CMSC 420 (0201) - Midterm Exam 2

Problem 1. (10 points) Consider the kd-tree shown in Fig. 1. Assume a "standard" kd-tree where the cutting dimensions alternate between x and y with each level.

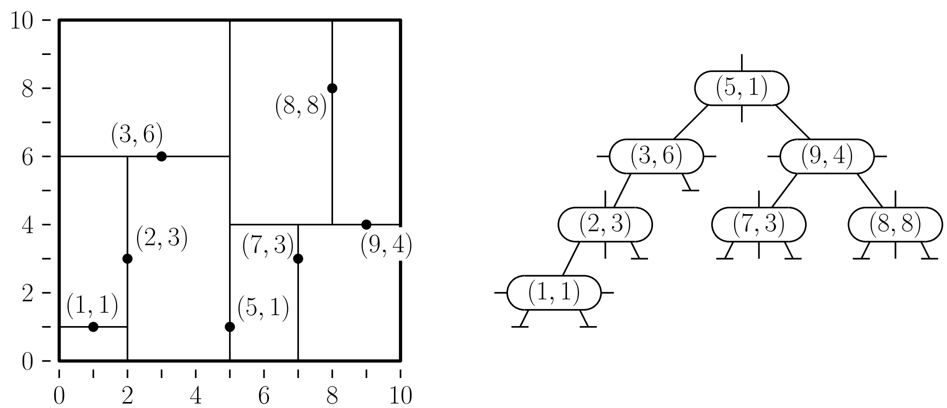


Figure 1: kd-Tree operations.

(a) (5 points) Show the final tree after the operation `insert((6,6))`. You need only show the tree, not the spatial subdivision.

(b) (5 points) Starting with the *original tree*, show the final tree after `delete((3,6))`. Indicate which nodes were used as replacement nodes. (Intermediate results are not required, but may be given for partial credit.)

Problem 2. (35 points) Short answer questions. No explanations required, but can be given for partial credit.

(a) (7 points) Consider a 2-dimensional point quadtree with m nodes. As an exact function of m, how many `null` pointers does it have? (Partial credit given depending on how close.)

(b) (7 points) You have a scapegoat tree, but you make two changes. First, a rebuild is triggered when an inserted node's depth exceeds $\log_{10/9} n$ (instead of $\log_{3/2} n$), and second a node `p` is declared a scapegoat if `size(p.child)/size(p)` $> 9/10$ (instead of $2/3$). Compared to the standard scapegoat tree, what changes? (Select all that apply.)

(1) The tree's height will tend to be *larger*

(2) The tree's height will tend to be *smaller*

(3) Subtrees will tend to be rebuilt *more often*

(4) Subtrees will tend to be rebuilt *less often*

(c) (3 points) What is the *maximum* number of subtrees that may need to be rebuilt as a result of a single insertion into a scapegoat tree? (Select the best option.)

(1) 1
(2) More than one, but a constant number
(3) $O(d)$, where d is the depth of the inserted node
(4) $O(h)$, where h is the overall height of the tree (even if the node is inserted at a much smaller depth)
(5) Larger than $O(h)$

(d) (7 points) You have a skip list with n nodes. Suppose that rather than using a fair coin to decide a node's height, you instead use a coin that comes up heads with probability 3/5 and tails with probability 2/5. All nodes start at level 0, and a node survives to the next higher level if the coin toss comes up heads. As a function of n, what is the expected number of nodes that survive to level 2 and higher?

(e) (3 points) You have a skip list containing n keys, where n is a large number. Suppose you perform a find operation. The search algorithm visits one or more nodes at each level of the structure. How many nodes do you *expect to visit* at level 4 of the search structure?

(1) None of them
(2) $O(1)$
(3) $O(\log n)$
(4) $O(n/(2^4))$
(5) All of them

(f) (5 points) Splay trees operate by performing most rotations in groups of two (zig-zag and zig-zig). Why is this necessary? In particular, why not just perform single rotations from the bottom up to bring the node up to the root?

