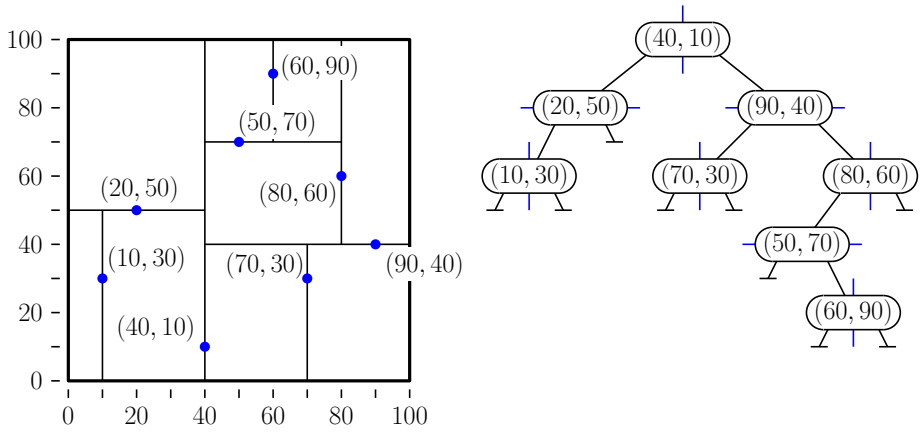
CMSC 420: Spring 2023

Homework 3: kd-Trees, Splay Trees, and More

Handed out Tue, Apr 4. Due **Tue, Apr 11, 9:30am** (that is, by the start of class).

Note: Solutions will be discussed in class, so **no late submissions will be accepted**.

Problem 1. (10 points) Consider the kd-tree shown in the figure below. Assume a "standard" kd-tree where the cutting dimensions alternates between x and y with each level. (Intermediate results are not required, but may be given for partial credit.)

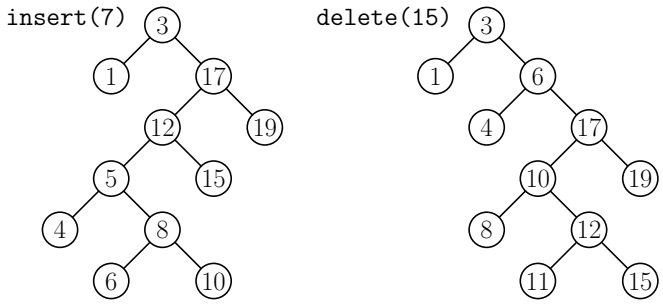


(a) (5 points) Show the final tree after the operation `insert((80,20))`. You need only show the tree, not the spatial subdivision.

(b) (5 points) Starting with the *original tree*, show the final tree after `delete((40,10))`. Indicate which nodes were used as replacement nodes.

Problem 2. (10 points) Consider the splay trees shown in the figure below. In both cases, apply the exact algorithms described in the lecture notes (Lecture 13).

(a) (5 points) Show the steps involved in operation `insert(7)` for the tree on the left.



(b) (5 points) Show the steps involved in operation `delete(15)` for the tree on the right.

In both cases, in addition to the final tree, show the result after each splay. Indicate which node was splayed on and the tree that resulted after splaying. (Intermediate results may be shown for partial credit.)

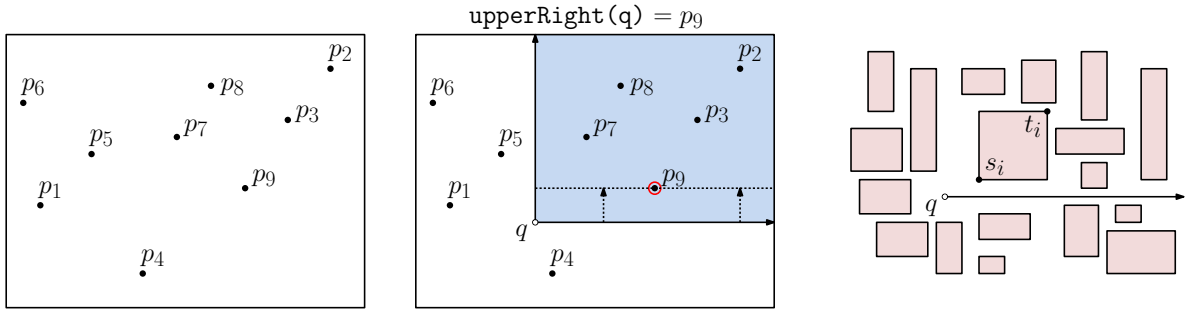
Problem 3. (10 points) Suppose you are given a kd-tree storing a set of points in $\mathbb{R}^2$. Recall from Lecture 10 that the `delete` operation on standard kd-trees makes use of the helper function, `findMin(Node p, int i)`, which returns the point in node `p`'s subtree that has the smallest ith coordinate ($i = 0$ for x and $i = 1$ for y). Let us assume that your kd-tree satisfies the "standard assumptions", namely that the cutting dimensions alternate between x and y with each level of the tree, and for each internal node `p`, there are an equal number of points in its left and right subtrees.

