CMSC 420: Spring 2021

Practice Problems for Midterm 2

This exam will be asynchronous and online. The exam will be made available through Gradescope for a 48-hour period starting at **12:00am the morning of Thu, Mar 29** and running through **11:59pm the evening of Fri, Mar 30** (Eastern Time). The exam is designed to be taken over a 1.5-hour time period, but to allow time for scanning and uploading, you will have **2 hours** to submit the exam through Gradescope once you start it. The exam will be open-book, open-notes, open-Internet, but it must be done on your own without the aid of other people or software. (You may use a simple arithmetic calculator, but I don't expect that you will need one.)

Disclaimer: These practice problems have been extracted from old homework assignments and exams. Material changes from semester to semester. These do **not** necessarily reflect the actual coverage, difficulty, or length of the midterm exam.

Problem 0. Expect at least one problem that involves working through some operations on a data structure that we have covered since the previous exam.

Problem 1. Short answer questions. Except where noted, explanations are not required but may be given for partial credit.

(a) What is the purpose of the *next-leaf* pointer in B+ trees?

(b) Suppose you know that a very small fraction of the keys in a data structure are to be accessed most of the time, but you do not know which these keys are. Among the data structures we have seen this semester, which would be best for this situation? Explain briefly.

(c) You are using hashing with open addressing. Suppose that the table has just one empty slot in it. In which of the following cases are you *guaranteed* to succeed in finding the empty slot? (Select all that apply.)

(1) Linear probing (under any circumstances)
(2) Quadratic probing (under any circumstances)
(3) Quadratic probing, where the table size m is a prime number
(4) Double hashing (under any circumstances)
(5) Double hashing, where the table size m and hash function $h(x)$ are relatively prime
(6) Double hashing, where the table size m and secondary hash function $g(x)$ are relatively prime

(d) What is the maximum number of points that can be stored in a 3-dimensional point quadtree of height h? Express your answer as an exact (not asymptotic) function of h. (Hint: It may be useful to recall the formula for any $c > 1$, $\sum_{i=0}^{m} c^i = (c^{m+1})-1)/(c-1)$.)

(e) We have n uniformly distributed points in the unit square, with no duplicate x- or y-coordinates. Suppose we insert these points into a kd-tree in *random* order (see the left side of Fig. 1). As in class, we assume that the cutting dimension alternates between x and y. As a function of n what is the expected height of the tree? (No explanation needed.)

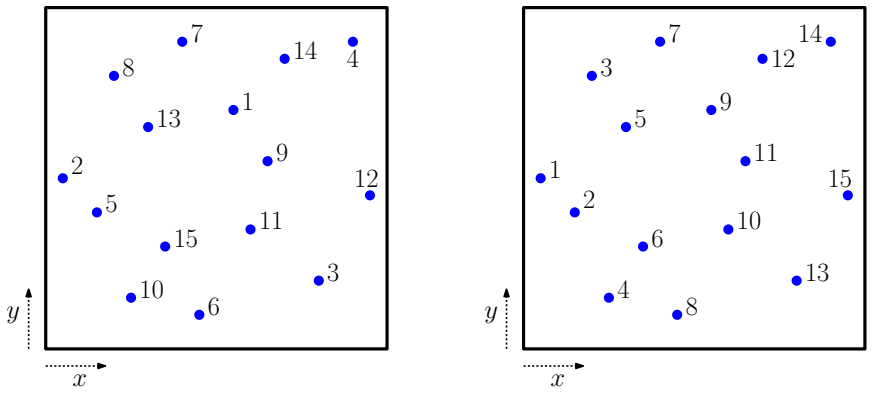

Figure 1: Height of kd-tree.

(f) Same as the previous problem, but suppose that we insert points in *ascending* order of x-coordinates, but the y-coordinates are *random* (see the right side of Fig. 1). What is the expected height of the tree? (No explanation needed.)

