CMSC 420: Fall 2022

Practice Problems for Midterm 2

The exam will be held in class on **Thu, Nov 17**. It is close-book, closed-notes, but you will be allowed two sheets of notes, front and back.

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. (Good candidates are kd-trees and hashing, but I may draw something from the material shortly before the midterm, such as treaps, skiplists, or splay trees.)

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

(a) We have studied many classes of binary trees this semester. For this problem, let us ignore the keys and consider just the tree's node structure. Given any binary tree T, define its *reversal* to be the tree that results by flipping the left and right children at every node in the tree. A class of trees is said to be *symmetrical* if it is invariant under reversals. That is, given any valid tree T in the class, its reversal is also a valid member of the class. Which of the following classes of binary trees are symmetrical? (Select all that apply.)

(1) Leftist heaps
(2) AVL trees
(3) Red-black trees
(4) AA trees
(5) Treaps
(6) Splay trees

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

(c) 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 $c > 1$, $\sum_{i=0}^{m} c^i = (c^{m+1})-1)/(c-1)$.)

(d) In high dimensional spaces (say, dimensions greater than 10), kd-trees are preferred over quadtrees. Why is this?

(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. 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? (You may express your answer in asymptotic form.)

(f) Same as the previous problem, but suppose that we insert points in *ascending* order of x-coordinates, but the y-coordinates are *random*.

(g) 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 (that is, they share no common factors)

Problem 2. Suppose that you are given a treap data structure storing n keys. The node structure is shown in Fig. 1. 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
}
```

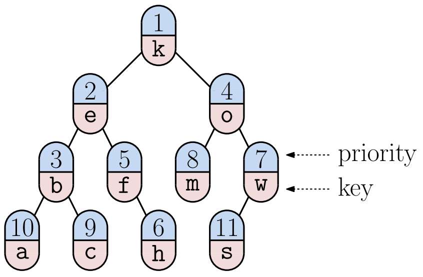

Figure 1: Treap node structure and an example.

(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 $x_0 \leq x \leq x_1$. 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. 1 the query `minPriority("c", "g")` would return 2 from node `"e"`, since it is the lowest priority among all keys x where `"c"` $\leq x \leq$ `"g"`.

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

Problem 3. We usually like our trees to be balanced. Here we will consider *unbalanced* trees. Given a node `p`, recall that `size(p)` is the number of nodes in `p`'s subtree. A binary tree is *left-heavy* if for each node `p`, where `size(p)` ≥ 3, we have `size(p.left)/size(p)` $\geq 2/3$ (see the figure below). Let T be a left-heavy tree that contains n nodes.

(a) Consider any left-heavy tree T with n nodes, and let `s` be the leftmost node in the tree. Prove that `depth(s)` $\geq \log_{3/2} n$. (If you are super careful in your proof, you may discover this is not quite true. The actual bound is $(\log_{3/2} n) - c$, for a constant c. Don't worry about this small correction term.)

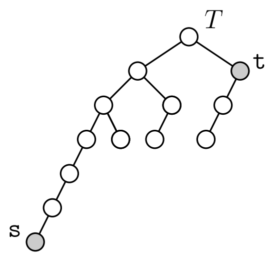

Figure 2: A left-heavy tree.

(b) Consider any left-heavy tree T with n nodes, and let `t` be the rightmost node in the tree. Prove that `depth(t)` $\leq \log_3 n$.

Problem 4. In ordered dictionaries, a *finger search* is one where the search starts at some node of the structure, rather than the root. This is useful when you believe that the next key you are searching for is close to the last one you visited.

Let us suppose we have a skiplist, with the node structure as shown in Fig. 3. Suppose that we have two keys, `x` $<$ `y`, and we have already found the node `p` that contains the key `x`. In order to find `y`, it would be wasteful to start the search at the head of the skip list. Instead, we start at `p`. The operation `forwardSearch(p, y)` searches for key `y` starting at node `p` (whose key is smaller than `y`). If `y` is found it returns the associated value, and otherwise it returns `null`.