Under these assumptions, prove that for any node `p`, the running time for `findMin(Node p, int i)` is $O(\sqrt{m})$, where m denotes the number of points in `p`'s subtree.

Hints: Your analysis is asymptotic, meaning that you should focus on the case when m is very large. You may use fact that, given any positive constants a, b, and c, the recurrence $T(n) = aT(n/b) + c$ solves to $O(n^{\log_b a})$, for all sufficiently large n.

Problem 4. (10 points) You are given a set $P = \{p_1, \ldots, p_n\}$ of n points in $\mathbb{R}^2$ stored in a "standard" point kd-tree, as given in Lecture 10.

You are given a query point $q = (q_x, q_y)$, and the objective of an *upper-right query* is to compute the the bottommost point (that is, the point having the minimum y-coordinate) among all the points of P whose x-coordinates are greater than or equal to q_x and whose y-coordinates are greater than or equal to q_y (see the figure below). If there is no such point, the query returns `null`. To avoid messy edge cases, you may assume that there are no duplicate x- or y-coordinates among the points (including q).



(a) (b) (c)

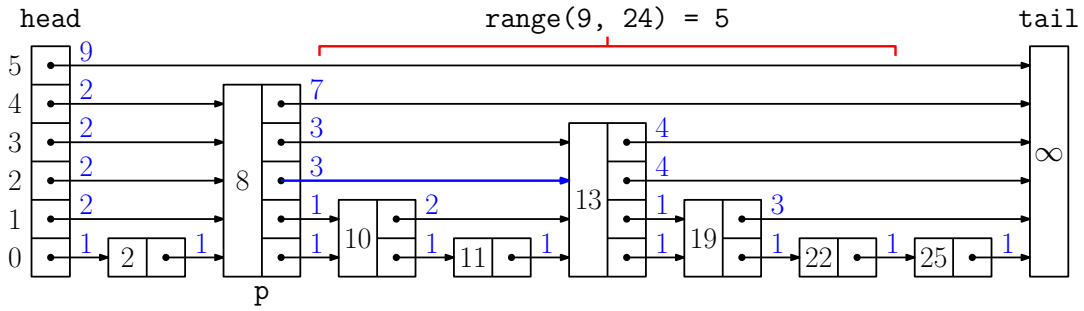
Present pseudo-code for an algorithm, `Point upperRight(Point q)` for answering upper-right queries given a kd-tree storing the points of P.

For full credit, your query algorithm should run in $O(\sqrt{n})$ time, but you do not need to prove this.

Hint: As usual, create a recursive helper function and explain how it is initially invoked. This is similar to nearest-neighbor searching in the sense that you will keep track of each node's `cell` and the `best` point seen so far. For efficiency, you should visit subtrees in an efficient order and avoid visiting nodes that cannot possibly contribute to the final answer.

Problem 5. (10 points) In this problem we consider an enhanced version of a skip list. As usual, each node `p` stores a key, `p.key`, and an array of next pointers, `p.next[]`. To this we add an array `p.span[]` which is parallel to `p.next[]`. This array is defined as follows. If `p.next[i]` refers to a node `q`, then `p.span[i]` contains the number of nodes from `p` to `q` (at level 0) of the skip list.

For example, in the figure below, the node `p` (with key "8") the link `p.next[2]` (shown in blue) jumps 3 nodes forward to the node with key 13, and so `p.span[2] = 3`.



Assuming this enhanced structure, present pseudo-code for a function `int range(Key lo, Key hi)`, which returns a count of the number of nodes in the entire skip list whose key values are greater than or equal to `lo` and less than or equal to `hi`. For example, in the figure above, the operation `range(9,24)` would return 5, since there are five items in this interval (namely, 10, 11, 13, 19, and 22).

Assuming that the skip list contains a total of n keys, your procedure should run in time expected-case time $O(\log n)$ (over all random choices), irrespective of the number of elements that lie within the range. Briefly explain how your function works.

Hint: It may help to approach this by first solving the simpler problem to answering semi-infinite range counting queries, namely, counting all the points whose key is $\leq x$.

Challenge Problem: You are given a set $R = \{r_1, \ldots, r_n\}$ of n disjoint rectangles in the $\mathbb{R}^2$. Each rectangle r_i is represented by a pair of points s_i is the lower left corner and t_i is the upper right corner. You may store this information in any data structure you choose with the purpose of answering the following horizontal ray-shooting queries efficiently.

You are given a query point $q = (q_x, q_y)$ which is guaranteed to lie outside of all of these rectangles. You shoot an infinite ray horizontally to the right of q. Your objective is to determine whether this ray hits any of the rectangles. To avoid messy edge cases, you may assume that there are no duplicate x- or y-coordinates among the points (including q).

Describe your data structure (what is stored? how is it organized?), and explain how to answer these horizontal ray-shooting queries from it. For full credit, your data structure should use $O(n)$ storage, and queries should be answered in $O(\sqrt{n})$ time. You may express your answer either in plain English or using pseudo-code (your choice), as long as it is clear and unambiguous how your algorithm can be implemented. Justify the correctness of your solution.

Hint: It is possible to reduce this problem to the kd-tree data structure from Problem 4, but you will need to augment the data structure with additional information.