Problem 2. Suppose that you are given a treap data structure storing n keys. The node structure is shown in Fig. 2. You may assume that *all keys and all priorities are distinct.*

```
class TreapNode {
    Key key           // key
    int priority      // priority
    TreapNode left    // left child
    TreapNode right   // right child
}
```

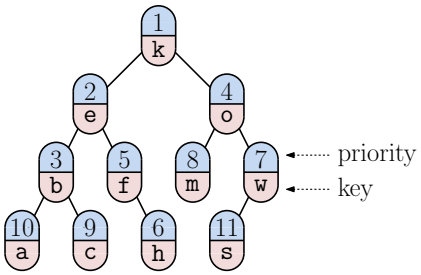

Figure 2: Treap node structure and an example.

(a) Present pseudocode for the operation `int minPriority(Key x0, Key x1)`, which is given two keys x_0 and x_1 (which may or may not be in the treap), and returns the lowest priority among all nodes whose keys x lie in the range $x_0 \le x \le x_1$. If the treap has no keys in this range, the function returns `Integer.MAX_VALUE`. Briefly explain why your function is correct.

For example, in Fig. 2 the query `minPriority("c", "g")` would return 2 from node `"e"`, since it is the lowest priority among all keys x where `"c"` $\le x \le$ `"g"`.

(b) Assuming that the treap stores n keys and has height $O(\log n)$, what is the running time of your algorithm? (Briefly justify your answer.)

Problem 3. Define a new treap operation, `expose(Key x)`. It finds the key `x` in the tree (throwing an exception if not found), sets its priority to $-\infty$ (or more practically `Integer.MIN_VALUE`), and then restores the treap's priority order through rotations. (Observe that the node containing `x` will be rotated to the root of the tree.) Present pseudo-code for this operation.

Problem 4. In scapegoat trees, we showed that if `size(u.child)/size(u)` $\leq \frac{2}{3}$ for every node of a tree, then the tree's height is at most $\log_{3/2} n$. In this problem, we will generalize this condition to:

$$\frac{\texttt{size(u.child)}}{\texttt{size(u)}} \leq \alpha, \qquad (*)$$

for some constant α.

(a) Why does it **not** make sense to set α larger than 1 or smaller than $\frac{1}{2}$?

(b) If every node of an n-node tree satisfies condition $(*)$ above, what can be said about the height of the tree as a function of n and α? Briefly justify your answer.

Problem 5. In class we demonstrated a simple idea for deleting keys from a hash table with open addressing. Namely, whenever a key is deleted, we stored a special value "`deleted`" in this cell of the table. It indicates that this cell contained a deleted key. The cell may be used for future insertions, but unlike "`empty`" cells, when the probe sequence searching for a key encounters such a location, it should continue the search.

Suppose that we are using *linear probing* in our hashing system. Describe an alternative approach, which does not use the "`deleted`" value. Instead it moves the table entries around to fill any holes caused by a deleted items.

In addition to explaining your new method, justify that dictionary operations are still performed correctly. (For example, you have not accidentally moved any key to a cell where it cannot be found!)

Problem 6. Given a set P of n points in the real plane, a *partial-range max query* is given two x-coordinates x_1 and x_2, and the problem is to find the point $p \in P$ that lies in the vertical strip bounded by x_1 and x_2 (that is, $x_1 \leq p.x \leq x_2$) and has the maximum y-coordinate (see Fig. 3).

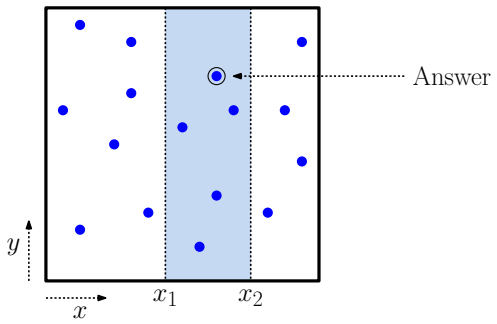

Figure 3: Partial-range max query.

