CMSC 420 (0201) - Solutions to the Final Exam

Solution 1:

(a) True: Every non-leaf node has at least one incoming thread, one from each child that is non-null. For example, if the left child of some node `u` is not null, then the rightmost leaf in the subtree `u.left` has `u` as its inorder successor, and so its forward inorder thread pointing to `u`.

(b) (2) and (3): In an *inorder traversal*, internal and external node *alternate* with each other. The first node is the leftmost leaf. (4) is almost true, but fails only in the case where the tree consists of a single external node.

(c) (3) and (4): A replacement node is needed whenever the node containing the deleted key has two non-null children. (Replacements are never needed for leaves and may not be needed for the root if it has only one child.) Selecting the replacement only from the right subtree can lead to less balanced trees over time, and so even though it is a common convention, selecting exclusively from the right subtree is not an optimal strategy. To see (4), observe that a replacement node is either the largest key in the left subtree or the smallest key in the right subtree. Such a node can have at most one child. Hence, once a replacement is performed, the node to be recursively deleted has just a single child and does not need a replacement!

(d) $n - k$: Initially every element is in its own set. As stated in the problem, every time a union occurs, it merges two distinct sets. Each time two sets are merged, we get one fewer set. So after k merges, we have $n - k$ sets.

(e) 4 (but we will give partial credit for values in the range 3–7): A naive argument is that the third largest element cannot reside below depth two, so there are at most 7 nodes to consider. A more refined argument observes that the root can never be the third largest element. It can be either of the children of the root (whichever child is smaller), and it can also be either of the children of the larger child of the root. So, the best you hope to achieve is to inspect the two children of the root, and the two children of the smaller child of the root, for a total of 4 nodes.

(f) Min: 7, Max: 13 (The minimum is a complete binary tree of height 2, and the other is a complete ternary tree of height 2.)

(g) 1: Once a key-rotation (adoption) is performed, the tree structure is restored

(h) L: You may need to perform a skew at every level of the tree.

(i) (2) $1/n$: The smallest key (and indeed any key) is placed at the root when its priority is the smallest among all the keys in treap. Since there are n keys, the probability that any fixed key has the smallest priority is $1/n$ (assuming distinct priorities).

(j) Skew: The skew operation enforces the black left-child constraint. (In contrast, the split operation is used to enforce the condition that if a node is red, then both its children are black.)

(k) (1) $O(1)$: The skiplist analysis shows that the number of hops that are made at any level of the skiplist is expected to be 2, which is $O(1)$.

(l) $n(9/16)$: All n contribute to level 0, $n(3/4)$ are expected to contribute to level 1, and $n(3/4)(3/4) = n(9/16)$ are expected to contribute to level 2.

(m) key-rotation (adoption): Preferred because it does not generate any new nodes and once done, does not propagate.

Solution 2: In both parts, we employ a utility function `size`, which returns the size of a node `p`, returning the value zero if `p` is `null`.

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

(a) The rotate code is the same as usual, but we update the sizes of `p` and `q` as the sum of their left and right children plus one (for the node itself). Clearly, this runs in $O(1)$ time.

```
Node rotateLeft(Node p) {
    Node q = p.right                              // do the rotation
    p.right = q.left
    q.left = p
    p.size = size(p.left) + size(p.right) + 1     // update p's size
    q.size = size(q.left) + size(q.right) + 1     // update q's size
    return q
}
```

(b) We use a helper, `Key getKth(int k, Node p)`, which returns the value of the kth smallest key in the subtree rooted at `p`. The initial call is `getKth(k, root)`. If the size of the current node's left subtree is at least k, we know that the smallest k nodes lie in the left subtree, and so we recurse on it. If the size of the left subtree is $k-1$, the kth smallest key is stored in `p`'s node, and we return its value. Otherwise, we know that the kth smallest key is in `p`'s right subtree. We recurse on this subtree, but first we decrease the value of k by `size(p.left) + 1` to compensate for the elements we have skipped over.

```
Key getKth(int k, Node p) {
    if (p == null) return null
    else if (size(p.left) >= k) return getKth(k, p.left)
    else if (size(p.left) == k-1) return p.key
    else return getKth(k - (size(p.left) + 1), p.right)
}
```

Since we recurse on only one subtree, this clearly runs in time proportional to the height of the tree.

Solution 3: See Fig. 1.

