CMSC 420: Spring 2022

Homework 3: More Search Trees and kd-Trees

Handed out Wed, Mar 30. Due **Wed, Apr 6, 11:59pm**. (Solutions will be discussed in class on Thu, Apr 7, so submissions will not be accepted after 9:30am, Apr 7.)

Problem 1. (15 points) Show the result of executing the operation `splay(8)` on the tree in Fig. 1.

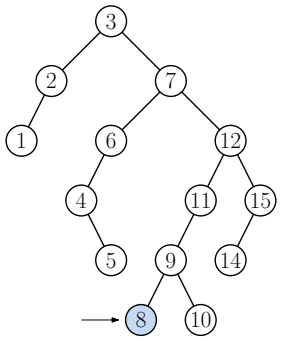

Figure 1: Splaying.

Problem 2. (15 points) Consider the scapegoat tree shown in Fig. 2. We will trace through the insertion of the key "p" into this tree (which fits between "o" and "q" alphabetically). For this problem, you may assume that `m == n` and both are equal to the number of nodes in the tree.

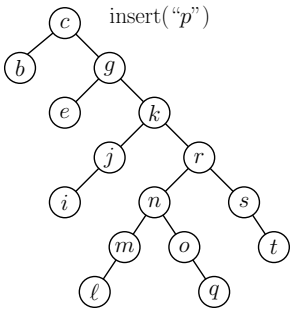

Figure 2: Insertion into a scapegoat tree.

(a) Following the insertion of "p", explain why a rebuild event will be triggered.

(b) Tracing the search path towards the root, which is the scapegoat node?

(c) Rebuild the subtree rooted at the scapegoat, and show the final tree with the rebuilt subtree inserted into its place. (Use the same algorithm given in the lecture notes for building the tree, with ties broken in the same manner.)

Problem 3. (10 points) In our lecture on Scapegoat Trees, we proved that if the tree has a node p at depth at least $\log_{3/2} n$, then somewhere along the search path to this node, there must exist a scapegoat node u. Recall that a node u is a *scapegoat* if

$$\frac{\text{size}(u.\text{child})}{\text{size}(u)} > \frac{2}{3},$$

where size(u) denotes the number of nodes in the subtree rooted at node u, and u.child denotes the child of u along the search path.

In this problem, we will ask you to generalize this result. Let α be any real constant, such that $\alpha > 1$. Complete the following lemma, and present a proof for it.

Lemma: Given a binary search tree of n nodes and any constant $\alpha > 1$, if there exists a node p such that depth(p) $> \log_\alpha n$, then p has an ancestor (possibly p itself) such that

$$\frac{\text{size}(u.\text{child})}{\text{size}(u)} > [\textit{"You fill this in"}]$$

For fullest credit, your answer to [*"You fill this in"*] should be as small as possible in the limit as $n \to \infty$. (**Hint:** Modify the proof from the Scapegoat tree lecture notes. You may also assume that `m == n`.)

Problem 4. (10 points) Throughout this problem we are given a set $P = \{p_1, \ldots, p_n\}$ of n points in 2D space stored in a point kd-tree (see Fig. 3(a)).

In a *vertical line-sliding query*, you are given an (infinitey) vertical line specified by its coordinate x_0 (see Fig. 3(b)). The query returns the first point $p_i \in P$ that is first hit if we slide the segment to its right. If no point of P are hit, the query returns `null`.

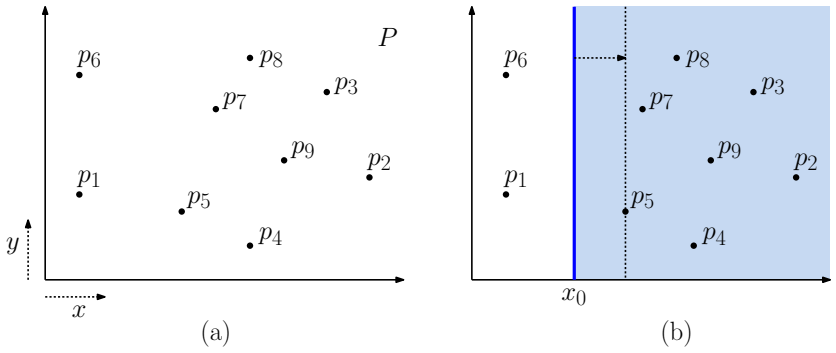

Figure 3: Segment-sliding and minimum box queries.

Present pseudo-code for an efficient algorithm, `Point vertLineSlide(scalar x0)`, which given x_0, the x-coordinate of the vertical line.