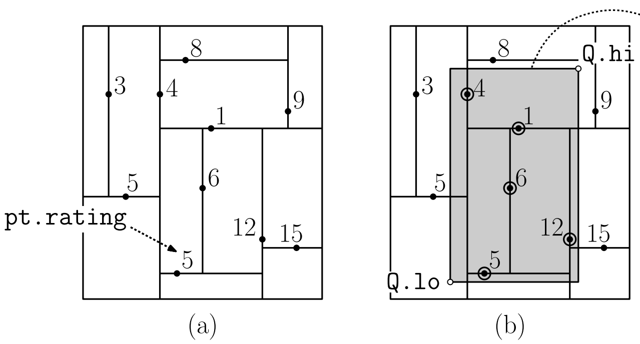
(g) (3 points) You have a splay tree with a large number n of keys, and you perform a long series of m `find` operations (where m is much larger than n). Suppose that one key is extremely popular, say `"Taylor Swift"`. Indeed, *every third* `find` is made to this popular key. What can you say about the time needed for the `find` operations on this popular key?

(1) They will have an amortized cost $O(1)$
(2) They will have an amortized cost of $O(\log n)$, but not $O(1)$
(3) They will have an amortized cost of $O((\log n)^3)$, but not $O(\log n)$
(4) They will have an amortized cost of $O(n)$, but not $O((\log n)^3)$
(5) They will have an amortized cost exceeding $O(n)$

Problem 3. (15 points) You are given a 2-dimensional point set stored in a standard kd-tree (cutting dimensions alternate). Let `root` denote its root, and let `rootCell` denote the root's rectangular cell.

Our objective is to answer queries of the form "What is the top rated restaurant in a given region?" To do this, in addition to its coordinates, each point pt $\in P$ stores a positive integer rating, `pt.rating` (see Fig. 2(a)). We are given a query rectangle Q, with lower-left and upper-right corner points, `Q.lo` and `Q.hi`, respectively. The function `rangeMax(Q)` returns the *maximum rating* among the points of P that lie within Q (see Fig. 2(b)). If there are no points in the range, it returns 0.

To help, each node `p` in the tree stores a field `maxRating`, which is the maximum rating over all the points in `p`'s subtree (see Fig. 2(c)).




```
class KDNode {
  Point point   // node's point
  int cutDim    // node's cutting dim
  int maxRating // max rating in subtree
  KDNode left, right // children
}
```

(c)

Figure 2: Range max queries.

(a) (10 points) Present pseudo-code for the kd-tree function `rangeMax(Rectangle Q)` that efficiently answers these queries. For full credit, it should run in $O(\sqrt{n})$ time.

Hint: You may use whatever helper you like. Here is a suggestion.

```
int rangeMax(Rectangle Q, KDNode p, Rectangle cell)
```

You may assume that any geometric primitive involving a constant number objects (e.g., "is `Q` disjoint from `cell`") can be computed in constant time.

(b) (5 points) How do we update node `maxRating` values with each insertion? Edit/Modify the insertion helper for the kd-tree so that it both inserts a point `pt` with rating `pt.rating` and efficiently *updates* the `maxRating` values for the affected nodes of the tree.

Problem 4. (10 points) Suppose you are given a splay tree storing the keys $X = \{x_1, x_2, \ldots, x_n\}$. Design a new splay-tree operation called `bulkDelete(a, b)`. It is given two keys $a, b \in X$, where $a < b$, and it *deletes* all the keys between a and b, *exclusive*, that is, it deletes $\{x \in X : a < x < b\}$. (The elements a and b are *not* deleted.) For example, in Fig. 3(b), the operation `bulkDelete(4, 12)` deletes the keys $\{5, 6, 7, 8, 10\}$.

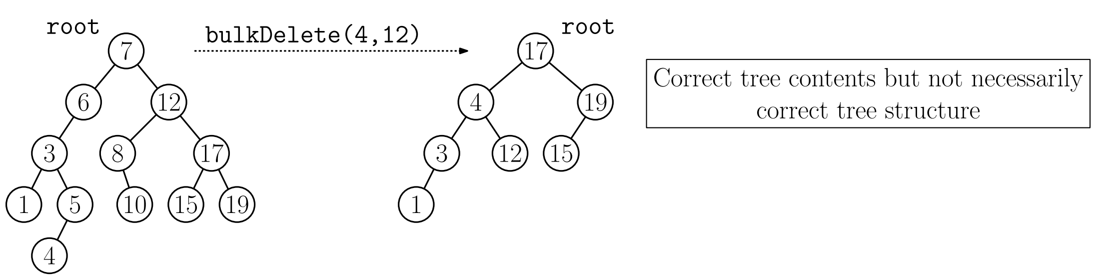


Figure 3: The `bulkDelete` operation in splay trees.

