ICS 621: Analysis of Algorithms Fall 2019

Lecture 12: Interval Trees, Segment Trees

Prof. Nodari Sitchinava *Scribe: Mojtaba Abolfazli, Muzamil Yahia*

1 Priority search tree: cont

This section is a continuation of the last lecture on Priority Search Trees (PSTs) where we are interested to dive deeper into the analysis of PSTs. Before going into details, we need to recall PST with P_{ymax} as the point with the largest y-coordinate and x_{med} as the median x-coordinate among points in the input set S. A PST can be built up as shown in Figure 1 where S_{left} and S_{right} are obtained as follows:

$$S_{left} = \{P \in S \setminus \{p_{ymax}\} \mid P.x \leq x_{med}\}$$
$$S_{right} = \{P \in S \setminus \{p_{ymax}\} \mid P.x > x_{med}\}$$

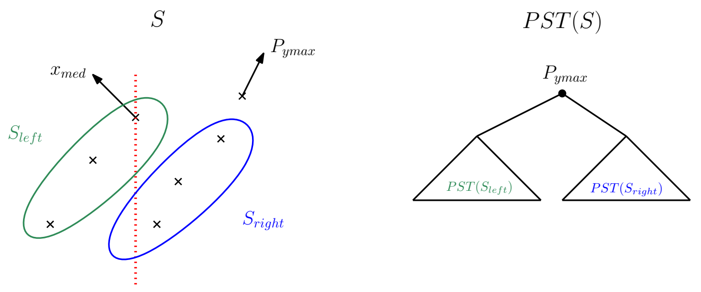


Figure 1: Building priority search tree.

To report all nodes in a subtree rooted at v, we can use the following pseudocode:

Algorithm 1 Report *all* points stored in the subtree rooted at node v of a PST

```
1: function REPORT($v$)
2:     if $v \neq NULL$ then
3:         OUTPUT($v$)
4:         REPORT($v.left$)
5:         REPORT($v.right$)
```

1.1 Range Query

Let's consider a range query in one dimensional space where we're asked to report all points greater than or equal to a real number q_y. In this case, we can use the following pseudocode to report all nodes above q_y in a subtree rooted at v.

Algorithm 2 Report only points that are above q_y among the ones stored in the subtree rooted at node v of a PST

```
1: function REPORT($v, q_y$)
2:     if $v \neq NULL$ then
3:         if $v.y \geq q_y$ then
4:             OUTPUT($v$)
5:             REPORT($v.left, q_y$)
6:             REPORT($v.right, q_y$)
```

The above pseudocode reports all the points above q_y because any path from root v to any node is in a non-decreasing order of y-coordinate (otherwise the property of PST doesn't hold). Likewise, the subtree of nodes needs to be reported is either connected and contains the root or it is empty.

The performance of PSTs is summarized in the following theorem.

Theorem 1. *The total running time to report all points in a query range $[q_y, +\infty]$ in a PST rooted at v is:*

$$T(v) = c \cdot k_v + c - 1$$

where k_v is the number of reported points and $c \geq 1$.

Proof. We prove the theorem by using the induction.
Inductive hypothesis: For all descendant nodes u of v:

$$T(u) = c \cdot k_u + c - 1$$

where k_u is the total number of nodes above q_y in a PST rooted at u.

For the base case where there is no point in the query above q_y, the theorem is true. Now we show that it's also true for the PST rooted at v.

$$\begin{aligned} T(v) &= 1 + T(v.left) + T(v.right) \\ &= 1 + c \cdot k_{v.left} + c - 1 + c \cdot k_{v.right} + c - 1 \\ &= c - 1 + c(1 + k_{v.left} + k_{v.right}) \\ &= c - 1 + c \cdot k_v \end{aligned}$$ □

1.2 3-sided Range Query

Now, let's look at 2D range query where we aim to find nodes within a 3-sided range (i.e. $[x_L, x_R] \times [y, +\infty]$) as shown in Figure 2.

