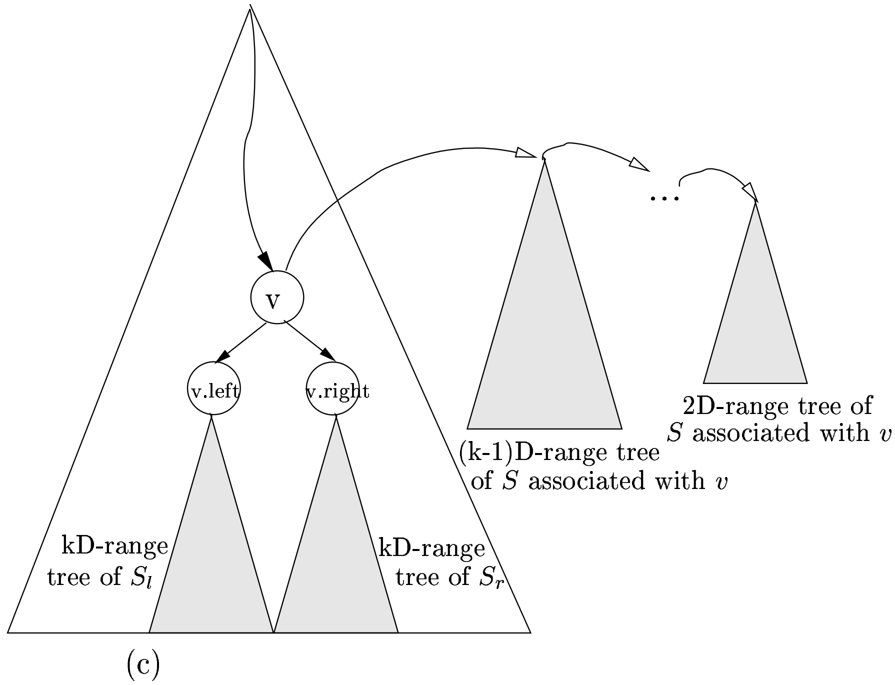


FIGURE 1.4: A schematic representation of a (layered) kD-range tree, where S is the set associated with node v.

to generalize it to higher dimensions. Recall we have a set S of n points in the plane $\Re^2$ and 2D range query $Q = [x_\ell, x_r] \times [y_\ell, y_r]$. Let us assume that a 2D-range tree rooted at 2D_Range_Tree_root(S) is available. Recall also that associated with each node v in the range tree there is a *standard range* for the set S_v of points represented in the subtree rooted at node v, in this case $[v.B, v.E]$ where $v.B = \min\{p_i.y\}$ and $v.E = \max\{p_i.y\}$ for all $p_i \in S_v$. $v.key$ will split the standard range into two standard subranges $[v.B, v.key]$ and $[v.key, v.E]$ each associated with the root nodes $v.left$ and $v.right$ of the left and right subtrees of v respectively.

The following pseudo code reports in F the set of points in S that lie in the range $Q = [x_\ell, x_r] \times [y_\ell, y_r]$. It is invoked by 2D_Range_Search($v, x_\ell, x_r, y_\ell, y_r, F$), where v is the root, 2D_Range_Tree_root(S), and F, initially empty will return all the points in S that lie in the range $Q = [x_\ell, x_r] \times [y_\ell, y_r]$.

procedure 2D_Range_Search($v, x_\ell, x_r, y_\ell, y_r, F$)
/* It returns F containing all the points in the range tree rooted at node v that lie in $[x_\ell, x_r] \times [y_\ell, y_r]$. */

Input: A set S of n points in $\Re^2$, $S = \{p_i | i = 1, 2, \ldots, n\}$ and each point $p_i = (p_i.x, p_i.y)$, where $p_i.x$ and $p_i.y$ are the x- and y-coordinates of p_i, $p_i.x, p_i.y \in \Re, i = 1, 2, \ldots, n$.

Output: A set F of points in S that lie in $[x_\ell, x_r] \times [y_\ell, y_r]$.

Method:

1. Start from the root node v to find the split-node s, s = Find_split-node(v, y_ℓ, y_r), such that $s.key$ lies in $[y_\ell, y_r]$.

