CMSC 420: Fall 2022

Solutions to Homework 3: Geometric Search and Hashing

Solution 1:

(a) To insert $(30,30)$, we trace the search path. We lie on the low side of the x-splitter $(40,60)$, on the low side of the y-splitter $(20,45)$, and on the high side of the x-splitter $(10,20)$. Here we fall out of the tree and create a new leaf node to store $(30,30)$. Since the parent is an x-splitter, this is a y-splitter (see Fig. 1).

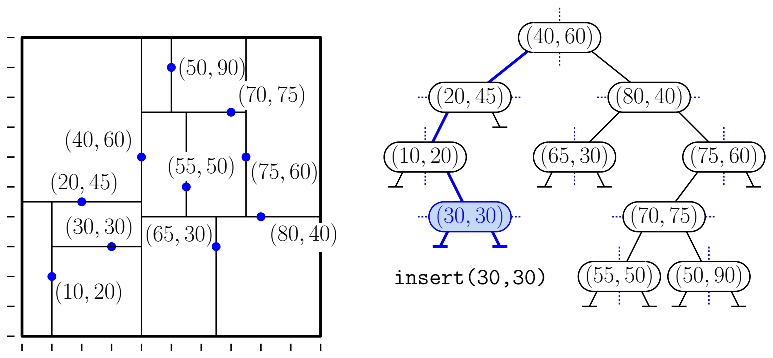

Figure 1: Inserting $(30,30)$.

(b) To delete $(40,60)$, we first find this node, which is easy since it is at the root.

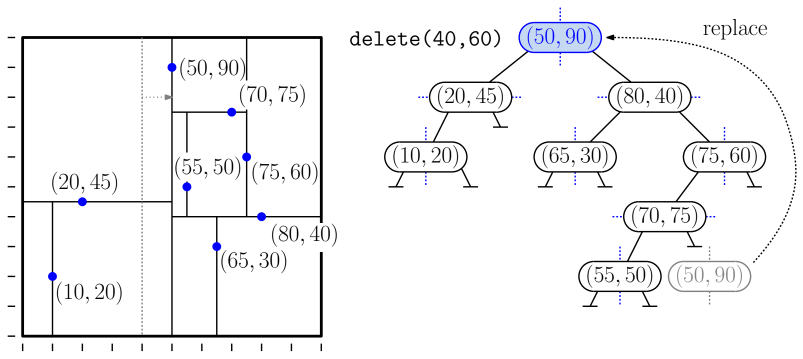

Figure 2: kd-tree operations.

We then need to find the replacement point. Because the root is an x-splitter, by convention the replacement is the point with the smallest x-coordinate in the right subtree. (The alternative choice would be the point with the largest x-coordinate in the left subtree.) To find this point, recurse into the right subtree. Whenever we visit an x-splitter, we recurse into its left child. Whenever we visit a y-splitter, we recurse into both of its children. So, we visit both of the children of $(80,40)$. When we visit $(65,30)$, we just return this point. When we visit $(75,60)$, we recurse into its left child. When we visit $(70,75)$, we visit both of its children. The right child yields the point $(50, 90)$, which has the smaller x-coordinate. So we return this up the tree, and since it has the smallest x-coordinate, it provides final replacement point. We copy the point $(50, 90)$ to the root node. (Note that even if $(50, 90)$ was a y-splitter, which it is not, we would not change the root's cutting dimension.)

Solution 2: Let N denote the set of null-pointer cells of the tree that intersect R. Let us classify these nodes into two groups. Let N' be those that partially overlap R (see Fig. 3(a)), and let N'' be those that are totally contained within R (see Fig. 3(b)). Since $|N| = |N'| + |N''|$, it suffices to prove (1) $|N'| = O(\sqrt{n})$ and (2) $|N''| = O(k)$.