You may assume the standard 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. You

may assume that there are no duplicate coordinate values among the points of P and no point of P lies on the vertical line $x = x_0$.

Briefly explain your algorithm and derive its execution time.

Hint: For fullest credit, your program should run in time $O(\sqrt{n})$, where n is the number of points in the structure. You may assume that the cutting dimensions alternate between x and y and that the tree is perfectly balanced, which means that if p's subtree contains m points then its children's subtrees each contain at most $m/2$ points.

Challenge Problem: All modern text editors provide an *undo* command (e.g., "control-Z"). In this problem, we would like to implement such an operation for a splay tree. Each time the undo operation is applied, the most recent splay operation is undone (see Fig. 4). If undo is performed multiple times, we step backwards undoing multiple splay operations until there are no more.

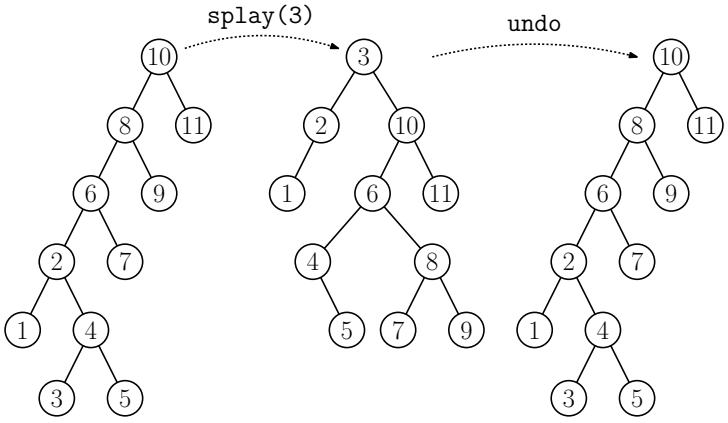

Figure 4: Undo operation on a splay tree.

The question to explore is how to do this efficiently. Obviously, we could just store the entire contents of every tree after each operation, but this would be hopelessly inefficient if the tree is very large. We would like to store the minimum amount of information in order to efficiently undo each splay.

(a) Here is an idea. There are essentially six different operations that a splay tree may perform at a given node p. These are named based on the relationship between p and its grandparent (see Fig. 5):

LL: Zig-zig, when p is a left-left grandchild (and the symmetrical **RR** operation)

LR: Zig-zag, when p is a left-right grandchild (and the symmetrical **RL** operation)

L: Zig, when p is the left child of the root (and the symmetrical **R** operation)

Each time we perform one of these operations as part of splaying, we push one of the symbols $\{LL, RR, LR, RL, L, R\}$ onto a stack S. Since each splay may consist of a variable number of rotations, before pushing these symbols, we push a special stack entry, $\perp$, which indicates the end of the rotations that constitute the splay operation.

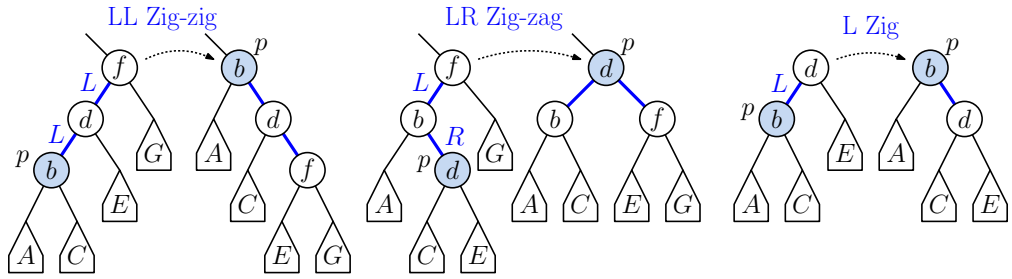

Figure 5: Undo operation on a splay tree.

For example, the splay operation of Fig. 4 involves the three rotations RL, LL, and L (from bottom to top). So the last four symbols in our stack would be $\langle \ldots, \bot, RL, LL, L \rangle$.)

Present an algorithm which given such a stack S containing a (valid) combination of the 7 symbols $\{LL, RR, LR, RL, L, R, \bot\}$, applies the appropriate modifications to restore the previous splay tree in the sequence. In particular, as each symbol is popped off the stack, explain exactly what operation is performed on the tree to undo the rotation.

(b) Taking this one step farther, suppose that we whenever the operation `splay(x)` is performed, there is a node containing `x`, and this node is a leaf node in the current tree. Show that under this assumption, it is possible to implement the undo operation *without* the need of the special symbol $\bot$.