Solution 4:

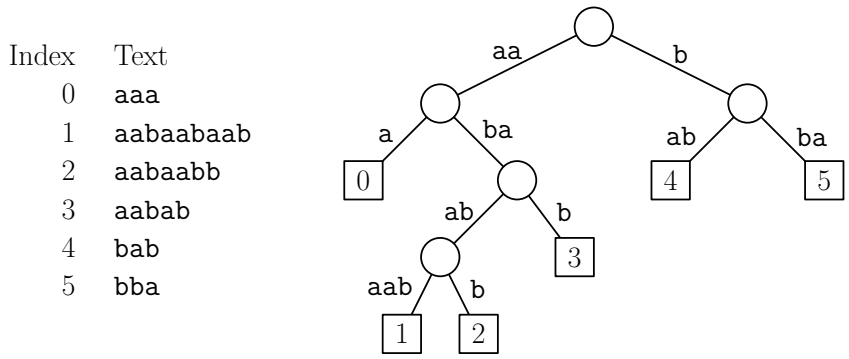

Figure 1: Patricia trie.

(a) The 16-word block starting at address 16 is broken into blocks of size 8, 4, 2, and 2, and the block of size 2 starting at 16 is returned as the answer.

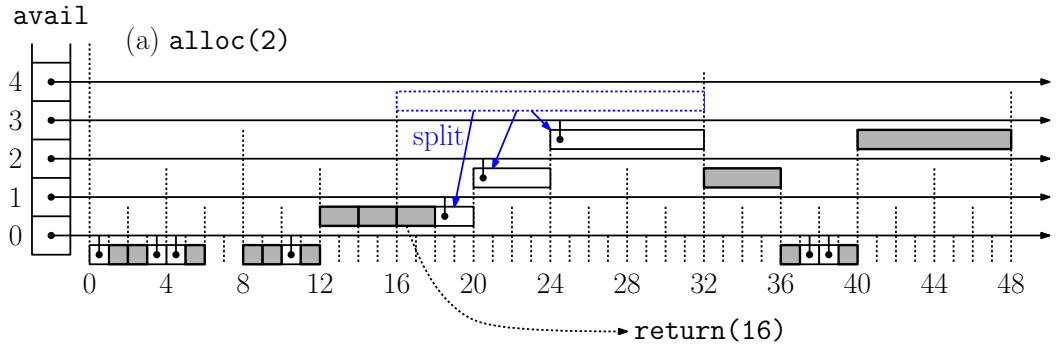

Figure 2: Buddy system allocation.

(b) The deallocated block is merged with its buddy of size 1 at 27, then its buddy of size 2 at 24, then its buddy of size 5 at 28, then its buddy of size 8 at 16, resulting in a block of size 16 at 16.

Solution 5:

(a) Let's assume that we have two utilities to help us classify cells. Define the *region of interest* to be the set of points (x, y) where $x \leq d$ and $y \geq t$. The first utility determines whether a cell is disjoint from the region of interest, and the second determines whether it is contained within it.

```
boolean contained(Point pt, int d, int t) { return pt.x <= d && pt.y >= t }
boolean contained(Rect r, int d, int t) { return r.hi.x <= d && r.lo.y >= t }
boolean isDisjoint(Rect r, int d, int t) { return r.lo.x > d || r.hi.y < t }
```

We 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. If we fall out of the

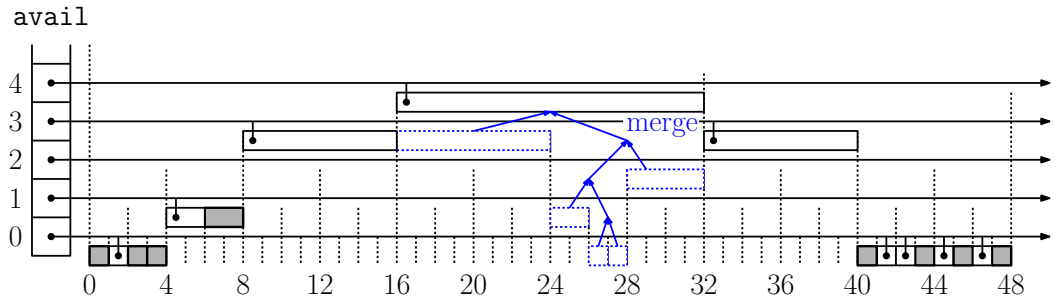

Figure 3: Buddy system deallocation.

