CMSC 420: Fall 2022

Solutions to Practice Problems for Midterm 2

Solution 1:

(a) AVL trees, red-black trees, treaps, and splay trees are all symmetrical. Leftist heaps are not symmetrical because of the leftist condition, and AA trees are not symmetrical because the skew condition implies that left subtrees are never deeper than right subtrees.

(b) Splay tree: This sort of efficiency is called *static optimality* (where the access probabilities are non-uniform but do not change over time). Among the dictionary data structures we saw this semester, only the *splay tree* is efficient with respect to static optimality, since it restructures itself so that more frequently accessed keys are placed near the root of the tree.

(c) In d-dimensional space, each node in a point quadtree stores one point and has up to 2^d children. In dimension 3, this means 8 children. The root level has one node, level 1 has 8 nodes, level 2 has 64 nodes, and generally, level i has 8^i nodes. Since each node stores a single point, the maximum number of points in a tree of height h is

$$\sum_{i=0}^{h} 8^i \;=\; \frac{8^{h+1}-1}{8-1} \;=\; \frac{8^{h+1}-1}{7} \;\approx\; 8^h.$$

(d) Each node of a quadtree in dimension d has 2^d children. If the dimension is high, the resulting tree has an insanely high degree. (For example, in $d = 10$, each node has 1024 child pointers.)

(e) The expected height is $O(\log n)$. The analysis is essentially the same as that of a standard (unbalanced) binary search tree, since, irrespective of the dimension, the next splitting chosen randomly from among the points to be inserted into the subtree.

(f) Remarkably, the expected height is still $O(\log n)$, although with about twice the constant factor. While each x-split insertion is entirely degenerate, each y-insertion is essentially random. Thus, there are in expectation $O(\log n)$ y-cutting levels, and hence the total height is roughly twice this. We can get a more formal analysis in the special case where the y-splits are perfectly balanced. Then with every two levels of the kd-tree, we decrease the points by half. Thus, the number of levels satisfies the recurrence $T(n) = 2 + T(n/2)$, and this solves to $T(n) = 2 \lg n$.

(g) The two that guarantee finding a single empty slot are (1) linear probing and (6) double hashing when m and g are relatively prime. All the others may visit only a strict subset of entries and so may miss an empty slot.

Solution 2:

(a) Remember that the priorities in a treap are heap ordered, so that a parent has a lower priority than its children. Any node `p` where $x_0 \leq$ `p.key` $\leq x_1$ is a candidate answer, and by heap ordering, the answer is the highest such node in the treap, that is, the first such node we encounter.

We present below pseudo-code for the helper. The initial call is `return minPriority(x0, x1, root)`. If `p.key` $> x_1$, we recurse on the left subtree, and if `p.key` $< x_0$, we recurse on the right subtree. Otherwise, we are in the interval and return the current key.

```
int minPriority(Key x0, Key x1, TreapNode p) {
    if (p == null) return Integer.MAX_VALUE   // fell out of the tree?
    else if (p.key > x1)                      // range to left of p.key
        return minPriority(x0, x1, p.left)    // ...search left
    else if (p.key < x0)                      // range to right of p.key
        return minPriority(x0, x1, p.right)   // ...search right
    else                                      // p.key in range?
        return p.priority                     // this has lowest priority
}
```

(b) In the worst case, the algorithm traverses a single path in the tree, and so the running time is proportional to the treap's height, which is $O(\log n)$ in expectation.

Solution 3:

(a) Let s denote the leftmost node in the tree. The size of the root is n. By left-heaviness, the size of its left child is at least $(2/3)n$, the size of its left-left grandchild is $(2/3)((2/3)n) = (2/3)^2n$, and generally the size of its d-fold left descendant is at least $(2/3)^dn$.

