Solutions to CMSC 420 (0201) - Midterm Exam 2

Solution 1: See Fig. 1

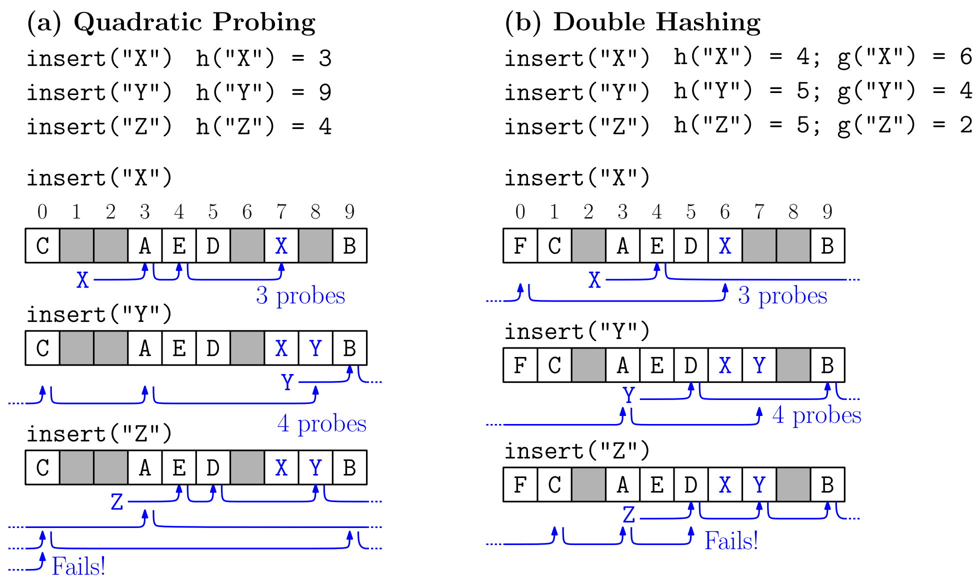

Figure 1: Hashing with open addressing.

Solution 2:

(a) Min: 4, Max: 7 (The minimum is a tree with a root where one subtree is a single node, and the other is a two-node subtree. The maximum is a complete tree of height two.)

(b) 2-3 trees (it is also used by B-trees, but we have not covered them yet)

(c) $n/9$: All n contribute to level 0, $n/3$ are expected to contribute to level 1, and $(n/3)/3 = n/9$ are expected to contribute to level 2.

(d) 9, 19: For the min, we store two points in each external node. This gives 10 external nodes and hence 9 internal nodes. For the max, we store one point in each external node for 20 externals and 19 internals. (I will also accept 18 as an answer for the max. The reason is that if the tree is built using the algorithm given in the programming assignment, and points are inserted one by one, then at least one bucket must have two points. Thus, there are 19 external nodes and 18 internal nodes.)

(e) Double hashing: The use of the second hash function implies that, even if two keys collide, they are very unlikely to have the same probe sequences.

(f) The load factor λ is defined to be n/m, that is, the fraction of the table that is utilized.

Solution 3: The recursive helper is presented below. The initial call is `findUp(x, root)`. First, we check whether we fall out of the tree, and if so, we return `null`. If we find the key `x` at this node, we return the key. Otherwise, if `x` is larger than `p.key`, we know that the answer cannot be in the left subtree, so we recurse on the right subtree. (Note that it might not be here either, but that is okay.) Finally, if `x` is smaller than `p.key`, we recurse on the left subtree. If we find the answer there, we just return it. However, if the answer is not found there (`result == null`), it must be that `x` is strictly larger than every key in our left subtree, and the next larger key in the tree is the key stored in this node, so we return `p.key`.

```
Key findUp(Key x, Node p) {
    if (p == null) return null
    else if (x == p.key) return p.key
    else if (x > p.key) return findUp(x, p.right)
    else /* x < p.key */ {
        Key result = findUp(x, p.left)
        return (result == null ? p.key : result)
    }
}
```

The function makes one recursive call for each level of the tree, so clearly the running time is $O(h)$, where h is the tree's height.

Solution 4:

(a) We adapt the kd-tree range searching algorithm. Our helper function `int crcHelper(Point c, double r, KDNode p, Rectangle cell)` is given a node `p` and its cell `cell`. The initial call is `crcHelper(c, r, root, bbox)`, where `bbox` is the bounding box for the entire kd-tree.

