Homework 3: Hashing, Geometry, Tries

Deadline: Part 1 is due **11am, Thu, Dec 10**. (**Note:** "AM" not "PM"! This is a **hard deadline**, since solutions will be discussed in class.) Part 2 is due 11pm Mon, Dec 14. (Yes, I realize that this is the same as the programming assignment, but we're running out of time.) Submit your homework through Gradescope.

Note: Because this has been handed out rather late, we have changed the grading policy for homeworks this semester. When averaging the homework scores, we will drop the lowest of the three scores.

Part 1: (Due 11am, Thu, Dec 10)

Problem 1. In this problem, you will show the result of inserting a sequence of three keys into a hash table, using various open-addressing conflict-resolution methods. In each case, at a minimum you should indicate the following:

- Was the insertion successful? (The insertion fails if the probe sequence loops infinitely without finding an empty slot.)
- Show contents of the hash table after inserting all three keys.
- For each case, give a count of the number of *probes*, that is, the number of entries in the hash table that were accessed in order to find an empty slot in which to perform the insertion. (The initial access counts as a probe, so this number is at least 1. For example, in Fig. 3 in the Lecture 14 LaTeX lecture notes, `insert(z)` makes 1 probe and `insert(t)` makes 4 probes.)

For assigning partial credit, you can illustrate the actual probes that were performed, as we did in Fig. 4 from Lecture 14. But be sure to also list the probe count.

(1.1) Show the results of inserting the keys "`X`", "`Y`", and "`Z`" into the hash table shown in Fig. 1(a), assuming that conflicts are resolved using *linear probing*.

(a) Linear probing

```
insert("X")  h("X") = 9
insert("Y")  h("Y") = 1
insert("Z")  h("Z") = 11
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
E	H	A	▨	M	▨	J	L	▨	P	C	K	▨	G	I	W

(b) Quadratic probing

```
insert("M")  h("M") = 4
insert("D")  h("D") = 8
insert("Q")  h("Q") = 3
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
▨	▨	▨	F	▨	▨	▨	L	N	P	▨	▨	▨	▨	▨	▨

Figure 1: Hashing with open addressing.

(1.2) Show the results of inserting the keys "`M`", "`D`", and "`Q`" into the hash table shown in Fig. 1(b) using *quadratic probing*. (Hint: If you are unsure whether quadratic probing has gone into an infinite loop, you may find it helpful to delve into the topic of quadratic residues.)

