Solutions to Homework 3: Hashing, Geometry, Tries

Solution 1:


![Figure 1: Hashing with linear probing.](image)

Insertion and hash calculations:

- $insert("Z")$ \quad h("Z") = 11
- $insert("Y")$ \quad h("Y") = 1
- $insert("X")$ \quad h("X") = 9

(1.2) See Fig. 2. M: 1 probe, inserted at $T[4]$. D: 3 probes, inserted at $T[12]$. Q: Failure! (Infinite probe sequence. This follows from the fact that the quadratic residues of 16 are 1, 4, and 9, so the only entries probed are at the indices 3, 3 + 1, 3 + 4 and 3 + 9 mod 16, and these entries are already occupied).

![Figure 2: Hashing with quadratic probing.](image)

Insertion and hash calculations:

- $insert("M")$ \quad h("M") = 4
- $insert("D")$ \quad h("D") = 8
- $insert("Q")$ \quad h("Q") = 3

(1.3) See Fig. 3. Q: 2 probes, inserted at $T[10]$. T: Failure! Since $g("T") = 4$, the only entries probed are 0, 4, 8, and 12, and these entries are already occupied. D: 15 probes, inserted at $T[1]$.

![Figure 3: Hashing with double hashing.](image)

Insertion and hash calculations:

- $insert("R")$ \quad h("R") = 7; $g("R") = 3$
- $insert("T")$ \quad h("T") = 0; $g("T") = 4$
- $insert("D")$ \quad h("D") = 8; $g("D") = 7$
Solution 2:

(2.1) We give code for a helper function `findUp(float x, KDNode p)`, and the initial call is `findUp(x, root)`. As a basis case, if `p == null`, we return `Float.MAX_VALUE`. Otherwise, let `pX = p.point[0]` be the x-coordinate of the point stored in this node.

If the current node is a y-splitter `(p.cutDim == 1)`, we invoke the function on both children, and let `minX` be the minimum of these two answers. If `px >= x0` return the minimum of `minX` and `pX`, and otherwise return `minX`.

Otherwise, this node is an x-splitter. If `pX >= x0` we invoke the procedure on the left subtree, and we return the minimum of that result and `px`. Otherwise, we invoke the procedure on the right subtree and return whatever result it generates. The code is presented in the code block below.

```
float findUp(float x, KDNode p) {
    if (p == null) return Float.MAX_VALUE // empty subtree
    else {
        float pX = p.point[0] // the x-coordinate of this point
        if (cutDim == 1) { // y-splitter - search both sides
            float minX = min(findUp(x, p.left), findUp(x, p.right))
            if (pX >= x0) return min(minX, pX) // pX is eligible?
                else return minX
        } else { // x-splitter
            if (pX >= x0) // px is eligible?
                return min(findUp(x, p.left), px) // min of left subtree and px
            else
                return findUp(x, p.right) // search the right subtree
        }
    }
}
```

(2.2) The derivation of the running time is similar to that for orthogonal range searching. Whenever we visit a y-splitter, we recurse on both children. Whenever we visit an x-splitter, we recurse on one of the two children. Thus, for every two levels of descent in the three, we visit two out of four grandchildren. Since we assume the tree is perfectly balanced, if the tree has a total of `n` nodes, then each grandchild has roughly `n/4` nodes. This leads to the following recurrence:

\[
T(n) = \begin{cases} 
1 + 2T(n/4) & \text{if } n \geq 2, \\
1 & \text{if } n = 1.
\end{cases}
\]

As shown in class, we can apply the Master Theorem for recurrences (from CLRS). We have the case \(T(n) = aT(n/b) + n^c\), where \(a = 2\), \(b = 4\), and \(c = 0\). In this case \(c \leq \log_b a\), and so the Master Theorem yields \(T(n) = O(\sqrt{n})\).

Solution 3: We distinguish two different senses in which nodes may be visited. (And we will give full credit for either answer.) A node is weakly visited if a recursive call is made involving this node. A node is strongly visited if it is weakly visited and it makes recursive calls on its children.
A cell is strongly visited if its cell stabs the query range (that is, it overlaps the query range but is not fully contained within the query range). The weakly and strongly visited sets of nodes are shown in Fig. 4. For example, node $a$ is strongly visited because the range stabs its cell, and so it invokes the search function recursively on both of its children, implying that both $b$ and $c$ are visited. Note that $b$’s cell does not overlap the range, and hence it makes no further recursive calls. Thus, $b$ is weakly visited.