2. **if** s is a leaf, **then** 1D_Range_Search($s.aux, x_\ell, x_r, F$) that checks in the sorted array pointed to by $s.aux$, which contains just a point p, to see if its x-coodinate $p.x$ lies in the x-range $[x_\ell, x_r]$
3. $v = s.left$.
4. **while** v is not a leaf **do**
   if ($y_\ell \leq v.key$) **then**
   1D_Range_Search($v.right.aux, x_\ell, x_r, F$)
   $v = v.left$

   else $v = v.right$
5. (/* v is a leaf, and check node $v.aux$ directly */)
   1D_Range_Search($v.aux, x_\ell, x_r, F$)
6. $v = s.right$
7. **while** v is not a leaf **do**
   if ($y_r > v.key$) **then**
   1D_Range_Search($v.left.aux, x_\ell, x_r, F$)
   $v = v.right$

   else $v = v.left$
8. (/* v is a leaf, and check node $v.aux$ directly */)
   1D_Range_Search($v.aux, x_\ell, x_r, F$)

procedure 1D_Range_Search(v, x_ℓ, x_r, F) is very straightforward. v is a pointer to a sorted array SA. We first do a binary search in SA looking for the first element no less than x_ℓ and then start to report in F those elements no greater than x_r. It is obvious that **procedure** 2D_Range_Search finds all the points in Q in $O(\log^2 n)$ time. Note that there are $O(\log n)$ nodes for which we need to invoke 1D_Range_Search in their auxiliary sorted arrays. These nodes v are in the canonical covering[3] of the y-range $[y_\ell, y_r]$, since its associated standard range $[v.B, v.E]$ is totally contained in $[y_\ell, y_r]$, and the 2D-range search problem is now reduced to the 1D-range search problem.

This is not difficult to see that the 2D-range search problem can be answered in time $O(\log^2 n)$ plus time for output, as there are $O(\log n)$ nodes in the canonical covering of a given y-range and for each node in the canonical covering we spend $O(\log n)$ time for dealing with the 1D-range search problem.

However, with a modification to the auxiliary data structure, one can achieve an optimal query time of $O(\log n)$, instead of $O(\log^2 n)$[6, 7, 24]. This is based on the observation that in each of the 1D-range search subproblem associated with each node in the canonical covering, we perform the same query, reporting points whose x-coordinates lie in the x-range $[x_\ell, x_r]$. More specifically we are searching for the smallest element no less than x_ℓ.

The modification is performed on the sorted array associated with each of the node in the 2D_Range_Tree(S).

Consider the root node v. As it is associated with the entire set of points, $v.aux$ points to the sorted array containing the x-coordinates of *all* the points in S. Let this sorted array be denoted $SA(v)$ and the entries, $SA(v)_i, i = 1, 2, \ldots, |S|$, are sorted in nondecreasing order of the x-coordinate values. In addition to the x-coordinate value, each entry also contains

[3]See the definition of the canonical covering defined in Section 1.3.1.

the index of the corresponding point. That is, $SA(v)_i.key$ and $SA(v)_i.index$ contain the x-coordinate of p_j respectively, where $SA(v)_i.index = j$ and $SA(v)_i.key = p_j.x$.

We shall augment each entry $SA(v)_i$ with two pointers, $SA(v)_i.left$ and $SA(v)_i.right$. They are defined as follows. Let v_ℓ and v_r denote the roots of the left and right subtrees of v, *i.e.*, $v.left = v_\ell$ and $v.right = v_r$. $SA(v)_i.left$ points to the entry $SA(v_\ell)_j$ such that entry $SA(v_\ell)_j.key$ is the smallest among all key values $SA(v_\ell)_j.key \geq SA(v)_i.key$. Similarly, $SA(v)_i.right$ points to the entry $SA(v_r)_k$ such that entry $SA(v_r)_k.key$ is the smallest among all key values $SA(v_r)_k.key \geq SA(v)_i.key$.

These two augmented pointers, $SA(v)_i.left$ and $SA(v)_i.right$, possess the following property: If $SA(v)_i.key$ is the smallest key such that $SA(v)_i.key \geq x_\ell$, then $SA(v_\ell)_j.key$ is also the smallest key such that $SA(v_\ell)_j.key \geq x_\ell$. Similarly $SA(v_r)_k.key$ is the smallest key such that $SA(v_r)_k.key \geq x_\ell$.