```
class SkipNode {
    Key key
    Value value
    SkipNode[] next
}
```

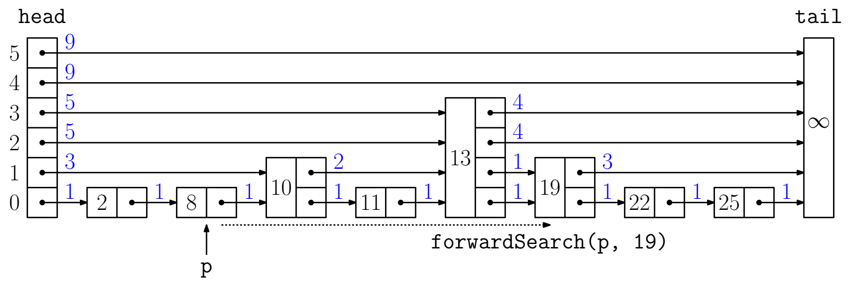

Figure 3: The operation `forwardSearch`.

Of course, we could crawl along level 0, but this would be slow. Suppose that there are m nodes between `x` and `y` in the skip list. We want the expected search time to be $O(\log m)$, not $O(\log n)$.

Present pseudo-code for an algorithm for an efficient function. You do not need to analyze the running time. (**Hint:** The height of any node `p` can be determined as the length of its array of next pointers, that is, `p.next.length`.)

Problem 5. You are given a set P of n points in the real plane stored in a kd-tree, which satisfies the *standard assumptions*. A *partial-range max query* is given two x-coordinates `lo` and `hi`, and the problem is to find the point $p \in P$ that lies in the vertical strip bounded by `lo` and `hi` (that is, `lo` $\leq$ `p.x` $\leq$ `hi`) and has the maximum y-coordinate (see Fig. 4).

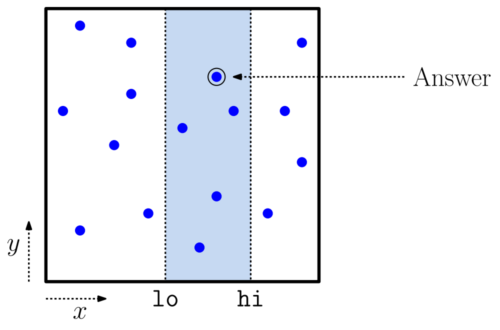

Figure 4: Partial-range max query.

(a) Present pseudo-code for an efficient algorithm to solve partial-range max queries, assuming that the points are stored in a point kd-tree. To simplify notation, let's assume we have an vertical strip object, `Strip`, where `strip.lo` and `strip.hi` are the strip bounds. You may make use of any primitive operations on points and rectangles (but please explain them). **Hint:** A possible signature for your helper function might be:

```
Point partialMax(Strip s, KDNode p, Rectangle cell, Point best)
```

(b) Show that your algorithm runs in time $O(\sqrt{n})$.

Problem 6. 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. 5). You may assume that both the x- and y-coordinates of all the points of P are strictly positive real numbers.

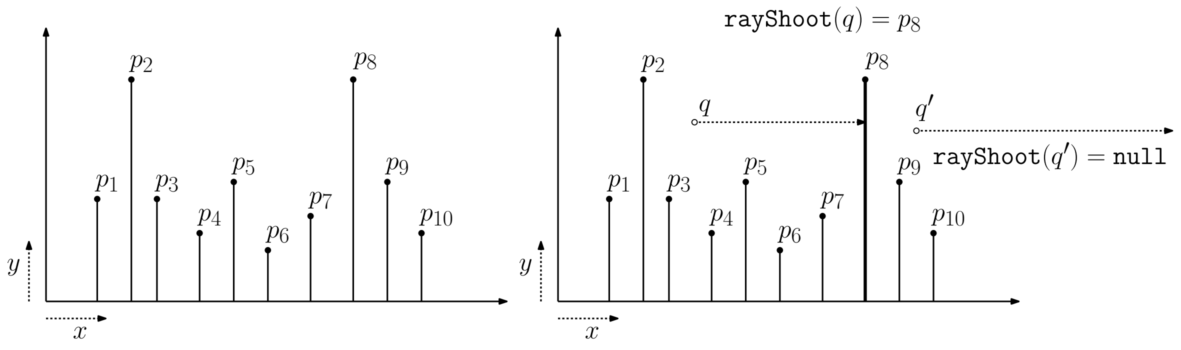

Figure 5: Ray shooting in a kd-tree.

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.

Hint: You might wonder how to store segments in a kd-tree. It turns out that to answer this query you do not need to store segments, just points. The function `rayShoot(q)` will invoke a recursive helper function. Here is a suggested form, which you are *not* required to use:

