CMSC 420: Spring 2022

CMSC 420 (0101) - Midterm Exam 2

Problem 1. (20 points) This problem involves splay trees. In both cases, apply the splay-tree algorithm given in the lecture notes.

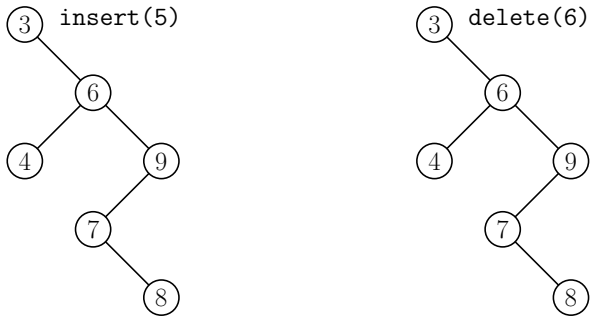


Figure 1: Splay tree operations

(a) (10 points) Show the steps involved in operation `insert(5)` for the tree in the figure. Indicate what splays are performed, and what trees result from each splay. Also indicate what node(s) are created, and show the final tree after insertion.

(b) (10 points) Show the steps involved in operation `delete(6)` for the tree in the figure. In particular, indicate what splays are performed, and what trees result from each splay. Also indicate what node(s) are removed, and show the final tree after deletion.

Problem 2. (30 points) Short answer questions. Unless requested, explanations are not required, but may always be given to help with partial credit.

(a) (4 points) By mistake, two keys in your treap happen to have the same priority. When you attempt to execute `find`, which of the following is a possible consequence? (Select all that apply.)

(1) It might go into an infinite loop
(2) It might return the wrong result
(3) It might take longer than $O(\log n)$ expected time
(4) There are no significant negative consequences

(b) (4 points) You have a splay tree storing n keys $x_1 < x_2 < \cdots < x_n$. Suppose that you perform n consecutive splay operations on these keys in sorted order:

$$\texttt{splay}(x_1),\ \texttt{splay}(x_2),\ \ldots,\ \texttt{splay}(x_n).$$

Which of the following is the *total* running time for all of these operations? (Give the tightest correct bound.)

(1) $O(n)$ (worst-case)

(2) $O(n \log n)$ (worst-case)
(3) $O(n)$ (expected case)
(4) $O(n \log n)$ (expected case)
(5) None of the above

(c) (6 points) Consider a B-tree of order $m = 13$. Answer the following questions for a single node that is *not* the root and *not* a leaf.

(a) What is the maximum number of children: Answer: _____

(b) What is the minimum number of children: Answer: _____

(c) What is the maximum number of keys: Answer: _____

(d) What is the minimum number of keys: Answer: _____

(d) (2 points) True or false: After performing an insertion in a scapegoat tree (but before any rebuilding), two or more nodes on the search path may satisfy the scapegoat condition (`size(p.child)/size(p)` $> 2/3$).

(e) (2 points) True or false: As part of performing an insertion in a scapegoat tree, two or more subtrees may be rebuilt.

(f) (4 points) In high dimensional spaces (say, dimensions greater than 10), kd-trees are preferred over quadtrees. Why is this?

(g) (4 points) Given a balanced kd-tree storing n points in 2-dimensional space, where we alternate the splitting axes, and given an *axis-parallel line* ℓ, what is the maximum number of nodes whose cell intersects ℓ?

(1) $O(\log n)$
(2) $O(\sqrt{n})$
(3) $O(n)$
(4) None of the above

(h) (4 points) Repeat (g), but this time ℓ may have an *arbitrary slope*.

Problem 3. (15 points) 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) (8 points) 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.)

(b) (7 points) Consider any left-heavy tree T with n nodes, and let `t` be the rightmost node in the tree. Prove that `depth(t)` $\leq \log_3 n$.

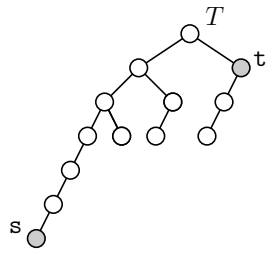


Figure 2: A left-heavy tree.

Problem 4. (15 points) You are given a set $P = \{p_1, \ldots, p_n\}$ of n points in 2D space, all having positive x- and y-coordinates (see Fig. 3(a)). They have been stored in a perfectly balanced kd-tree using alternating cutting dimensions, $x, y, x, y, \ldots$. The kd-tree stores a bounding box `bbox` that contains all the points and a pointer `root` to the root of the kd-tree.

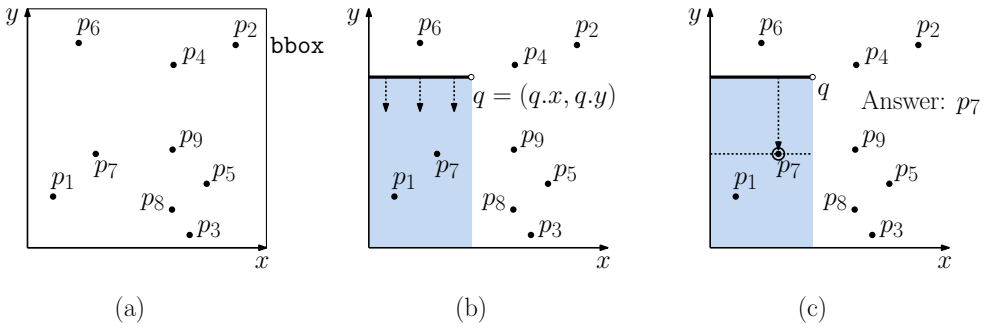


Figure 3: Platform dropping queries.

