Solutions to Midterm Exam 2

Solution 1:

(1.1) (d): The table is less than half full ($\lambda < \frac{1}{2}$), and the table size $m$ is a prime number.

(Justification: If $m$ is prime, the first $\lfloor m/2 \rfloor$ probes are distinct, and if the table is less than half full, the probe sequence will find an empty slot.)

(1.2) (a) High utilization $\Rightarrow$ (1) $\lambda_{\text{min}}$ should not be too low; (b) Fast search times $\Rightarrow$ (4) $\lambda_{\text{max}}$ should not be too high; (c) Rebuild not too often (5) $\lambda_{\text{max}} - \lambda_{\text{min}}$ should not be too low.

(1.3) 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.

(1.4) 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.

Solution 2: The results of insert(15) and delete(25) are shown in the figure below. In the first case, a single key rotation suffices to correct the imbalance. In the second case, two merges are required.

Solution 3:

(3.1) The size of a parent is strictly larger than the size of either of its children, so

$$\frac{\text{size}(\text{u.child})}{\text{size}(\text{u})} < 1,$$

which implies that $\alpha < 1$. We also claim that we cannot have $\alpha < 1/2$. Suppose we did. If $n$ were even, then excluding the root, one child must at least $\lceil (n-1)/2 \rceil$ of the remaining nodes. Since $n-1$ is odd, we have $\lceil (n-1)/2 \rceil = n/2 > \alpha n$, implying that this subtree would violate the size condition, a contradiction.
(3.2) If every node satisfies condition (\(\ast\)), then the height of the tree would be at most \(\log_{1/\alpha} n\).

(3.3) Let \(h\) denote the height of the tree. Consider any path in the tree from the root to a node of depth \(h\). The root has size \(n\). By the balance condition (\(\ast\)), the child of this node along this path has size at most \(\alpha n\), the grandchild along the path has size at most \(\alpha^2 n\), and generally a node at level \(h\) has size at most \(\alpha^h n\). This leaf node defines a subtree with 1 node. Therefore, we have \(\alpha^h n \geq 1\). Equivalently, we have \((1/\alpha)^h \leq n\), and taking logs base \(1/\alpha\) on both sides, we have \(h \leq \log_{(1/\alpha)} n\).

Solution 4: Let us assume the general case where we want to construct a tree for the subarray \(A[i..j-1]\). There are two major differences with respect to the standard rebuilding process. First, the splitter is chosen as the mean element, that is, \(s = (A[i] + A[j-1])/2\). Second, rather than splitting at the median, we find the smallest element \(A[m] > s\), and partition the array as \(A[i..m-1]\) (for the left subtree) and \(A[m..j-1]\) (for the right subtree).

The construction is presented in the code block below. The initial call is \(buildSubtree(A, 0, n)\). We assume we have a helper function \(search(s, A, i, j)\) that searches the sorted subarray \(A[i...j-1]\) for the smallest index \(m (i \leq m < j)\) such that \(A[m] > s\). Since \(A\) is sorted, this can be done in \(O(\log k)\) time by binary search (but \(O(k)\) is good enough for full credit).

```java
def buildSubtree(Key A[], int i, int j)
    if (j-i == 1) // one key?
        return new ExternalNode(A[i]); // ... external node
    else { // subdivide at upper median
        Key s = (A[i] + A[j-1])/2; // splitter is midpoint
        int m = search(s, A, i, j); // A[i..m-1] <= s < A[m..j-1]
        Node left = buildTree(list, i, m); // recursively build subtrees
        Node right = buildTree(list, m, j);
        return new InternalNode(s, left, right);
    }

def search(Key splitter, Key A[], int i, int j)
    for (int m = i; m < j; m++)
        if (A[m] > splitter) return m;
    return j;
```

Solution 5: We apply the standard approach for answering range searching queries. We visit nodes of the kdtree 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 \(cell\). We maintain a point \(best\), which among all candidates seen so far, has the largest \(x\)-coordinate. The recursive function is given in the code block below.

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 \(y\) (horizontal), we should recurse on the right child before the left child, since it is more likely to yield a point with a higher \(y\)-coordinate. Second, if
Partial-range maximum query

Point partialMax(float x1, float x2, KDNode p, Rectangle cell, Point best) {
    if (p == null) // fell out of tree?
        return best;
    else if (cell.low.x > x2 || cell.high.x < x1) // no overlap with strip
        return best;
    else {
        if (p.point.x >= x1 && p.point.x <= x2) // p.point in the strip?
            if (p.point.y > best.y) best = p.point; // p.point is better?
                // get children cells
        Rectangle leftCell = cell.leftPart(p.cutDim, p.point);
        Rectangle rightCell = cell.rightPart(p.cutDim, p.point);

        best = partialMax(q, p.left, leftCell, best); // search left subtree
        best = partialMax(q, p.right, rightCell, best); // search right subtree

        return best;
    }
}

Recall that the tree is associated with a bounding box cell that includes all the points of the kd-tree. This is the root’s cell. The initial call at the root level is partialMax(q, root, boundingBox, initialBest), where initialBest can be taken to be a sentinel point with the y-coordinate $-\infty$. cell lies entirely below best (that is, cell.high.y < best.y), there is no need to visit this cell, since any point it can provide will be worse than the current Pareto candidate.