Interval Trees
Storing and Searching Intervals

- Instead of points, suppose you want to keep track of axis-aligned *segments*:

  ![Diagram of axis-aligned segments]

- Range queries: return all segments that have any part of them inside the rectangle.

- *Motivation:* wiring diagrams, genes on genomes
Simpler Problem: 1-d intervals

• Segments *with at least one* endpoint in the rectangle can be found by building a 2d range tree on the 2n endpoints.
  - Keep pointer from each endpoint stored in tree to the segments
  - Mark segments as you output them, so that you don’t output contained segments twice.

• Segments *with no* endpoints in range are the harder part.
  - Consider just horizontal segments
  - They must cross a vertical side of the region
  - Leads to subproblem: Given a vertical line, find segments that it crosses.
  - (y-coords become irrelevant for this subproblem)
Recursively build tree on interval set $S$ as follows:

1. Sort the $2n$ endpoints.
2. Let $x_{\text{mid}}$ be the median point.
3. Store intervals that cross $x_{\text{mid}}$ in node $N$.
4. Intervals that are completely to the left of $x_{\text{mid}}$ in $N_{\text{left}}$.
5. Intervals that are completely to the right of $x_{\text{mid}}$ in $N_{\text{right}}$. 
Another view of interval trees
Interval Trees, continued

- Will be approximately balanced because by choosing the median, we split the set of end points up in half each time
  - Depth is $O(\log n)$

- Have to store $x_{\text{mid}}$ with each node

- Uses $O(n)$ storage
  - each interval stored once, plus
  - fewer than $n$ nodes (each node contains at least one interval)

- Can be built in $O(n \log n)$ time.

- Can be searched in $O(\log n + k)$ time
  $[k = \# \text{ intervals output}]$
Interval Tree Searching

- Query: vertical line (aka $x_q$)
- Suppose we’re at node N:
  - if $x_q < x_{\text{med}}$, then can eliminate right subtree
  - if $x_q \geq x_{\text{med}}$, then can eliminate left subtree
  - Always have to search the intervals stored at current node => leads to another trick (next slide)
Searching intervals at current node

- Store each interval in *two* sorted lists stored at node:
  - List L sorted by increasing left endpoint
  - List R sorted by decreasing right endpoint

- Search list depending on which side of $x_{\text{med}}$ the query is on:
  - If $x_q < x_{\text{med}}$ then search L, output all until you find a left endpoint $> x_q$.
  - If $x_q \geq x_{\text{med}}$ then search R, output all until you find a right endpoint $< x_q$.

- Only works because we know each segment intersects $x_{\text{med}}$. 
Vertical **SEGMENT** searching

- Instead of infinite vertical lines, we have finite segments as a query
- Start with same idea:
  - Interval trees => candidates
  - But somehow have to remove the ones that don’t satisfy the y-constraints

- **Idea**: use 2-d range trees instead of sorted lists to hold segments at each node
Vertical Segment Searching

- Consider the segments stored at a given node and a query segment:

\[-\infty, x_q\] by \([y_1, y_2]\)

- Execute a range query on a semi-infinite range on the 2d-range tree on the end points stored at each node of the interval tree.

  - optimization: keep two range trees \(R_{\text{left}}\) and \(R_{\text{right}}\) that store points to the left and to the right of \(x_{\text{mid}}\).
Vertical Segment Queries: Runtime & Space

• Query time is $O(\log^2 n + k)$:
  - $\log n$ to walk down the interval tree.
  - At each node $v$ have to do an $O(\log n + k_v)$ search on a range tree (assuming your range trees use fractional cascading)

• $O(n \log n)$ space:
  - each interval stored at one node.
  - Total space for set of range trees holding $\leq 2n$ items is $O(n \log n)$.

• Priority search trees reduce the storage to $O(n)$
Priority Search Trees
Handling queries that are unbounded on one side

- Easy in the 1-d case:
  - just walk sorted list from left to right or right to left

- But then how long does an insert take?
  - Can we do better?
1-sided Range Queries in 1-d

Heap on x-values:

Query: $x < 20$

2-d case:
$x < 20$ AND $25 < y < 70$

Any ideas?
Unbounded range queries in 2d

• In 2d-case:
  – Want to find points with low x-values
  – Within a range of y-values

• Idea:
  – Find low values ---> heap
  – 1-d range queries (on y-values) --> BST

• Combine them:
  – Priority Search Trees
1-sided Range Queries in 2-d

Heap on x-values:

Query: $x < 20$

2-d case:

$x < 20$ AND $25 < y < 70$

Heap on the x-values
BST on the y-values

Range searching for y-range can be done as in 1-d range trees

Then each of the subtrees found in that 1-d range search is a heap, so you just output the “top” of the heap.
2-d range queries with one unbounded side, cont.

Range search on $[y_1, y_2]$ based on the y-keys

Points in the gray subtrees all satisfy the y-constraint (fall into the $[y_1, y_2]$ range)

Points along the search paths may or may not

All the points that fall in the x-range are at the tops of the discovered subtrees
PST Searching:

- Query: \([-\infty, x] \text{ by } [y_1, y_2]\)
- Range search on \([y_1, y_2]\)
- Then output “tops” of each subtree between the paths found during the range search.
- Also, must check each node along both paths because they store points.

- Time: \(O(\log n)\) to find trees + \(O(k)\) to output their tops.
  - faster than the \(O(\log^2 n + k)\) time required if you use range trees with fractional cascading
  - Also simpler
Recursive Definition of PST

- Given a set of points $P$, let
  - point $P_{\min x}$ = one with smallest $x$
  - $y_{\text{mid}}$ = median of the y-coordinates of $P \setminus \{P_{\min x}\}$

- Store point $P_{\min x}$ and $y_{\text{mid}}$ in node a N.
  - note that $y_{\text{mid}}$ need not correspond to point $P_{\min x}$.

