Solutions to the Midterm-2 Practice Problems

Solution 1:

(a) Sorted array and balanced binary search tree. (Unbalanced trees do not provide good worst-case performance, splay trees only guarantee good performance over a sequence of operations, and hashing scatters keys about so finding nearby keys is not easy.)

(b) If the second hash function and table size share a common factor, then the probe sequence may not visit every entry of the table. (For example, if both are even, then the probe sequence will hit only even cells or only odd cells, depending on whether \( h(x) \) is even or odd, respectively.) Thus, you may insert a key into a table that is not full, but still not find a slot for it.

(c) This was done to facilitate range reporting queries, where you want to list all the keys in some range \([x_{\text{low}}, x_{\text{high}}]\). You first locate the smallest key at the leaf level that is greater than or equal to \( x_{\text{low}} \) and then keep following these leaf links until you first encounter a key that is larger than \( x_{\text{high}} \).

(d) By the algorithm given in the lecture notes, there are \( k \) multiplications and \( k \) additions. If you are more efficient, however, you can avoid the first multiplication and addition (since they produce the value zero) for \( k - 1 \) multiplications and \( k - 1 \) additions. (Either answer is acceptable.)

(e) Secondary clustering is clustering in the hash table that occurs because the collision resolution method fails to disperse keys effectively. Linear probing is most susceptible (since the keys are not well dispersed), and double hashing is least susceptible (because it disperses the keys by nearly random offsets).

(f) 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

\[
\sum_{i=0}^{h} 8^i = \frac{8^{h+1} - 1}{8 - 1} = \frac{8^{h+1} - 1}{7}.
\]

Solution 2: The helper function `makeSum(BinaryNode p)` is given below. We recursively compute the sum of the left and right subtrees, stores the value in the node’s key, and then returns this value. The initial call is `makeSum(root)`. The algorithm spends \( O(1) \) time at each node of the tree, so the total running time is \( O(n) \) for an \( n \)-node tree.

```cpp
Key makeSum(BinaryNode p) {
    if (p == null)
        return 0;
    else {
        // Code for recursive computation of the sum
        // of left and right subtrees
    }
}
```
p.key += makeSum(p.left) + makeSum(p.right);
return p.key;
}
}

**Solution 3:** The helper function `kthSmallest(BinaryNode p, int k)` computes the `k`th smallest key in the subtree rooted at `p`. The initial call is `kthSmallest(root, k)`. (We assume that `1 ≤ k ≤ T.size()`) We first define a short utility function `size(p)`, which returns the size the subtree rooted at node `p` or zero if `p` is null. There are three cases, depending on `size(p.left)`:

- If `size(p.left) = k - 1`, then `p.key` is the `k`th smallest, and we return it.
- If `size(p.left) > k - 1`, then this subtree contains the `k`th smallest key. We solve the problem recursively by calling `kthSmallest(p.left, k)`.
- Finally, if `size(p.left) < k - 1`, then the `k`th smallest key is located in its right subtree. To compensate for the fact that we have skipped over `1 + size(p)` keys in `p` and its left subtree, the recursive call is `kthSmallest(p.right, k - (1 + size(p.left)))`.

The code is presented below. Note that if `k` is in bounds, we should never invoke this function on an empty subtree, so we won’t bother checking the case where `p` is null. Since we make one recursive call to a node’s child, the running time is proportional to the tree’s height.

```java
int size(BinaryNode p) {
    return (p == null ? 0 : p.size);
}

Key kthSmallest(BinaryNode p) {
    if (size(p.left) == k-1)
        return p.key;
    else if (size(p.left) > k-1)
        return kthSmallest(p.left, k);
    else
        return kthSmallest(p.right, k - (1 + size(p.left)));
}
```

**Solution 4:** We will give just a high-level explanation of the “hole-free” method of deletion for hashing with linear probing. The insertion process works just as for standard linear-probing hashing. When we wish to delete a key `x`, we first compute the hash value `h(x)` and then start searching the probe sequence `h(x), h(x) + 1, h(x) + 2`,... until we find the key to be deleted. (If we don’t find it, we throw an exception.) Suppose that the cell holding the deleted key is at `table[i]`. We set `table[i] = empty` to indicate that the key is deleted, but this leaves a hole in our table, which needs to be filled.

