CMSC 420: Spring 2023

Practice Problems for Midterm 2

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, scapegoat trees, splay trees, and skip lists.)

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

(a) 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 (ignoring keys) its structural properties are 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 (3) Red-black trees (5) Scapegoat trees
(2) AVL trees (4) AA trees (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) 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, where 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.)

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

(f) 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?

(g) Unbalanced search trees and skip lists both support dictionary operations in $O(\log n)$ "expected time." What difference is there (if either) in the meaning of "expected time". Which would be preferred from the perspective of performance?

(h) You are given a skip list storing n items. What is the expected number of nodes that are at levels 3 and higher in the skip list? (Express your answer as a function of n. Assume that level 0 is the lowest level, containing all n items. Also assume that the coin is fair, return heads half the time and tails half the time.)

Problem 2. Recall that in an extended binary search tree, the keys are stored only in the external nodes, and each internal node stores a splitter key. Keys smaller than the splitter are stored in the left subtree and keys greater than or equal to the splitter are stored in the right subtree.

We say that an extended binary search tree is *geometrically-balanced* if the splitter value stored in each internal node `p` is midway between the smallest and largest keys of its external nodes. More formally, if the smallest external node in the subtree rooted at `p` has the value $x_{\min}$ and the largest external node has the value $x_{\max}$, then `p`'s splitter is $(x_{\min} + x_{\max})/2$ (see Fig. 1).

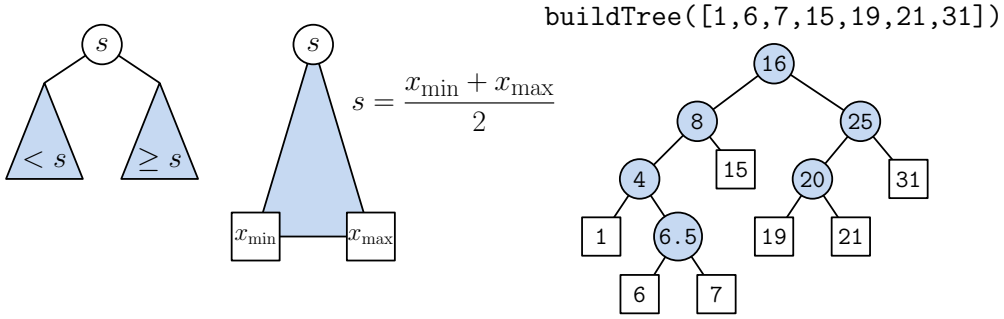

Figure 1: Geometrically balanced tree

Given a sorted array $A[0 \ldots n-1]$ containing $n \geq 1$ numeric keys, present pseudo-code for a function that builds a geometrically-balanced extended binary search tree, whose external nodes are the elements of A. (You may assume that you have access to a function for extracting a sublist of an array, but explain them.)

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

$$\frac{\texttt{size(p.left)}}{\texttt{size(p)}} \geq \frac{2}{3}$$

(see the figure below). Let T be a left-heavy tree that contains n nodes.

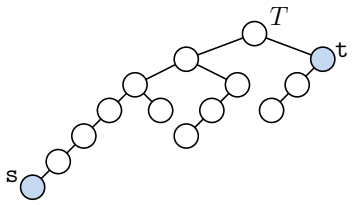

Figure 2: A left-heavy tree.

(a) Consider any left-heavy tree T with n nodes, and let `s` be the leftmost node in the tree (the first node in an inorder traversal). Prove that `depth(s)` $\geq (\log_{3/2} n) - c$, for some constant c. (The term c is just a small correction term. Let's assume that n is very large, so c may be ignored.)

(b) Consider any left-heavy tree T with n nodes, and let `t` be the rightmost node in the tree (the last node in an inorder traversal). 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. In this problem, we will consider how to perform finger searches in a skip list.

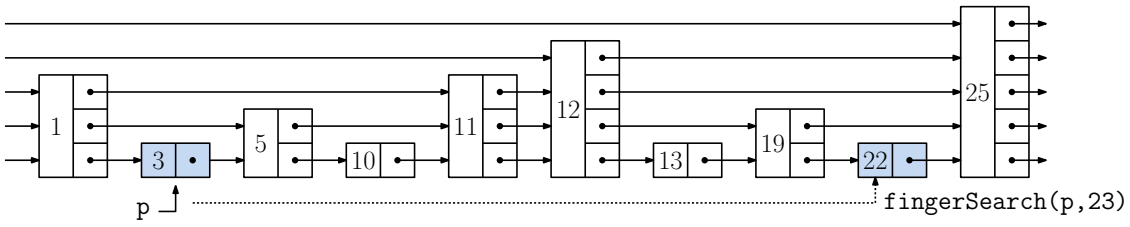

Figure 3: The operation `fingerSearch(p,23)`.

You are given a pointer `p` to a node in a skip list. You are also given a key `y`, where `p.key <= y`. Present pseudocode for a function `fingerSearch(p, y)` that performs a find operation on `y`, but rather than starting at the head node of the skip list, it starts at `p`. It returns a pointer to the node `q` that contains the largest key that is less than or equal to `y` (see Fig. 3).

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

Present pseudo-code for an algorithm for an efficient function. You do not need to analyze the running time.

