Solutions to the Final Exam

Solution 1:

(1.1) An extended binary tree with \( n \) internal nodes has \( n + 1 \) external nodes.

(1.2) Only (c): in an inorder traversal, internal and external node alternate with each other.

(1.3) 2: In AVL tree-insertion, after the first rotation operation (single or double) the subtree to which the rotation is applied has exactly the same height it did prior to the insertion. It follows that this subtree and all the others in the tree are properly balanced with respect to the AVL height criteria. Since we count a double rotation as two rotations, the answer is 2.

(1.4) \( O(\log n) \): In AVL-tree deletion, the subtree height may change after a rotation, and this may result in the parent’s balance factors being altered. This pattern can propagate all the way to the root.

(1.5) Min: 0, Max: \( h + 1 \). A tree of height \( h \) has \( h + 1 \) levels of nodes. Each node at a given level may give rise to a single red node (if the path goes through a 3-node) or not (if the path goes through a 2-node).

(1.6) This sort of efficiency is called static optimality (where the access probabilities are non-uniform but do not change over time). Among the dictionary data structures we saw this semester, only the splay tree is efficient with respect to static optimality, since it restructures itself so that more frequently accessed keys are placed near the root of the tree.

(1.7) The treap data structure has the property that it behaves the same as a standard (unbalanced) binary search tree in which the keys are inserted in priority order. In this case, the priorities are the same as the insertion order, this is exactly the behavior that the treap will exhibit in this case.

(1.8) The skip list would be preferred. The expectation in the performance of the binary search tree is over all possible insertion orders and the expectation in the skip list is over the algorithm’s random choices. Thus, if there is some systematic bias in the insertion order, the binary search tree will always perform poorly while the skip list will only rarely perform poorly.

(1.9) Internal fragmentation refers to the wastage of memory within (as opposed to between) the allocated blocks. The variable-sized allocator only allocates one additional word for each block allocated. In contrast, the buddy system may waste up to half of the allocated block by rounding the size up to the next power of 2.

Solution 2:

(2.1) The original tree and the corresponding AA tree are both shown in Fig. 1.

(2.2) The intermediate steps are shown in Fig. 2, and the final result is shown in the middle of the lower row.
Solution 3:

(a) Since $h(x) = 0$, it visits the table indices $0, 1, 4, 9, 16, 25, \ldots, i^2$.

(b) The line $c = c \% m$ causes the index to wrap-around, which means that we will never index outside the array bounds.

(c) Let $c_i$ denote the value of $c$ after the $i$th pass through the loop. After the $i$th pass through the loop, the value of $c_i$ satisfies $c_i = c_{i-1} + 2i - 1$. We assert that $c_i = i^2$. The basis case of $i = 0$ is trivial. Otherwise, let us assume inductively that $c_{i-1} = (i-1)^2$. Then, we have

$$c_i = c_{i-1} + 2i - 1 = (i-1)^2 + 2i - 1 = (i^2 - 2i + 1) + 2i - 1 = i^2,$$

as desired. If you are not fond of algebra, the figure below provides a nice pictorial proof of this fact.

![Figure 1: 2-3 and AA tree correspondence.](image1)

![Figure 2: AA insertion.](image2)
**Solution 4:** There are a number of cases to consider. First, if $p$ is the root, it has no predecessor. Otherwise, if $p$ is a left child, then its preorder predecessor is its parent (see Fig. 3(a)). If $p$ is a right child, there are two cases. If its parent has no left child, then its preorder predecessor is its parent (see Fig. 3(b)).

![Figure 3: Cases arising in computing the preorder predecessor.](image)