We follow the subsequent steps to report all nodes corresponding to the specified range:

- Find splitting vertex v_{split}.
- Follow left path down to x_L (left boundary) and report all points in the subtrees hanging from the right.

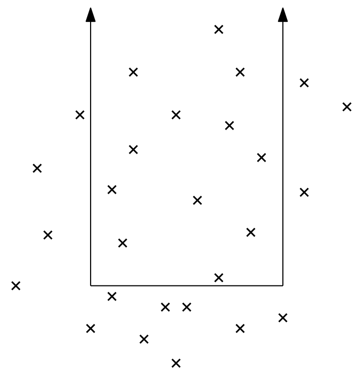


Figure 2: 3-sided Range Query.

- Follow right path down to x_R (right boundary) and report all points in the subtrees hanging from the left.

The pseudocode shown in Algorithm 3 describes the query algorithm.

We start by finding the vertex where the search for x_L and x_R splits into different subtrees, as shown in Figure 3. Below this spplitting vertex, all shaded subtrees are those whose x-coordinate lies in the specified range. Therefore, we can report all points stored there that lie above the y-coordinate using the Algorithm 2.

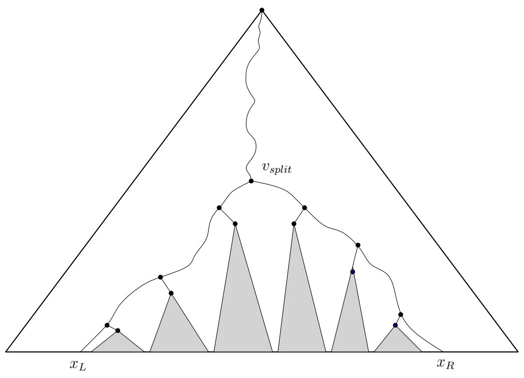


Figure 3: Querying a PST.

Finding the splitting vertex and traversing the two paths takes $O(\log n)$ time, and reporting the k_v points within a subtree rooted at v takes $O(k_v)$ time, as discussed in the previous section. Thus, the running time to answer the 3-sided query is $T(v) = O(\log n) + \sum_{v} k_v = O(\log n + k)$, where k is the number of reported points.

The PST for n points uses $O(n)$ space to store the data because every point is stored in exactly one node.

Finally, the preprocessing time to build the data structure takes $O(n \log n)$ time. The reason is that the y-coordinate PST associated with each node at x-coordinate PST takes $O(n)$ to be constructed with bottom-up approach. Considering the fact that the depth of the x-coordinate PST is of the order $O(\log n)$, the total preprocessing time is bounded by $O(n \log n)$.

Algorithm 3 3-sided Range Query

```
function 3-SIDED QUERY($v, [x_L, x_R], y$)
                                                        ▷ Find the vertex where the search paths split
    while $(v \neq NULL)$and$(v.x_{med} \leq x_L$ or $x_R < v.x_{med})$ do
        if $v \in [x_L, x_R] \times [y, +\infty]$ then     ▷ Check if the point on the path needs to be reported
            OUTPUT($v$)
        if $v.x_{med} \leq x_L$ then
            $v = v.right$
        else
            $v = v.left$
    $v_{split} = v$
    $v = v.left$
    if $v_{split} \in [x_L, x_R] \times [y, +\infty]$ then
        OUTPUT($v_{split}$)
                                                        ▷ Follow the left path
    while $v \neq NULL$ do
        if $v \in [x_L, x_R] \times [y, +\infty]$ then     ▷ Check if the point on the path needs to be reported
            OUTPUT($v$)
        if $x_L \leq v.x_{med}$ then
            REPORT($v.right, y$)                  ▷ Report points in the right-hanging (shaded) subtree
            $v = v.left$
        else
            $v = v.right$
    $v = v_{split}.right$
                                                        ▷ Follow the right path
    while $v \neq NULL$ do
        if $v \in [x_L, x_R] \times [y, +\infty]$ then     ▷ Check if the point on the path needs to be reported
            OUTPUT($v$)
        if $v.x_{med} \leq x_R$ then
            REPORT($v.left, y$)                   ▷ Report points in the left-hanging (shaded) subtree
            $v = v.right$
        else
            $v = v.left$
```

