Solutions to Homework 3: Hashing, kd-Trees and More

Solution 1: The solutions are shown in Figs. 1–3. In the case of $\text{insert}(9)$, it has a sibling that can absorb an extra key and so we do a key rotation. In the case of $\text{insert}(2)$, the sibling is full, so we split the node. This causes the parent to split, which results in the creation of a new root. In the case of $\text{delete}(22)$, the sibling node cannot provide a key, so we merge these nodes together.

![Figure 1: Solution to Problem 1.1: $\text{insert}(9)$.](image1)

![Figure 2: Solution to Problem 1.2: $\text{insert}(2)$.](image2)

![Figure 3: Solution to Problem 1.3: $\text{delete}(22)$.](image3)
Solution 2: See Fig. 4.

Solution 3:

(3.1) See Fig. 5. We find the point (80, 10) and fall out of the tree on the right child of (70, 20).

We insert the new node here. Since the parent \(y\)-splitter, the new node is an \(x\)-splitter.
(3.2) See Fig. 6. The deleted node is at the root. Since it is an \( x \)-splitter, its replacement is the node with the smallest \( x \)-coordinate in the root’s right child, which is \((50, 85)\). Since \((50,85)\) is an \( x \)-splitter, its replacement is the node with the smallest \( x \)-coordinate in its right subtree, which is \((60,90)\). Since \((60,90)\) is a \( y \)-splitter and its right subtree is empty, we find the point with smallest \( y \)-coordinate in its left subtree, which is \((90,55)\), and we move this subtree to become the right child of \((60,90)\). Finally, we delete \((90,55)\) from this subtree. Since it is a leaf, it can simply be unlinked from the tree.

![Figure 6: Solution to Problem 3.2: kd-tree deletion.](image)

**Solution 4:** Let us start by presenting a natural solution, which is not quite correct. Then we will fix it. It is easy to see that for all sufficiently large \( n \), it is better to start with a 3-node in the root than a 2-node. This suggests the following simple bottom-up recursive solution. Let \( k \) denote the length of the current subarray.

- If \( k \) is sufficiently small (e.g., \( k \leq 4 \)) build an optimal 2-3 subtree explicitly and return it.

- For all larger values of \( k \), split the array into three roughly equal subarrays as follows: Select two “median” elements at ranks \( m_1 \approx k/3 \) and \( m_2 \approx 2k/3 \). Then let \( A_{\text{left}} \) be the subarray to the left of \( m_1 \), let \( A_{\text{mid}} \) be the subarray between \( m_1 \) and \( m_2 \), and let \( A_{\text{right}} \) be the subarray to the right of \( m_2 \).

- Recursively build the minimum height subtrees for \( A_{\text{left}}, A_{\text{mid}}, \) and \( A_{\text{right}} \), and make them the subtrees of a 3-node whose keys are \( m_1 \) and \( m_2 \).

Sadly, this simple strategy does not work. The reason is that by independently computing minimum-height subtrees for each of the subarrays, one of them may have a lower height than the others, and as a result the leaves of the tree will not all appear at the same level of the final tree.
This violates a fundamental property of 2-3 trees. (To see this, try the case of \( n = 28 \), which results in subtrees of sizes 9, 9, and 8.)

To fix this, we include an additional parameter \( h \) in the `buildSubtree` function that indicates the desired height of subtree being constructed. It is not hard to show that any 2-3 tree containing \( n \) keys requires a height of \( \lfloor \log_3 n \rfloor \). (To see this, consider a 2-3 tree of height \( h \) in which all the nodes are 3-nodes. It is easy to prove that such a tree has \( 3^h - 1 \) keys.) In order that the final tree have height \( h \), we modify the above strategy so that when we make recursive calls to our children, we require that each produces a subtree of height exactly \( h - 1 \), even though this may not be the minimum. To build a tree with \( n \) keys, the initial call to this function will be

\[
\text{buildSubtree}(A, 0, n, \text{floor}(\log_3 n))
\]

This raises the further complication of how to control the height of a 2-3 tree? There are a number of ways to do this. Our approach will be to use a 2-node for the root, whenever it is feasible to artificially increase the height of the tree. How to we know whether a 2-node can be used at the root? Such a root node will contain one key and the remaining \( k - 1 \) keys are evenly split among the two children. Thus, each subtree will store at most \( k' = \lceil \frac{k-1}{2} \rceil \) keys. We know that we can build a 2-3 of height \( \lfloor \log_3 k' \rfloor \), and therefore, we conclude that it is safe to make the root a 2-node if \( \lfloor \log_3 \lceil \frac{k-1}{2} \rfloor \rfloor \leq h - 1 \). We write a little helper function to test this condition.

\[
\text{boolean twoNodeIsOkay(int k, int h) { return floor(log_3 ceil((k-1)/2)) <= h-1; }}
\]

All that remains is to determine how to trisect an array. This is a rather messy exercise in floor-ceiling arithmetic. We want the subarrays to have sizes that are within \( \pm 1 \) of each other. There are a number of ways to do this. Our approach is to set the sizes of the left and middle subarrays to \( k_{\text{left}} = k_{\text{mid}} = \lfloor (k - 1)/3 \rfloor \), and set the size of the right subarray to hold the remaining elements, \( k_{\text{right}} = k - (k_{\text{left}} + k_{\text{mid}} + 2) \). It is easy to see that this results in subarrays that have sizes that are within \( \pm 1 \) of each other. The final pseudo-code is given below.

```
Node23 buildSubtree(Key A[], int i, int k, int h) {
  if (k == 1)
    return new Node23(null, A[i], null); // return a single 2-node
  else if (k == 2)
    return new Node23(null, A[i], null, A[i + 1], null); // a single 3-node
  else if (twoNodeIsOkay(k, h)) { // okay for root to be 2-node?
    int m = (int) Math.floor(k / 2.0); // median rank
    Node23 left = buildSubtree(A, i, m, h - 1); // build left/right subtrees
    Node23 middle = buildSubtree(A, i + m + 1, k - m - 1, h - 1);
    return new Node23(left, A[i + m], middle); // join them together
   }
}
```

Figure 7: Solution to Problem 4.
else {
    int kLeft = (int) Math.floor((k - 1) / 3.0);  // size of left subarray
    int m1 = kLeft;  // left-mid splitter
    int kMid = kLeft;  // size of middle subarray
    int m2 = kLeft + 1 + kMid;  // rank of mid-right splitter
    int kRight = k - (kLeft + kMid + 2);  // rest in the right subtree
    Node23 left = buildSubtree(A, i, kLeft, h-1);  // recursively build subtrees
    Node23 middle = buildSubtree(A, i + m1 + 1, kMid, h-1);
    Node23 right = buildSubtree(A, i + m2 + 1, kRight, h-1);
    return new Node23(left, A[i + m1], middle, A[i + m2], right);  // and join
}