The helper is presented below. If we fall out of the tree or the cell is disjoint from the query range (its min distance from `c` is greater than `r`) we return zero. If the node's cell is completely contained within the disk (that is, its maximum distance from `c` is at most `r`) we return all the points in this node's subtree. If neither holds, we check whether this point lies within the disk (that is, the distance from `c` to `p.point` is at most `r`), and if so we count it. Finally, we recurse on both children, passing in their corresponding cells.

```
int crcHelper(Point c, double r, KDNode p, Rectangle cell) {
    if (p == null || minDist(c, cell) > r)    // trivial cases
        return 0
    else if (maxDist(c, cell) <= r)           // cell contained in disk
        return p.size                         // take all its points
    else {
        int ct = 0
        if (dist(c, p.point) <= r) ct += 1    // consider this point
                                              // apply to children
        ct += crcHelper(c, r, p.left,  cell.leftPart(p.cutDim, p.point))
        ct += crcHelper(c, r, p.right, cell.rightPart(p.cutDim, p.point))
        return ct
    }
}
```

(b/c) The worst-case running time is $O(n)$, even if no points lie within the disk. Although the algorithm is probably quite practical, it is possible to create a scenario where almost all the points lie barely outside the disk, but the disk overlaps all the n leaf cells of the tree (see Fig. 2). The algorithm needs to visit all of these leaf cells, but no points are reported.

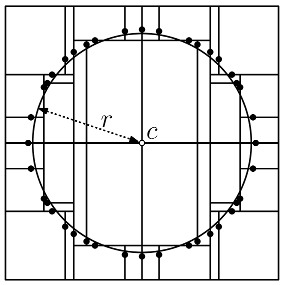

Figure 2: Circular range counting worst case. Every cell in the tree needs to be visited, but none of the points lies in the circular disk.

Solution 5: The answer is based on a 2-dimensional range tree. There are two approaches. The first is to sort the primary tree by y-coordinates, then build the auxiliary tree sorted by x-coordinates, and use `findUp` to locate the smallest item to the right of the segment. The alternative is to use the classical ordering (x- then y-), but each node of the auxiliary tree stores its descendant with the minimum x-coordinate. We will describe the former approach.

(a) The data structure is a two-layer range tree. The primary (first-level) tree stores the points of P sorted by their y-coordinates. The second-layer auxiliary trees each store points of the associated primary subtree sorted by their x-coordinates. We assume that the auxiliary trees support the operation `findUp(x)`, which returns the smallest key in the tree whose value is greater than or equal to `x`.

Since this is a standard two-layer range tree, the space is $O(n \log n)$.

(b) We answer a query as follows. Let `s` denote the query segment. We first apply the standard range search to the primary tree to identify all the nodes u corresponding to the maximal subtrees lying in the interval [`s.ylo`, `s.yhi`]. For each such u, we then access its auxiliary tree and perform the operation `findUp(s.x)`. This identifies the result of applying the segment sliding query to this horizontal substrip. We return the overall minimum of these values as the final answer.

The query visits a total of $O(\log^2 n)$ nodes ($O(\log n)$ from the primary tree, and each of these spawns an additional $O(\log n)$ from its auxiliary tree.) We can collect all the find-up values and determine their minimum in the same time bound.

Solution 6: In both parts, let us consider what happens during a run that starts with a tree of size n, and ends when the number of active entries falls below $n/2$.

(a) To see that `find` operation takes $O(\log n_a)$, consider any find that takes place during the run. Thus, we have $n/2 \leq n_a \leq n$. The original tree was balanced, so (even with all the inactive nodes clogging things up) any `find` operation takes time $O(\log n)$. Since $n/2 \leq n_a$, it follows that $\lg(n/2) \leq \lg n_a$, and since $\lg(n/2) = (\lg n) - 1$, we have $\lg n \leq (\lg n_a) + 1$. Therefore, a running time of $O(\log n)$ is also $O(\log n_a)$, as desired.

(b) We assert that the amortized running time of `delete` is $O(\log n)$. To see this, consider the run. The actual running time of each `delete` and each `find` is $O(\log n)$. Initially, all the entries are active, and to reduce the number of active entries below $n/2$, we need to perform at least $n/2$ deletions. If we assess a charge of $t = O(\log n)$ tokens for each operation, we use half of this to pay for the actual cost of the operation, and bank the other half. Thus, we bank a total of at least $t(n/2) = O((n/2)\log n) = O(n \log n)$ tokens during the run. Clearly, we have accumulated enough tokens to pay the $O(n)$ cost to rebuild the tree.

Note that we had quite a bit of excess "slack" in our analysis. We would have collected enough tokens if we had charged just one token for every `delete` operation. Unfortunately, we cannot assert that the amortized running time is $O(1)$, because the amortized time can never be smaller than the actual time. (After all, it is the average of the actual costs.) The actual time for each `delete` and each `find` is $O(\log n)$.