By left-heaviness, we may assume that s is a leaf, and therefore `size(s)` $= 1$. Letting d denote `s`'s depth, we have $1 =$ `size(s)` $\geq (2/3)^dn$. Solving for d, we have $(3/2)^d \geq n$, which implies that $d \geq \log_{3/2} n$, as desired. (This proof is not formally correct, since the left-heaviness condition only applies if `size(s)` ≥ 3, but we can correct this by adjusting the constant c.)

(b) Let t denote the rightmost node in the tree. The size of the root is n. By left-heaviness, the size of its left child is at least $(2/3)n$, and therefore the size of its right child is at most $n-1-(2/3)n \leq n/3$. The size of its right-right grandchild is at most $(1/3)((1/3)n) = (1/3)^2n$, and generally the size of its d-fold right descendant is at most $(1/3)^dn$.

We have `size(t)` ≥ 1. Letting d denote `t`'s depth, we have $1 \leq$ `size(t)` $\leq (1/3)^dn$. Solving for d, we have $3^d \leq n$, which implies that $d \leq \log_3 n$, as desired.

Solution 4: We start at level 0, since we know that every node exists at this level. The search involves two phases, called *up-phase* and *down-phase*. During the up-phase, we try to move up to higher levels if possible. If not, we follow the `next` pointer. If we see that the `next` link leads us beyond the search key `y`, we initiate the down-phase. This is the same as the standard skip-list search. It attempts to move forward if the move does not take us beyond the search key. If it would, we instead move down a level. The process terminates when the level number becomes negative. If the key is in the structure, it will be at this final node (see Fig. 1).

```
Value forwardSearch(SkipNode p, Key y) {
    int i = 0                              // start at the lowest level
    while (p.next[i].key <= y) {           // up-phase
        if (i+1 < p.next.length) i++       // move up, if possible
        else p = p.next[i]                 // move horizontal, if not
    }
    while (i >= 0) {                       // down-phase
        if (p.next[i].key <= y) {          // can we move forward?
            p = p.next[i]                  // do so
        else i--                           // drop down a level
    }
    return (p.key == y ? p.value : null)  // return value if found
}
```

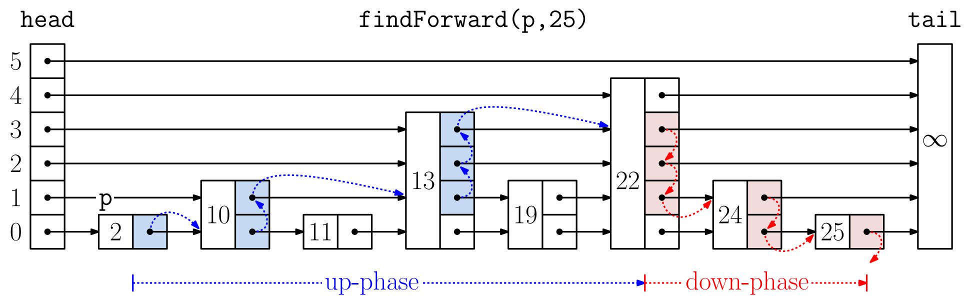

Figure 1: Find-forward in a skip list.

We will not analyze the running time, but intuitively, we expect to make a constant number of hops at each level, and each time we move up a level, we are doubling the distance traveled. Thus, if there m nodes between our starting and ending node, we expect to spend $O(\log m)$ hops in the up-phase and $O(\log m)$ hops in the down-phase.