2 Interval trees for orthogonal stabbing queries

In certain geometric applications like planar graphs, we might be interested in reporting the set of line segments L associated with a window query $w = [x_0, x_1] \times [y_0, y_1]$. (See Figure 4 for an example.) This set L contains all line segments with at least one end point inside w, and those segments that span the window w without having an end point inside w.

We can report all line segments with at least one endpoint inside w using algorithms and techniques from Section 1. We can use 2 instances of 3-sided query i.e. $q_0 = [x_0, x_1] \times [y_0, +\infty]$ and $q_1 = [x_0, x_1] \times [-\infty, y_1]$, and then report the intersections of the two queries in time $O(\log n + k_0) + O(\log n + k_1)$. An algorithm to find intersections $q_0 \cap q_1$, would be to keep sorted lists of q_0 and q_1

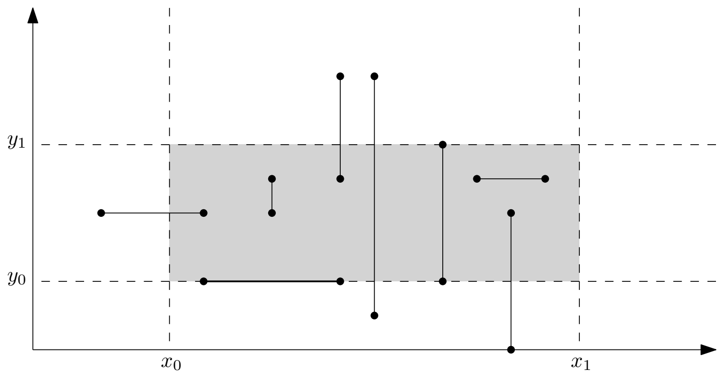


Figure 4: Query $w = [x_0, x_1] \times [y_0, y_1]$.

and then find the intersection in linear time $O(k_0+k_1)$ using modified merge algorithm in mergesort. However, in the worst case scenario, the two queries may report all $n = k_0 + k_1$ endpoints while the intersection may contain only one element. By constructing 2D range trees with storage $O(n \log n)$, and time $O(\log^2 n + k)$ we can report all line segments with ends points in w. Applying fractional cascading can reduce the query time further to $O(\log n + k)$ which is faster than $O(\log n + n)$.

What is left is how to handle efficiently those line segments that span our window query w without any endpoint inside w? In other words, we are looking for line segments that cross two boundaries of our window query w. If we know how to report all line segments that cross one boundary, say $[x_0, x_1]$, of w, we can easily check if they cross any of the remaining boundaries. We present first a solution to a slightly simpler problem that can help us towards reporting those line segments crossing w but with no end point inside w.

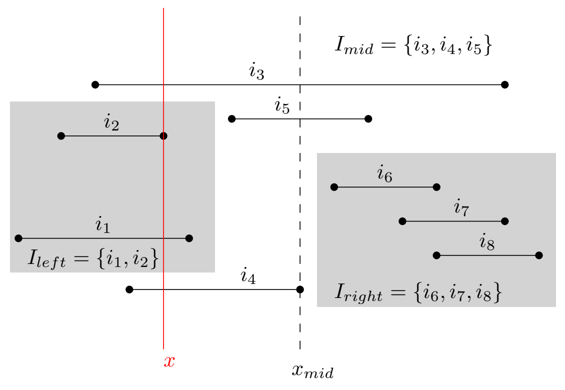
Stabbing query: Given a set of intervals $I = \{i_1, \dots, i_n\}$, and a vertical line query $\ell(q = x)$, find all intervals that are crossed by ℓ.

