CMSC 420: Fall 2022

CMSC 420 (0201) - Midterm Exam 2

Problem 1. (18 points) Hashing:

(a) (9 points) Show the results of inserting the sequence "`X`" then "`Y`" then "`Z`" into the hash table shown in Fig. 1(a), assuming *quadratic probing*. The operations are performed *as a sequence* (that is, prior insertions affect later ones). Indicate the number of *probes*, that is, array accesses. (The final insertion counts as a probe.) If the operation fails, give the number of probes as "∞".

(b) (9 points) Repeat (a) with the hash table shown in Fig. 1(b) assuming *double hashing*, where `g()` is the jump size.

(a) Quadratic Probing

```
insert("X")  h("X") = 3
insert("Y")  h("Y") = 9
insert("Z")  h("Z") = 4
```

0	1	2	3	4	5	6	7	8	9
C			A	E	D				B

(b) Double Hashing

```
insert("X")  h("X") = 4; g("X") = 6
insert("Y")  h("Y") = 5; g("Y") = 4
insert("Z")  h("Z") = 5; g("Z") = 2
```

0	1	2	3	4	5	6	7	8	9
F	C		A	E	D				B

Figure 1: Hashing.

Problem 2. (32 points) Short answer questions. No explanations required.

(a) (8 points) What are the min and max number of nodes in an AVL tree of height 2?

(b) (4 points) Which data structure did we see this semester that used the operation of *key rotation* (also called *adoption*) to maintain its structure?

(c) (4 points) You have a skip list with n nodes. Suppose that rather than using a fair coin to decide a node's height, you instead use a coin that comes up heads with probability 1/3 and tails with probability 2/3. All nodes start at level 0, and a node survives to the next higher level if the coin toss comes up heads. As a function of n, what is the expected number of nodes that survive to level 2?

(d) (8 points) Suppose you store 20 points in $\mathbb{R}^2$ in an extended kd-tree with a bucket size of two (as in Programming Assignment 2). What are the minimum and maximum number of internal nodes this tree might have?

(e) (3 points) Among the open-addressing hashing methods we have covered (linear probing, quadratic probing, double hashing), which does the best job of *avoiding* clustering?

(f) (5 points) In a hash table, what is the definition of the *load factor*? (Let m denote the table size, and let n be the current number of entries.)

Problem 3. (10 points) You are given a standard (unbalanced) binary search tree, where each node `p` has a key (`p.key`) and left and right child pointers (`p.left` and `p.right`).

The operation `Key findUp(Key x)` returns the smallest key in the tree whose value is greater than or equal to `x`. If `x` appears in the tree, it returns `x`, and if all the keys in the tree are strictly smaller than `x`, it returns `null` (see Fig. 2).

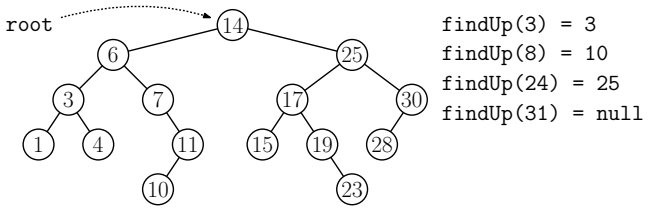

Figure 2: Find-up queries.

Present pseudocode for this function. Briefly explain how your function works. For full credit, your function should run in time $O(h)$, where h is the height of the tree. As always, you may create whatever helper functions you like, but you cannot alter the tree structure (e.g., you may not assume there are parent links or threads).

Problem 4. (15 points) You are given a set $P = \{p_1, \ldots, p_n\}$ of n points in $\mathbb{R}^2$ stored in a kd-tree. Assume that every node `p` of the tree stores a point (`p.point`), its cutting dimension (`p.cutDim`), its cutting value (`p.cutVal`), and its size (`p.size`), which is defined to be the number of points in the subtree rooted at `p`. Assume that all the points are contained in a bounding box `bbox` (see Fig. 3(a)).

In a *circular range counting query* (CRC), you are given a center point $c = (c_x, c_y)$ and a radius r, and the problem is count the number of points of P that lie within the circular disk of radius r centered at c (see Fig. 3(b)).