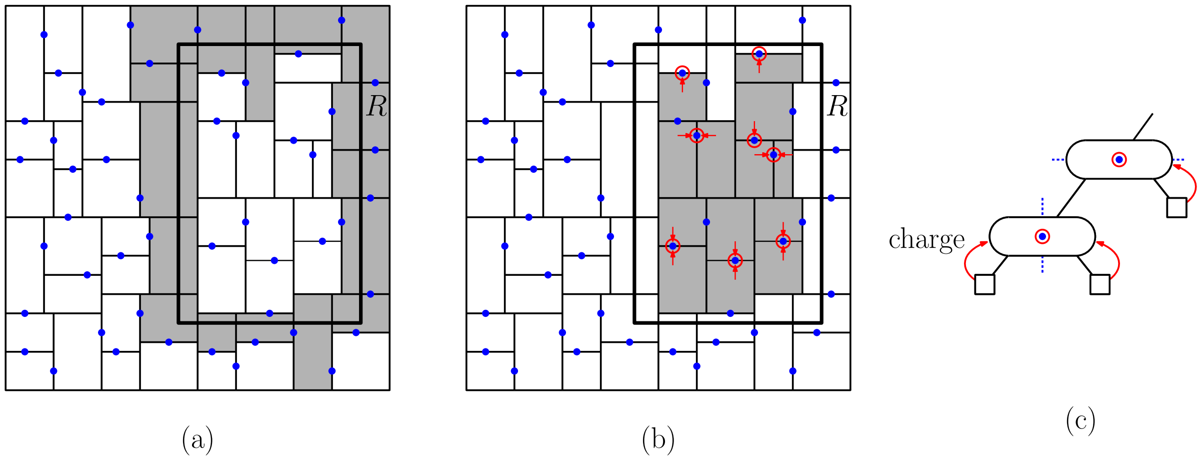

Figure 3: Counting null-pointer cells.

To show (1), observe that by Lemma A, the number of null-pointer cells that intersect each of the four (infinite) lines that make up R's boundary intersect $O(\sqrt{n})$ cells. Thus, we have $|N'| \leq 4 \cdot O(\sqrt{n}) = O(\sqrt{n})$.

To show (2), we use a charging argument. We will "charge" each null-pointer cell to one of the k points lying within R. Observe that if a cell lies entirely within R, then all four of its sides lie completely within R. For any cell, one of these four sides is the splitter for its parent node, and therefore the null-pointer's parent node lies within R. We charge this null-pointer cell to this point of $P \cap R$. Observe that any point of the k points of $P \cap R$ can be charged by at most twice, one from its left child and one from its right (see the red arrows in Fig. 3(b) and (c)). Some points of $P \cap R$ may not be charged at all, but this only makes the total number of charges smaller. Since each null-pointer cell makes a charge, the total number of charges is $|N''|$. Since each of the k points of $P \cap R$ receives at most two charges, we have $|N''| \leq 2|P \cap R| = 2k$. Thus, $|N''| = O(k)$, as desired.

Solution 3:

(a) This problem follows the standard form we have seen in other kd-tree range searching problems. Computing the answer is equivalent to computing the point of P whose y-coordinate lies in the interval [`seg.ylo`, `seg.yhi`] and whose x-coordinate is the largest that is less than or equal to `seg.x`. Define the *strip* to be the semi-infinite rectangle lying to the left of the segment. We first define two utilities. The first determines whether a point is in the strip, and the second determines whether a node's cell is disjoint from the strip.

```
boolean inStrip(Point pt, VertSeg seg)
  { return pt.x <= seg.x && pt.y >= seg.ylo && pt.y <= q.yhi }

boolean isDisjoint(Rectangle cell, VertSeg seg)
  { return cell.low.x > seg.x || cell.high.y < seg.ylo || cell.low.y > seg.yhi }
```

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

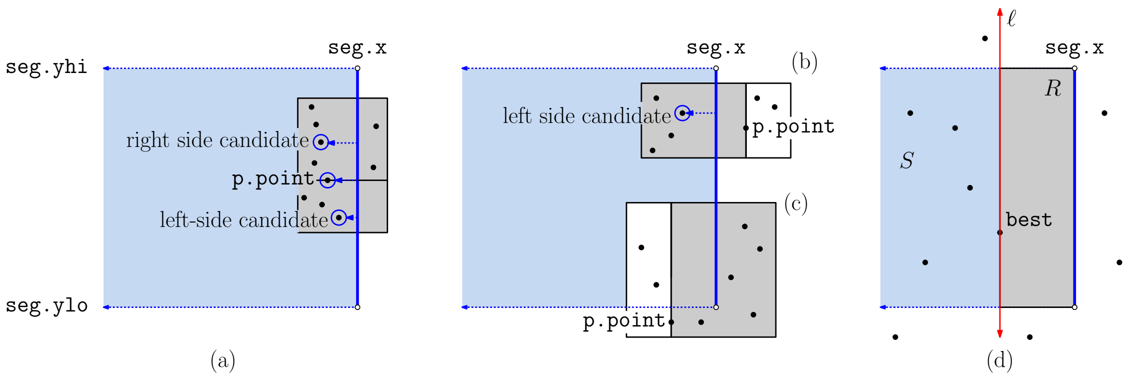

Figure 4: Segment sliding queries.

When we visit a node `p`, we first check the trivial cases, of whether `p` is `null` or its cell is disjoint from the strip. If so, we simply return the current `best`. Otherwise, we test whether `p`'s point is in the strip and to the right of `best`, and if so we make it the new best point. Next, we compute the cells of the node's children. There are two cases to consider. First, if the node is a horizontal splitter (`cutDim == 1`) (see Fig. 4(a)), we recurse on both children, since either may hold the answer.