Solution 5:

(a) We assume we have an object called `Strip`, where `strip.lo` and `strip.hi` are the strip's left and right endpoints, respectively. Let's assume we have two utility functions. The first tests whether a rectangular cell is disjoint from the strip, and the second determines whether a point is contained in the strip.

```
boolean inStrip(Point pt, Strip s) { return pt.x >= s.lo && pt.x <= s.hi }

boolean isDisjoint(Rectangle r, Strip s) { return r.lo.x > s.hi || r.hi.x < s.lo }
```

We apply the standard approach for answering range searching queries. We visit nodes of the kd-tree recursively. Let `p` denote the node currently being visited. If we fall out of the tree or if the cell is disjoint from the strip, we return the current best. If the node's point is in the strip and higher than the current best, it replaces best. Otherwise, we construct the two child cells. If the node is a vertical splitter, we search both subtrees. If it is a horizontal splitter, we try the right (upper) subtree first. If the left subtree is still relevant to the search, then we visit it. The initial call at the root level is `partialMax(strip, root, bbox, null)`, where `bbox` is the kd-tree's bounding box.

```
Point partialMax(Strip s, KDNode p, Rectangle cell, Point best) {
  if (p == null || isDisjoint(cell, s)))           // trivial cases
    return best
  if (inStrip(p.point, s) && (best == null || p.point.y > best.y))
      best = p.point                               // new best
                                                   // children cells
  Rectangle leftCell = cell.leftPart(p.cutDim, p.point)
  Rectangle rightCell = cell.rightPart(p.cutDim, p.point)

  if (cutDim == 0) {                               // vertical cut?
    best = partialMax(s, p.left, leftCell, best)   // try both sides
    best = partialMax(s, p.right, rightCell, best)
  } else {                                         // horizontal cut?
    best = partialMax(s, p.right, rightCell, best) // try right first
    if (best == null || p.point.y > best.y)        // is left relevant?
      best = partialMax(s, p.left, leftCell, best) // try left
  }
  return best
}
```

(b) The running time of the algorithm is $O(\sqrt{n})$ under the standard assumptions about kd-trees. To see this, observe first that, by applying the standard orthogonal range search analysis for kd-trees, the number of nodes whose cells are stabbed by the two vertical sides of the range is $O(\sqrt{n})$. The remaining nodes have cells that are either entirely inside or outside the vertical strip. If the cell is outside the strip, the search will return immediately.

All that remains is to analyze the number of nodes whose cell lies entirely inside the strip. For each horizontal splitter, we never need to visit the lower child (since the point stored in this node will provide a larger y-coordinate than any point in this subtree). Therefore, we will visit at most two out of the four grandchildren of any such node. It follows that the total number of nodes of this last type that are visited by the search satisfies the recurrence $T(n) = 2T(n/4) + 3$, which by the Master Theorem solves to $O(\sqrt{n})$.

Solution 6: A key observation is that this problem, which apparently has to do with line segments can be answered by a data structure that just stores points! It is not hard to see that answering a left-to-right horizontal ray-shooting query at point $q = (q_x, q_y)$ is equivalent to computing the point of P with the minimum x-coordinate that lies within the northeast quadrant of q (see Fig. 2). To see why, observe that in order to hit any segment, its topmost point must lie to q's right and have a higher y-coordinate, thus it lies in q's northeast quadrant. We seek the first such point, that is, the one with the lowest x-coordinate. A point lying in q's northeast quadrant is called a *candidate* (points $\{p_8, p_9, p_{10}\}$ in the figure), and the *best* candidate is the one with the smallest x-coordinate, that is, p_8.

We will apply the standard approach for answering range searching queries. We visit nodes of the kd-tree recursively. Let `p` denote the node currently being visited, and let `cell` denote its associated cell. Let `best` denote the best candidate seen so far. The initial call at the root level is `rayShoot(q, root, bbox, null)`, where `bbox` is the bounding box for the entire tree.

Let Q denote the region lying to the northeast of q (shaded in blue in Fig. 2). Let's assume we have two utility functions. The first tests whether a rectangular cell is disjoint from Q, and the second determines whether a point is contained within Q.