Thus if we have performed a binary search in the auxiliary sorted array $SA(v)$ associated with node v locating the entry $SA(v)_i$ whose key $SA(v)_i.key$ is the smallest key such that $SA(v)_i.key \geq x_\ell$, then following the left (respectively right) pointer $SA(v)_i.left$ (respectively $SA(v)_i.right$) to $SA(v_\ell)_j$ (respectively $SA(v_r)_k$), the entry $SA(v_\ell)_j.key$ (respectively $SA(v_r)_k.key$) is also the smallest key such that $SA(v_\ell)_j.key \geq x_\ell$ (respectively $SA(v_r)_k.key \geq x_\ell$). Thus there is no need to perform an additional binary search in the auxiliary sorted array $SA(v.left)$ (respectively $SA(v.right)$).

With this additional modification, we obtain an *augmented* 2D-range tree and the following theorem.

THEOREM 1.5 *The 2D-range search problem for a set of n points in the 2-dimensional space can be solved in time $O(\log n)$ plus time for output, using an augmented 2D-range tree that requires $O(n \log n)$ space.*

The following procedure is generalized from **procedure** 2D_Range_Search$(v, x_\ell, x_r, y_\ell, y_r, F)$ discussed in Section 1.4.2 taken into account the augmented auxiliary data structure. It is invoked by kD_Range_Search(k, v, Q, F), where v is the root kD_Range_Tree_root(S) of the range tree, Q is the k-range, $[x_\ell^1, x_r^1] \times [x_\ell^2, x_r^2] \times \ldots \times [x_\ell^k, x_r^k]$, represented by a two dimensional array, such that $Q_i.\ell = x_\ell^i$ and $Q_i.r = x_r^i$, and F, initially empty, will contain all the points that lie in Q.

procedure kD_Range_Search(k, v, Q, F). /* It returns F containing all the points in the range tree rooted at node v that lie in k-range, $[x_\ell^1, x_r^1] \times [x_\ell^2, x_r^2] \times \ldots \times [x_\ell^k, x_r^k]$, where each range $[x_\ell^i, x_r^i], x_\ell^i = Q_i.\ell, x_r^i = Q_i.r \in \Re, x_\ell^i \leq x_r^i$ for all $i = 1, 2, \ldots, k$, denotes an interval in $\Re$. */

Input: A set S of n points in $\Re^k$, $S = \{p_i | i = 1, 2, \ldots, n\}$ and each point $p_i = (p_i.x^1, p_i.x^2, \ldots, p_i.x^k)$, where $p_i.x^j \in \Re$, are the jth-coordinates of p_i, $j = 1, 2, \ldots, k$.

Output: A set F of points in S that lie in $[x_\ell^1, x_r^1] \times [x_\ell^2, x_r^2] \times \ldots \times [x_\ell^k, x_r^k]$.

Method:

1. **if** $(k > 2)$ **then**
   - Start from the root node v to find the split-node s, s = Find_split-node(v, Q_ℓ^k, Q_r^k), such that $s.key$ lies in $[Q_\ell^k, Q_r^k]$.
   - **if** s is a leaf, **then** check in the sorted array pointed to by $s.aux$, which contains just a point p. $p \Rightarrow F$ if its coodinate values lie in Q. **return**
   - $v = s.left$.

- **while** v is not a leaf **do**
  if $(Q_\ell^k \leq v.key)$
  then kD_Range_Search$(k-1, v.right.aux, Q, F)$.
  $v = v.left$
  else $v = v.right$
- (/* v is a leaf, and check node $v.aux$ directly */)
  Check in the sorted array pointed to by $v.aux$, which contains just a point p. $p \Rightarrow F$ if its coodinate values lie in Q. **return**
- $v = s.right$
- **while** v is not a leaf **do**
  if $(Q_r^k > v.key)$
  then kD_Range_Search$(k-1, v.left.aux, Q, F)$.
  $v = v.right$
  else $v = v.left$
- (/* v is a leaf, and check node $v.aux$ directly */)
  Check in the sorted array pointed to by $v.aux$, which contains just a point p. $p \Rightarrow F$ if its coodinate values lie in Q. **return**