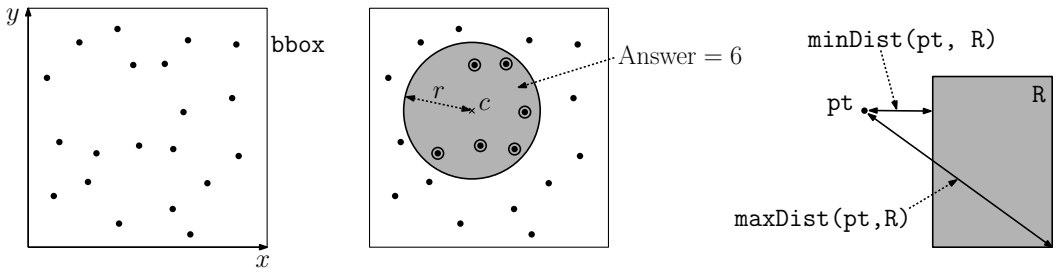

Figure 3: Circular range counting (CRC) queries.

(a) (10 points) Give pseudocode for an efficient algorithm, `int crc(Point c, double r)` for answering this CRC queries in the kd-tree.

Hint: Create a recursive helper function and explain how it is initially called. You may assume you have access to any geometric primitives you like (for example):

- `double dist(Point pt, Point q)`: Distance between `pt` and `q`
- `double minDist(Point pt, Rectangle R)`: Minimum distance between `pt` and `R`
- `double maxDist(Point pt, Rectangle R)`: Maximum distance between `pt` and `R`

(b) (3 points) No doubt, your algorithm is as efficient as possible for a kd-tree. But, what is the *worst-case* query time of your algorithm?

You may make the "standard assumptions" that the cutting dimensions alternate and the tree is well-balanced. Select the best option below. (No justification needed.)

(1) $O(\log n)$
(2) $O(\sqrt{n})$, irrespective of the number of points in the disk
(3) $O(\sqrt{n} + k)$, where k is the number of points in the circular disk
(4) $O(n)$
(5) $O(n \log n)$
(6) None of the above. (Indicate what you think it should be)

(c) (2 points) Suppose that your query algorithm reports that there are zero points of P in the circular disk. What is the *worst case* query time subject to this condition? (Same choices as in (b).)

Problem 5. (10 points) You are given a set $P = \{p_1, \ldots, p_n\}$ of n points in $\mathbb{R}^2$ (see Fig. 4(a)). In a *segment sliding query*, you are given a vertical line segment s, and the query returns the first point $p_i \in P$ that is hit if we were to slide the segment to its right (see Fig. 4(b)). If a point of P lies on the segment, then the answer is this point. If there is no point of P hit by the segment, the query returns `null`. You may assume there are no duplicate x-coordinates. Given a `Segment` object `s`, let `s.x` denote its x-coordinate, and let `s.yhi` and `s.ylo` denote its upper and lower y-coordinates, respectively.

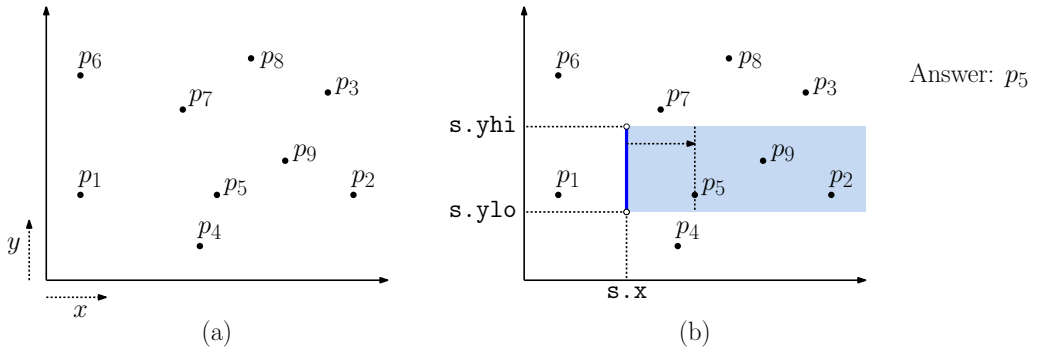

Figure 4: Vertical segment-sliding.