Present an efficient algorithm for this operation. As with other splay-tree operations, you are allowed to perform splay operations, either on the entire tree or on subtrees, and you can access and modify nodes. However, you are *not* allowed to iterate through the tree or apply recursive functions to the tree (other than calling `splay`).

You may present your algorithm either in pseudo-code or in English. You may assume that $a < b$, and both keys appear in tree. **Hint:** It is possible to do this with a *constant* number of splays, no matter how many entries are deleted.

Problem 5. (10 points) In this problem we consider an enhanced version of a skip list. As usual, each node `p` stores a key, `p.key`, and an array of next pointers, `p.next[]`. To this we add a parallel array `p.span[]`, where `p.span[i]` stores the number of nodes that `next[i]` skips.

Present pseudo-code for a function `Key getKth(int k)`, which returns the kth smallest key in the entire skip list. For example, in Fig. 4, the call `getKth(6)` would return 19, since 19 is the sixth smallest key. You may assume that $1 \leq k \leq n$, where n is the total number of nodes in the skip list.

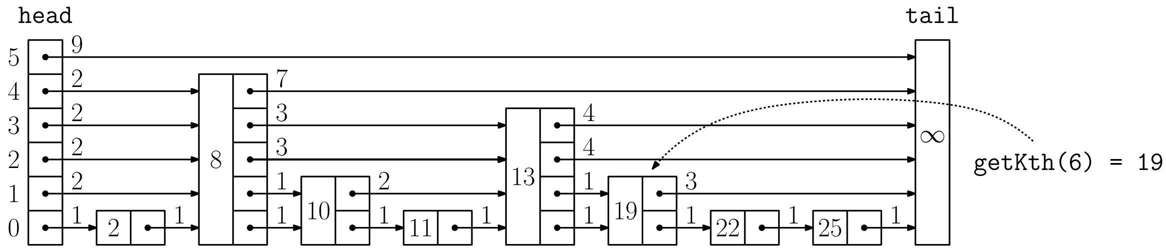


Figure 4: Skip list with span values.

Your procedure should run in time expected-case time $O(\log n)$ (over all random choices), but you don't need to prove this.

Problem 6. (20 points) Recall that an extended binary search tree consists of internal nodes, which have exactly two children, and external nodes, which have no children. A node's *weight* is defined to be the number of external nodes its subtree. (Internal nodes are not counted.) For example, in Fig. 5 each node is labeled with its weight.

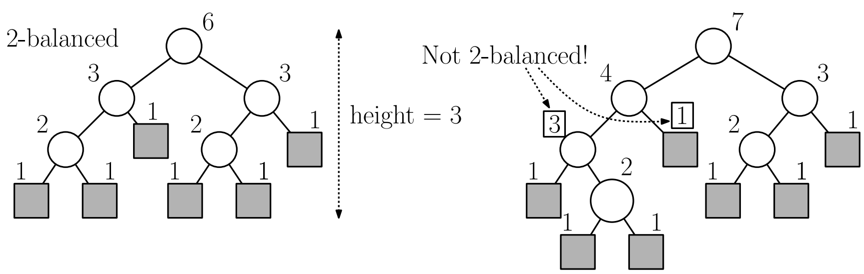


Figure 5: Weight-balanced extended trees.

Given $\alpha \geq 1$, an external tree is *α-balanced* if for every internal node `u`,

$$\frac{1}{\alpha} \;\leq\; \frac{\text{weight(u.left)}}{\text{weight(u.right)}} \;\leq\; \alpha.$$

In Fig. 5, the tree on the left is 2-balanced. But the tree on the right is not because there are two siblings with weight ration $3:1$, exceeding the allowed ratio of $2:1$.

(a) (10 points) Prove that if an extended binary tree of total weight $n \geq 1$ is 2-weight balanced, then its height is at most $\log_{3/2} n$.

(b) (5 points) Generalize the result from (a). Given an extended tree that is α-balanced for some $\alpha \geq 1$, its height is at most $\log_{\beta} n$ for some β that depends on α. What is the value of β as a function of α? (You do not not need to give the proof, just the formula).

(c) (5 points) **True or False**: Given a weight balanced tree with total weight n, if there is a node at depth greater than $\log_{3/2} n$, then some ancestor of this node is *not* 2-weight balanced. Give your answer and a brief (1–2 sentence) justification.