2. **else** /* $k \leq 2$*/
3. **if** $k = 2$ **then**

- Do binary search in sorted array $SA(v)$ associated with node v, using $Q_1.\ell$ (x_ℓ^1) as key to find entry o_v such that $SA(v)_{o_v}$'s key, $SA(v)_{o_v}.key$ is the smallest such that $SA(v)_{o_v}.key \geq Q_1.\ell$,
- Find the split-node s, $s =$ Find_split-node(v, x_ℓ^2, x_r^2), such that $s.key$ lies in $[x_\ell^2, x_r^2]$. Record the root-to-split-node path from v to s, following *left* or *right* tree pointers.
- Starting from entry o_v $(SA(v)_i)$ follow pointers $SA(v)_{o_v}.left$ or $SA(v)_{o_v}.right$ according to the v-to-s path to point to entry $SA(s)_{o_s}$ associated with $SA(s)$.
- **if** s is a leaf, **then** check in the sorted array pointed to by $s.aux$, which contains just a point p. $p \Rightarrow F$ if its coodinate values lie in Q. **return**
- $v = s.left$, $o_v = SA(s)_{o_s}.left$.
- **while** v is not a leaf **do**
  if $(Q_2.\ell \leq v.key)$
  then $\ell = SA(v)_{o_v}.right$
  while $(SA(v.right)_\ell.key \leq Q_1.r)$ **do**
  point $p_m \Rightarrow F$, where $m = SA(v.right)_\ell.index$
  $\ell{+}{+}$
  $v = v.left$, $o_v = SA(v)_{o_v}.left$
  else $v = v.right$, $o_v = SA(v)_{o_v}.right$
- (/* v is a leaf, and check node $v.aux$ directly */)
  Check in the sorted array pointed to by $v.aux$, which contains just a point p. $p \Rightarrow F$ if its coodinate values lie in Q.
- $v = s.right$, $o_v = SA(s)_{o_s}.right$.
- **while** v is not a leaf **do**
  if $(Q_2.r > v.key)$
  then $\ell = SA(v)_{o_v}.left$

while $(SA(v.left)_\ell.key \le Q_1.r)$ **do**
point $p_m \Rightarrow F$, where $m = SA(v.left)_\ell.index$
ℓ++

else $v = v.left$, $o_v = SA(v)_{o_v}.left$

- (/* v is a leaf, and check node $v.aux$ directly */)
Check in the sorted array pointed to by $v.aux$, which contains just a point p. $p \Rightarrow F$ if its coodinate values lie in Q.

The following recurrence relation for the query time $Q(n, k)$ of the kD-range search problem, can be easily obtained:

$$Q(n,k) = \begin{cases} O(1) & \text{if } n = 1 \\ O(\log n) + \mathcal{F} & \text{if } k = 2 \\ \Sigma_{v \in CC} Q(n_v, k-1) + O(\log n) & \text{otherwise} \end{cases}$$

where $\mathcal{F}$ denotes the output size, and n_v denotes the size of the subtree rooted at node v that belongs to the canonical covering CC of the query. The solution is $Q(n,k) = O(\log^{k-1} n) + \mathcal{F}$[5, 22].

We conclude with the following theorem.

THEOREM 1.6 *The kD-range search problem for a set of n points in the k-dimensional space can be solved in time $O(\log^{k-1} n)$ plus time for output, using an augmented kD-range tree that requires $O(n \log^{k-1} n)$ space for $k \ge 1$.*

1.5 Priority Search Trees

The priority search tree was originally introduced by McCreight[19]. It is a hybrid of two data structures, binary search tree and a priority queue.[13] A *priority queue* is a queue and supports the following operations: insertion of an item and deletion of the minimum (highest priority) item, so called *delete_min* operation. Normally the delete_min operation takes constant time, while updating the queue so that the minimum element is readily accessible takes logarithmic time. However, searching for an element in a priority queue will normally take linear time. To support efficient searching, the priority queue is modified to be a priority search tree. We will give a formal definition and its construction later. As the priority search tree represents a set S of elements, each of which has two pieces of information, one being a key from a totally ordered set, say the set $\Re$ of real numbers, and the other being a notion of priority, also from a totally ordered set, for each element, we can model this set S as a set of points in 2-dimensional space. The x- and y-coordinates of a point p represents the key and the priority respectively. For instance, consider a set of jobs $S = \{J_1, J_2, \ldots, J_n\}$, each of which has a release time $r_i \in \Re$ and a priority $p_i \in \Re, i = 1, 2, \ldots, n$. Then each job J_i can be represented as a point q such that $q.x = r_i, q.y = p_i$.

