Announcements 4/11

- Midterm 2
  - Thu, Apr 13
    - Closed book / closed notes
    - 2 cheat sheets / front + back
    - Coverage through this week - kd-trees, splay, scapegoat, skip-lists

- Coverage:
  - Old stuff - Briefly
  - Quad + kd-trees
    - insert
    - delete \(\Rightarrow\) + replacements
  - Queries -
    - range counting
    - nearest neighbor
    - upper-right (HW3)
  - Scapegoat Trees
  - Splay Trees
  - Skip lists
Solution 1:

(a) To perform \textbf{insert}((80,20)), we trace the search path to this point, falling out of the tree on the right child of (70,30). We create a new node at this point. Since the parent splits vertically (cutting dimension is \textit{x}), the new node splits horizontally (cutting dimension is \textit{y}).
(b) To perform delete((40,10)), we find the point at the root node. We search for the replacement point in the right subtree, which returns (50,70) (see the figure below). We copy its contents to the root and recursively delete (50,70) from the right child. We then search for the replacement point in the right subtree, which is clearly (60,90). We copy its contents to the (50,70) node, and recursively delete (60,90).
delete(p)

find_min

delete(q)

delete(q)

\[ p \]

\[ q \]

\[ \emptyset \]
Solution 2:

1. To perform \texttt{insert}(7), we first search for 7 in the tree, falling out at node 6. We first perform a zig-zag rotation, followed by a zig-zig, followed by a zig to bring 6 to the root (see Fig. 1).

\begin{figure}
\centering
\includegraphics[width=\textwidth]{splay_tree_insertion.png}
\caption{Splay tree insertion.}
\end{figure}
(b) To perform delete(15), we first search for 15 in the tree. We first perform a zig-zig rotation, followed by a zig-zag, followed by a zig to bring 15 to the root (see Fig. 2). Next, we perform splay(15) on the root’s right subtree, but since we fall out of the tree at node 17 itself, so there is no change. Finally, we replace the root with 17, moving 15’s left child to be 17’s left child.

Figure 2: Splay tree deletion.
**Solution 3:** Let $T(m)$ denote the running time for `findMin(Node p, int i)`, where $m$ is the number of points in $p$’s subtree. At levels where the cutting dimension equals $i$, we recurse on one side and at the other levels we recurse on both children. This means that for every two levels of descent, we visit at most two out of the four possible grandchildren. Since the tree is balanced, each of these recursive calls involves at most $m/4$ points. The running time at each node is $O(1)$. Thus, up to constant factors, the running time is given by the recurrence $T(m) = 2T(m/4) + 1$. By the Master Theorem (where $a = 2$ and $b = 4$), the running time is $O(m^{\log_b a}) = O(m^{\log_4 2}) = O(m^{1/2}) = O(\sqrt{m})$. Therefore, the asymptotic running time of `findMin` is $O(\sqrt{m})$, where $m$ is the number of points in the subtree.
Solution 4: Our approach to answering upper-right queries is similar to that used in nearest-neighbor searching. We will design a recursive helper function. The function is given the query point \( q = (q.x, q.y) \), the current node \( p \) being visited, the cell associated with this node, and the current best point seen so far in the search.

The helper first checks the various trivial cases. This includes any of the following cases (see Fig. 3(a)):

- We have fallen out of the tree \( (p == \text{null}) \)
- The current cell is fully to the left or fully below the query point \( (\text{cell.hi.x} < q.x \ || \ \text{cell.hi.y} < q.y) \)
- The current cell lies entirely above best \( (\text{cell.lo.y} >= \text{best.y}) \)

If any of these conditions hold, the current node cannot offer a better solution, so we just return best.

Figure 3: Answering upper-right queries.
The initial call at the root level is \( \text{upperRight}(q, \text{root}, \text{rootCell}, (+\infty, +\infty)) \), where rootCell is the bounding box for the entire point set.

```java
Point upperRight(Point q, Node p, Rectangle cell, Point best) {
    if (p == null) return best // handle trivial cases
    if (cell.hi.x < q.x || cell.hi.y < q.y) return best // cell is disjoint?
    if (cell.lo.y >= best.y) return best // cell is too high?
    if (p.point.x >= q.x && p.point.y >= q.y && p.point.y < best.y) // p’s point better?
        best = p.point

    Rectangle leftCell = cell.leftPart(p.cutDim, p.point) // child cells
    Rectangle rightCell = cell.rightPart(p.cutDim, p.point)
    best = upperRight(q, p.left, leftCell, best) // try both sides
    best = upperRight(q, p.right, rightCell, best)
    return best
}
```
Solution 5: There are a couple of ways to solve the range searching problem for an interval \([lo, hi]\). The simpler approach is to design a function that computes the total number of keys that are strictly smaller than a key \(x\). Call this \texttt{smallerThan}(x).