Otherwise, $p$’s parent has a left child. Let $q$ be this child (see Fig. 3(c)). The desired node is the last preorder node in $q$’s subtree. Computing this correctly takes a bit of thought. The key observation is that such a node must be a leaf (since an internal node comes earlier in preorder than either of its children). If a node has a single right child, the last preorder node comes from this child. If it has just a left child, it will come from there. We will give a recursive function to implement this (see the function `preorderLast` in the code block below).

```java
Node preorderPred(Node p) { // p's preorder predecessor
    if (p.parent == null) // p is the root?
        return null; // ...no predecessor
    else if (p == p.parent.left) // p is a left child?
        return p.parent; // ...parent is predecessor
    else { // p must be a right child
        if (p.parent.left == null) // no left sibling?
            return p.parent; // ...parent is predecessor
        else {
            return preorderLast(p.parent.left); // preorder last of parent's left
        }
    }
}

Node preorderLast(Node q) { // preorder last in q's subtree
    if (q.right != null) // right subtree is non-empty?
        return preorderLast(q.right); // ...look for it here
    else if (q.left != null) // left subtree is non-empty?
        return preorderLast(q.left); // ...look for it here
    else // arrived at a leaf
        return q; // ...this is it!
}
```

**Solution 5:** A cool observation is that this problem, which apparently has to do with line segments can be answered by a data structure that just stores points! It is not hard to see that answering a left-to-right horizontal ray-shooting query at point $q = (q_x, q_y)$ is equivalent to computing the point
of \( P \) with the minimum \( x \)-coordinate that lies within the northeast quadrant of \( q \) (see Fig. 4). To see why, observe that in order to hit any segment, its topmost point must lie to \( q \)'s right and have a higher \( y \)-coordinate, thus it lies in \( q \)'s northeast quadrant. We seek the first such point, that is, the one with the lowest \( x \)-coordinate. A point lying in \( q \)'s northeast quadrant is called a candidate (points \( \{p_8, p_9, p_{10}\} \) in the figure), and the best candidate is the one with the smallest \( x \)-coordinate.

Figure 4: Horizontal ray shooting via the kd-tree.

We 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. We pass in \( p \)'s cell into the function as the parameter cell. The point best is the best among all candidates seen so far. The initial call at the root level is \( \text{rayShoot}(q, \text{root}, \text{boundingBox}, \text{sentinel}) \), where sentinel is the point \((+\infty, +\infty)\). If this point is returned from the search, we return null as the answer.

When we visit a node \( p \), we consider the relationship of \( p \)'s cell to \( q \)'s northeast quadrant. If \( p \) is null or if its cell does not overlap the quadrant (\( \text{cell.high.x} < q.x \) or \( \text{cell.high.y} < q.y \)), we may ignore this node and its contents, since it cannot possibly provide a candidate.

```java
Point ray Shoot(Point q, KDNode p, Rectangle cell, Point best) {
    if (p == null) // fell out of tree?
        return best;
    else if (cell.high.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;
    }
}
```

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 smaller \( x \)-coordinate than \( best \)). If so, we update
best. Finally, we recurse on p’s two children, and keep the point with the smaller x-coordinate. We use the utility functions leftPart and rightPart to compute the cells associated with the left and right children.

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 left child before the right child, since it is more likely to yield a point with a higher x-coordinate. Second, if cell lies entirely to the right of best (that is, cell.low.x > best.x), there is no need to visit this cell, since any candidate it provides cannot be better than the current best.

Solution 6: It is tempting to maintain a single variable that stores the current minimum, but when the minimum element is popped off the stack, we would need to find a new minimum, which would take O(n) time. (Note that if elements are pushed in random order, then the probability that the minimum is at the top of the stack of size n is 1/n, and hence the expected amortized time would be (1/n)n = 1. But the assumption of randomness is critical for this to work.)

Our solution is to maintain two parallel stacks. The first, stack, stores the standard stack contents. The second, min, maintains the invariant that its top element is the minimum among all the items in the stack. Letting top denote the index of the stack’s top, getMin simply returns min[top]. Whenever a new element is pushed, we update stack in the standard manner, and we push on min the minimum of the new item or the previous min. To pop the stack, we simply pop both stacks simultaneously. The pseudo-code appears in the following code block.

```java
class MinStack {
    int top; // top of stack
    int stack[]; // stack contents
    int min[]; // minimum in stack

    MinStack(int n) { // constructor
        stack = new int[n];
        min = new int[n];
        top = -1;
    } // operations (no error checking)

    boolean isEmpty( return top == -1; )
    int pop() { return(stack[top--]); }
    int getMin() { return(min[top]); }

    void push(int x) { // push x on stack
        int newMin = (isEmpty() ? x : Math.min(x, min[top])); // new minimum
        stack[++top] = x; // push x
        min[top] = newMin; // push new minimum
    }
}
```