Claim 2. *We can solve stabbing the query in $O(n \log n)$ and $O(n)$ space.*

We apply divide and conquer strategy to solve stabbing query problem. We construct recursively a balanced BST over all end points of intervals in I, i.e. $E_I = \bigcup_{[i_l, i_r]=i \in I} \{i_l.x, i_r.x\}$. While doing so, we augment this BST with extra data containing intervals from I that contains $v.val$ in their boundaries as shown in Algorithm 4.

Algorithm 4

```
1: function BuildInt($I$)
2:     $v.val = median(E_I)$                                   ▷ key stored at node $v$
3:     $I_{mid} \leftarrow \{i \in I \mid i_l.x \leq v.val \leq i_r.x\}$
4:     $I_{left} \leftarrow \{i \in I \mid i_r.x < v.val\}$
5:     $I_{mid} \leftarrow \{i \in I \mid v.val < i_l.x\}$
6:     $v.data \leftarrow I_{mid}$                                  ▷ augmented data
7:     $v.left \leftarrow$ BuildInt($I_{left}$)
8:     $v.right \leftarrow$ BuildInt($I_{right}$)
9:     return $v$.
```



$I_{mid} = \{i_3, i_4, i_5\}$
i_3
i_5
i_2
i_6
i_7
i_1
i_8
$I_{left} = \{i_1, i_2\}$
i_4
$I_{right} = \{i_6, i_7, i_8\}$
x
x_{mid}

Figure 5: $I = I_{left} \cup I_{mid} \cup I_{right}$, and $\textsc{Stab}(I, x) = \{i_3, i_4, i_1, i_2\}$.

Figure 5 shows 8 line segments (intervals) with median being the right end point of interval i_4. The root of the interval tree divides the set of intervals $I = \{i_1, \ldots, i_8\}$ into 3 disjoint sets $I_{mid}, I_{left}, I_{right}$, where the root contains $I_{mid} = \{i_3, i_4, i_5\}$. The left subtree will recursively store $I_{left} = \{i_1, i_2\}$ and the right subtree will recursively store $I_{right} = \{i_6, i_7, i_8\}$.

How do we store I_{mid}? In order to retrieve all intervals that are stabbed by $q = x$, we can scan the list of intervals stored at node v for intervals that contain x before going to the left or right subtree depending on whether x is less than or greater than $v.val$. However, it may happen that all n intervals are stored in the root, and checking for intervals containing x may take $O(n)$ then instead of $O(k)$, the output size!

In order to prevent that, we store I_{mid} in a more efficient way by storing I_{mid} sorted by the x-coordinate of start point in an increasing order, and another copy of I'_{mid} of I_{mid} ordered by the x-coordinate of end point in a decreasing order. To check if $q = x$ is contained in some intervals in v, we compare x with $v.val$ and accordingly scan the list I_{mid} or I'_{mid} till we find an interval that does not contain x. In Figure 5, at the root of the tree, $I_{mid} = \{i_3 \leq i_4 \leq i_5\}$ and $I'_{mid} = \{i_3 \geq i_5 \geq i_4\}$. To report the intervals that stored at this node and that contain the query point, if the query point x is to the left (respectively, right) of $v.val$, we scan the intervals stored in I_{mid} (respectively, I'_{mid}) until the first interval that does not contain x. Note, that the ordering we chose for intervals in I_{mid} and I'_{mid} ensures that we encounter all the intervals that contain the query point before we hit the first one that doesn't. The pseudocode for answering the query on interval tree is presented in Algorithm 5.