- Split the points up by y-coordinate:
  - $P_{\text{left}} = \{ p \in P \setminus \{P_{\min x}\} : p.y < y_{\text{mid}} \}$
  - $P_{\text{right}} = \{ p \in P \setminus \{P_{\min x}\} : p.y \geq y_{\text{mid}} \}$

- Recursively built left and right subtrees of N on each of these children sets.

- $\Rightarrow O(n \log n)$ algorithm to build PST
Segment Trees
Arbitrarily Oriented Segments

- No longer assume that segments are parallel to the x- or y-axis.

One trick: store the bounding boxes of each segment as a collection of 4 axis-parallel segments.

- Know how to handle range queries on these kinds of segments
- If a vertical line crosses a segment, it crosses its bounding box (good)
- It may be that a vertical line crosses a bounding box but doesn’t cross the segment (bad)
• Interested in Vertical Segment Stabbing Queries:
• Return all segments that intersect a vertical query segment
• (Assume segments don’t cross)
Why don’t interval trees work?

Interval trees answer vertical segment stabbing queries for axis-parallel datasets, so why don’t they work for slanted segments?

- No longer true that a query like \([-\infty, x]\) by \([y_1, y_2]\) will find the endpoints of satisfying segments:

  Segment intersects range and endpoint falls in half-infinite query (as in interval trees)

  Segment intersects query, but there are no endpoints in the range
Again, we consider 1-d case

Induce a partitioning of the line

Build a balanced BST on that partitioning

Partitioning = open intervals & endpoints

Induce a partitioning of the line

1-d segments
Segment Trees

Forget for a moment the segments we’re trying to store.

This BST we’ve built recursively partitions 1-d space.

Leaves store an elementary region.

For internal node $u$, $\text{Region}(u) =$ union of elementary regions in the subtree rooted at $u$. 
So,

- We’ve divided up space into a set of basic “building-block” units.

- Subdivision of space is customized to our needs:
  - Every segment we want to store is the union of some set of these basic building block units (elementary regions)

- How do we store the actual set of intervals?
Where to store segments

**Rule:** store segment $s$ at any node $u$ for which

- segment covers the entire Region($u$), but
- doesn’t cover the entire Region(parent($u$))

(in other words, we propagate segments up until we reach a node whose Region is not a subset of the segment)
Space usage:

- Segments may be stored at several nodes, but...

- Each segment is stored at most twice at each level
  - if it were stored 3 times, there would be a parent should contain it
  - contradicts that intervals are not stored at both a child and its parent

- $O(\log n)$ height because tree is balanced.

- Therefore: $O(n \log n)$ total space.
Searching with vertical line queries

• Find segments that intersect a given x.
  - Binary Search traversal of tree
  - At each step: Output every segment stored at the current node \( u \) (x must intersect them all because they all span Region(\( u \)))
  - Note that Region(\( u \)) = Region(leftchild(\( u \))) \cup \text{Region(rightchild}(u)))
  - If x falls into Region(leftchild(\( u \))), take the left branch
  - If x falls into Region(rightchild(u)), take the right branch

• \( O(\log n + k) \) time: follow a path of \( O(\log n) \) nodes down to a leaf. Output all k segments encountered along the way.
Segment Tree Construction

- Build the tree:
  - Sort segments
  - Break into elementary building blocks
  - Building balanced BST on these building blocks

- For every segment to insert:

```python
def InsertSegment(u, x_1, x_2):
    # if the interval spans the region represented by u
    # store it in the linked list “segs”
    if Region(u) subset of [x_1, x_2]:
        u.segs.append(x_1, x_2)
    else:
        # otherwise, walk down both subtrees
        if [x_1, x_2] intersects Region(u.left):
            InsertSegment(u.left, x_1, x_2)
        if [x_1, x_2] intersects Region(u.right):
            InsertSegment(u.right, x_1, x_2)
```
Why is construction $O(n \log n)$?

If we visit node $u$ while inserting, one of 3 things happen:

- interval spans $\text{Region}(u)$  \[\leq 2 \text{ nodes / level}\]
- $\text{Region}(u)$ contains $x_1$  \[\leq 1 \text{ node / level}\]
- $\text{Region}(u)$ contains $x_2$  \[\leq 1 \text{ node / level}\]

Therefore, $\leq 4$ nodes visited per level $\Rightarrow O(\log n)$ nodes visited on each segment insert.
Segment Trees vs. Interval Trees

- **Storage:**
  - Interval trees: \( O(n) \)
  - Segment trees: \( O(n \log n) \)

- **Construction:**
  - Interval trees: \( O(n \log n) \)
  - Segment trees: \( O(n \log n) \)

- **Vertical line queries:**
  - Interval trees: \( O(\log n + k) \)
  - Segment trees: \( O(\log n + k) \)

So why are segment trees interesting?

- Partition the space in a application specific manner
- *All* intervals encountered will be output
  So: instead of using aux data structure to find subset of intervals to output, we can use it for other things.)
2-d case

Segments stored at \( u \) all span \( \text{Region}(u) \) by definition.

Because we assume segments don’t overlap, they can be linearly ordered from top to bottom

So, store segments in BST (aka 1-d range tree) sorted by this ordering.

Do a range search for those segments that are below \( y_2 \) and above \( y_1 \).