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 (see Fig. 3(b)). Our objective is to report the first point $p \in P$ that would be hit if we drop the platform. Formally, this point has the maximum y-coordinate such that $p_x \leq q_x$ and $p_y \leq q_y$ (see Fig. 3(c)).

In this problem we will derive an efficient algorithm to answer such queries. You will fill in the missing gaps to a recursive helper function. The helper is given the query point `q`, the current node `p` being visited (which might be `null`), the associated rectangular `cell` for this node, and `best`, the best answer seen so far. The function returns the new best point.

(a) (4 points) Give a condition on `cell` so that *no* point in `p`'s subtree can be hit by the falling platform? Express your answer in terms of `q = (q.x, q.y)` and the low and high cell corner points `cell.lo` and `cell.hi`.

(b) (3 points) What conditions must `p.point` satisfy to replace `best` as the new best answer to the query?

(c) (4 points) Suppose that `cell` lies entirely to the left of `q` (that is, `cell.hi.x` $\leq$ `q.x`), and further suppose that `p` has a cutting dimension of 1 (horizontal cut). Explain why only one of `p`'s children need be searched for the answer. How would you determine which child it is?

(d) (2 points) Based on your above answers, fill in the boxes boxes to complete the helper function for answering platform dropping queries. It is given the query point `q`, the current node `p` being visited (which might be `null`), the associated rectangular `cell` for this node, and `best`, the best answer seen so far. The function returns the new best point. (If there is not change from the above, you can just say "Same as (a)".)

`Note:` If the structure of our helper function does not make sense to you, you can optionally provide your own helper function in its entirety at the end of the exam. Please put in note in the first fill-in box that you are doing this.

```
Point platformDrop(Point q, KDNode p, Rectangle cell, Point best) {
    if (p == null) return best
    if (                                                    ) return best   (a)
    if (                                                    ) best = p.point   (b)
    Rectangle leftCell = cell.leftPart(p.cutDim, p.point)
    Rectangle rightCell = cell.rightPart(p.cutDim, p.point)
    if (cell.hi.x <= q.x && p.cutDim == 1) {
        if (                                                )   (c)
            best = platformDrop(q, p.left, leftCell, best)
        else
            best = platformDrop(q, p.right, rightCell, best)
    } else {
        best = platformDrop(q, p.right, rightCell, best)
        best = platformDrop(q, p.left, leftCell, best)
    }
    return best
}
```

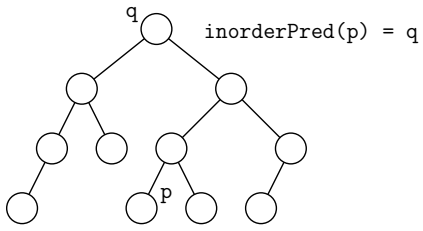
(e) (2 points) What is the initial call to the helper function?

Problem 5. (20 points)

(a) (10 points) Suppose that you are given a standard rooted binary tree, where in addition to left and right, each node has a parent link. (There are no keys.)

```
class Node {                    // a node in the tree
    Node      left              // left child
    Node      right             // right child
    Node      parent            // parent (null if root)
}
```

Present pseudocode for a function `Node inorderPred(Node p)`, which returns a pointer to inorder predecessor of `p`. If `p` has no inorder predecessor, it should return `null`.



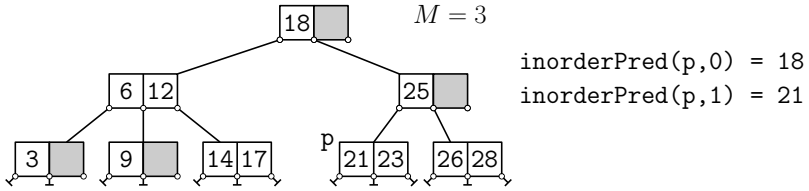
Briefly explain how your function works. If you wish, you may define additional helper functions. Your function should run in time proportional to the height of the tree.

(b) (10 points) Suppose that we are given a B-tree of order M, for some $M \geq 3$. Let us assume the following node structure:

```
class BNode {                   // a node in a B-tree
    int       nc                // number of children
    BNode     child[M]          // children pointers
    Key       key[M-1]          // keys
    BNode     parent            // parent
}
```

The children are `child[0..nc-1]` and the keys are `key[0..nc-2]`. The entry `key[i]` sits between subtrees `child[i]` and `child[i+1]`.

Present pseudocode for a function `Key inorderPred(BNode p, int i)`, which returns the key of the B-tree that immediately precedes `p.key[i]` in sorted order.



Briefly explain how your function works. If you wish, you may define additional helper functions. Your function should run in time proportional to the height of the tree (assuming M is a constant).

Hint: There are a number of cases. You can get partial credit if you can solve some of the subcases correctly. For example, you might start by considering what happens when `p` is a leaf node.