If the node is an vertical splitter (`cutDim == 0`) we first try the right child, since it is more likely to hold the answer. (You might wonder why we didn't first check whether the right cell overlaps the strip, to avoid the situation shown in Fig. 4(b). We don't need to to since the call will return immediately when we do the cell-strip disjointness test.) Note that the recursion on the right side might not provide a viable candidate (see Fig. 4(c)), so if `best` lies to the left of `p.point`, we need to consider the left child. The code is presented below.

```
Point slideLeft(VertSeg seg, KDNode p, Rectangle cell, Point best) {
  if (p == null || isDisjoint(cell, seg)) return best   // trivial cases
                                                        // this point better?
  if (inStrip(p.point, seg) && (best == null || p.point.x > best.x))
      best = p.point

  Rectangle leftCell = cell.leftPart(p.cutDim, p.point) // child cells
```

```
        Rectangle rightCell = cell.rightPart(p.cutDim, p.point)
        if (cutDim == 1) {                                  // horizontal cut?
            best = slideLeft(seg, p.left, leftCell, best)    // try both sides
            best = slideLeft(seg, p.right, rightCell, best)
        } else {                                            // vertical cut?
            best = slideLeft(seg, p.right, rightCell, best)  // try right first
            if (best == null || p.point.x > best.x)          // is left relevant?
                best = slideLeft(seg, p.left, leftCell, best) // try left
        }
        return best;
    }
```

(b) We claim that the algorithm runs in $O(\sqrt{n})$ time. To establish this, let S denote the semi-infinite strip lying to the left of the vertical segment. Since every node is associated with a cell, let's just use "cell" when referring to both. Observe that if a cell is disjoint from S, then we return immediately from this call, so we only need to account for nodes whose cells either partially overlap S or are completely contained in S. From Lemma A, we know that the number of cells that partially overlap S is $O(\sqrt{n})$.

Finally, let's consider those cells that lie entirely within S. Consider the cells of this group that overlap the vertical line ℓ passing through the final best point. By applying Lemma B to the rectangle R whose right side is the segment and whose left side is ℓ (see Fig. 4(c)), there are $O(\sqrt{n}$ such cells. For each of these cells, if it is a vertical splitter, we will recurse only its right child (which overlaps ℓ). If it is a horizontal splitter, we will recurse on both of its children (but they both overlap ℓ). Since there are $O(\sqrt{n})$ nodes whose cells overlap ℓ in total, it follows that the total number of cells visited that lie entirely within S is $O(\sqrt{n})$, and this completes the analysis.

Solution 4:

(a) The answer is based on a 2-dimensional range tree. This is just the standard tree, where the first-level tree contains the points sorted by their x-coordinates and the second-level auxiliary trees are sorted by their y-coordinates. The added wrinkle is, each node of the tree stores the minimum cost of any point in the subtree rooted at this node. That is, consider any node u, and let $S(u)$ denote the points lying in u's subtree. Define `u.minCost` to be the minimum cost value c_i for all points $p_i \in S(u)$. It is an easy matter to compute these values while we are constructing the tree, by propagating this information upwards from the leaves. (We only need to do this for the auxiliary trees.)

(b) We answer a query as follows. Let q be the user's location and let w and h denote the width and height of the query region. We create the query rectangle $R(q, w, h)$ centered at q with width w and height h. (The lower-left corner is $(q_x - w/2, q_y - h/2)$ and the upper-right corner is $(q_x + w/2, q_y + h/2)$.) We then apply the standard query algorithm for range trees. We first visit the primary tree, and identify all the nodes u corresponding to the maximal subtrees lying in the interval $[q_x - w/2, q_x + w/2]$. For each such u, we then access its auxiliary tree and repeat the above process, but now for the interval $[q_y - h/2, q_y + h/2]$. This yields a collection of nodes v in the auxiliary tree whose x- and y-coordinates both lie within $R(q, w, h)$. We return the value of `v.minCost` for each of these nodes, and 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 min-cost values and determine their minimum in the same time bound.

Solution 5: See Fig. 5.

(a) Linear Probing: Remember that the each access to the table is counted as a probe (including finding the first empty entry), so the probe counts are 4, 6, and 4, respectively.