Solution 7:

(7.1) This layout is the standard one used in the HeapSort algorithm. Basically, moving up and down the tree corresponds to halving and doubling indices. We can compute these using
the usual arithmetic operations or by bit operations (where left shift doubles and right shift halves).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arithmetic</th>
<th>Bitwise</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>left(x)</td>
<td>2*x</td>
<td>x&lt;&lt;1</td>
<td></td>
</tr>
<tr>
<td>right(x)</td>
<td>2*x + 1</td>
<td>(x&lt;&lt;1) + 1</td>
<td></td>
</tr>
<tr>
<td>parent(x)</td>
<td>x/2</td>
<td>x &gt;&gt; 1</td>
<td>[x/2]</td>
</tr>
<tr>
<td>sibling(x)</td>
<td>x+1-(x%2)</td>
<td>x^1</td>
<td>Flip lowest order bit</td>
</tr>
</tbody>
</table>

(7.2) This layout is much harder than the heap layout. We will focus primarily on using bit operations. The figure below shows the indices as bit strings:

![Binary Tree Diagram](image)

It will be helpful to first derive a function that yields the level of a node (where leaves are at level 0). Observe that the indices of nodes at level \( k \) are multiples of \( 2^k \). Thus, \( x \)'s level equals the number of times we can halve \( x \) until it becomes odd.

```c
int level(int x) {
    int k = 0;
    while (x % 2 == 0) { k++; x = x / 2; }
    return k;
}
```

In the following, level \( k = \text{level}(x) \). The quantity \( 2^k \) will be important, and we can compute it easily by shifting a 1-bit over \( k \) positions, that is, \( 2^k \equiv (1 << k) \).

It is easy to see that the left and right children of any node are just negative and positive offsets by \( 2^{k-1} \), or equivalently \( x - (1 << (k-1)) \) and \( x + (1 << (k-1)) \), respectively.

Sibling is a bit simpler than parent. It is easiest to see the pattern at the leaf level. Two leaf nodes are siblings if they have the bit pattern 01 and 11 in their lowest order bits. Thus, to convert from one sibling to the other, we should flip bit-1. This is equivalent to \( x^10 \) or alternately, \( x^{1<<1} \). To generalize this to level \( k \), the flipped bit is bit \( k + 1 \). Thus we have

```
sibling(x) :: (x ^ (1 << (k+1))
```

Parent is trickier. Again, let’s start at the leaf level. If we consider just the two low order bits, we see that the left, parent, and right have the bit patterns 01, 10, and 11, respectively. Thus, we can compute the parent by subtracting 1 (thus zeroing out the lowest order bit) and then or-ing with the bit pattern 10. Thus, leaf-level parent can be computed as \( (x-1) | 10 \) or equivalently \( (x-1) | (1<<1) \). This can be generalized to higher levels of the tree by shifting everything \( k \) bits to the left. Thus, \( x-1 \) becomes \( x - (1 << k) \) and \( 1<<1 \) becomes \( 1<<(k+1) \). We have
In summary we have the following rules. (We have omitted the arithmetic rules, since they are somewhat messier.)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arithmetic</th>
<th>Bitwise</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>left(x)</td>
<td>$x - 2^{k-1}$</td>
<td>$x - (1 &lt;&lt; (k-1))$</td>
<td></td>
</tr>
<tr>
<td>right(x)</td>
<td>$x + 2^{k-1}$</td>
<td>$x + (1 &lt;&lt; (k-1))$</td>
<td></td>
</tr>
<tr>
<td>sibling(x)</td>
<td>$x \oplus (1 &lt;&lt; (k+1))$</td>
<td>$x \oplus (1 &lt;&lt; (k+1))$</td>
<td>Flip bit $k + 1$</td>
</tr>
<tr>
<td>parent(x)</td>
<td>$(x - (1 &lt;&lt; k)) \mid (1 &lt;&lt; (k+1))$</td>
<td>$(x - (1 &lt;&lt; k)) \mid (1 &lt;&lt; (k+1))$</td>
<td></td>
</tr>
</tbody>
</table>