CMSC 420: Spring 2021

Midterm Exam 2

The exam is asynchronous and online. It is open-book, open-notes, open-Internet, but it must be done on your own without the aid of other people or software. The total point value is 100 points. Good luck!

Problem 1. (20 points) In this problem we will consider a sequence of operations performed on a hash table. In each case the operations are performed one after the other in sequence. In each case, indicate how many probes are made to the table. (A *probe* is defined to be any access to the table, whether to read or write an entry.) Intermediate results are not required, but may be given to help assigning partial credit.

(a) Double hashing

```
insert("X")  h("X") = 7; g("X") = 2
insert("Y")  h("Y") = 13; g("Y") = 6
insert("Z")  h("Z") = 4; g("Z") = 5
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				M			L		P	K			B	I

(b) Linear probing/Deletion

```
insert("R")  h("R") = 2
delete("B")  h("B") = 13
insert("D")  h("D") = 1
```

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

Figure 1: Hasing with open addressing

(1.1) (10 points) Show the results of performing the insertion operations given in Fig. 1(a) assuming *double hashing*. (The second hash function g is shown in the figure.)

(1.2) (10 points) Show the result of performing the sequence of insert and delete operations shown in Fig. 1(b) assuming *linear probing*. The shaded entries are "empty".

Problem 2. (25 points) Short answer questions. Unless requested, explanations are not required, but may always be given to help with partial credit.

(2.1) (3 points) In the skip list data structure we assumed that the coin we tossed was even, coming up heads or tails with equal probability. If we were instead to use a biased coin that comes up heads more often than tails (but both still with constant probability), what would the effect be on the *expected storage* needed by the data structure? Select one and explain briefly:

(a) The expected storage requirements would go up (by at least a constant factor).
(b) The expected storage requirements would go down (by at least a constant factor).
(c) The expected storage requirements would remain unchanged.

(2.2) (3 points) Same as (2.1), but what would the effect be on the *expected query time*? Select one and explain briefly:

(a) The expected query time would go up (by at least a constant factor).
(b) The expected query time would go down (by at least a constant factor).
(c) The expected query time would remain unchanged.

(2.3) (3 points) What properties of B-trees make them particularly attractive for use in external (disk) memory?

(2.4) (4 points) Both scapegoat trees and splay trees provide $O(\log n)$ amortized time for standard dictionary operations (insert, delete, and find). Suppose that your application involves many more find operations than insertions or deletions. Which of these two structures would you prefer and why?

(2.5) (4 points) A treap and an standard (unbalanced) binary search tree both have the property that they support dictionary operations in $O(\log n)$ expected time. How is the notion of "expected" different in each case, and which structure would be preferred? Explain briefly.