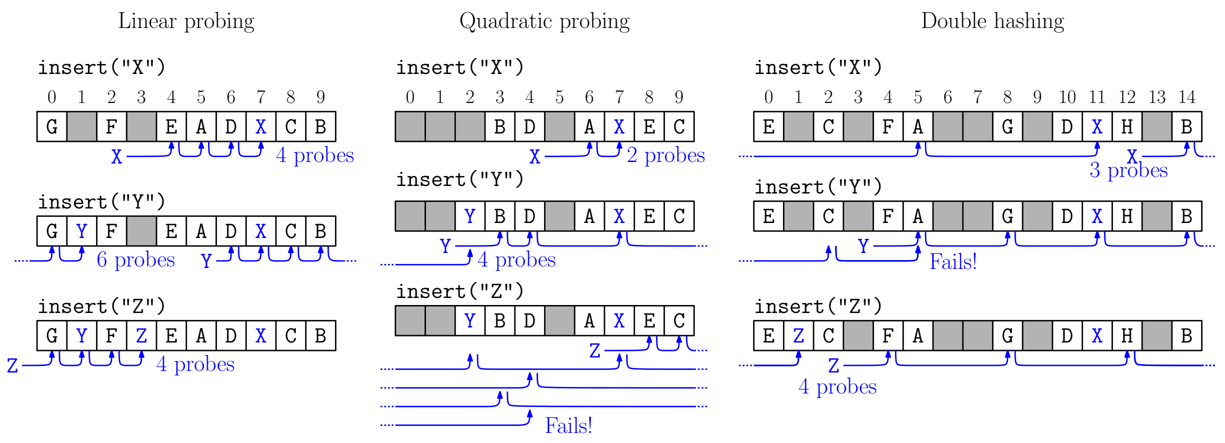

Figure 5: Hashing with linear and quadratic probing.

(b) Quadratic probing: Note that the quadratic offsets are taken with respect to the *initial* hash position. For example, for `Z`, we access locations

$$(\texttt{h(Z)} + \{0,1,4,9,\ldots\}) \bmod 10 \;=\; (8 + \{0,1,4,9,\ldots\}) \bmod 10 \;=\; \{8,9,2,7,\ldots\}.$$

For `X`, we succeed after two probes (6, 7), and for `Y`, we succeed after 4 probes (3, 4, 7, 2). For `Z`, we go into an infinite loop. This follows from the fact that whenever you square a number, its value mod 10 is in the set $\{0,1,4,5,6,9\}$, the so-called *quadratic residues* of 10. Since we started at 8, this means that the only cells we can possibly probe are $(8 + \{0,1,4,5,6,9\}) \bmod 10 = \{8,9,2,3,4,7\}$. Since all these entries are already occupied, we cannot insert `Z`.

(c) For `X`, we succeed after three probes (14, 5, 11). For `Y`, we fail because all the eligible entries (5, 8, 11, 14, 2) are already full. For `Z`, we succeed after 4 probes (4, 8, 12, 1).

Solution to the Challenge Problem:

(a) To determine the number of null-pointer nodes of a 3-dimensional kd-tree that intersect an axis-orthogonal plane, consider any three consecutive levels of the tree, splitting along x then y then z. Suppose that the plane is orthogonal to the y-axis, for example, it has the equation

$y = y_0$. This plane may intersect both of the cells at the x- and z- nodes, but it will only intersect one of the cells of the y-node, depending on whether y_0 is greater than or less than the splitting value. Therefore, of the 8 great-grandchildren of the node, we can intersect at most 4 (two for x, and each of these may split to form two for z). There is nothing special about y. The cases for the other axes are exactly analogous.

Letting n denote the number of points in each node, by our balance assumption each great-grandchild has $n/8$ points. Therefore, we obtain the following recurrence on the number of null-pointer nodes that intersect the hyperplane:

$$T(n) \;=\; \begin{cases} 1 & \text{if } n = 1 \\ 4T(n/8) & \text{otherwise.} \end{cases}$$

(This is a bit of a simplification. Since each node consumes a point, the recurrence is really $4T((n-1)/8)$ and the basis case is at $n = 0$, but for the purposes of an asymptotic analysis, the above is close enough.) We can solve this using the Master Theorem for recurrences. This provides a solution for recurrencs of the form $T(n) = aT(n/b) + f(n)$. In our case, $a = 4$, $b = 8$, and $f(n) = 0$. Since $0 \leq \log_b a$, the Master Theorem states that this solves asymptotically to $T(n) = O(n^{\log_b a}) = O(n^{\log_8 4}) = O(n^{2/3})$. This is more than the planar case, where we had $O(\sqrt{n})$, but it is still sublinear.

(b) The case for the line is similar, but the parameter values are different. Suppose that the line is parallel to the y-axis. In this case, we will intersect only one of the two children of the x-node and one of the two children of the z-node, but we'll intersect both of the children of the y-node. This yields the recurrence

$$T(n) \;=\; \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/8) & \text{otherwise.} \end{cases}$$

Again, we apply the Master Theorem with $a = 2$, $b = 8$ and $f(n) = 0$, which reveals that $T(n) = O(n^{\log_b a}) = O(n^{\log_8 2}) = O(n^{1/3})$.