Present an efficient data structure and algorithm for answering these queries. Our objective is a query time of $O(\log^a n)$ using space $O(n \log^b n)$, where $a, b > 0$ are small constants. Partial credit will be given if your answer is correct, but not as efficient as it might be.

(a) (5 points) Describe your data structure and derive its space bound. (**Hint:** Use a variant of range trees. You will *not* get any credit for a kd-tree solution. Explain what layers

you use, how each layer is sorted, and what auxiliary information (if any) you store in each node.)

(b) (5 points) Explain how queries are answered and derive the query time. (**Note:** Pseudocode is not required. A high-level English description is fine.)

Problem 6. (15 points) Throughout this problem, we start with a large, perfectly balanced binary search tree (see Fig. 5(a)). The only operations we perform are `delete` and `find`.

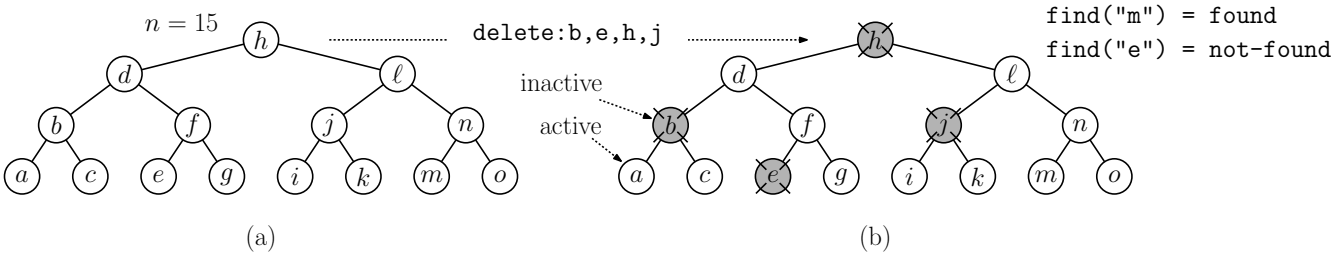

Figure 5: (a) A balanced binary search tree and (b) lazy deletion.

An alternative to the standard `delete` operation is called *lazy deletion*. We do not actually remove any nodes. Instead, to delete a node, we mark this node as *inactive*. When we perform a `find` operation, if the node found is *active*, we return `found`. But, if it is not found or *inactive*, we return `not-found` (see Fig. 5(b)).

As we perform deletions, the tree becomes burdened with many inactive nodes—not good. Whenever the number of inactive nodes exceeds the number active nodes, we *rebuild* the tree as follows. We first traverse the tree keeping only active nodes. We then build a perfectly balanced binary search tree containing only the active nodes (Fig. 6). Let n denote the total number of nodes in the tree (both active and inactive).

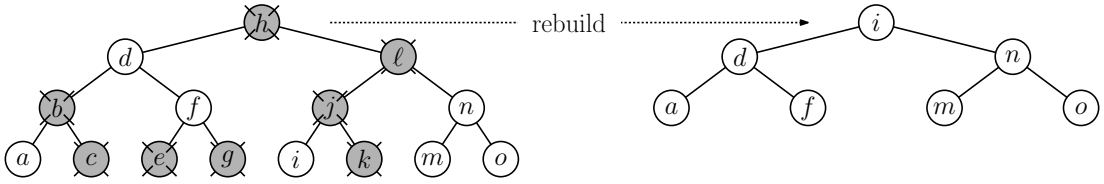

Figure 6: Rebuilding (with 8 inactive nodes and 7 active nodes).

We will analyze the time for `find` and `delete`. If rebuild does not occur, both `delete` and `find` take actual time $O(\log n)$ where n is the total number of nodes (both active and inactive). The actual time for rebuilding is $O(n)$.

(a) (5 points) Show that the *worst-case time* for any `find` operation is $O(\log n_a)$, where n_a is the current number of *active nodes* in the tree. (Note that $n_a \leq n$, where n is the total number of nodes.)

(b) (10 points) Derive the *amortized time* of lazy deletion. Express your answer asymptotically as a function of n (e.g., $O(1)$, $O(\log n)$ or $O(n)$.)

Hint: Analyze just a single run from an initial tree of size n until the next rebuilding. Since this is asymptotic, think of n has being very large and constant factors do not matter.