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).

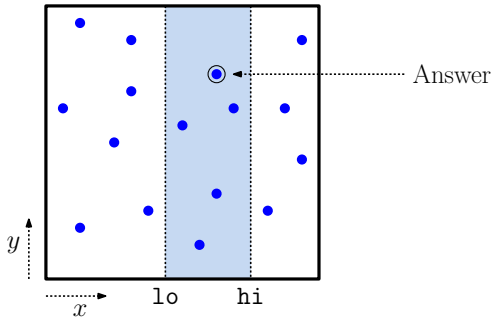

Figure 4: Partial-range max query.

(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. You may make use of any primitive operations on points and rectangles (but please explain them). **Hint:** A possible signature for your helper:

```
Point partialMax(double lo, double hi, 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.

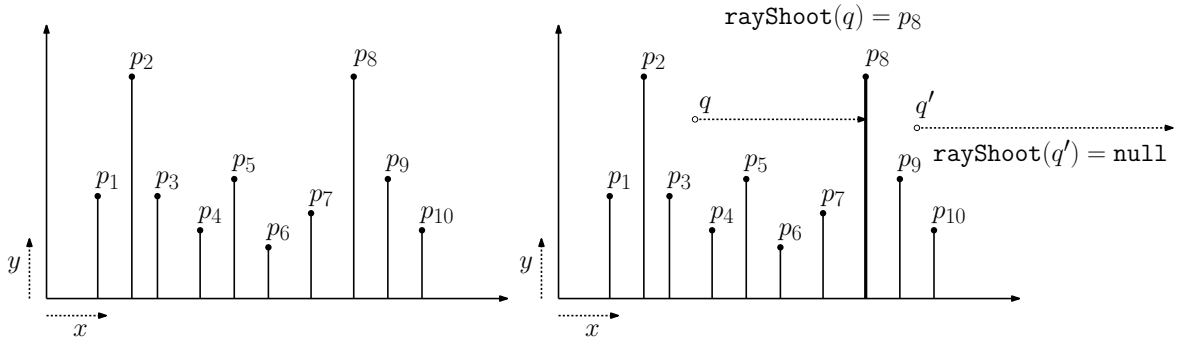

Figure 5: Ray shooting in a kd-tree.

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 need only store points in your kd-tree. Here is a suggested helper.

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

What is the initial call to the helper?

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. It is easy to see that, if you splay twice on the same key in a splay tree (`splay(x); splay(x)`), the tree's structure does not change as a result of the second call.

Is this true when we alternate between two keys? Let T_0 be an arbitrary splay tree, and let x and y be two keys that appear within T_0. Let:

- T_1 be the result of applying `splay(x); splay(y)` to T_0.
- T_2 be the result of applying `splay(x); splay(y); splay(x); splay(y)` to T_0.

Question: Irrespective of the initial tree T_0 and the choice of x and y, is $T_1 = T_2$? (That is, are the two trees structurally identical?) Either state this as a theorem and prove it or provide a counterexample, by giving the tree T_0 and two keys x and y for which this fails.

Problem 9. Throughout this problem, we start with a large, perfectly balanced binary search tree (see Fig. 6(a)). The only operations we perform are `delete` and `find`.

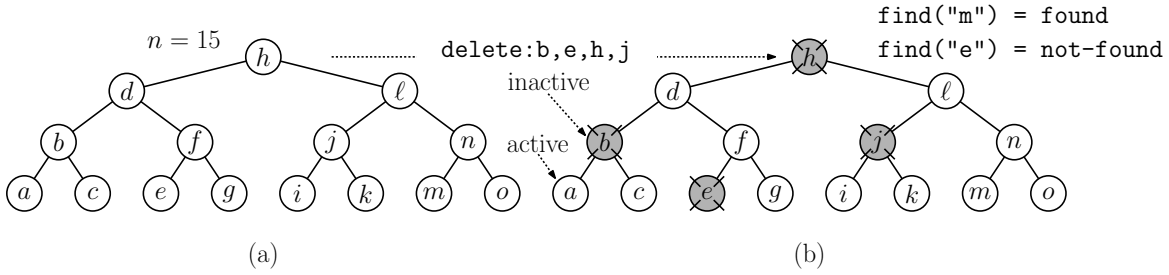

Figure 6: (a) A balanced binary search tree and (b) lazy deletion.

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. 6(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. 7). Let n denote the total number of nodes in the tree (both active and inactive).

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) 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.)

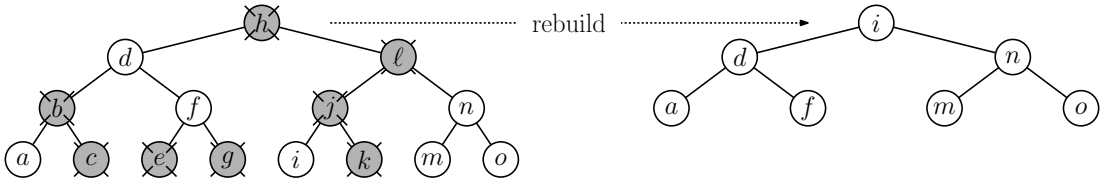

Figure 7: Rebuilding (with 8 inactive nodes and 7 active nodes).

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