(2.6) (4 points) Give one example of an operation that can be performed more efficiently (on average) with hashing compared to AVL trees. Explain briefly.

(2.7) (4 points) Give one example of an operation that can be performed more efficiently (on average) on AVL trees compared to hashing. Explain briefly.

Problem 3. (20 points) This problem involves the question of how the depths of nodes change when splay operations are performed. Consider the splay tree shown in Fig. 2(a) with each node's depth indicated in blue. After performing `splay(a)` observe that some node depths have increased, some have decreased, and some of remain unchanged. The greatest increase is node g, whose depth increases by $+2$.

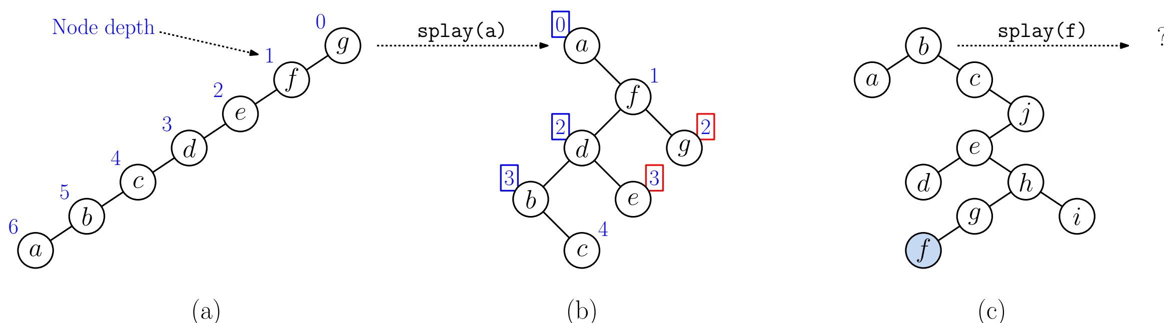

Figure 2: How depths change in a splay tree.

(3.1) (8 points) Consider the tree shown in Fig. 2(c). Show the result of performing the operation `splay(f)` on this tree. (We only need the final tree, but intermediate results can be shown to help with partial credit.)

(3.2) (2 points) Indicate in your drawing which nodes have increased in depth, and in each case by how much the depth has increased.

(3.3) (10 points) Select one of the two options below, and justify your choice.

- Given any splay tree T and any key x in this tree, after performing `splay(x)` if a node's depth increases, *this increase is at most 2.*
- There exists a splay tree T and a key x such that, after performing `splay(x)` there is a node in this tree such that *its depth increases by 3 or more.*

If you believe that first option is true, provide a short sketch of a proof. If you believe that the second option is true, provide an example of a tree T and x where this happens.

Problem 4. (20 points) Throughout this problem, assume that you are given a standard (not wrapped) kd-tree storing a set P of n points in $\mathbb{R}^2$ (see Fig. 3(a)). Assume that the cutting dimension alternates between x and y. You may also assume that the tree stores a root cell `rootCell`, which is a 2-dimensional rectangle containing all the points of P. You may also assume that that any geometric computations on primitive objects (distances, disjointness, containment, etc.) can be computed in constant time, without explanation.

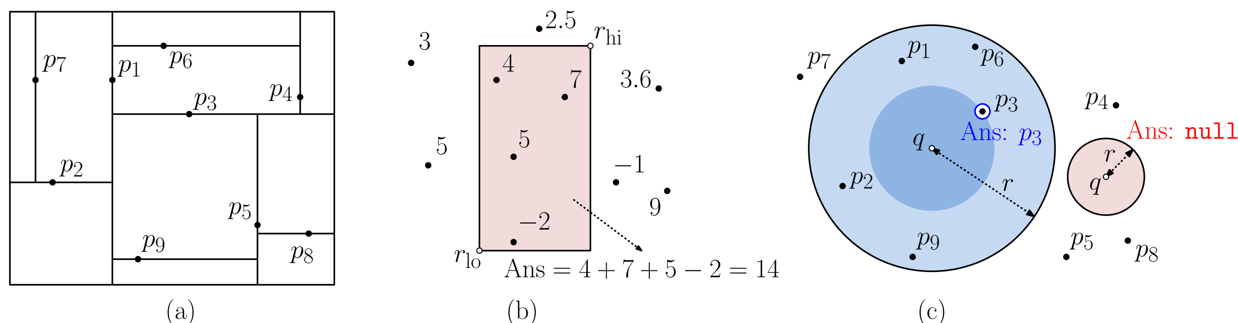

Figure 3: Queries on kd-trees.

(4.1) (7 points) In a standard range-counting query, we want to count the number of points in the range. Suppose that each point $p_i \in P$ has an associated real-valued weight w_i. In a *weighted orthogonal range query*, we are given a query rectangle R, given by its lower-left corner r_{lo} and upper-right corner r_{hi}, and the answer is the sum of the weights of the points that lie within R (see Fig. 3(b)). If there are no points in the range, the answer is 0.

Explain how to modify the kd-tree (by adding additional fields to the nodes, if you like) so that weighted orthogonal range queries can be answered efficiently. Based on your modified data structure, present an efficient algorithm in pseudo-code for answering these queries and explain. (For full credit, your algorithm should run in $O(\sqrt{n})$ time).

You may handle the edge cases (e.g., points lying on the boundary of R) however you like. **Hint:** You may use whatever helper function(s) you like, but I would suggest using:

```
float weightedRange(Rectangle R, KDNode p, Rectangle cell)
```

where `p` is the current node in the kd-tree, `cell` is the associated cell. The initial call is `weightedRange(R, root, rootCell)`.

(4.2) (3 points) Briefly analyze the running time of your algorithm, assuming that the tree is balanced. (You may apply/modify results proved in class.)

(4.3) (10 points) In a *fixed-radius nearest neighbor query*, we are given a point $q \in \mathbb{R}^d$ and a radius $r > 0$. Let C denote the circular disk centered at q whose radius is r. If no points of P lie within this disk, the answer to the query is `null`. Otherwise, it returns the point of P within the disk that is closest to q. Present (in pseudo-code) an efficient kd-tree algorithm that answers such a query.

You may handle the edge cases (e.g., multiple points at the same distance or points lying on the boundary of C) however you like. **Hint:** You may use whatever helper function(s) you like, but I would suggest using:

```
Point frnn(Point q, float r, KDNode p, Rectangle cell, Point best)
```

where `p` is the current node in the kd-tree, `cell` is the associated cell, and `best` is the best point seen so far. The initial call is `frnn(q, r, root, rootCell, null)`.

To receive full credit, your algorithm should not recurse on the children of any node p that (based on p's cell) cannot contribute a (better) answer to the query.

Briefly explain your algorithm, but you *do not* need to derive its running time.

Problem 5. (15 points) In this problem, we will consider how to use/modify range trees to answer two related queries. While the answer should be based on range trees, you may need to make modifications including possibly transforming the points and even adding additional coordinates. In each case, describe the points that are stored in the range tree and how the search process works. An English explanation (as opposed to pseudo-code) is sufficient. Justify your algorithm's correctness and derive its running time.

(5.1) (5 points) Assume you are given an n-element point set P in $\mathbb{R}^2$ (see Fig. 4(a)). In an *empty square annulus query* a query is given by a query point $q \in \mathbb{R}^2$ and two positive radii $r_1 < r_2$. Let $S_1 = S(q, r_1)$ be the square centered at q whose half side length is r_1 and define S_2 similarly for q and r_2. The square annulus $A(q, r_1, r_2)$ is defined to be the region between these two squares. The query returns `true` if $A(q, r_1, r_2)$ contains no points of P, and `false` otherwise (see Fig. 4(b)).

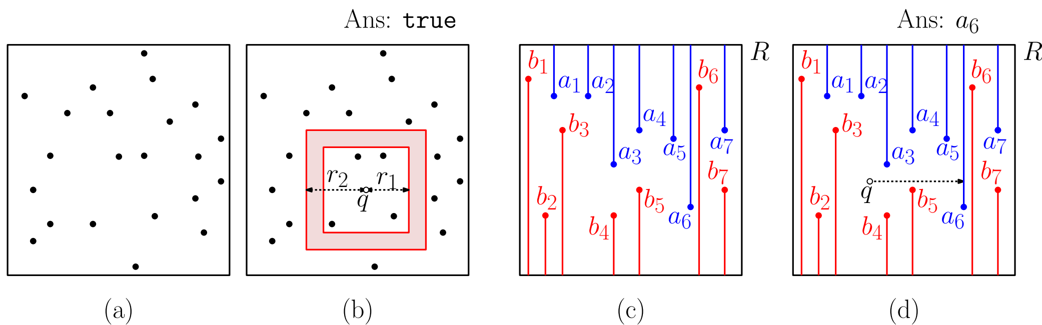

Figure 4: Queries on range trees.

Describe how to preprocess the point set P in a data structure that can efficiently answer any empty annulus query (q, r_1, r_2). Your data structure should use $O(n \log n)$ storage and answer queries in $O(\log^2 n)$ time. (I don't care how you handle edge cases, such as points lying on the boundary of the annulus as being inside or outside.)

(5.2) (10 points) This problem takes place inside a large axis-aligned rectangle R. You are given two sets of points $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_n\}$, all lying within R. At each point a_i there is a vertical line segment from a_i to the top edge of R, and at each point b_j there is a vertical line segment from b_j down to the bottom edge of R (see Fig. 4(d)). A *horizontal ray-shooting query* is presented by a point $q \in R$. The answer

to the query is the first line segment (either from the the A set or B set) that is hit by a horizontal ray shot to the right from q (see Fig. 4(d)).

Describe how to preprocess the point sets A and B into a data structure that can efficiently answer any horizontal ray-shooting query. Your data structure should use $O(n \log n)$ storage and answer queries in $O(\log^2 n)$ time. (I don't care how you handle edge cases, such as if the ray passes through a point from A or B.)