(c) Double hashing

```
insert("R")  h("R") = 7; g("R") = 3
insert("T")  h("T") = 0; g("T") = 4
insert("D")  h("D") = 8; g("D") = 7
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
E	▨	A	F	M	B	J	L	N	P	▨	K	U	G	I	W

Figure 2: Hashing with open addressing.

(1.3) Show the results of inserting the keys "R", "T", and "D" into the hash table shown in Fig. 2(c) using *double hashing*. (The second hash function g is shown in the figure.)

(Intermediate results are not required, but may be given to help assigning partial credit.)

Problem 2. You are given a 2-dimensional point kd-tree, as described in Lectures 15 and 16. The operation `float findUp(float x0)` is given a scalar x_0, and it returns the smallest x-coordinate in the tree that is greater than or equal to x_0. If every point has x-coordinate strictly smaller than x_0, the function returns the special value `Float.MAX_VALUE`.

For example, given the points in Fig. 3, the result of `findUp(x1)` would be b's x-coordinate, `findUp(x2))` would be k's x-coordinate, and `findUp(x3)` would be `Float.MAX_VALUE`.

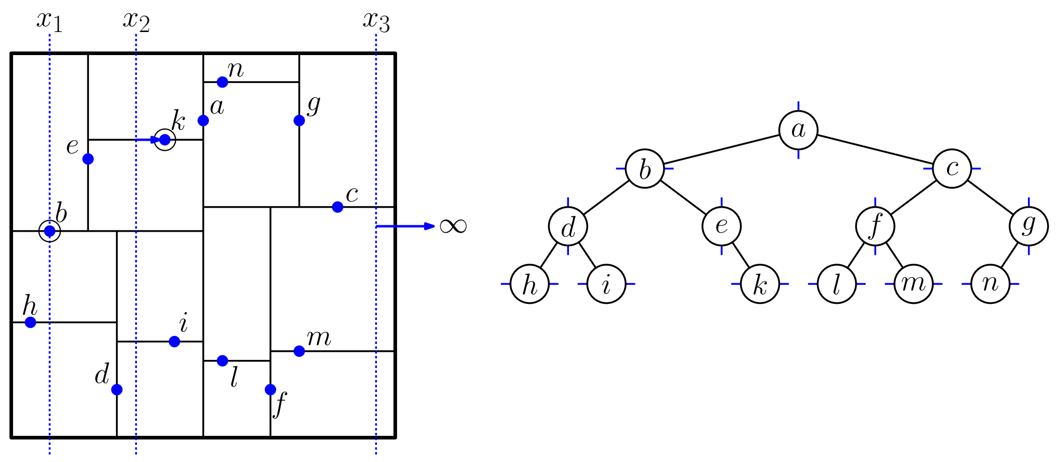

Figure 3: FindUp operation in a point kd-tree.

(2.1) Present pseudocode for this function. (Hint: Use a recursive helper function `float findUp(float x0, KDNode p)`.) Assuming that the tree is balanced and has n points, your function should run in time $O(\sqrt{n})$. (See part 4.2 below.)

(2.2) Derive the running time of your procedure, under the assumption that there are n points and the tree is balanced. (Hint: It may help to recall the analysis of the orthogonal range-search algorithm from Lecture 16. As we did in class, you may make the idealized assumption that the left and right subtrees of each node have equal numbers of points.)

Part 2: (Due 11pm, Mon, Dec 14)

Problem 3. Consider the point kd-tree shown in Fig. 4. As in class, assume that the cutting dimensions alternate between x and y. Suppose that we perform a range query involving the

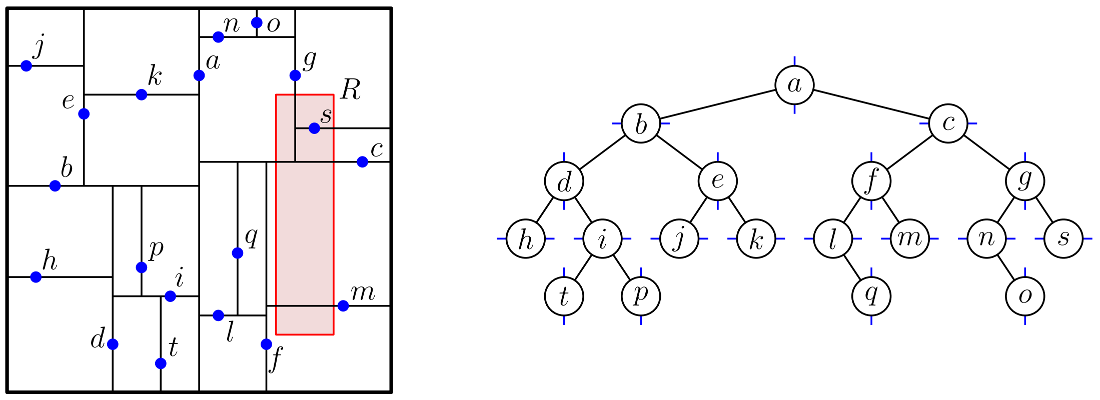

Figure 4: kd-tree operations.

red rectangle R shown in the figure. Indicate (e.g., by drawing the tree and circling them) which nodes of the tree are visited by the orthogonal-range search algorithm given in class. (A node p is "visited" if a call `rangeCount(R, p, cell)` is made.)

Problem 4. In this problem, we will consider how to use/modify range trees to answer two related queries. Given a set P of n points in d-dimensional space, recall that a range tree storing these points uses a total of $O(n \log^{d-1} n)$ storage. Given an axis-aligned rectangle R in d-dimensional space, in $O(\log^d n)$ time it is possible to identify a set of $O(\log^d n)$ subtrees in the range tree, such that the points lying within these subtrees form a partition of $P \cap R$. (This is the only fact about range trees that you need to solve this problem.)

The input set P is in the x, y-plane in all three cases. However, the tree that you construct might be for a different set of points, and it may even be in a different dimension. In each case, describe the points that are stored in the range tree and how the search process works. Justify your algorithm's correctness and derive its running time.