Present pseudo-code for an efficient algorithm to solve partial-range max queries, assuming that the points are stored in a point kd-tree. You may make use of any primitive operations on points and rectangles (but please explain them).

Assuming the tree is balanced and the splitting dimension alternates between x and y, show that your algorithm runs in time $O(\sqrt{n})$.

Problem 7. In class we showed that for a balanced kd-tree with n points in the real plane (that is, in 2-dimensional space), any *axis-parallel line* intersects at most $O(\sqrt{n})$ cells of the tree.

The purpose of this problem is to show that does not apply to lines that are not axis-parallel. Show that for every n, there exists a set of points P in the real plane, a kd-tree of height $O(\log n)$ storing the points of P, and a line ℓ, such that *every* cell of the kd-tree intersects this line.

Problem 8. In this problem we will see how to use kd-trees to answer a common geometric query, called *ray shooting*. You are given a collection of vertical line segments in 2D space, each starts at the x-axis and goes up to a point in the positive quadrant. Let $P = \{p_1, \ldots, p_n\}$ denote the upper endpoints of these segments (see Fig. 4). You may assume that both the x- and y-coordinates of all the points of P are strictly positive real numbers.

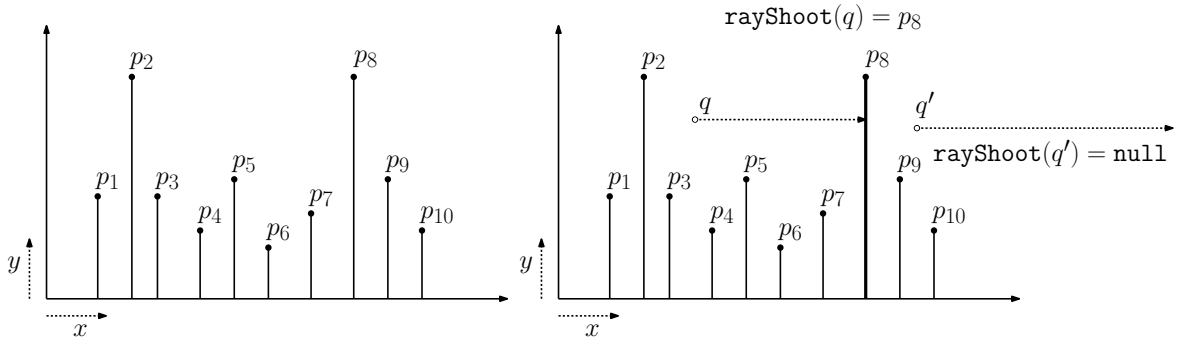

Figure 4: Ray shooting in a kd-tree.

Given a point q, we shoot a horizontal ray emanating from q to the right. This ray travels until it hits one of these segments (or perhaps misses them all). For example, in the figure above, the ray shot from q hits the segment with upper endpoint p_8. The ray shot from q' hits nothing.

In this problem we will show how to answer such queries using a standard point kd-tree for the point set P. A query is given the point $q = (q_x, q_y)$, and it returns the upper endpoint $p_i \in P$ of the segment the ray first hits, or `null` if the ray misses all the segments.

Suppose you are given a kd-tree of height $O(\log n)$ storing the points of P. (It does *not* store the segments, just the points.) Present pseudo-code for an efficient algorithm, `rayShoot(q)`, which returns an answer to the horizontal ray-shooting query (see the figure above, right).

You may assume the kd-tree structure given in class, where each node stores a point `p.point`, a cutting dimension `p.cutDim`, and left and right child pointers `p.left` and `p.right`, respectively. You may make use of any primitive operations on points and rectangles (but please explain them). You may assume that there are no duplicate coordinate values among the points of P or the query point.

Problem 9. (Expect a problem on range trees. I can't find any good practice problems, but the examples from Homework 2 are pretty good.)