```
Point rayShoot(Point2D q, KDNode p, Rectangle cell, Point best),
```

Be sure to indicate how `rayShoot(q)` makes its initial call to the helper function.

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.

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 consider how to use/modify range trees to answer two queries efficiently. Throughout, $P = \{p_1, \ldots, p_n\}$ is a set of n points in $\mathbb{R}^2$ (Fig. 6(a)). Your answer should be based on range trees, you may make modifications to P including possibly transforming the points and adding additional coordinates.

In each case, the various layers of your search structure (what points are stored there and how they are sorted) and explain how your search algorithm operates. An English explanation (as opposed to pseudocode) is sufficient. Justify your algorithm's correctness and derive its running time.

(a) Assume that all the points of P have positive x- and y-coordinates. In a *platform-dropping query*, we are given a point $q = (q_x, q_y)$ with positive coordinates. This defines a horizontal segment running from the y-axis to q. The objective is to report the first point $p \in P$ that would be hit if we drop the platform (see Fig. 6(b)). Formally, this point has the maximum y-coordinate such that $p_x \leq q_x$ and $p_y \leq q_y$. If no point of P is hit by the platform, the query returns `null`.

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

(b) In a *max empty-triangle query* you are given a point $q = (q_x, q_y)$. The objective is to compute the largest axis-parallel 45-45 right triangle that extends to the upper-right of q and contains no point of P in its interior. The answer to the query is the point of P that lies on the triangle's hypotenuse (see Fig. 6(c)). (Alternatively, you can think of this as sliding the 45° hypotenuse until it first hits a point of P). If the triangle can be grown to infinite size, return `null`.

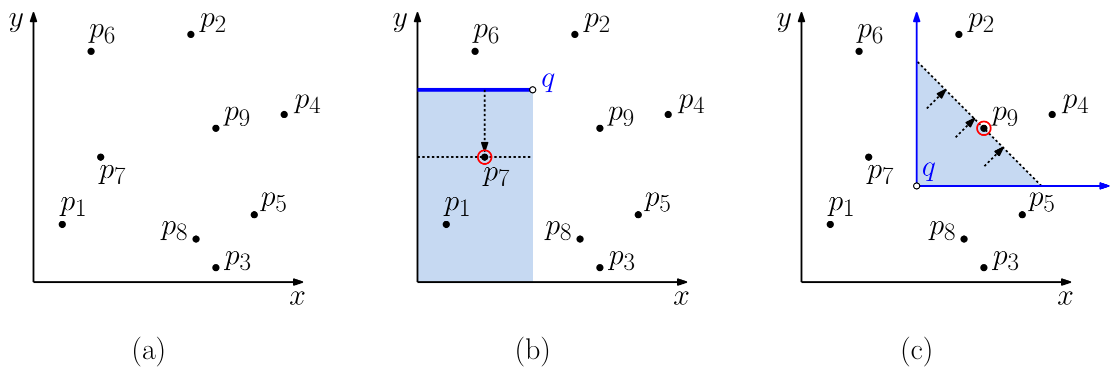

Figure 6: Platform-dropping and max empty-triangle queries.

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

Problem 9. You are designing an expandable hash table using open addressing. Let m denote the current table size. Initially $m = 4$. Let us make the ideal assumption that each hash operation takes exactly 1 time unit. After each insertion, if the number of entries in the table is greater than or equal to $3m/4$, we expand the table as follows. We allocate a new table of size $4m$, create a new hash function, and rehash all of the elements from the current table into the new table. The time to do this expansion is $3m/4$.

(a) Derive the amortized time to perform an insertion in this hash table (assuming that m is very large). State your amortized running time and explain how you derived it. (For fullest credit, your running time should as tight as possible.)

Hint: The amortized time need not be an integer.

(b) One approach to decrease the amortized time is to modify the table expansion factor, which in this case is 4. In order to reduce the amortized time, should we *increase* or *decrease* this factor? If you make this adjustment, what negative side effect (if any) might you observe regarding the space and time performance of the data structure? **Explain briefly. (Don't give a formal analysis)**