(4.1) In a *3-sided min query*, you are given two x-coordinates, x_0 and x_1, where $x_0 < x_1$, and one y-coordinate, y_0. Among all the points of P whose x-coordinates lie within the interval $[x_0, x_1]$ and whose y-coordinate is greater than or equal to y_0, the answer is the point of P with the smallest y-coordinate. If there are no points that satisfy these conditions, the query returns the value `null`. If there are multiple points that satisfy the x-conditions and have the same y-coordinates, any of them may be returned. (In Fig. 5(a), the query returns point s.)

Your data structure should use $O(n \log n)$ storage and answer queries in $O(\log^2 n)$ time.

(4.2) A *skewed rectangle* is defined by two points $q^- = (x^-, y^-)$ and $q^+ = (x^+, y^+)$. The range shape is a parallelogram that has two vertical sides and two sides with a slope of $+1$. The lower left corner is q^- and the upper right corner is q^+. The answer to the query is the number of points of P that lie within the parallelogram. (In Fig. 5(b), the answer to the given query is 3.)

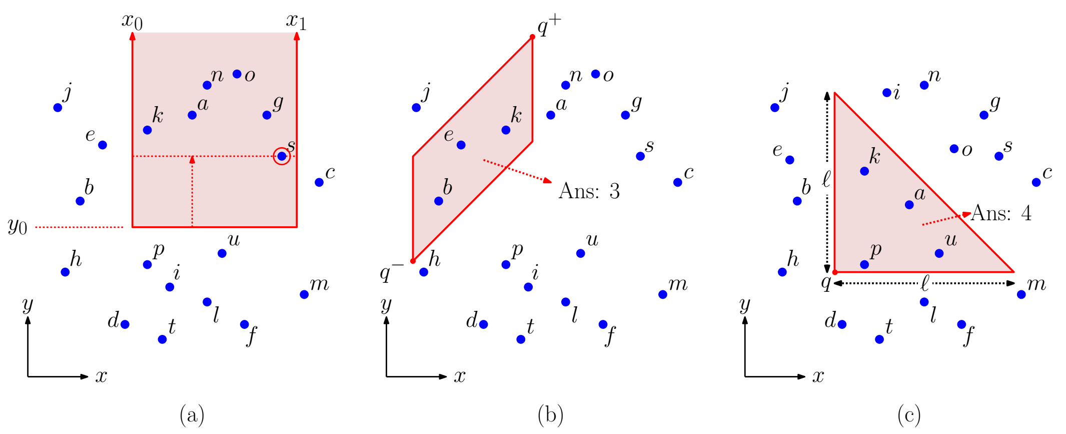

Figure 5: Using range trees to answer various queries. (Part (c) is for the Challenge Problem.)

Your data structure should use $O(n \log n)$ storage and answer queries in $O(\log^2 n)$ time. (Hint: Apply a transformation to space so that any skewed rectangle is mapped to an axis-aligned rectangle. Transform the points of P accordingly and build a range tree with the transformed points.)

Problem 5. In this problem we will build a suffix tree for the string $S =$ `babaabaabbabaab$`.

(5.1) List the 16 substring identifiers for the 16 suffixes of S. For the sake of uniformity, list them from back to front, so the first substring identifier will be "`$`", and the last will be the substring identifier for the entire string.

(5.2) Draw a picture of the suffix tree. For the sake of uniformity, when drawing your tree, use the conventions of Fig. 7 in the Lecture 19 LaTeX lecture notes. In particular, label edges of the final tree with substrings, index the suffixes from 0 to 15, and order subtrees in ascending lexicographical order (`a` $<$ `b` $<$ `$`).

Challenge Problem: Consider the same set-up as in Problem 4. Apply this to the following query.

A *NE right-triangle query* is defined by a point $q = (q_x, q_y) \in \mathbb{R}^2$ and a scalar $\ell > 0$. The NE right triangle has horizontal and vertical sides of length ℓ that share a vertex at q, and the triangle lies in the northeast quadrant relative to q. The answer to the query is the number of points of P that lie within this triangle. (In Fig. 5(c), the answer to the given query is 4.)

Your data structure should use $O(n \log^2 n)$ storage and answer queries in $O(\log^3 n)$ time.

(Hint: Map the problem into a 3-dimensional orthogonal range query. The z-coordinate is a function of the x and y coordinates, and it is used to enforce the constraint that points lie to the lower left of the diagonal edge of the triangle.)