Solution 1:

(a) \( n + 1 \): There is one thread for every null child link, and (as shown in class) a binary tree with \( n \) nodes has \( n + 1 \) null child links.

(b) True: If there are no duplicate keys, then in a max-heap, the node with the smallest key cannot have any children, and hence it must be a leaf.

(c) Min: 7, Max: 26. There is a minimum of 7 (all are 2-nodes) nodes in a 2-3 tree of height two and a maximum of 13 (all are 3-nodes). The number of keys per node is one in the first case and two in the second.

(d) Min: 0, Max: \( h + 1 \): There may be no splits at all. In the worst case, every node on the search path (of which there are \( h + 1 \)) may be forced to split.

(e) (1) and (3): The worst-case time of \( O(\log n) \) is guaranteed for find and delete. This is because the tree height is guaranteed to be \( O(\log n) \) (due to rebuilding). Deletion might normally take \( O(n) \) time, but since we have performed no deletions yet, the condition that triggers a deletion rebuild will not apply, so deletion will also take \( O(\log n) \) time. Depending on the tree structure, an insertion could cause a major rebuilding, which could take up to \( O(n) \) time.

(f) (1): Allocated blocks only have the header. Free blocks store items (1)–(3). Item (4) is not an element of the system described in class.

(g) (4): If, by luck, the adjacent key was pulled up close to the root, then (1) would apply, but there is no guarantee that this will happen. For example, if \( x \) was the original root, then the first splay would do nothing, and

(h) (3) and (4): The insertion process involves computing \( k \) hash functions and setting the corresponding bits, thus it take \( O(k) \) time. As you increase \( m \), the bits are more spread out, and the probability of hitting a given pattern grows smaller. This does not apply to \( k \), however. If \( k \) becomes too large the number of 1-bits in the table may grow so large that almost any bit pattern will be found by accident.

Solution 2: This is proved by induction on the height of the tree. For the basis cases, observe that an AVL tree of heights \( h = 0 \) or 1 has a root node, and so it is full at depth \( \lfloor h/2 \rfloor = 0 \).

Let us make the (strong) induction hypothesis that for any \( h \geq 2 \), an AVL tree of strictly smaller height \( h' < h \) is full at level \( \lfloor h'/2 \rfloor \), and we will use this to prove the result for \( h \) itself.

An AVL tree of height \( h \) is formed from two AVL trees, one of height exactly \( h - 1 \) and the other of height either \( h - 1 \) or \( h - 2 \). By the induction hypothesis, both these subtrees are all full up to depth at least \( \lfloor (h - 2)/2 \rfloor \). Therefore, by including the root level, the entire tree is full at
one higher depth, that is, \( \left\lfloor \frac{h - 2}{2} \right\rfloor + 1 \). Using the identity that \( \left\lfloor x - 1 \right\rfloor = \left\lfloor x \right\rfloor - 1 \), we conclude that the entire tree is full up to depth
\[
\left\lfloor \frac{h - 2}{2} \right\rfloor + 1 = \left\lfloor \frac{h}{2} - 1 \right\rfloor + 1 = \left\lfloor \frac{h}{2} \right\rfloor - 1 + 1 = \left\lfloor \frac{h}{2} \right\rfloor,
\]
as desired.

**Solution 3:**

(a) Inserting 13 first finds the leaf node \([12, 14, 15]\) in which the key is to be inserted. The insertion results in an overflow (see Fig. 1). Since both siblings ([6, 8, 9] and [20, 21, 24]) have the maximum number of allowed keys, key rotation (adoption) is not an option. So, we split this node, generating \([12]\) and \([14, 15]\) and promoting the middle key 13. The parent node now contains \([5, 10, 13, 17]\) which is too full. In this case, however, we can perform a key rotation with the right neighbor. We rotate 17 up, 27 down, and move the subtree \([20, 21, 24]\) over to the sibling.