The priority search tree can be used to support queries of the form, find, among a set S of n points, the point p with minimum $p.y$ such that its x-coordinate lies in a given range $[\ell, r]$, *i.e.*, $\ell \le p.x \le r$. As can be shown later, this query can be answered in $O(\log n)$ time.

1.5.1 Construction of Priority Search Trees

As before, the root node, Priority_Search_Tree_root(S), represents the entire set S of points. Each node v in the tree will have a key $v.key$, an auxiliary data $v.aux$ containing the index of the point and its priority, and two pointers $v.left$ and $v.right$ to its left and right subtrees respectively such that all the key values stored in the left subtree are less than $v.key$, and all the key values stored in the right subtree are greater than $v.key$. The following is a pseudo code for the recursive construction of the priority search tree of a set S of n points in $\Re^2$. See Fig. 1.5(a) for an illustration.

function Priority_Search_Tree(S)
/* It returns a pointer v to the root, Priority_Search_Tree_root(S), of the priority search tree for a set S of points. */

Input: A set S of n points in $\Re^2$, $S = \{p_i | i = 1, 2, \ldots, n\}$ and each point $p_i = (p_i.x, p_i.y)$, where $p_i.x$ and $p_i.y$ are the x- and y-coordinates of p_i, $p_i.x, p_i.y \in \Re, i = 1, 2, \ldots, n$.
Output: A priority search tree, rooted at Priority_Search_Tree_root(S).
Method:

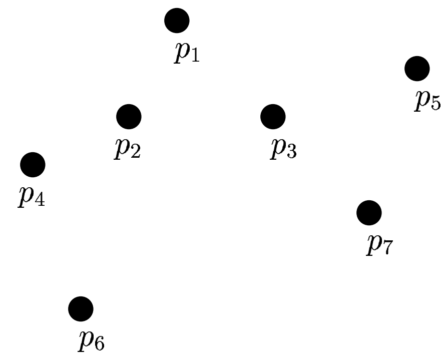
1. **if** $S = \emptyset$, **return nil**.
2. Create a node v such that $v.key$ equals the *median* of the set $\{p.x | p \in S\}$, and $v.aux$ contains the index i of the point p_i whose y-coordinate is the minimum among all the y-coordinates of the set S of points *i.e.*, $p_i.y = \min\{p.y | p \in S\}$.
3. Let $S_\ell = \{p \in S \setminus \{p_{v.aux}\} | p.x \leq v.key\}$ and $S_r = \{p \in S \setminus \{p_{v.aux}\} | p.x > v.key\}$ denote the set of points whose x-coordinates are less than or equal to $v.key$ and greater than $v.key$ respectively.
4. $v.left$= Priority_Search_Tree_root(S_ℓ).
5. $v.right$= Priority_Search_Tree_root(S_r).
6. **return** v.

Thus, Priority_Search_Tree_root(S) is a minimum heap data structure with respect to the y-coordinates, *i.e.,* the point with minimum y-coordinate can be accessed in constant time, and is a balanced binary search tree for the x-coordinates. Implicitly the root node v is associated with an x-range $[x_\ell, x_r]$ representing the span of the x-coordinate values of all the points in the whole set S. The root of the left subtree pointed to by $v.left$ is associated with the x-range $[x_\ell, v.key]$ representing the span of the x-coordinate values of all the points in the set S_ℓ and the root of the right subtree pointed to by $v.right$ is associated with the x-range $[v.key, x_r]$ representing the span of the x-coordinate values of all the points in the set S_r. It is obvious that this algorithm takes $O(n \log n)$ time and linear space. We summarize this in the following.

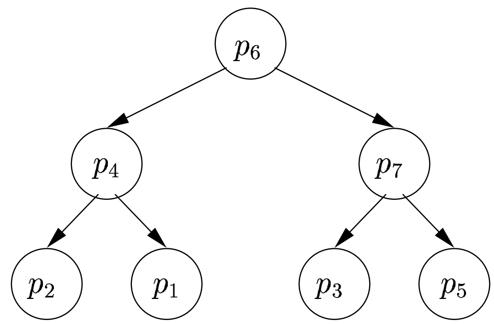
THEOREM 1.7 *The priority search tree for a set S of n points in $\Re^2$ can be constructed in $O(n \log n)$ time and linear space.*

1.5.2 Examples and Its Applications

Fig. 1.5 illustrates an example of a priority search tree of the set of points. Note that the root node contains p_6 since its y-coordinate value is the minimum.