Given this, we can determine the number of keys in any half-open interval \([lo, hi]\) by computing the difference the number of elements strictly smaller than \(hi\) and the number strictly smaller than \(lo\), that is, \texttt{smallerThan}(hi) - \texttt{smallerThan}(lo). Unfortunately, this will not count the element \(hi\) itself if it is in the skip list. We fix this by explicitly searching for \(hi\) and adding it to the count if present. Thus, the function \texttt{rangeCount(Key lo, Key hi)} can be implemented as

\[
\texttt{smallerThan}(hi) - \texttt{smallerThan}(lo) + (\texttt{find}(hi) = 1 : 0)
\]
The function `smallerThan(\text{Key } x)` is essentially the same as `find(x)`, except we avoid visiting `x` itself. We maintain a count, `ct`, of the number of entries we skip over with each jump. We initialize `ct` to zero. Whenever the search traverses a next link, we add the span of this link to `ct` (see Fig. 4 (top)). Note that this will not count the very last node visited, but this is what we want since we only count elements strictly smaller than `x`. Because the search process is essentially the same as for any skip list, the running time is $O(\log n)$ in expectation.

```c
int smallerThan(\text{Key } x) {  // count keys < x
    int i = topmostLevel  // start at topmost nonempty level
    SkipNode p = head  // start at head node
    int ct = 0  // number of smaller elements
    while (i >= 0) {  // while levels remain
        if (p.next[i].key < x) {
            ct += p.span[i]  // count number of skipped items
            p = p.next[i]  // advance along same level
        }
        else i--  // drop down a level
    }
    return ct  // return final count
}
```
Figure 4: Smaller-than queries in a skip list (top) and range counting (bottom).
int rangeCount(Key lo, Key hi) { // count keys in [lo, hi]
    SkipNode p = head
    int i = maxLevel // find last node < lo
    while (i >= 0) {
        if (p.next[i].key < lo) p = p.next[i]
        else i--
    }
    int ct = 0 // initialize count
    i = p.next.length - 1 // top level of p
    while (p.next[i].key <= hi) { // up phase
        ct += p.span[i] // count nodes skipped
        p = p.next[i]
        i = p.next.length - 1 // go to top level
    }
    while (i >= 0) { // down phase
        if (p.next[i].key <= hi) { // count nodes skipped
            ct += p.span[i]
            p = p.next[i]
        } else i-- // drop down a level
    }
    return ct // final count
Solution to the Challenge Problem:

Let us first construct a point set $P$ of size $2n$ consisting of all the lower-right and upper-right corners of the rectangles. We claim that we can determine the answer to any horizontal ray-shooting query emanating from a point $q$ from information stored within $q$'s lowest upper-right point.

Figure 5: Answering ray-shooting queries.

1. Using the upper-right data structure, find the lowest point $p$ that lies to the upper right of $q$.
2. If $p$ is an upper endpoint, we report that the ray hits some segment (see Fig. 5(c), top).
3. If $p$ is a lower endpoint, let $\ell_x$ and $r_x$ be the $x$-coordinates of the left and right rays shot from $p$.
   
   3a) If $r_x \neq +\infty$, we report that the ray hits some segment (see Fig. 5(c), center).
   
   3b) If $\ell_x \geq q_x$, we report that the ray hits some segment (see Fig. 5(c), bottom).
4. Otherwise, we report that the ray does not hit any segment.
To prove the correctness of this algorithm, we will show that the horizontal ray emanating from $q$ hits some segment if and only if one of the conditions of the lemma is satisfied. Suppose first that the horizontal ray emanating from $q$ hits some segment. Translate the ray upwards until it first hits a segment endpoint. If the first endpoint it hits is an upper endpoint, then we satisfy condition (2). If it hits a lower endpoint, there are two cases. If the $x$-coordinate of the segment endpoint is to the left of the ray intersection point, then we satisfy condition (3a). Otherwise, if the $x$-coordinate of the segment endpoint is to the right of the ray intersection point, then we satisfy condition (3b). Therefore, we will detect the ray intersection. Conversely, suppose that any of the three conditions holds. In all three cases, there is a segment lying to the upper-right of $q$ that clearly hit (as seen in Fig. 4) provided that the endpoint of this segment lies below $q_y$. But if the lower endpoint were above $q_y$, then the upper-right query would have returned this other endpoint instead. This implies that the algorithm is correct.

Because the query algorithm involves a single application of the upper-right query, it has the same running time.