(b) Deleting 42 first find the containing node \([42]\) and removes it (see Fig. 2). This leaves a node with zero keys. The sibling \([49]\) has only a single key, and so key rotation is not possible. We merge with our sibling \([35]\), which causes the parent \([40]\) to be demoted.
Solution 4: See Fig. 3.

Solution 5: The key observation is that the set of segments hit by the ray is equivalent to the set of upper endpoints that lie within the infinite axis-aligned rectangle $x \leq q.x$ and $y \geq q.y$. Thus, this is an instance of orthogonal range counting involving the semi-infinite rectangle to the northwest of $q$.

(a) We make the standard assumptions that the cutting dimension alternates between vertical and horizontal and that subtrees are balanced, in the sense that they have (roughly) equal numbers of points. The tree stores an axis-aligned rectangle, rootCell, which contains all the points. We will also assume that each node $p$ has a member $p.size$, which indicates the number of points that lie within the subtree rooted at $p$.

(b) The pseudocode is a very minor adaptation of the orthogonal range counting algorithm given in class. The initial call is $seghit(q, root, rootCell)$.

```c
int seghit(Point q, KDNode p, Rect cell) {
    if (p == null) return 0 // empty subtree
    else if (cell.lo.x > q.x || cell.hi.y < q.y) // cell is disjoint?
        return 0
    else if (cell.hi.x <= q.x && cell.lo.y >= q.y) // entire cell in range?
        return p.size
    else { // cell overlaps
        int count = 0
        if (p.point.x <= q.x && p.point.y >= q.y) // consider this point
            count += 1 // recurse on children
        count += rangeCount(R, p.left, cell.leftPart(p.cutDim, p.point))
        count += rangeCount(R, p.right, cell.rightPart(p.cutDim, p.point))
        return count
    }
}
```
(c) Since this just a minor modification of range searching, the running time is $O(\sqrt{n})$, as shown in class.

**Solution 6:**

(a) We create a helper, \texttt{cmHelper(Node p, int c)} that takes a current node \texttt{p}, a current character value \texttt{c}, and returns a count of all the strings in the subtree rooted at \texttt{p} whose current character values are numerically greater than or equal to \texttt{c}. Whenever we visit a subtree with some character value \texttt{i}, we set \texttt{c} to \texttt{i}, so we only visit subtrees of greater or equal character value. The initial call is \texttt{cmHelper(root, 0)} (see Fig. 4).

```
int cmHelper(Node p, int c)
    if (p.isExternal) return 1 // external? Count it
    else { // internal?
        sum = 0
        for (int i = c; i < 3; i++) { // recurse on >= char values
            sum += cmHelper(p.child[i], i)
        }
        return sum
    }
```

(b) We follow the same strategy as in (a), but tailored to the different type of tree. We create a helper that takes a current node \texttt{p}, a current character value \texttt{c}, and returns a count of all the strings in this subtree whose character values are numerically greater than or equal to \texttt{c}. As before, when we reach an external node we count it. If we encounter a node whose key value is at least as large as \texttt{c}, we recurse into its subtree. In either case, we recurse on it next sibling (see Fig. 5).

```
int cmHelper(Node p, int c)
    if (p == null) return 0 // fell off a right chain?
    else if (p.isExternal) return 1 // external? Count it
    else { // internal?
        ct = 0
        if (p.key >= c) { // recurse on >= char values
            for (int i = c; i < 3; i++) { // recurse on >= char values
                sum += cmHelper(p.child[i], i)
            }
            return sum
        }
    }
```

![Figure 4: Monotone Strings: Standard trie.](image-url)
ct += cmHelper(p.firstChild, p.key)
}
ct += cmHelper(p.nextSibling, c) // continue with sibling
return ct // return the final count
}
}

Figure 5: Monotone Strings: de la Briandais trie.