(a)



(b)

FIGURE 1.5: Priority search tree of $S = \{p_1, p_2, \ldots, p_7\}$

We now illustrate a usage of the priority search tree by an example. Consider a so-called *grounded 2D range search problem* for a set of n points in the plane. As defined in Section 1.4.2, a 2D range search problem is to find all the points $p \in S$ such that $p.x$ lies in an x-range $[x_\ell, x_r], x_\ell \le x_r$ and $p.y$ lies in a y-range $[y_\ell, y_r]$. When the y-range is of the form $[-\infty, y_r]$ then the 2D range is referred to as *grounded* 2D range or sometimes as *1.5D* range, and the 2D range search problem as *grounded* 2D range search or *1.5D* range search problem.

Grounded 2D Range Search Problem Given a set S of n points in the plane $\Re^2$, with preprocessing allowed, find the subset F of points whose x- and y-coordinates satisfy a grounded 2D range query of the form $[x_\ell, x_r] \times [-\infty, y_r], x_\ell, x_r, y_r \in \Re, x_\ell \le x_r$.

The following pseudo code solves this problem optimally. We assume that a priority search tree for S has been constructed via procedure Priority_Search_Tree(S). The answer will be obtained in F via an invokation to Priority_Search_Tree_Range_Search(v, x_ℓ, x_r, y_r, F), where v is Priority_Search_Tree_root(S).

procedure Priority_Search_Tree_Range_Search(v, x_ℓ, x_r, y_r, F)
/* v points to the root of the tree, F is a queue and set to nil initially. */

Input: A set S of n points, $\{p_1, p_2, \ldots, p_n\}$, in $\Re^2$, stored in a priority search tree, Priority_Search_Tree(S) pointed to by Priority_Search_Tree_root(S) and a 2D grounded range $[x_\ell, x_r] \times [-\infty, y_r], x_\ell, x_r, y_r \in \Re, x_\ell \leq x_r$.

Output: A subset $F \subseteq S$ of points that lie in the 2D grounded range, *i.e.*, $F = \{p \in S | x_\ell \leq p.x \leq x_r$ and $p.y \leq y_r\}$.

Method:

1. Start from the root v finding the first split-node v_{split} such that $v_{split}.x$ lies in the x-range $[x_\ell, x_r]$.
2. For each node u on the path from node v to node v_{split} **if** the point $p_{u.aux}$ lies in range $[x_\ell, x_r] \times [\infty, y_r]$ **then** report it by $(p_{u.aux} \Rightarrow F)$.
3. For each node u on the path of x_ℓ in the left subtree of v_{split} **do** **if** the path goes left at u **then** Priority_Search_Tree_1dRange_Search($u.right, y_r, F$).
4. For each node u on the path of x_r in the right subtree of v_{split} **do** **if** the path goes right at u **then** Priority_Search_Tree_1dRange_Search($u.left, y_r, F$).

procedure Priority_Search_Tree_1dRange_Search(v, y_r, F)
/* Report in F all the points p_i, whose y-coordinate values are no greater than y_r, where $i = v.aux$. */

1. **if** v is nil **return**.
2. **if** $p_{v.aux}.y \leq y_r$ **then** report it by $(p_{v.aux} \Rightarrow F)$.
3. Priority_Search_Tree_1dRange_Search($v.left, y_r, F$)
4. Priority_Search_Tree_1dRange_Search($v.right, y_r, F$)

procedure Priority_Search_Tree_1dRange_Search(v, y_r, F) basically retrieves all the points stored in the priority search tree rooted at v such that their y-coordinates are all less than and equal to y_r. The search terminates at the node u whose associated point has a y-coordinate greater than y_r, implying **all** the nodes in the subtree rooted at u satisfy this property. The amount of time required is proportional to the output size. Thus we conclude that

THEOREM 1.8 *The* **Grounded 2D Range Search Problem** *for a set S of n points in the plane $\Re^2$ can be solved in time $O(\log n)$ plus time for output, with a priority search tree structure for S that requires $O(n \log n)$ time and $O(n)$ space.*

Note that the space requirement for the priority search tree in linear, compared to that of a 2D-range tree, which requires $O(n \log n)$ space. That is, the **Grounded 2D Range Search Problem** for a set S of n points can be solved optimally using priority search tree structure.