Algorithm 5

```
1: function QueryInt($v, x$)
2:     if $v \neq NULL$ then
3:         if $t \leq v.val$ then
4:             ReportInt($I_{mid}, left, x, \leq$)          ▷ $O(k_v + 1)$
5:             QueryInt($v.left, t$)
6:         else
7:             ReportInt($I'_{mid}, right, x, \geq$)        ▷ $O(k_v + 1)$
8:             QueryInt($v.right, t$)
```

Algorithm 6

```
1: function ReportInt($I, direction, x, comparator$)
2:     for $j = 1$ to $|I|$ do
3:         if $compartor(I[j].direction.x, x) = true$ then
4:             output $I[j]$
5:         else
6:             break
```

Since, we store only 2 copies of each interval i in the entire tree, interval trees takes $O(n)$ space. To report all intervals containing x at node v, we need $k_v + 1$ comparisons. The recurrence relation is then given by

$$\begin{aligned} T(v) &= O(1 + k_v) + \max\{T(v.left), T(v.right)\} \\ &= O(\log n + k) && \text{(because the tree is balanced).} \end{aligned}$$

The preprocessing time takes sorting complexity, as we need to build a balanced BST, sort all intervals by start point, and again by end points. We can write the recurrence relation of BuiltInt as follows:

$$\begin{aligned} T(n, h) &= O(n) + T(|I_{left}|, h - 1) + T(|I_{right}|, h - 1) \\ &= O(nh) \\ &= O(n \log n). \end{aligned}$$

Answering the window query $w = [x_0, \times x_1] \times [y_0 \times y_1]$

In order to answer our original window query w, we can modify the data structure of augmented data I_{mid} stored in interval tree to be 1D range tree over y-axis instead of an ordered list over x-axis. Range tree takes $O(|I_{mid}|)$ to store data, $O(\log |I_{mid}| + k_v)$ for querying its data, and $O(|I_{mid}| \log |I_{mid}|)$ for preprocessing. Since we need to query a range tree at each node v visited while querying the interval tree, it takes $\sum_{v \in Path(\text{QueryInt}(v,x))} O(\log n + k_v) = O(\log^2 n + k)$ to answering the window query.

3 Segment trees

Segments tree is another data structure that can be used to answer stabbing queries. It is more general than interval trees as it can process any type of non-intersecting non-vertical line segments (i.e., not only the horizontal ones), but at the price of increased storage $O(n \log n)$. As we did with interval trees we first show how segment trees answer stabbing queries and then highlight how to answer window queries.

Consider the set of line segments $L = \{\ell_1, \ell_2, \ell_3, \ell_4, \ell_5\}$ in the following diagram:

Observe that we cannot always define a total ordering over slanted line segments that preserves the transitivity property. For example, if we consider $\ell_5 \leq \ell_4 \leq \ell_2$ then by transitivity we would have $\ell_5 \leq \ell_2$. On the other hand, if we consider $\ell_2 \leq \ell'_4 \leq \ell_5$ then we would have $\ell_2 \leq \ell_5$. However, if

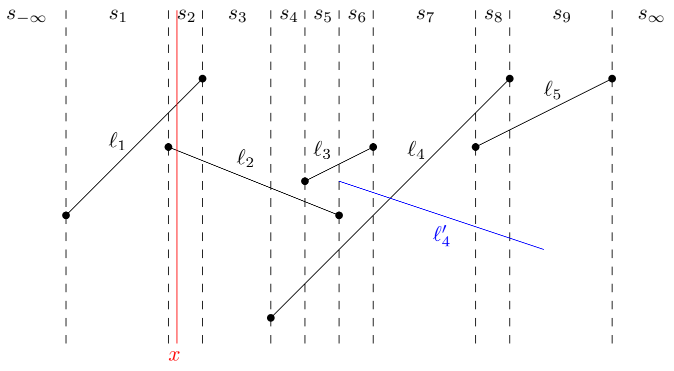


Figure 6: Set of line segments $L = \{\ell_1, \ell_2, \ell_3, \ell_4, \ell_5\}$ with slabs $S = \{s_{-\infty}, s_1, \ldots, s_9, s_\infty\}$.

we divide the plane into slabs defined by all endpoints (shown by the dashed lines in Figure 6), we can define a total ordering within each slab that is transitive.