tree or if the cell is disjoint from the strip, we return the current best. If the node's point is in the region of interest and to the right of the current best, it replaces best. Otherwise, we construct the two child cells. If the node is a horizontal splitter (`p.cutDim == 1`), we search both subtrees. If it is a vertical splitter, we try the right subtree first. If the left subtree is still relevant to the search (`p.point.x > best.x`), then we visit it as well. The initial call at the root level is `tempQuery(d, t, root, bbox, null)`, where `bbox` is the kd-tree's bounding box.

```
double tempQuery(double d, double t, KDNode p, Rect cell, double best) {
  if (p == null || isDisjoint(cell, d, t)))          // trivial cases
    return best
  if (contained(p.point, d, t) && (best == null || p.point.x > best))
      best = p.point.x                                // new best
                                                      // children cells
  Rect leftCell = cell.leftPart(p.cutDim, p.point)
  Rect rightCell = cell.rightPart(p.cutDim, p.point)

  if (p.cutDim == 1) {                                  // horizontal cut?
    best = tempQuery(d, t, p.left, leftCell, best)   // try both sides
    best = tempQuery(d, t, p.right, rightCell, best)
  } else {                                            // vertical cut?
    best = tempQuery(d, t, p.right, rightCell, best) // try right first
    if (best == null || p.point.x > best.x)           // is left relevant?
      best = tempQuery(d, t, p.left, leftCell, best) // try left
  }
  return best
}
```

(b) (2): The running time of the algorithm is $O(\sqrt{n})$ under the standard assumptions about kd-trees. To see this, observe first that, by applying the standard orthogonal range search analysis for kd-trees, the number of nodes whose cells are stabbed by the sides of the region of interest is $O(\sqrt{n})$. If the cell is outside the strip, the search will return immediately.

It remains is to analyze the nodes whose cell lies entirely inside the region of interest. For each vertical splitter, we never need to visit the left child (since the point in this node dominates the left subtree). Therefore, we will visit at most two out of the four grandchildren of any

such node. Thus, the number of nodes visited satisfies the recurrence $T(n) = 2T(n/4)+O(1)$, which solves to $O(\sqrt{n})$.

Solution 6: The answer is based on a 2-dimensional range tree. There are two approaches. The first is to sort the primary tree by y-coordinates in order to filter out high temperatures, then build the auxiliary tree sorted by x-coordinates, and use `findDown` to locate the largest date to the left of the query point. The alternative is to use the classical ordering (x- then y-), but each node of the auxiliary tree stores its descendant with the maximum x-coordinate. We take the maximum of all of the eligible entries. We will describe the former approach.

(a) The data structure is a two-layer range tree. The primary (first-level) tree stores the points sorted by their y-coordinates (temperature). The second-layer auxiliary trees each store points of the associated primary subtree sorted by their x-coordinates (date). We assume that each auxiliary tree has access to a function `findDown(d)`, which finds the largest entry that is less than or equal to `d`.

Since this is a standard two-layer range tree, the space is $O(n \log n)$.

(b) We answer a query as follows. Let $(\mathtt{d}, \mathtt{t})$ denote the query point. We first apply the standard range search to the primary tree to identify all the nodes u corresponding to the maximal subtrees whose y-coordinates $\geq \mathtt{t}$. For each of these trees, we invoke `findDown(d)` to find the point with the largest date that is $\leq \mathtt{d}$. Among these, we return the one with the largest date value.

There are $O(\log n)$ such maximal subtrees and each query takes time $O(\log n)$, for a total query time of $O(\log^2 n)$.

It is tempting to think that this can be solved with a 1-dimensional range tree as follows. The primary tree stores points sorted by the x-coordinate (date) and for each node u in this tree, we store the highest temperature of any descendant of u. To answer a query $(\mathtt{d}, \mathtt{t})$, we identify a maximal set of subtrees that cover the dates that are $\leq \mathtt{d}$, and we select the maximum temperature in this subtree. We repeat this from right to left until we find the first whose maximum temperature exceeds `t`. This fails, however, since the highest temperature in this subtree may not be the most recent one prior to `d`.

Solution 7: First observe that each `union` takes $O(1)$ time, and since there are at most m `unions`, the total cost for all the `unions` is $O(m)$. To bound the time spent in the `finds` we classify the tree links as being of two types. Links that go directly to a root are colored green and all others are colored red. Since there are at most m `unions`, there are at most $m - 1$ links in the tree. As we perform `finds`, path compression causes the red links along the search path to become green.

The time needed to perform any `find` is proportional to the number red links traversed plus the number of green links traversed. Since each `find` can traverse at most one green link (it will be the last link on the path) the total cost for traversing green links is equal to the number of `finds`, which is at most m. To bound the total time spent traversing red links, observe that every time we traverse a red link as part of a `find`, its color changes to green as a result of path compression. Since the `unions` all precede the `finds`, once a link becomes green it cannot change back to red. Since there are at most m red links, we cannot spend more than $O(m)$ time traversing these red links until they are all changed to green. Thus the total time spent in the `finds` is $O(m)$.