Solution 7:

(a) The following function is almost correct. It follows the probe sequence for \( x \) until either (1) it finds a matching key, (2) it finds a null entry, or (3) it finds an entry of lower priority. In the first case, we have clearly found \( x \). In the second case (because there are no deletions) \( x \) cannot be in the table, since it would have been inserted at this null entry. In the third case, we also infer that \( x \) is not in the table, because if we arrived at this entry, we would have stored \( x \) here and bumped the existing item of lower priority.

```java
boolean find(Key x) { // find key x in table T[0..m-1]
    int i = h(x)
    return findHelper(x, i)
}

boolean findHelper(Key x, int i) { // try to insert x at T[i]
    i = i % m // wrap around
    Key y = T[i] // y is the key at T[i]
    if (y == x) { // keys match?
        return true // ...success
    } else if (y == null || p(y) < p(x)) { // not here
        return false // ...failure
    } else { // might still be in table
        return findHelper(x, i + g(x)) // ...keep looking
    }
}
```

There is a small bug here. If \( x \) is not in the table, but it has a lower priority than every cell we probe, we may go into an infinite loop. To remedy this, we can check whether we ever loop all the way around (\( i == h(x) \)). If so, we return `false`. 
(b) True. The obvious answer is that since \( g(x) \) is prime relative to \( m \) for all keys \( x \), we will never repeat an index until trying all available slots in the table.\(^1\) However, this simple answer fails to consider the effect of switching offsets between keys. Suppose, for example that \( m = 64 \), \( g(x) = 3 \) for half the keys in the table and \( g(x) = 5 \) for the other half. Both 3 and 5 are prime relative to 64. But since the offset varies based on the priorities, half of the time we might use 3 as the offset, and the other half of the time we might use 5 as the offset. So, it may be that with every two probes, our effective offset is \( 3 + 5 = 8 \), which does share a common factor with \( m \). Thus, we could go into an infinite loop, repeatedly alternating between offsets of size 3 and 5.

What saves us is the fact that the bumping process switches offset values only when we come to a key of lower priority. There are a finite number of keys, and eventually we hit the smallest priority we will ever encounter, and we stop changing the offset value. Thus, in the above case, we cannot alternate infinitely between 3 and 5. We must settled down to using just one, and then we are guaranteed to find an empty slot. Here now is the proof.

Suppose to the contrary that the table is not full, but the priority hashing insert algorithm goes into an infinite loop. Each time we change offset values, we are continuing with a key of lower priority. Because there are finitely many keys, and hence finitely many priority values, at some point, we will arrive at the lowest priority we will ever see. Call this key \( z \). Because the offset value \( g(z) \) and the table size \( m \) are relatively prime, the probe sequence \( (i \cdot g(z)) \mod m \) will hit all of the integers from 0 to \( m - 1 \). Therefore, if there is an empty slot, we will succeed in inserting the key, a contradiction.

(c) False. To see this, suppose that \( m \), \( h(x) \), and \( g(x) \) are all even. Also, suppose that \( x \) has the lowest priority. We will use \( g(x) \) to move the key \( x \) around the table, but it can only hit cells with indices of the form \( (h(x) + i \cdot g(x)) \mod m \), which are all even. If the empty entries are in odd valued indices, we will never find one of them.

\(^1\)To see why, suppose that for two different integers \( i \) and \( j \), we had \( (i \cdot g(z)) \equiv (j \cdot g(z)) \pmod{m} \). By simple manipulations, we have \( (i \cdot g(z)) - (j \cdot g(z)) \equiv 0 \pmod{m} \), or equivalently \( (i - j)g(z) \equiv 0 \pmod{m} \). Since \( g(z) \) and \( m \) are relatively prime, it must be that \( (i - j) \equiv 0 \pmod{m} \). This implies that \( |i - j| \geq m \). In other words, before we encounter any repetitions of indices we must probe at least \( m \) distinct entries. Thus, if there is an empty entry, we will find it.