To fill the hole, we consider the subsequent probe sequence `i + 1, i + 2`,... (wrapping around if necessary) until we first find an empty cell, say at index `j`. (To simplify the presentation, let’s assume that we didn’t need to wrap around, implying that `j > i`.) An overly simple idea is to shift each entry in the subarray `table[i+1 ... j]` back one position to fill in the hole. This moves the hole to `table[j]`, but that is okay because we are beyond the end of the probe sequence. There
is a problem, however. If some key is located at its ideal location (that is, if $h(table[k]) == k$) then we do not want to move it, or else we will never find it again. So, whenever we encounter such a node, we skip over it and try the next one. To do this, we will maintain two “fingers” in the list. One indicating the location of hole to be filled ($p$) and the other indicating the location of the element to fill it ($q$).

The following is a simplified version of this hole-filling code. (This code is not complete, because it doesn’t consider the possibility of falling off the end of the table and wrapping around.)

```cpp
// let table[i] be the location of the deleted key
table[i] = empty; // delete the key from the table
int p = i; // index of the hole
int q = i+1; // index of the item to fill the hole
while (table[q] != empty) { // walk the probe sequence to its end
    while (table[q] != empty && h(table[q]) == q) { // can’t move this key
        q++; // ...try the next
    }
    table[p++] = table[q++]; // slide key forward to fill the hole
}
```

Solution 5: 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. 1). 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).

![Figure 1: A line that stabs all the cells of a kd-tree.](image)

Solution 6: Computing the Pareto predecessor of a point $q$ is equivalent to computing the point in the kd-tree with the maximum $x$-coordinate lying within $q$’s northwest quadrant, that is, the semi-infinite axis-parallel rectangle range whose lower-right corner is $q$. This is similar to an orthogonal range query, but we are interested in just one point of the range, the one with the largest $x$-coordinate.

To avoid dealing with special cases, let’s make the simplifying assumption that no point in the kd-tree has the same $x$- or $y$-coordinate as $q$. 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. Since we do not store a node’s cell with each node, we pass the cell into the function.
as the parameter cell. (Recall that the tree is associated with a bounding box cell that includes all the points of the kd-tree, and this is the root’s cell.) We also maintain a point best, which among all candidates seen for the Pareto predecessor, is the one with the largest $x$-coordinate. The initial call at the root level is \texttt{paretoPred(q, root, boundingBox, initialBest)}, where initialBest can be taken to be a sentinel point with the $x$-coordinate $-\infty$.

Because this takes place in the plane, we’ll use the more intuitive notation $q_x$ and $q_y$ to refer to a point’s coordinates, rather than the more proper $q[0]$ and $q[1]$.

When we visit a node $p$, we consider the relationship of $p$’s cell to $q$’s northwest quadrant. If $p$ is null or if its cell does not overlap the quadrant ($cell.low.x > q_x$ or $cell.high.y < q.y$) we may ignore this node and its contents, since it cannot possibly provide a valid Pareto predecessor. To do this, we just return best.

Otherwise, we first consider whether the point stored in this node offers a better choice (that is, $p.point$ lies within the quadrant and has a larger $x$-coordinate than best). If so, we update best. Finally, we recurse on $p$’s two children, and keep the point with the larger $x$-coordinate. We use the utility functions leftPart and rightPart to compute the cells associated with the left and right children.

```java
Point paretoPred(Point q, KDNode p, Rectangle cell, Point best) {
    if (p == null) // fell out of tree?
        return best;
    else if (cell.low.x > q.x || cell.high.y < q.y) // no overlap
        return best;
    else {
        if (p.point.x <= q.x && p.point.y >= q.y) // p's point is in quadrant?
            if (p.point.x > best.x) best = p.point; // p's point is better?
        // get children cells
        Rectangle leftCell = cell.leftPart(p.cutDim, p.point);
        Rectangle rightCell = cell.rightPart(p.cutDim, p.point);

        best = paretoPred(q, p.left, leftCell, best); // search left subtree
        best = paretoPred(q, p.right, rightCell, best); // search right subtree
        return best;
    }
}
```

There are a couple of further refinements we could make to the above algorithm to improve its efficiency. First, if the cell’s cutting dimension is $x$ (vertical), we should recurse on the right child before the left child, since it is more likely to yield a point with a higher $x$-coordinate. Second, if cell lies entirely to the left of best (that is, $cell.high.x < best.x$), there is no need to visit this cell, since any point it can provide will be worse than the current Pareto candidate.