We can build a balanced BST with leaves corresponding to each of the slabs $S = \{s_{-\infty}, s_1, \ldots, s_9, s_\infty\}$, ordered on the natural left-to-right ordering of the slabs (by the x-coordinate). Each internal node v in the BST will correspond to slab that is the union of all slabs within the subtree rooted at v. The root will then correspond to the slab $[-\infty, \infty]$, that is the whole x-axis. Next we augment the BST node v with a list of line segments ℓ that fully cross any slab within its subtree but do not cross the parent node's slab.

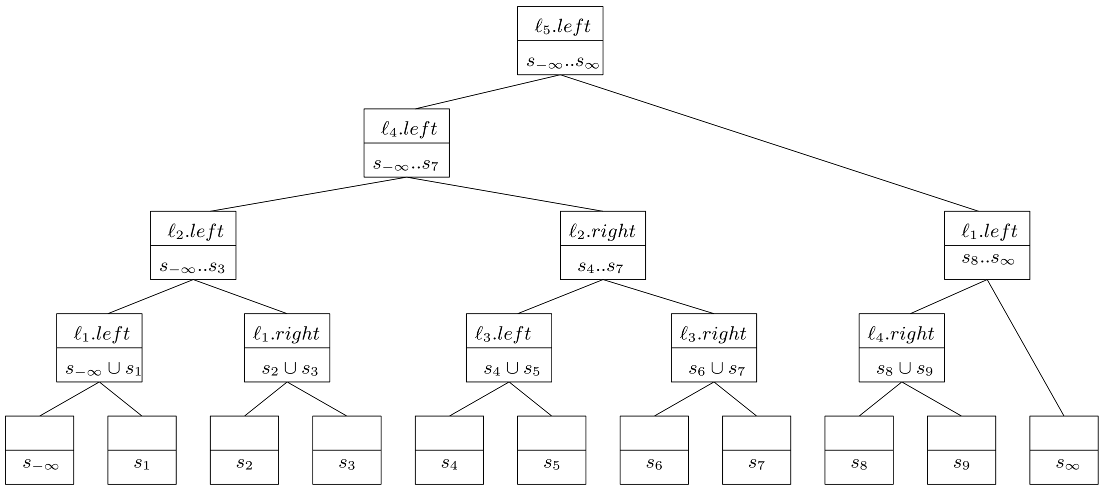


Figure 7: A segment tree for Figure 6 with slabs as leaves, and union of slabs as internal nodes.

Invariant Each line segment ℓ is stored in a node of v such that

$$[v.x_{left}, v.x_{right}] \subseteq [\ell.x_{left}, \ell.x_{right}] \ \wedge\ [parent(v).x_{left}, parent(v).x_{right}] \not\subseteq [\ell.x_{left}, \ell.x_{right}].$$

Claim 3. *Each segment will be stored in at most 2 nodes at each level of segment tree.*

The proof of the claim above can be found in chapter 10 in [BCKO08]. Based on the result above, the segment tree would take $O(n \log n)$ space because we store at most $2n$ nodes at each level of the BST, and because the tree is balanced. In order to answer stabbing queries, we locate the slabs that contains the query $q = x$ from root till leaves level (a single branch) and report all line segments containing q. This takes $O(\log n)$ to locate the leaf node, and k to report the output. Hence, it takes $O(\log n + k)$ to answer stabbing query.

Answering the window query $w = [x_0, \times x_1] \times [y_0 \times y_1]$

In order to answer our original window query w, we use 1D range tree to store line segments sorted by y-axis. Note that within each slab in segment tree, we have total ordering of line segments and therefore we can build up a BST without affecting the result. The same analysis for interval trees applies here and we would have $O(\log^2 n + k)$ query time for segment trees. Observe that, compared to interval trees, segment trees don't improve the query time, yet require more space. The power of segment trees comes from the fact that we can answer queries on non-horizontal segments, not just horizontal line segments (intervals) that interval trees are restricted to.

References

[BCKO08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications.* Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008.