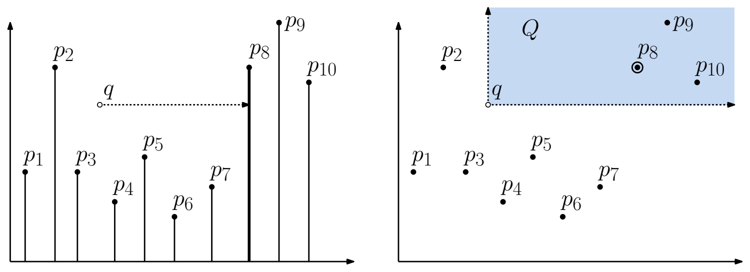

Figure 2: Horizontal ray shooting via the kd-tree.

```
boolean inQ(Point pt, Point q) { return pt.x >= q.x && pt.y >= q.y }

boolean isDisjoint(Rectangle r, Point q) { return r.hi.x < q.x || r.hi.y < q.y }
```

If we fall out of the tree or if the cell is disjoint from Q, we return the current best. If the node's point is in Q and left of the current best, it replaces best. Otherwise, we construct the two child cells. If the node is a horizontal splitter, we search both subtrees. If it is a vertical splitter, we try the left subtree first. If the right subtree is still relevant to the search, then we visit it. The initial call at the root level is `rayShoot(q, root, bbox, null)`, where `bbox` is the kd-tree's bounding box.

```
Point rayShoot(Point q, KDNode p, Rectangle cell, Point best) {
  if (p == null || isDisjoint(cell, q)))           // trivial cases
    return best
  if (inQ(p.point, q) && (best == null || p.point.x < best.x))
    best = p.point                                 // new best
                                                   // children cells
  Rectangle leftCell = cell.leftPart(p.cutDim, p.point)
  Rectangle rightCell = cell.rightPart(p.cutDim, p.point)

  if (cutDim == 1) {                               // horizontal cut?
    best = rayShoot(q, p.left, leftCell, best)     // try both sides
    best = rayShoot(q, p.right, rightCell, best)
  } else {                                         // vertical cut?
    best = rayShoot(q, p.left, leftCell, best)     // try left first
    if (best == null || p.point.x < best.x)        // is right relevant?
      best = rayShoot(q, p.right, rightCell, best) // try right
  }
  return best
}
```

The running time of the algorithm is $O(\sqrt{n})$ under the standard assumptions about kd-trees. (The proof is similar that of the previous problem.)

Solution 7: We create a line with slope of -1 and place the points either on or very close to this line. To keep the tree balanced, we insert them in a balanced manner, recursively inserting the median point (see Fig. 3). It is easy to see that the line will stab all the leaf cells of the kd-tree, and hence it stabs all the cells (ancestors as well).

Solution 8:

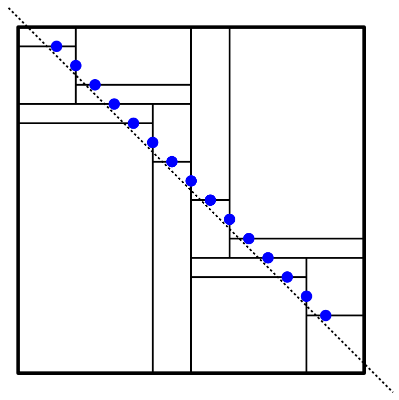

Figure 3: A line that stabs all the cells of a kd-tree.

(a) The preprocessing consists of building a 2-layer structure. The first layer is a standard 1-dimension range tree sorted by x-coordinates. Each auxiliary tree is just a 1-dimensional range tree sorted by y-coordinates. We assume that the auxiliary trees can efficiently answer a query called `findDown(y0)`. Given a set of points, this query is given a y-coordinate value y_0 and returns the point (if any) with the largest y-coordinate that is less than or equal to y_0.[1] As this is a standard 2-layer tree, the space is $O(n \log n)$.