Acknowledgement

This work was supported in part by the National Science Council under the Grants NSC-91-2219-E-001-001 and NSC91-2219-E-001-002.

References

[1] J. L. Bentley, "Decomposable searching problems," *Inform. Process. Lett.*, vol. 8, 1979, pp. 244- 251.

[2] J. L. Bentley, "Multidimensional divide-and-conquer," *Commun. ACM*, (23,4), 1980, pp. 214-229.

[3] J. L. Bentley and J. B. Saxe, "Decomposable searching problems I: Static-to-dynamic transformation," *J. Algorithms*, vol. 1, 1980, pp. 301-358.

[4] J.-D. Boissonnat and F. P. Preparata, "Robust plane sweep for intersecting segments," *SIAM J. Comput.* (29,5), 2000, pp. 1401-1421.

[5] M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf, Computational Geometry: Algorithms and Applications, Springer-Verlag, Berlin, 1997.

[6] B. Chazelle and L. J. Guibas, "Fractional cascading: I. A data structuring technique," *Algorithmica*, (1,3), 1986, pp. 133–162

[7] B. Chazelle and L. J. Guibas, "Fractional cascading: II. Applications," *Algorithmica*, (1,3), 1986, pp. 163-191.

[8] H. Edelsbrunner, "A new approach to rectangle intersections, Part I," *Int'l J. Comput. Math.*, vol. 13, 1983, pp. 209-219.

[9] H. Edelsbrunner, "A new approach to rectangle intersections, Part II," *Int'l J. Comput. Math.*, vol. 13, 1983, pp. 221-229.

[10] M. L. Fredman and B. Weide, "On the complexity of computing the measure of $\bigcup[a_i, b_i]$," *Commun. ACM,* vol. 21, 1978, pp. 540-544.

[11] P. Gupta, R. Janardan, M. Smid and B. Dasgupta, "The rectangle enclosure and point-dominance problems revisited," *Int'l J. Comput. Geom. Appl.*, vol. 7, 1997, pp. 437-455.

[12] U. Gupta, D. T. Lee and J. Y-T. Leung, "An optimal solution for the channel-assignment problem," *IEEE Trans. Comput.*, Nov. 1979, pp. 807-810.

[13] E. Horowitz, S. Sahni and S. Anderson-Freed, Fundamentals of Data Structures in C, Computer Science Press, 1993.

[14] H. Imai and Ta. Asano, "Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane," *J. Algorithms*, vol. 4, 1983, pp. 310-323.

[15] D. T. Lee and F.P. Preparata, "An improved algorithm for the rectangle enclosure problem," *J. Algorithms,* 3,3 Sept. 1982, pp. 218-224.

[16] D. T. Lee, "Maximum clique problem of rectangle graphs," in Advances in Computing Research, Vol. 1, ed. F.P. Preparata, JAI Press Inc., 1983, 91-107.

[17] J. van Leeuwen and D. Wood, "The measure problem for rectangular ranges in d-space," *J. Algorithms*, vol. 2, 1981, pp. 282-300.

[18] G. S. Lueker and D. E. Willard, "A data structure for dynamic range queries," *Inform. Process. Lett.*, (15,5), 1982, pp. 209-213.

[19] E. M. McCreight, "Priority search trees," *SIAM J. Comput.*, (14,1), 1985, pp. 257-276.

[20] L. Monier, "Combinatorial solutions of multidimensional divide-and-conquer recurrences," *J. Algorithms*, vol. 1, 1980, pp. 60-74.

[21] M. H. Overmars and J. van Leeuwen, "Two general methods for dynamizzing decomposable searching problems," *Computing*, vol. 26, 1981, pp. 155-166.

[22] F. P. Preparata and M. I. Shamos, Computational Geometry: An Introduction, 3rd edition, Springer-Verlag, 1990.

[23] M. Sarrafzadeh and D. T. Lee, "Restricted track assignment with applications," *Int'l J. Comput. Geom. Appl.*, vol. 4, 1994, pp. 53-68.

[24] D. E. Willard, "New data structures for orthogonal range queries," *SIAM J. Comput.*, vol. 14, 1985, pp. 232-253.

Index