![Strongly visited: \{a, c, f, g, m, n, s\}]  
Weakly visited: \{a, b, c, f, g, l, m, n, o, s\}

**Solution 4:**

(4.1) Our solution will make use of the operation $\text{findUp}$, which was described in the second programming assignment. It runs in time $O(\log n)$, assuming that the tree is balanced.

The range tree is constructed from the original point set $P$. Given a 3-sided query $(x_0, x_1, y_0)$, apply the standard range-search algorithm in the $x$-range tree to determine a collection of $O(\log n)$ nodes, such that all the points whose $x$-coordinates lie within the interval $[x_0, x_1]$ lie within one of these subtrees. For each auxiliary tree associated with each of these, apply the query $\text{findUp}(y_0)$, to obtain the point with the smallest $y$-coordinate that is greater than or equal to $y_0$ within this subtree. If all of these queries return null, then return null. Otherwise, among those that are non-null, return the point having the smallest $y$-coordinate.

We know from earlier assignments this semester, $\text{findUp}$ queries can be answered in $O(\log n)$ time from a tree of size $n$. Since each auxiliary tree has size at most $n$, and there are $O(\log n)$ auxiliary trees, the overall query time is the product of these, which is $O(\log^2 n)$.

(4.2) Consider the transformation $T : (x, y) \rightarrow (x, y - x)$. This transformation maps vertical lines to vertical lines (because it does not alter $x$-coordinates). It also maps lines of slope +1 to horizontal lines. (To see this, suppose that we have two points $(x, y)$ and $(x', y')$ that both lie on a line of slope +1. This means that $y' - y = x' - x$, or equivalently $y - x = y' - x'$. After applying our transformation, the new coordinates are $(x, y - x)$ and $(x', y' - x')$. These two points have the same $y$-coordinates, and hence they lie on a horizontal line.) It follows that the transformation $T$ maps skewed rectangles to axis-aligned rectangles.

Rather than building the range tree on the original point set, we apply the transformation $T$ to each point of $P$, resulting in a modified set of points, $T(P)$. We create a standard...
orthogonal range search tree for the points of \( T(P) \). Then, given a skewed query rectangle
query with corners \( q^- \) and \( q^+ \), we apply the transformation, resulting in an axis-aligned
rectangle with corner points \( T(q^-) \) and \( T(q^+) \). We apply our range tree on the transformed
query, and return the associated count. By the observations made above, a point of \( P \) lies in
the skewed rectangle if and only if the transformed point lies in the transformed rectangle.
Therefore, the resulting count is correct.

Given a set \( P \) of \( n \) points, we can compute the transformed set \( T(P) \) in \( O(n) \) time, and we
can then build the range tree in time and space \( O(n \log n) \). Given a query, we transform it
into axis-aligned form in constant time, and then the query can be answered (by the standard
range-tree algorithm) in \( O(\log^2 n) \) time.

Solution 5: The substring identifiers and final suffix tree are shown in Fig. 5.

![Suffix Tree](image)

Figure 5: Suffix tree construction.

Solution to the Challenge Problem:: We define a transformation \( T \) that maps each point
\((x, y)\) to \((x, y, x + y)\) in \( \mathbb{R}^3 \). Given a query \((q, \ell)\), create the 3-dimensional range whose lower-left
corner is \((q_x, q_y, -\infty)\) and whose upper-right corner is \((+\infty, +\infty, q_x + q_y + \ell)\).

We assert that a point \( p = (px, py) \) lies within the query triangle if and only if \( T(p) \) lies within
this range. To see this, observe that a point \((x, y)\) lies within the query triangle if and only if
\( x \geq q_x, y \geq q_y, \) and \( x + y \leq q_x + q_y + \ell \). This means that the transformed point \((x, y, z)\) satisfies
\( x \geq q_x, y \geq q_y, \) and \( z \leq q_x + q_y + \ell \), which implies that it lies within the 3-dimensional range. The
converse applies similarly. (Note that the only points that we need to consider are those produced
by our transformation, and so we may assume that \( z = x + y.\))

Therefore, answering the 2-dimensional NE right-triangle query is equivalent to answering the
given 3-dimensional orthogonal range query on the transformed point set. To answer triangle
queries, we apply the transformation to every point of \( P \), letting \( T(P) \) denote the resulting point
set. Then, given a right-triangle query \((q, \ell)\), we generate the associated query rectangle in 3-
dimensional space, and return the resulting answer.

Since the orthogonal range query is computed in 3-dimensional space, it follows that the storage
is \( O(n \log^2 n) \) and the query time is \( O(\log^3 n) \).