Given a point $q = (q_x, q_y)$ the query is answered as follows. First, we consult the x-range tree to extract a set of $O(\log n)$ nodes such that the points contained within these subtrees cover the interval $[0, q_x]$. For each of these points, we perform the operation `findDown(q.y)` to obtain the point with the highest y-coordinate lying on or below q_y. Each of these queries takes $O(\log n)$ time, for a total query time of $O(\log^2 n)$. All of the resulting points are valid candidate answers for the query, since their x-coordinates lie in $[0, q_x]$, and their y-coordinates lie on or below q_y. Among all the result, we select the one with the largest y-coordinate.

(b) This problem is equivalent to sliding a line of slope -1 passing through q in the northeast direction until it first hits a point that lies in the northeast quadrant with respect to q.

The preprocessing consists of building a 3-layer structure. The first layer is a standard 1-dimension range tree sorted by x-coordinates. The second layer is a standard 1-dimensional range tree sorted by y-coordinates. And the third layer is a 1-dimensional range tree sorted by the additional coordinates $z = x + y$. We assume that the third layer can answer the query `findUp(z0)`, which returns the point with the smallest z-coordinate that is greater than or equal to z_0. As this is a standard 3-layer tree, the space is $O(n \log^2 n)$.

Given a point $q = (q_x, q_y)$ the query is answered as follows. First, we consult the x-range tree to extract a set of $O(\log n)$ nodes such that the points contained within these subtrees cover the interval $[q_x, +\infty]$. Among all these canonical subtrees, we access the 2nd-layer auxiliary tree to find all the subtrees y that cover the interval $[q.y, +\infty]$. All the points that survive this process have (x, y) coordinates such that $x \geq q_x$ and $y \geq q_y$. Finally, for each of these

[1] Assuming that the tree has parent links, this query can be answered in $O(\log n)$ time. First, a search is performed for the key y_0. If it is found, it is returned as the answer. If not, its inorder predecessor is returned. Using parent links, this inorder predecessor can be found in $O(\log n)$ time.

subtrees, we access the 3rd-layer invoking the function `findUp`$(q_x + q_y)$. Among all of these, we return the one with the smallest z-value. Since all the points involved in these searches lie in the northeast quadrant with respect to q, they are all candidates to be hit by the sliding line. The one with the smallest $z = x + y$ coordinate is the first to be hit by the sliding line.

The query involves first searching in the x-tree, which results in $O(\log n)$ auxiliary trees. We search of these y-trees. Since each takes $O(\log n)$ time, and there are $O(\log n)$ of them, this takes total $O(\log^2 n)$ time and results in $O(\log^2 n)$ z-trees. The `findUp` query in each of them takes $O(\log n)$ time, for a total of $O(\log n) + O(\log^2 n) + O(\log^3 n) = O(\log^3 n)$ time.

Solution 9:

(a) We will show that the amortized time $\alpha = 7/3 = 2.333\ldots$. Each time we perform an insertion, we receive α tokens. One of these tokens will be used to pay for the insertion, and the remaining $\alpha - 1$ are put in a bank account to pay for the next expansion. Let us assume that we have just expanded a table of size m resulting in a new table of size $m' = 4m$, which contains $3m/4$ entries. In order to induce the next expansion, the total number of entries must grow to $(3/4)m' = (3/4)(4m) = 3m$. This means that the number of new insertions is at least $3m - (3m/4) = (9/4)m$. Through these insertions we have collected $(9/4)m(\alpha - 1)$ tokens. We need to have enough tokens to pay the expansion cost, which is $3m$. Therefore, α must satisfy:

$$\frac{9m}{4}(\alpha - 1) \;\geq\; 3m \quad\Longrightarrow\quad \alpha \;\geq\; 1 + \frac{4}{3} \;=\; \frac{7}{3},$$

as desired.

(b) To decrease the amortized cost, we should *increase* the expansion factor, since this reduces the frequency with which expansions take place (but does not increase their cost). This increase has the negative side effect that we may waste more space if we never fill up the expanded table. For example, if we expanded the table by a factor of 400 instead of 4, expansions would be very infrequent, but the final expansion could potentially waste a lot of space.